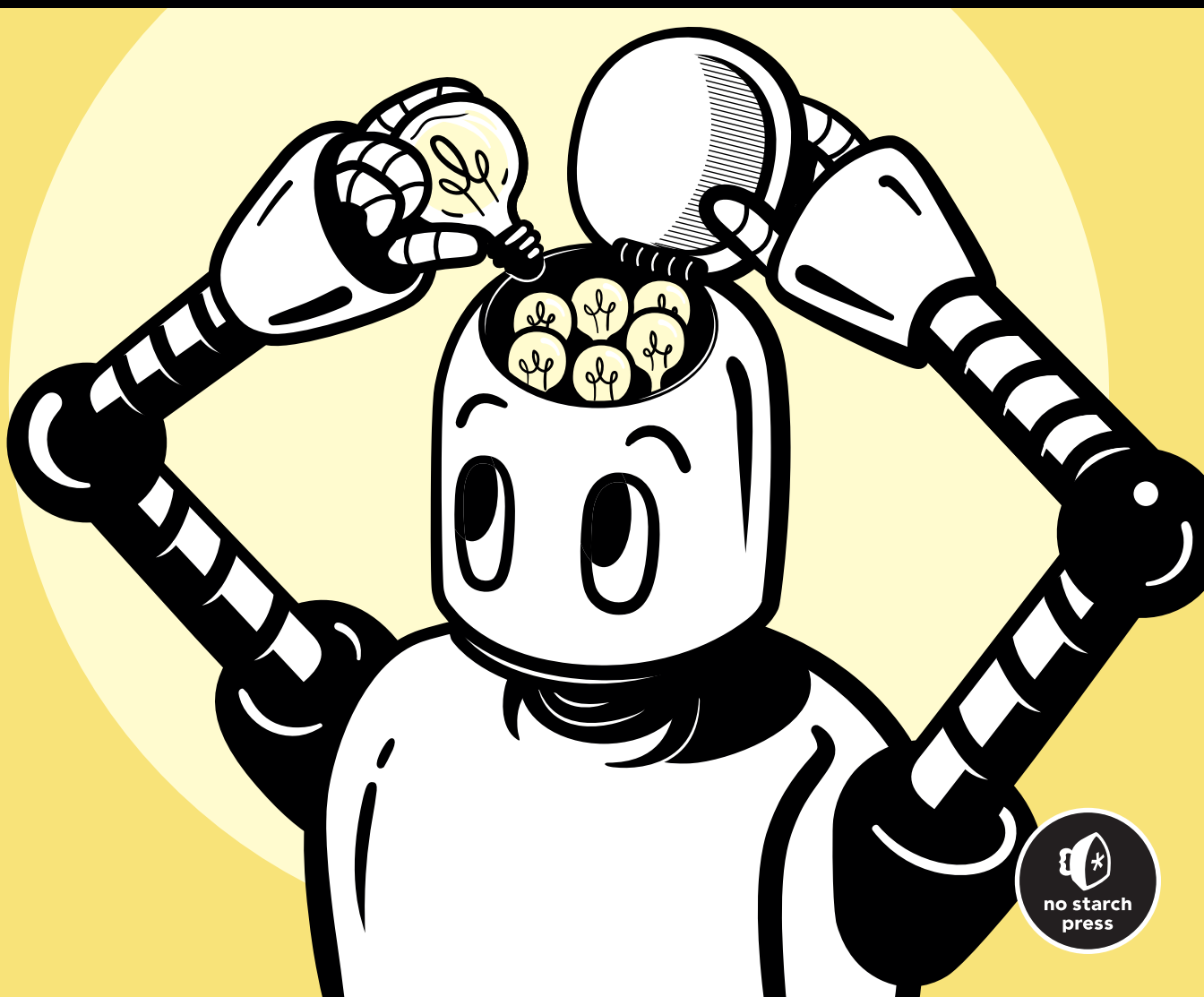
FULL COLOR

DEEP LEARNING

A VISUAL APPROACH

ANDREW GLASSNER



no starch press

DEEP LEARNING

DEEP LEARNING

A Visual Approach

Andrew Glassner



no starch
press

San Francisco

DEEP LEARNING: A VISUAL APPROACH. Copyright © 2021 by Andrew Glassner.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13: 978-1-7185-0072-3 (print)
ISBN-13: 978-1-7185-0073-0 (ebook)

Publisher: William Pollock
Executive Editor: Barbara Yien
Production Editors: Maureen Forys and Rachel Monaghan
Developmental Editor: Alex Freed
Cover and Interior Design: Octopod Studios
Cover Illustrator: Gina Redman
Technical Reviewers: George Hosu and Ron Kneusel
Copyeditor: Rebecca Rider
Compositor: Maureen Forys, Happenstance Type-O-Rama
Proofreader: James Fraleigh

With the exception of the images noted at the end of the book in the Image Credits, all the images in this book are produced by the author. All original images may be freely downloaded from *https://github.com/blueberrymusic* and used as the reader pleases.

For information on book distributors or translations, please contact No Starch Press, Inc. directly:
No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1-415-863-9900; info@nostarch.com
www.nostarch.com

Library of Congress Cataloging-in-Publication Data

```
Names: Glassner, Andrew S., author.
Title: Deep learning : a visual approach / Andrew Glassner.
Description: San Francisco, CA : No Starch Press, Inc., [2021] | Includes
   bibliographical references and index.
Identifiers: LCCN 2020047326 (print) | LCCN 2020047327 (ebook) | ISBN
   9781718500723 (paperback) | ISBN 9781718500730 (ebook)
Subjects: LCSH: Machine learning. | Neural networks.
Classification: LCC Q325.5 .G58 2021  (print) | LCC Q325.5  (ebook) | DDC
   006.3/1--dc23
LC record available at https://lccn.loc.gov/2020047326
LC ebook record available at https://lccn.loc.gov/2020047327
```

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

For Niko,
who's always there
with a smile,
a paw,
and a wag.

About the Author

Dr. Andrew Glassner is a Senior Research Scientist at Weta Digital, where he uses deep learning to help artists produce visual effects for film and television. He was Technical Papers Chair for SIGGRAPH '94, Founding Editor of the *Journal of Computer Graphics Tools*, and Editor-in-Chief of *ACM Transactions on Graphics*. His prior books include the *Graphics Gems* series and the textbook *Principles of Digital Image Synthesis*. Glassner holds a PhD from UNC-Chapel Hill. He paints, plays jazz piano, and writes novels. His website is *www.glassner.com*, and he can be followed on Twitter as *@AndrewGlassner*.

About the Technical Reviewers

George Hosu is a software engineer and world traveler with a broad interest in statistics and machine learning. He works as the lead machine learning engineer on an autoML and "explainable AI" project called Mindsdb. In his spare time, he writes to strengthen his understanding of epistemology, ML, and classical mathematics, and how they all come together to generate a meaningful map of the world. You can find his writing at *https://blog.cerebralab.com/*.

Ron Kneusel has been working with machine learning in industry since 2003 and completed a PhD in machine learning from the University of Colorado, Boulder, in 2016. Ron currently works for L3Harris Technologies, Inc. He has two books available from Springer: *Numbers and Computers*, and *Random Numbers and Computers*.

BRIEF CONTENTS

Acknowledgments xxi

Introduction xxiii

PART I: FOUNDATIONAL IDEAS 1

Chapter 1: An Overview of Machine Learning 3

Chapter 2: Essential Statistics 15

Chapter 3: Measuring Performance 47

Chapter 4: Bayes' Rule 83

Chapter 5: Curves and Surfaces 117

Chapter 6: Information Theory 133

PART II: BASIC MACHINE LEARNING 153

Chapter 7: Classification 155

Chapter 8: Training and Testing 181

Chapter 9: Overfitting and Underfitting 195

Chapter 10: Data Preparation 221

Chapter 11: Classifiers 263

Chapter 12: Ensembles 297

PART III: DEEP LEARNING BASICS 311

Chapter 13: Neural Networks 313

Chapter 14: Backpropagation 351

Chapter 15: Optimizers 387

PART IV: BEYOND THE BASICS 427

Chapter 16: Convolutional Neural Networks 429

Chapter 17: Convnets in Practice 473

Chapter 18: Autoencoders 495

Chapter 19: Recurrent Neural Networks 539

Chapter 20: Attention and Transformers 565

Chapter 21: Reinforcement Learning 601

Chapter 22: Generative Adversarial Networks 649

Chapter 23: Creative Applications 675

References 693

Image Credits 717

Index 721

CONTENTS IN DETAIL

ACKNOWLEDGMENTS xxi

INTRODUCTION xxiii

Who This Book Is For xxiv
This Book Has No Math and No Code xxv
There Is Code, If You Want It xxv
The Figures Are Available, Too! xxv
Errata xxvi
About This Book xxvi
Part I: Foundational Ideas xxvi
Part II: Basic Machine Learning xxvii
Part III: Deep Learning Basics xxvii
Part IV: Deep Beyond the Basics xxvii
Final Words xxviii

PART I: FOUNDATIONAL IDEAS 1

1
AN OVERVIEW OF MACHINE LEARNING 3

Expert Systems 4
Supervised Learning 6
Unsupervised Learning 8
Reinforcement Learning 9
Deep Learning 10
Summary 13

2
ESSENTIAL STATISTICS 15

Describing Randomness 16
Random Variables and Probability Distributions 17
Some Common Distributions 21
Continuous Distributions 21
Discrete Distributions 26
Collections of Random Values 28
Expected Value 28
Dependence 29
Independent and Identically Distributed Variables 29
Sampling and Replacement 29
Selection with Replacement 30
Selection Without Replacement 30
Bootstrapping 31
Covariance and Correlation 35
Covariance 36
Correlation 37

Statistics Don't Tell Us Everything . . . 40
High-Dimensional Spaces . . . 42
Summary . . . 44

3
MEASURING PERFORMANCE **47**

Different Types of Probability . . . 48
Dart Throwing . . . 48
Simple Probability . . . 50
Conditional Probability . . . 50
Joint Probability . . . 53
Marginal Probability . . . 55
Measuring Correctness . . . 56
Classifying Samples . . . 57
The Confusion Matrix . . . 60
Characterizing Incorrect Predictions . . . 61
Measuring Correct and Incorrect . . . 63
Accuracy . . . 64
Precision . . . 65
Recall . . . 66
Precision-Recall Tradeoff . . . 67
Misleading Measures . . . 69
f1 Score . . . 71
About These Terms . . . 72
Other Measures . . . 72
Constructing a Confusion Matrix Correctly . . . 74
Summary . . . 81

4
BAYES' RULE **83**

Frequentist and Bayesian Probability . . . 84
The Frequentist Approach . . . 84
The Bayesian Approach . . . 85
Frequentists vs. Bayesians . . . 85
Frequentist Coin Flipping . . . 86
Bayesian Coin Flipping . . . 87
A Motivating Example . . . 87
Picturing the Coin Probabilities . . . 88
Expressing Coin Flips as Probabilities . . . 90
Bayes' Rule . . . 94
Discussion of Bayes' Rule . . . 95
Bayes' Rule and Confusion Matrices . . . 97
Repeating Bayes' Rule . . . 101
The Posterior-Prior Loop . . . 102
The Bayes Loop in Action . . . 103
Multiple Hypotheses . . . 109
Summary . . . 115

5
CURVES AND SURFACES **117**

The Nature of Functions 118
The Derivative 119
Maximums and Minimums 119
Tangent Lines 122
Finding Minimums and Maximums with Derivatives 125
The Gradient 126
Water, Gravity, and the Gradient 127
Finding Maximums and Minimums with Gradients 128
Saddle Points 130
Summary 131

6
INFORMATION THEORY **133**

Surprise and Context 134
Understanding Surprise 134
Unpacking Context 135
Measuring Information 136
Adaptive Codes 137
Speaking Morse 138
Customizing Morse Code 141
Entropy 143
Cross Entropy 145
Two Adaptive Codes 146
Using the Codes 148
Cross Entropy in Practice 150
Kullback–Leibler Divergence 151
Summary 152

PART II: BASIC MACHINE LEARNING **153**

7
CLASSIFICATION **155**

Two-Dimensional Binary Classification 156
2D Multiclass Classification 160
Multiclass Classification 161
One-Versus-Rest 161
One-Versus-One 163
Clustering 166
The Curse of Dimensionality 168
Dimensionality and Density 169
High-Dimensional Weirdness 175
Summary 179

8
TRAINING AND TESTING 181

Training 182
Testing the Performance 183
Test Data 186
Validation Data 187
Cross-Validation 190
k-Fold Cross-Validation 192
Summary 194

9
OVERFITTING AND UNDERFITTING 195

Finding a Good Fit 196
Overfitting 196
Underfitting 197
Detecting and Addressing Overfitting 197
Early Stopping 202
Regularization 203
Bias and Variance 204
Matching the Underlying Data 205
High Bias, Low Variance 207
Low Bias, High Variance 209
Comparing Curves 210
Fitting a Line with Bayes' Rule 212
Summary 219

10
DATA PREPARATION 221

Basic Data Cleaning 222
The Importance of Consistency 223
Types of Data 225
One-Hot Encoding 226
Normalizing and Standardizing 227
Normalization 228
Standardization 229
Remembering the Transformation 230
Types of Transformations 231
Slice Processing 232
Samplewise Processing 232
Featurewise Processing 233
Elementwise Processing 234
Inverse Transformations 234
Information Leakage in Cross-Validation 239
Shrinking the Dataset 242
Feature Selection 243
Dimensionality Reduction 243
Principal Component Analysis 244
PCA for Simple Images 250
PCA for Real Images 255
Summary 260

11
CLASSIFIERS **263**

Types of Classifiers . . . 264
k-Nearest Neighbors . . . 264
Decision Trees . . . 269
Introduction to Trees . . . 269
Using Decision Trees . . . 271
Overfitting Trees . . . 275
Splitting Nodes . . . 280
Support Vector Machines . . . 282
The Basic Algorithm . . . 282
The SVM Kernel Trick . . . 287
Naive Bayes . . . 290
Comparing Classifiers . . . 295
Summary . . . 296

12
ENSEMBLES **297**

Voting . . . 298
Ensembles of Decision Trees . . . 299
Bagging . . . 299
Random Forests . . . 301
Extra Trees . . . 302
Boosting . . . 302
Summary . . . 309

PART III: DEEP LEARNING BASICS **311**

13
NEURAL NETWORKS **313**

Real Neurons . . . 314
Artificial Neurons . . . 315
The Perceptron . . . 315
Modern Artificial Neurons . . . 317
Drawing the Neurons . . . 319
Feed-Forward Networks . . . 322
Neural Network Graphs . . . 323
Initializing the Weights . . . 325
Deep Networks . . . 326
Fully Connected Layers . . . 328
Tensors . . . 328
Preventing Network Collapse . . . 329
Activation Functions . . . 331
Straight-Line Functions . . . 331
Step Functions . . . 333
Piecewise Linear Functions . . . 336
Smooth Functions . . . 339

Activation Function Gallery . . . 344
Comparing Activation Functions . . . 344
Softmax . . . 345
Summary . . . 348

14
BACKPROPAGATION 351

A High-Level Overview of Training . . . 352
Punishing Error . . . 352
A Slow Way to Learn . . . 354
Gradient Descent . . . 355
Getting Started . . . 356
Backprop on a Tiny Neural Network . . . 358
Finding Deltas for the Output Neurons . . . 360
Using Deltas to Change Weights . . . 366
Other Neuron Deltas . . . 368
Backprop on a Larger Network . . . 372
The Learning Rate . . . 376
Building a Binary Classifier . . . 378
Picking a Learning Rate . . . 379
An Even Smaller Learning Rate . . . 383
Summary . . . 386

15
OPTIMIZERS 387

Error as a 2D Curve . . . 388
Adjusting the Learning Rate . . . 389
Constant-Sized Updates . . . 391
Changing the Learning Rate over Time . . . 396
Decay Schedules . . . 398
Updating Strategies . . . 400
Batch Gradient Descent . . . 401
Stochastic Gradient Descent . . . 403
Mini-Batch Gradient Descent . . . 405
Gradient Descent Variations . . . 407
Momentum . . . 408
Nesterov Momentum . . . 414
Adagrad . . . 417
Adadelta and RMSprop . . . 418
Adam . . . 420
Choosing an Optimizer . . . 421
Regularization . . . 422
Dropout . . . 422
Batchnorm . . . 424
Summary . . . 425

PART IV: BEYOND THE BASICS 427

16
CONVOLUTIONAL NEURAL NETWORKS 429

Introducing Convolution 430
Detecting Yellow 431
Weight Sharing 433
Larger Filters 434
Filters and Features 437
Padding 440
Multidimensional Convolution 443
Multiple Filters 444
Convolution Layers 446
1D Convolution 446
1×1 Convolutions 447
Changing Output Size 449
Pooling 449
Striding 453
Transposed Convolution 457
Hierarchies of Filters 461
Simplifying Assumptions 461
Finding Face Masks 462
Finding Eyes, Noses, and Mouths 465
Applying Our Filters 467
Summary 472

17
CONVNETS IN PRACTICE 473

Categorizing Handwritten Digits 473
VGG16 478
Visualizing Filters, Part 1 481
Visualizing Filters, Part 2 487
Adversaries 491
Summary 493

18
AUTOENCODERS 495

Introduction to Encoding 496
Lossless and Lossy Encoding 496
Blending Representations 498
The Simplest Autoencoder 500
A Better Autoencoder 505
Exploring the Autoencoder 508
A Closer Look at the Latent Variables 508
The Parameter Space 508
Blending Latent Variables 513
Predicting from Novel Input 515
Convolutional Autoencoders 516
Blending Latent Variables 517
Predicting from Novel Input 519

Denoising 519
Variational Autoencoders 521
Distribution of Latent Variables 522
Variational Autoencoder Structure 523
Exploring the VAE 530
Working with the MNIST Samples 530
Working with Two Latent Variables 533
Producing New Input 535
Summary 538

19
RECURRENT NEURAL NETWORKS 539

Working with Language 540
Common Natural Language Processing Tasks 540
Transforming Text into Numbers 541
Fine-Tuning and Downstream Networks 542
Fully Connected Prediction 542
Testing Our Network 543
Why Our Network Failed 546
Recurrent Neural Networks 548
Introducing State 548
Rolling Up Our Diagram 549
Recurrent Cells in Action 552
Training a Recurrent Neural Network 552
Long Short-Term Memory and Gated Recurrent Networks 553
Using Recurrent Neural Networks 554
Working with Sunspot Data 554
Generating Text 555
Different Architectures 557
Seq2Seq 561
Summary 564

20
ATTENTION AND TRANSFORMERS 565

Embedding 566
Embedding Words 569
ELMo 571
Attention 574
A Motivating Analogy 574
Self-Attention 576
Q/KV Attention 579
Multi-Head Attention 580
Layer Icons 581
Transformers 581
Skip Connections 582
Norm-Add 583
Positional Encoding 584
Assembling a Transformer 586
Transformers in Action 589

BERT and GPT-2 590
BERT 590
GPT-2 593
Generators Discussion 596
Data Poisoning 598
Summary 599

21
REINFORCEMENT LEARNING **601**

Basic Ideas 602
Learning a New Game 603
The Structure of Reinforcement Learning 605
Step 1: The Agent Selects an Action 605
Step 2: The Environment Responds 606
Step 3: The Agent Updates Itself 607
Back to the Big Picture 608
Understanding Rewards 608
Flippers 614
L-Learning 616
The Basics 616
The L-Learning Algorithm 619
Testing Our Algorithm 621
Handling Unpredictability 624
Q-Learning 626
Q-Values and Updates 627
Q-Learning Policy 630
Putting It All Together 632
The Elephant in the Room 633
Q-learning in Action 634
SARSA 638
The Algorithm 639
SARSA in Action 642
Comparing Q-Learning and SARSA 644
The Big Picture 646
Summary 648

22
GENERATIVE ADVERSARIAL NETWORKS **649**

Forging Money 650
Learning from Experience 652
Forging with Neural Networks 653
A Learning Round 655
Why Adversarial? 656
Implementing GANs 657
The Discriminator 657
The Generator 658
Training the GAN 658
GANs in Action 660
Building a Discriminator and Generator 662
Training Our Network 664
Testing Our Network 665

DCGANs 666
Challenges 669
Using Big Samples 670
Modal Collapse 671
Training with Generated Data 671
Summary 673

23
CREATIVE APPLICATIONS **675**

Deep Dreaming 675
Stimulating Filters 676
Running Deep Dreaming 678
Neural Style Transfer 680
Representing Style 680
Representing Content 683
Style and Content Together 683
Running Style Transfer 685
Generating More of This Book 688
Summary 690
Final Thoughts 690

REFERENCES **693**
IMAGE CREDITS **717**
INDEX **721**

ACKNOWLEDGMENTS

Authors like to say that nobody writes a book alone. We say that because it's true. It gives me great pleasure to thank my friends and colleagues who helped me make this book possible.

For their consistent and enthusiastic support of this project, and for helping me feel good about it all the way through, I am enormously grateful to Eric Braun, Steven Drucker, Eric Haines, and Morgan McGuire. Thanks also to Georgia, Jenn, and Michael Ambrose for always providing cheerful conversation after I'd spent too long at the computer.

Thanks to the reviewers of this book's first edition, whose generous and insightful comments greatly improved the presentation: Adam Finkelstein, Alex Colburn, Alyn Rockwood, Angelo Pesce, Barbara Mones, Brian Wyvill, Craig Kaplan, Doug Roble, Eric Braun, Greg Turk, Jeff Hultquist, Kristi Morton, Lesley Istead, Matt Pharr, Mike Tyka, Morgan McGuire, Paul Beardsley, Paul Strauss, Peter Shirley, Philipp Slusallek, Serban Porumbescu, Stefanus Du Toit, Steven Drucker, Wenhao Yu, and Zackory Erickson.

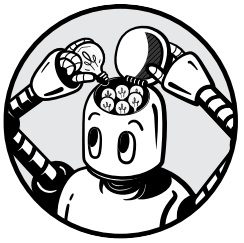
Special thanks to super reviewers Alexander Keller, Eric Haines, Jessica Hodgins, and Luis Alvarado, who read the whole first edition and offered wonderful feedback on both presentation and structure. Thank you to technical reviewers George Hosu and Ron Kneusel for their insights.

Thanks to Todd Szymanski for advice on the design and layout of the book's first edition and to Morgan McGuire for the Markdeep layout system, which enabled me to focus on writing and not word processing mechanics.

Thank you to the wonderful folks at No Starch Press for taking on this large project. Giant thanks to my editor Alex Freed, who read through the entire manuscript and offered numerous insightful comments and suggestions that greatly improved it throughout. Thank you to copyeditor Rebecca Rider and production editor Maureen Forys. Thank you to Rachel Monaghan for shepherding this project to completion with skill and grace. Thanks to publisher Bill Pollock for believing in the book and supporting the process.

My terrific colleagues at Weta Digital, Ltd. gave me professional and personal support and encouragement as I tackled this second edition. Thank you to Antoine Bouthors, Jedrzej Wojtowicz, Joe Letteri, Luca Fascione, Millie Maier, Navi Brouwer, Tom Buys, and Yann Provencher.

INTRODUCTION



Imagine that you're rubbing a golden lamp. You say, "Genie, for my three wishes, give me someone to love, great wealth, and a long and healthy life."

Now imagine that you're entering your home. You say, "House, bring the car around, ask Sarah if she's free for lunch, schedule a haircut, and make me a latte. Oh, and play some Thelonious Monk, please."

In both of these situations, you're asking a disembodied being of great power to hear you, understand you, and fulfill your desires. The first scenario is a fantasy going back millennia. The second scenario is commonplace reality today, thanks to artificial intelligence, or AI.

How did we invent these magic genies? The revolution in AI that's changing the world today is the result of three developments.

First, computers are getting bigger and faster every year, and special-purpose chips originally intended for generating images are now powering AI techniques as well.

Second, people keep developing new algorithms. Surprisingly, some of the algorithms powering AI today have been around for decades. They were more powerful than anyone knew and were just waiting for enough

data and compute power to let them shine. Newer and even more powerful algorithms are now pushing the field forward at an increasingly rapid pace. Some of the most powerful algorithms used today belong to a category of AI called *deep learning*, and those are the techniques we'll emphasize in this book.

Third, and perhaps most critically, there are now massive databases for these algorithms to learn from. Social networks, streaming services, governments, credit card agencies, and even supermarkets measure and retain every crumb of information they can gather. Public data on the web is also vast. As of late 2020, estimates suggest there are more than 4 billion hours of video freely available online, over 17 billion images, and an untold amount of text covering topics from sports history to weather patterns to municipal records.

In practice, these databases are often seen as the most valuable component of any new machine learning system. After all, anyone with money can buy computers, and the algorithms they run are almost all publicly available in research journals, books, and open source repositories. It's the data that organizations jealously hoard, to use for themselves, or to sell to the highest bidder. It's no stretch to say that databases are the new oil, the new gold.

Like any radical technology, AI has its rosy cheerleaders who foresee great advantages for the human race and dour predictors of doom who see nothing but ruin and the destruction of the societies and cultures that make up our world. Who is right? How can we judge the risks and rewards of this new technology? When should we embrace AI and when should we forbid it?

The best way to thoughtfully deal with a new technology is to understand it. When we know how it works, and the nature of its powers and limitations, we can determine where and how it should be used to create a future we want to inhabit. This book exists to help you understand what deep learning is and how it works. When you know the strengths and weaknesses of deep learning, you'll be in a better position to use it for your own work and understand its actual and potential impacts on our cultures and societies. It will also help you see when people and organizations that hold power are using these tools, and then determine for yourself whether they're using them for your advantage, or their own.

Who This Book Is For

I wrote this book for anyone who's interested in how deep learning works. You don't need math or programming experience. You don't need to be a computer whiz. You don't have to be a technologist at all!

This book is for anyone with curiosity and a desire to look behind the headlines. You may be surprised that most of the algorithms of deep learning aren't very complicated or hard to understand. They're usually simple and elegant and gain their power by being repeated millions of times over huge databases.

In addition to satisfying pure intellectual curiosity (which I think is a fine reason to learn anything), I wrote this book for people who come face to face with deep learning, either in their own work or when interacting with others who use it. After all, one of the best reasons to understand AI is so we can use it ourselves! We can build AI systems now that help us do our work better, enjoy our hobbies more deeply, and understand the world around us more fully.

If you want to know how this stuff works, you're going to feel right at home.

This Book Has No Complex Math and No Code

Everybody has their own way of learning. I wrote this book because I felt there were people who would like to understand deep learning but didn't want to get there by studying equations or programs.

So, rather than describe algorithms with equations or program listings, I use words and pictures. That doesn't mean the descriptions are sloppy or vague. Rather, I've worked hard to be as clear as possible, and, when appropriate, precise. When you're done with this book, you'll have a solid grasp of the general principles. If you later decide that you'd like to reframe your understanding in mathematical language, or write programs with a specific computer language or library, you'll find it much easier than starting from scratch.

There Is Code, If You Want It

Although programming skills aren't needed to read this book, translating the ideas into practice is important if you want to build and train your own systems. So, if you have that itch, I've provided the tools to scratch it.

I've provided three free bonus chapters on my GitHub at *https://github.com/blueberrymusic*. One chapter addresses using Python's free scikit-learn library, and the other two show how to build many of the deep learning systems we talk about in the book. With actual running code to build on and modify, you'll be able to use existing networks or design and train your own networks for your own applications. The code is presented in the popular form of Jupyter notebooks, so it's ready for your investigation, experimentation, adaptation, and use.

All of these programs are shared under the MIT license, which means you can use them in almost any way you like.

The Figures Are Available, Too!

Also on my GitHub at *https://github.com/blueberrymusic* you'll find almost every figure that I drew for this book, in high-quality 300 dpi PNG format.

The figures are provided under the same MIT license as the code, so you're free to use or modify them for classes, talks, presentations, papers, or any other use where you feel one of these figures would help you out.

Errata

No book of this size is going to be free of errors, from little typos to big mistakes. If you spot something that seems wrong, please let us know at *errata@nostarch.com*. We'll keep a list of the confirmed corrections at *https://nostarch.com/deep-learning-visual-approach*.

About This Book

The first part of this book covers foundational ideas from probability, statistics, and information theory that we'll need later on. Don't let these topic names intimidate you! We'll be sticking to the basic and essential ideas, with almost no mathematical notation. You may be surprised at how easy some of these topics are when they're presented in plain language and diagrams.

The second part of the book covers basic machine learning concepts, including basic ideas and some classic algorithms.

With that background in place, the third and fourth parts of the book are where it all comes together, and we get into deep learning itself.

Here's a brief overview of what you'll find in each chapter.

Part I: Foundational Ideas

Chapter 1: An Overview of Machine Learning. We'll look at the big picture and set the stage for how machine learning works.

Chapter 2: Essential Statistics. A core idea in deep learning is finding patterns in data. The language of statistics allows us to identify and discuss those patterns.

Chapter 3: Measuring Performance. When an algorithm answers a question, there is always some chance that answer is wrong. By carefully choosing how we measure, we can talk about what "wrong" really means.

Chapter 4: Bayes' Rule. We can discuss the likelihood that an algorithm is giving us correct results by considering both our expectations and what results we've seen so far. Bayes' Rule is a powerful way to do this.

Chapter 5: Curves and Surfaces. As a learning algorithm searches for patterns in data, it often makes use of abstracted curves and surfaces in imaginary spaces. To help us discuss those algorithms later in the book, here we discuss what those curves and surfaces look like.

Chapter 6: Information Theory. A powerful idea used in machine learning is that we are representing and modifying information. The ideas of information theory let us quantify and measure different kinds of information.

Part II: Basic Machine Learning

Chapter 7: Classification. We often want a computer to assign a specific class, or category, to a piece of data. For example, what animal is in a photo, or what word is being said into a phone? We look at the basic ideas for solving this problem.

Chapter 8: Training and Testing. To build a deep learning system for use in the world, we must first train it to learn how to do what we want and then test its performance to be sure it's doing the job well.

Chapter 9: Overfitting and Underfitting. A surprising result of training a deep learning system is that it can start memorizing the data we're using to train it. In an apparent paradox, this makes it worse at handling new data when the algorithm is released. We'll see where this problem comes from and how to reduce its impact.

Chapter 10: Data Preparation. We train a deep learning system by providing it with lots of data to learn from. We'll see how to prepare that data so training is as efficient as possible.

Chapter 11: Classifiers. We'll look at specific machine learning algorithms for classifying data. These are often a good way to get to know our data before investing the time and effort to train a deep learning system.

Chapter 12: Ensembles. We can assemble lots of very simple learning systems into a far more powerful composite system. Sometimes many small systems can return an answer faster, and more accurately, than a single big system.

Part III: Deep Learning Basics

Chapter 13: Neural Networks. We look at artificial neurons, and how to connect them together to form a network. These networks form the basis of deep learning.

Chapter 14: Backpropagation. The key algorithm that makes neural networks practical is a way of training them so that they learn from data. We look closely into the first of the two algorithms that make up the learning process.

Chapter 15: Optimizers. The second algorithm for training a deep network actually modifies the numbers that make up the network, improving its performance. We'll look at a variety of ways to do this effectively.

Part IV: Beyond the Basics

Chapter 16: Convolutional Neural Networks. Powerful algorithms have been developed to handle spatial data like images. We'll look at these algorithms and how they are used.

Chapter 17: Convnets in Practice. Having covered the techniques of handling spatial data, we'll look more closely at how we can use those techniques in practice to recognize objects.

Chapter 18: Autoencoders. We can simplify huge datasets so that they're smaller in size and easier to manage. We can also remove noise, enabling us to clean up damaged images.

Chapter 19: Recurrent Neural Networks. When we work with sequences, like text and audio clips, we need special tools. We'll see one popular class of them here.

Chapter 20: Attention and Transformers. Understanding text and language is particularly important. We'll look at algorithms originally designed to interpret and generate text, but which are proving to also be useful in other applications.

Chapter 21: Reinforcement Learning. Sometimes we don't know the answers we want the computer to provide, such as when scheduling real-world activities involving unpredictable groups of people. We'll see how to address these kinds of problem flexibly.

Chapter 22: Generative Adversarial Networks. We often want to create, or generate, new instances of the data we have, for example to create newspaper stories from raw data, or perhaps worlds for people to explore in games. We'll look at a powerful way to train such generators.

Chapter 23: Creative Applications. We wrap up with some fun, applying the tools of deep learning to make psychedelic images, apply an artist's signature style to a photograph, and generate new text in the style of any author.

Final Words

There's a lot of material in this book!

It's all there so that when you're done, you'll really know your stuff. You'll be able to talk to other people about deep learning, sharing your insights and experience, and learn from theirs. And if you're so motivated, you'll be able to grab one of the many free deep learning libraries and design, train, test, and deploy your own systems for any purpose you can dream up.

Deep learning is a fascinating field that combines ideas from many intellectual disciplines to build algorithms that are forcing us to ask fundamental questions about the nature of intelligence and understanding. It's also a whole lot of fun!

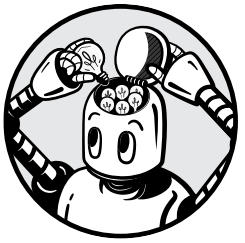
Welcome to the journey. You're going to have a great time!

PART I

FOUNDATIONAL IDEAS

1

AN OVERVIEW OF MACHINE LEARNING



This book is about *deep learning*, a subfield of *machine learning*. The phrase *machine learning* describes a growing body of techniques that share a common goal: extracting meaningful information from data. Here, *data* refers to anything that can be represented numerically. Data can be raw numbers (like stock prices on successive days, the masses of different planets, or the heights of people visiting a county fair), but it can also be sounds (the words someone speaks into their cell phone), pictures (photographs of flowers or cats), words (the text of a newspaper article or a novel), behavior (what activities someone enjoys), preferences (the music or films someone likes), or anything else that we can collect and describe with numbers.

Our goal is to discover meaningful information, where it's up to us to decide what's meaningful. We usually want to find patterns that help us understand the data or use past measurements to predict future events. For example, we might want to predict a movie someone would like based on movies they've already rated, read the handwriting on a note, or identify a song from just a few notes.

We generally find the information we're after in three steps: we identify the information that we want to find, we collect data that we hope will hold that information, and then we design and run algorithms to extract as much of that information as possible from that data.

In this chapter, we'll cover some of the major movements in machine learning. We'll begin by discussing an early approach to machine learning called an expert system. We'll then discuss three of the major approaches to learning: supervised learning, unsupervised learning, and reinforcement learning. We'll end the chapter by looking at deep learning.

Expert Systems

Before deep learning became practical on a widespread basis, a popular approach to learning from data involved creating *expert systems*. Still used today, these are computer programs intended to encapsulate the thought processes of human experts such as doctors, engineers, and even musicians. The idea is to study a human expert at work, watch what they do and how they do it, and perhaps ask them to describe their process out loud. We capture that thinking and behavior with a set of rules. The hope is that a computer could then do the expert's job just by following those rules.

These kinds of systems can work well once they're built, but they're difficult to create and maintain. It's worth taking a moment to see why. The problem is that the key step of producing the rules, called *feature engineering*, can require impractical amounts of human intervention and ingenuity. Part of deep learning's success is that it addresses exactly this problem by creating the rules algorithmically.

Let's illustrate the problem faced by expert systems with a practical example: recognizing digits. Let's say that we want to teach a computer to recognize the number 7. By talking to people and asking questions, we might come up with a set of three small rules that let us distinguish a 7 from all other digits: first, 7s have a mostly horizontal line near the top of the figure; second, they have a mostly northeast-southwest diagonal line; and third, those two lines meet in the upper right. The rules are illustrated in Figure 1-1.

This might work well enough until we get a 7 like Figure 1-2.

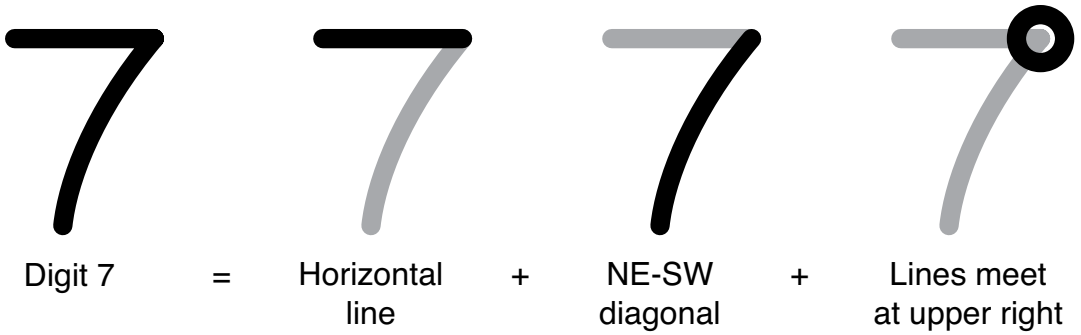


Figure 1-1: Top: A handwritten 7. Bottom: A set of rules for distinguishing a handwritten 7 from other digits.

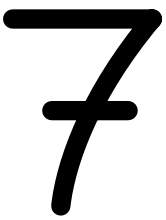


Figure 1-2: A 7 that would not be recognized by the rules of Figure 1-1 because of the extra horizontal line

Our set of rules won't recognize this as a 7, because we hadn't originally considered that some people put a bar through the middle of the diagonal stroke. So now we need to add another rule for that special case. In practice, this kind of thing happens over and over again to anyone developing an expert system. In a problem of any complexity, finding a good and complete set of rules is frequently an overwhelmingly difficult task. Turning human expertise into a series of explicit instructions often means laboriously uncovering inferences and decisions that people make without even realizing it, turning those into huge numbers of instructions, then adjusting and hand-tuning those instructions to cover all of the situations that were initially overlooked, debugging the rules where they contradict, and so on, in a seemingly never-ending series of tasks performed on a massive, complicated set of rules.

This process of finding the rules to accomplish a job is tough work: the rules human experts follow are often not explicit, and as we saw, it's easy to overlook exceptions and special cases. Imagine trying to find a comprehensive set of rules that can mimic a radiologist's thought process as they determine whether a smudge on an MRI image is benign or not, or the way an air-traffic controller handles heavily scheduled air traffic, or how someone drives a car safely in extreme weather conditions. To make things even more complex, the technology, laws, and social conventions around human activities are constantly changing, requiring us to constantly monitor, update, and repair this tangled web of interconnecting rules.

Rule-based expert systems can be made to work in some cases, but the difficulties of crafting the right set of rules, making sure they work properly across a wide variety of data, and keeping them up to date, makes them impractical as a general solution.

If we could only find and manage this set of rules, then computers could indeed emulate some forms of human decision making. This is just what deep learning is all about. These algorithms, given enough training data, can discover the decision-making rules *automatically*. We don't have to explicitly tell the algorithm how to recognize a 2 or a 7, because the system figures that out for itself. It can work out whether an MRI smudge is benign or not, whether a cellphone's photo has been ideally exposed, or whether a piece of text was really written by some historical figure. These are all among the many applications deep learning is already carrying out for us.

The computer discovers the decision-making rules by examining the input data and extracting patterns. The system never "understands" what it's doing, as a person does. It has no common sense, awareness, or comprehension. It just measures patterns in the training data and then uses those patterns to evaluate new data, producing a decision or result based on the examples it was trained on.

Generally speaking, we train deep learning algorithms in one of three different ways, depending on the data we have and what we want the computer to produce for us. Let's survey them briefly.

Supervised Learning

We'll first consider *supervised learning*. Here, the word *supervised* is a synonym for "labeled." In supervised learning, we typically give the computer pairs of values: an item drawn from a dataset, and a label that we've assigned to that item.

For example, we might be training a system called an *image classifier*, with the goal of having it tell us what object is most prominent in a photograph. To train this system, we'd give it a collection of images, and accompany each image with a label describing the most prominent object. So, for example, we might give the computer a picture of a tiger and a label consisting of the word *tiger*.

This idea can be extended to any kind of input. Suppose that we have a few cookbooks full of recipes that we've tried out, and we've kept records on how much we liked each dish. In this case, the recipe would be the input, and our rating of it would be that recipe's label. After training a program on all of our cookbooks, we could give our trained system a new recipe, and it could predict how much we'd enjoy eating the result. Generally speaking, the better we're able to train the system (usually by providing more pieces of training data), the better its prediction will be.

Regardless of the type of data, by giving the computer an enormous number of pairs of inputs and labels, a successful system designed for the task will gradually discover enough rules or patterns from the inputs that it will be able to correctly *predict* each provided label. That is, as a result of this training, the system has learned what to measure in each input so that

it can identify which of its learned labels it should return. When it gets the right answer frequently enough for our needs, we say that the system has been *trained*.

Keep in mind that the computer has no sense of what a recipe actually is, or how things taste. It's just using the data in the input to find the closest matching label, using the rules it learned during training.

Figure 1-3 shows the results of giving four photographs to a trained image classifier.

These photos were found on the web, and the system had never seen them before. In response to each image, the classifier tells us the likelihood for each of the 1,000 labels it was trained to recognize. Here we show the top five predictions for each photo, with their associated probabilities.

The picture in the upper left of Figure 1-3 is a bunch of bananas, so ideally we'd like to get back a label like bunch of bananas. But this particular classifier wasn't trained on any images labeled bunch of bananas. The algorithm can only return one of the labels it was trained on, in the same way that we can only identify objects by the words we know. The closest match it could find from the labels it was trained on was just banana, so that's the label it returned to us.

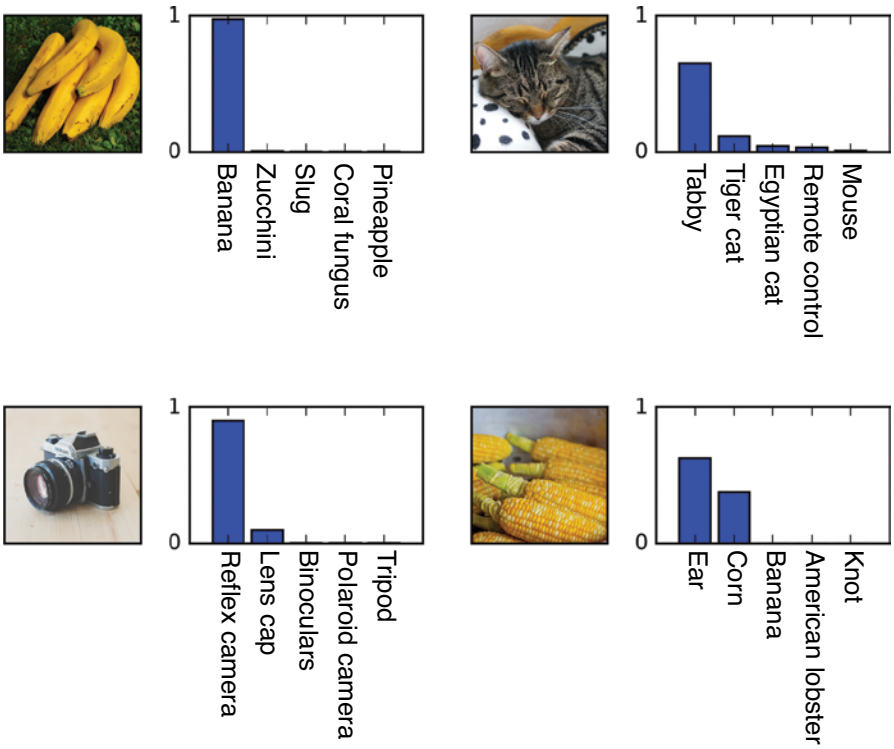


Figure 1-3: Four images and their predicted labels, with probabilities, from a deep learning classifier

In the upper left, the computer has very high confidence in the label of banana. In the lower right, the computer is about 60 percent confident that the proper label is ear. but it's about 40 percent confident it could be corn. If we follow a common practice and return just one label per image to the user, it would be helpful to also return the computer's confidence that the label is correct. If the confidence isn't reassuring, such as only about 60 percent for ear, we might decide to try again with a different algorithm, or perhaps even ask a human for help.

Unsupervised Learning

When we don't have labels associated with our data, we use techniques that are known collectively as *unsupervised learning*. These algorithms learn about relationships between the elements of the input, rather than between each input and a label.

Unsupervised learning is frequently used for *clustering*, or grouping, pieces of data that we think are related. For example, suppose that we're digging out the foundation for a new house, and while excavating, we find the ground is filled with old clay pots and vases. We call an archaeologist friend who realizes that we've found a jumbled collection of ancient pottery, apparently from many different places and perhaps even different times.

The archaeologist doesn't recognize any of the markings and decorations, so she can't say for sure where each one came from. Some of the marks look like variations on the same theme, but other marks look like different symbols. To get a handle on the problem, she takes rubbings of the markings and then tries to sort them into groups. But there are far too many for her to sort through, and since all of her graduate students are working on other projects, she turns to a machine learning algorithm to automatically group the markings together in a sensible way.

Figure 1-4 shows her captured marks and the groupings an algorithm might produce.

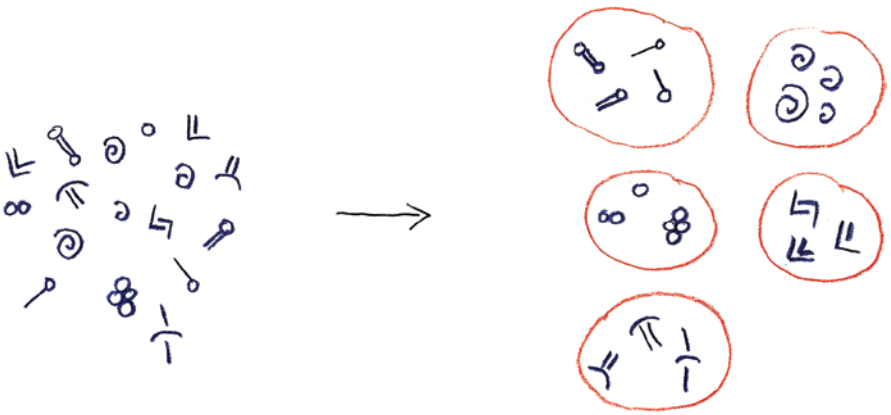


Figure 1-4: Using a clustering algorithm to organize marks on clay pots. Left: The markings from the pots. Right: The marks grouped into similar clusters.

Because this technique arranges our data into related groups (or *clusters*), we call the process *clustering*, and we refer to this algorithm as a *clustering algorithm*.

Unsupervised learning algorithms can also be used to improve the quality of measured data (for example, removing speckles in a photo taken with a cellphone camera) or compress datasets so they take up less room on our disks, without losing any qualities we care about (such as what MP3 and JPG encoders do for sound and images).

Reinforcement Learning

Sometimes we want to train a computer to learn how to perform a task, but we don't even know the best way to do it ourselves. Maybe we're playing a complex game or writing some music. What's the next best move to take or the next best note to choose? Often there isn't a single best answer. But we might be able to roughly say that one answer is better than another. It would be great to be able to train a computer to find the best steps toward a good result by letting it try out possible approaches, which we only need to rank in a very general manner, such as "probably good," or "better than the last one."

For example, suppose we're responsible for designing the operation of the elevators in a new office building, as in Figure 1-5. Our job is to decide where elevator cars should wait when they're not needed, and which car should answer a request when someone pushes a call button. Let's say that our goal is to minimize the average wait time for all riders.

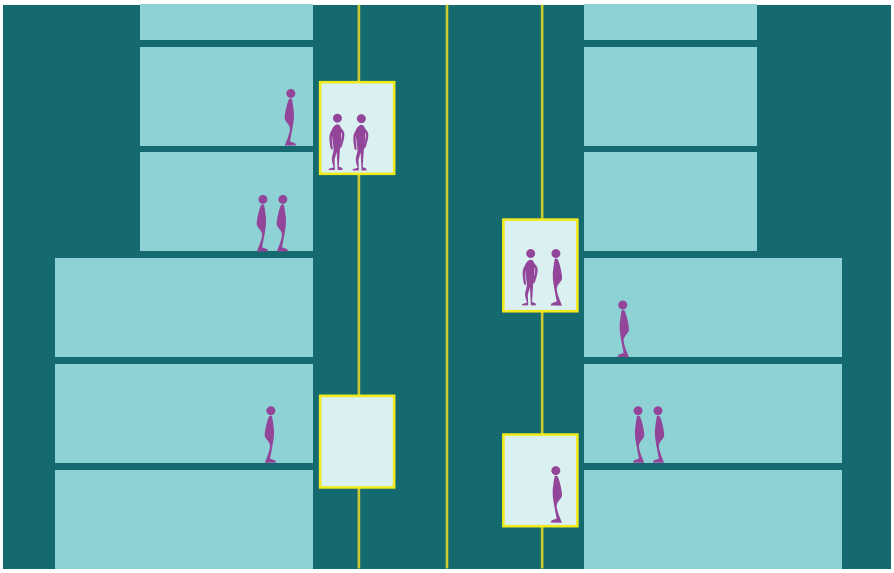


Figure 1-5: We want to find the best schedule for these elevators.

How should we do that? The quality of any solution we might imagine will depend entirely on the patterns of when and where people want to travel. Maybe in the morning everyone's arriving for work, so we should always bring empty cars to the first floor where they'll be ready for new arrivals. But perhaps at lunch everyone wants to go outside, so we should keep idle cars near the top floors, always ready to come down and take people to the ground floor. But if it's raining, perhaps most people will instead want to go to the cafeteria on the top floor. Day by day, hour by hour, what's the best policy?

There probably isn't any single best policy, so the computer can't learn to give it to us. All we can do is try out different approaches over time, and choose the one that seems to be giving us the best results. So we'll have the computer invent a policy, or maybe make a variation on an existing one, and see how well it performs. We'll then give it a score based on the average wait time of our riders. After trying out lots of variations, we can pick the policy with the best score. And then, over time, as patterns change, we can try out new approaches, always searching, but keeping the schedule with the best score.

This is an example of *reinforcement learning (RL)*. The technique of RL is at the heart of the recent game-playing algorithms that have been beating human masters at games like Go, and even online strategy games like *StarCraft*.

Deep Learning

The phrase *deep learning* refers to machine learning algorithms that use a series of steps, or *layers,* of computation (Bishop 2006; Goodfellow, Bengio, and Courville 2017). We can draw these layers on the page in any way we like as long the structure is clear. If we draw the layers vertically, we can imagine looking up from the bottom and saying that the system is tall, or, looking down from the top and calling it deep. If we draw many layers horizontally, we might say that the system is wide. For no particular reason, the "deep" language is what caught on, lending its name to the whole field of deep learning.

It's important to keep in mind that these systems are called "deep" only because of their appearance when we draw them stacked up vertically. They're not deep in the sense of having profound understanding or penetrating insights. When a deep learning system attaches a name to a face in a photo, it has no knowledge of what faces are, or what people are, or even that people exist. The computer just measures pixels and, using the patterns it learned from the training data, produces the most likely label.

Let's jump many chapters ahead and take a quick look at a deep network, pictured in Figure 1-6. In this simple network, we start with four input numbers, shown at the bottom of the figure. These might be the values of the four pixels in a 2 by 2 grayscale image, the closing price of a stock over four sequential days, or four samples from a snippet of voice data. Each input value is just a floating-point number, such as –2.982 or 3.1142.

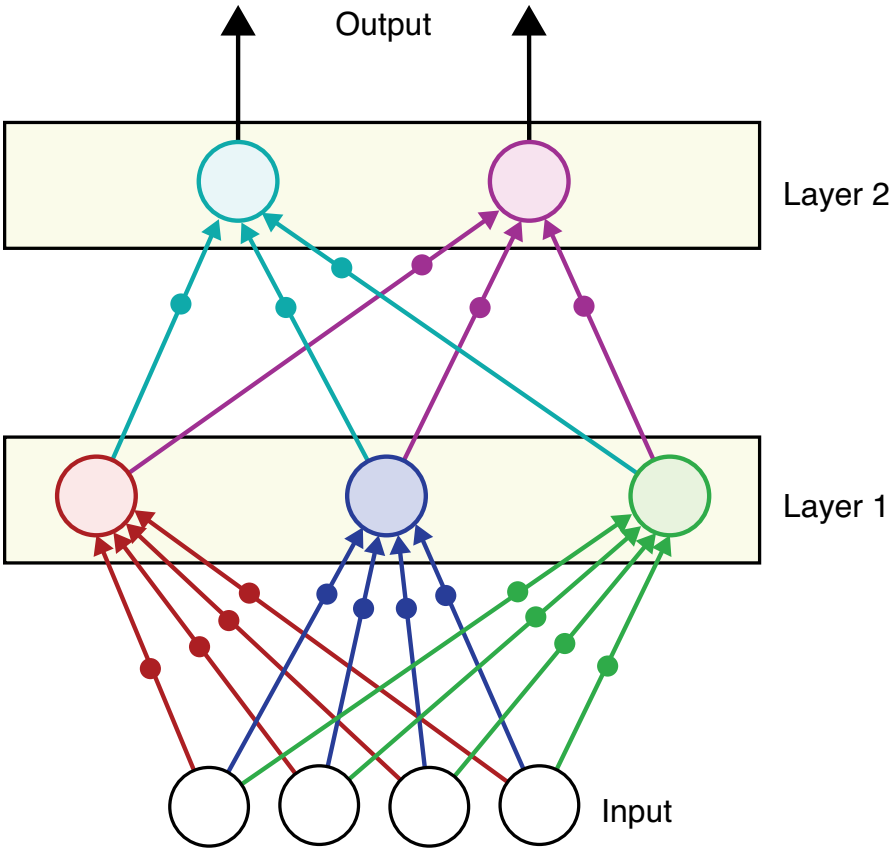


Figure 1-6: A simple deep neural network

Those four values go upward in the diagram into a *layer*, or grouping, of three *artificial neurons*. Although they have the word *neuron* in their name and were distantly inspired by real neurons, these artificial neurons are extremely simple. We'll see them in detail in Chapter 13, but it's best to think of them as units that perform a tiny calculation and are not remotely as complex as real neurons.

In this diagram, each neuron on layer 1 receives each of the four starting numbers as input. Note that each of the 12 lines that take an input value into a neuron has a little dot on it. In this figure, each dot represents the idea that the input value traveling on that line is multiplied by another number, called the *weight*, before it reaches the neuron. These weights are vital to the network, and we'll return to them in a moment.

The output of each artificial neuron in layer 1 is a new number. In Figure 1-6, each of those outputs is fed into each of the neurons on layer 2, and again each value is multiplied by a weight along the way. Finally, the values produced by the two neurons on layer 2 are the output of the network. We might interpret these output values to be the chance that the input is

in each of two classes, or the first and last name of the person who spoke that sound fragment, or the predicted price of the stock on each of the next two days.

Each of the big circles representing an artificial neuron turns its input values into a number. These computations are fixed: once we set up the network, each of these neurons will always compute the same output for a given input. So, once we've chosen the artificial neurons and arranged them into the network of Figure 1-6, almost everything is specified.

The only things in Figure 1-6 that can change are the inputs and the weights. That's the key insight that makes it possible to train the network. The weights start out as random numbers. That means the output of the network will initially be nonsense, and we'll never get back the results we want (unless we happen to occasionally get lucky).

To get the network to reliably produce the answers we want, we carefully change the weights, just a little at a time, after each mistaken output, to make the network more likely to produce the desired values at the output. If we do this carefully, then over time the output values will gradually get closer and closer to the results we want. Eventually, if we've done our job well, the network will produce the right answer in response to almost all the data in our training database, and we can release our network to the web, or offer it as a product or service.

In short, training this network, or teaching it, is nothing more than finding values for the weights so that every input produces the desired output. Amazingly, this is all there is to it! Even when our networks grow to hundreds of layers of many varieties, and tens of thousands of artificial neurons, and millions of weights, learning usually means just gradually changing the weights until we get the answers we want. More sophisticated networks might learn some other values as well, but the weights are always important.

One of the beautiful things about this process is that it makes good on the promise of feature engineering. For example, consider a system that takes a photo as input and tells us what breed of dog is in the picture. When the training process for this system has finished and the weights have settled into their best values, they have the effect of turning the neurons into little feature detectors. For example, one of the neurons on an early layer might produce a large value if it "sees" an eye, and another if it "sees" a floppy ear (we'll see just how this is done in Chapter 16). Then later neurons might look for combinations of these, such as a bushy tail along with short legs, or dark eyes with a long nose and large body, to help it determine the breed. In short, the neurons are looking for features, though we never explicitly guided them to. Feature detection is just a natural result of training the weights to produce the correct answers.

So although manually building an expert system that acts like a radiologist is a near-impossible task, creating a complex deep network and training it successfully can implement that agenda automatically. The system finds its own ways of combining the values of the pixels in each image into features and then using those features to make a determination about whether that image shows healthy tissue or not (Saba et al. 2019).

Summary

In this chapter, we got a general sense of deep learning. We began with expert systems, which required too much manual work to be successful in practice. We saw that training a deep learning system usually follows one of three approaches. Supervised learning means that we provide a label with every piece of data so that we can train the system to predict the correct label for new data. Unsupervised learning means that we give the system just the data, without labels, so we train the system to cluster the data into similar groups. And reinforcement learning means that we score various proposals put forth by the computer in the hope that it will eventually come up with an acceptably good solution.

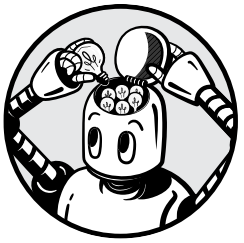
We then looked at a real, but tiny, deep learning system. The basic structure organized artificial neurons into layers. Neurons on each layer communicate with the neurons on the previous and following layer. It's the shape of this structure when we draw it in this form (like a tall tower of layers) that gives deep learning its name.

Finally, we saw the importance of the weights, or the values that multiply every number before it arrives at the input of an artificial neuron. When we teach a system, or we say it's learning, all we're usually doing is adjusting these weights. When the weights have found sufficiently good values, the system is able to do the job we've asked of it.

The next few chapters will dig into the background that we'll need to design and build deep learning systems.

2

ESSENTIAL STATISTICS



The better we understand our data, the better we can design a deep learning system that can make the most of that data.

By studying and analyzing the data we're starting with, we can pick the best algorithms for learning from it. The ideas and tools that let us do this analysis are generally bundled together under the heading of *statistics*. Statistical ideas and language appear everywhere in machine learning, from papers and source code comments to library documentation.

In this chapter, we'll cover the key statistical ideas essential to doing deep learning without delving into the math or details. The ideas fall roughly into two categories. The first category involves random numbers and how to describe them in ways that are of the most value in machine learning. The second category involves the ways in which we can choose objects (like numbers) from a collection and how to measure how well such selections represent the collection as a whole. Our goal here is to develop enough understanding and intuition to help us make good decisions when we do machine learning.

If you're already familiar with statistics and random values, at least skim the chapter. That way you'll know the language we use in this book, and you'll know where to come back to later if you want to brush up.

Describing Randomness

Random numbers play an important role in many machine learning algorithms. We use them to initialize our systems, to control steps during the learning process, and sometimes even to influence output. Picking random numbers properly is important: doing so can mean the difference between a system that learns from our data and produces useful results and a system that stubbornly refuses to learn anything. Rather than simply picking some arbitrary numbers out of thin air, we use a variety of tools to control what kinds of numbers we want to use and how we select them.

We typically select a random number between a given minimum and maximum, such as when someone has us "pick a number from 1 to 10." In this example, it's implied that our choice is limited to a finite number of options (the integers from 1 to 10). We'll frequently work with *real* numbers, which may lie between the integers. There are 10 integers from 1 to 10 (including the endpoints), but there is an infinite quantity of real numbers in that range.

When we talk about collections of numbers, random or otherwise, we often also talk about their *average*. This is a useful way to quickly characterize the collection of values. There are three different common ways to compute an average, and they come up frequently, so we'll identify them here. As a running example, let's work with a list of five numbers: 1, 3, 4, 4, 13.

The *mean* is the value usually meant in everyday language when we say *average*. It's the sum of all the entries divided by the number of entries in the list. In our example, adding together all the list elements gives us 1 + 3 + 4 + 4 + 13 = 25. There are five elements, so the mean is 25 / 5, or 5.

The *mode* is the value that occurs the most often in the list. In our example, 4 appears twice, and the other three values each appear only once, so 4 is the mode. If no value occurs more often than any other, we say the list has no mode.

Finally, the *median* is the number in the middle of the list when we write the values sorted from the smallest to the largest. In our list, which is already sorted, 1 and 3 make up the left side, 4 and 13 make up the right side, and another 4 is in the middle. So 4 is the median. If a list has an even number of entries, then the median is the mean of the two middle entries. For the list 1, 3, 4, 8, the median would be the average of 3 and 4, which is 3.5.

Averages are useful, but they don't tell us much about how the numbers in a collection are distributed. For example, they might be spread out equally across their range, or grouped into one or more clusters. We'll now look at the techniques that let us describe how numbers are distributed.

Random Variables and Probability Distributions

Before getting into details, let's build up our intuition with a running analogy. Suppose we're photographers who have been assigned to support an article on auto junkyards by taking lots of pictures of broken-down trucks and cars. Feeling adventurous, we go to a junkyard that contains many broken-down vehicles. We talk to the owner, and we agree that the best way to get great photos is for us to pay her to bring us vehicles to photograph, one by one. She makes it fun by using an old carnival wheel she has in her office. It has one equally sized slot for each car on the lot, as in Figure 2-1. Both the slots and the cars are numbered starting at 1.

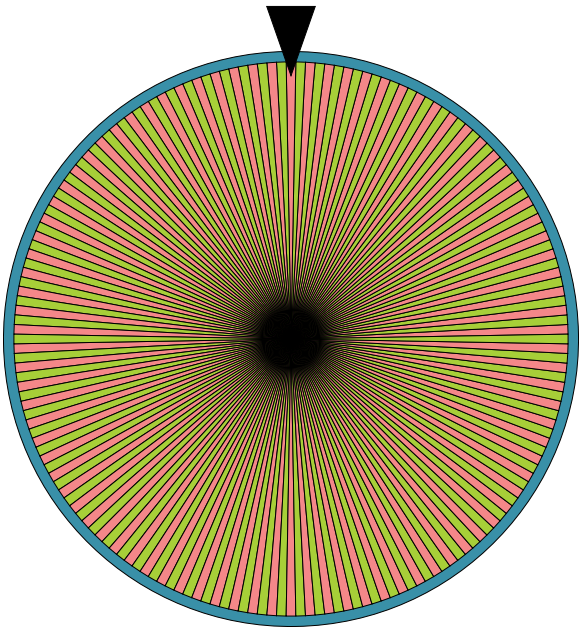


Figure 2-1: The junkyard owner's carnival wheel. Each equally sized sliver represents one car in her lot.

Once we pay her, she spins the wheel. When the wheel stops, she notes the number at the top, goes out with her tow truck, and drags the vehicle with the corresponding number back to us. We take some pictures, and she returns the vehicle to the lot. If we want to photograph another vehicle, we pay again, she spins the wheel again, and the process repeats.

Suppose that our assignment calls for us to get pictures of five different types of cars: a sedan, a pickup, a minivan, an SUV, and a wagon. For each type of car, we would like to know the chance that if she spins the wheel, we'll get that type. To work that out, let's suppose that we go into the lot to examine every vehicle, and we assign each one to one of these five categories. Our results are shown in Figure 2-2.

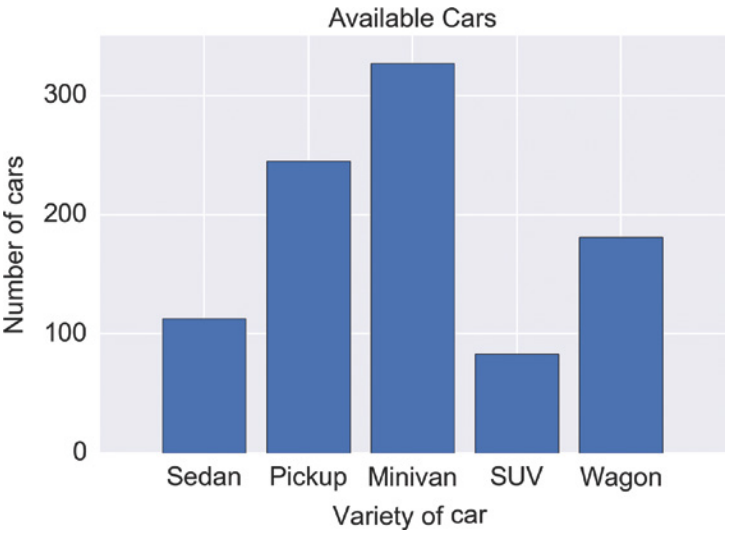


Figure 2-2: Our junkyard has five different types of cars in it. Each bar tells us how many cars we have of that type.

Of the almost 950 cars on her lot, the largest population is minivans, followed by pickups, wagons, sedans, and SUVs, in that order. Since every vehicle on her lot has an equal chance of being selected, on each spin of the wheel, we're most likely to get a minivan.

But specifically how *much* more likely are we to get a minivan? To determine how likely it is that we'll get each kind of vehicle, we can divide the height of each bar in Figure 2-2 by the total number of vehicles. This value gives us the probability that we'll get that given type of car, as shown in Figure 2-3.

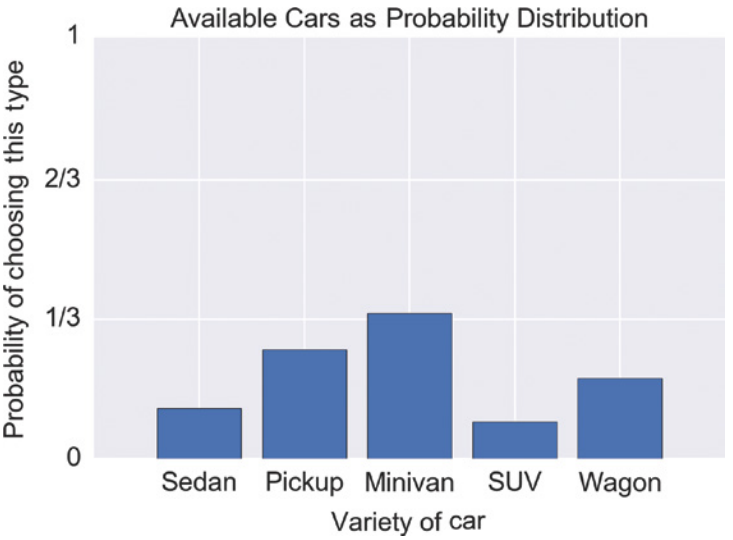


Figure 2-3: The probability of getting each type of car in the junkyard

To convert the numbers in Figure 2-3 into percentages, we multiply them by 100. For example, the height of the minivan bar is about 0.34, so we say that there's a 34 percent chance of getting a minivan. We say that the height of each bar is the *probability* that we'll get that kind of vehicle. If we add up the heights of all five bars, we'll find that the sum is 1.0. This illustrates the rules that turn any list of numbers into probabilities: the values must all be between 0 and 1, and add up to 1.

We call Figure 2-3 a *probability distribution* because it's distributing the 100 percent probability of getting some vehicle among the available options. We also sometimes say that Figure 2-3 is a *normalized* version of Figure 2-2, which means that the values all add up to 1.

We can use our probability distribution to draw a simplified carnival wheel, as in Figure 2-4.

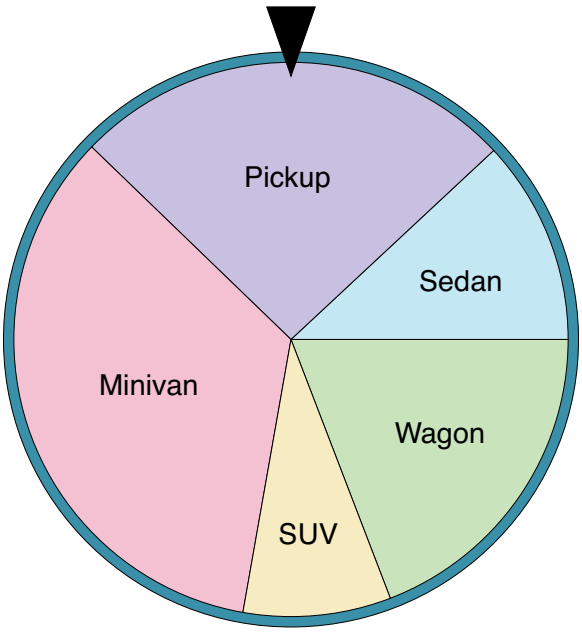


Figure 2-4: A simplified carnival wheel that tells us what kind of vehicle we'll get if the owner spins the big wheel of Figure 2-1

The chance that the wheel will end up with the pointer in a given region is given by the portion of the wheel's circumference taken up by that region, which we've drawn with the same proportions as those shown in Figure 2-3.

Most of the time we don't have carnival wheels around when we generate random numbers on the computer. Instead, we rely on software to simulate the process. For instance, we might give a library function a list of values, like the heights of the bars in Figure 2-3, and ask it to return a value. We expect that we'll get back `minivan` about 34 percent of the time, `pickup` about 26 percent of the time, and so on.

The job of picking a value at random from a list of options, each with its own probability, takes a bit of work. For convenience, we package up this selection process into its own conceptual procedure called a *random variable*.

This term can create confusion for programmers, because programmers think of a variable as being a named piece of storage that can hold data. In this context, rather than a piece of storage, a random variable is a *function* (Wikipedia 2017b), which takes a probability distribution as an input and produces a single value as an output. The process of selecting a value from a distribution is called *drawing* a value from the random variable.

We called Figure 2-3 a probability distribution, but we can also think of it as a function. We call the function, and it returns one of the types of vehicles, given these probabilities. This idea leads us to two more formal names for a distribution. When there are only a finite number of possible return values, like the five values in Figure 2-3, we sometimes use the oblique name *probability mass function* or *pmf* (this acronym is usually written in lowercase). A pmf is also sometimes called a *discrete probability distribution* (adding *function* at the end of this term is optional). These terms are meant to remind us that there are only a fixed number of possible outputs.

We can easily create probability distributions that are continuous. We use approximations of such functions when we initialize the values in a neural network. As an analogy, let's suppose that we want to know how much oil is left in each car that our junkyard dealer brings us. The amount of oil is a continuous variable, because it can take on any real number. Figure 2-5 shows a continuous graph for our oil measurements. This graph lets us find the probability of getting back a value within any given range, by adding up the area under the curve in that range. Suppose we want to find the probability of finding a car with, say, 0.45 units of oil. We don't just look up the value of the curve at 0.45. Instead, we imagine a little region around 0.45, such as from 0.44 to 0.46, and we add up the area under the curve in that region. This gives us the probability of getting back a value between 0.44 and 0.46. This means that the curve can take on values much greater than 1, as long as the curve is always positive, and the total area beneath the entire curve adds up to 1.

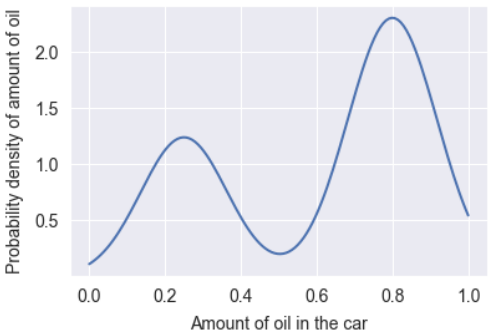


Figure 2-5: A distribution of probabilities for a continuous range of values

A distribution like Figure 2-5 is called a *continuous probability distribution* (or *cpd*), or a *probability density function* (or *pdf*). Sometimes people casually use the term probability density function, or pdf, to refer to discrete distributions, rather than continuous ones. The context usually makes it clear which interpretation is intended.

Recall that for the discrete case, all the possible return values need to add up to 1. In the continuous case, as in Figure 2-5, that means that the area under the curve is 1.

Most of the time, we obtain random numbers by selecting a distribution and then calling a library function to produce a value from that distribution (that is, it draws a random variable from our given distribution). We can make our own distributions when we want them, but most libraries provide a handful of distributions that have been found to cover most situations. That way we can just use one of these prebuilt distributions when we choose our random numbers. Let's take a look at some of these distributions.

Some Common Distributions

We've mentioned that we can draw a random variable from a distribution. Each time we draw a random variable, it takes on a number in accordance with the distribution: numbers with a large corresponding value in the distribution are more likely than those with a smaller value in the distribution. This makes distributions of great practical value, since different algorithms will want to use random variables that take on different values with particular probabilities. To achieve this, we merely need to pick an appropriate distribution.

Continuous Distributions

Most of the following distributions are offered as built-in routines by major libraries, so they're easy to specify and use. For simplicity, we'll demonstrate the following two distributions in their continuous forms. Most libraries offer us a choice between continuous and discrete versions, or they may offer a general-purpose routine to turn any continuous distribution into a discrete one on demand, or vice versa. We'll look at some discrete distributions later in the section.

The Uniform Distribution

Figure 2-6 shows the *uniform distribution*. The basic uniform distribution is 0 everywhere except between 0 and 1, where it has the value 1.

In Figure 2-6, it may appear that there are two values at 0 and two values at 1, but there aren't. Our convention is that an open circle (as on the lower line) means "this point is not part of the line," and a closed circle (as on the upper line) means "this point is part of the line." So, at the input values 0 and 1, our graph has an output of 1. This is a common way to define

this function, but some implementations make either or both of those outputs 0. It always pays to check.

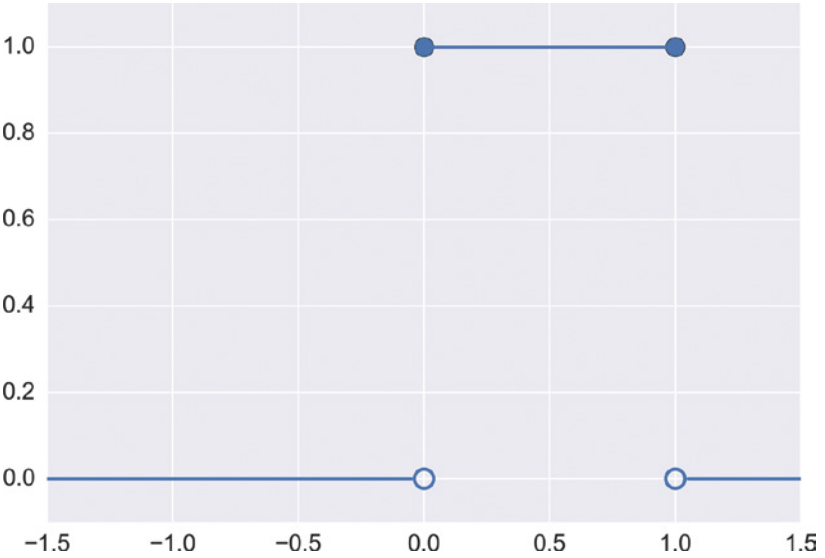


Figure 2-6: An example of a uniform distribution

This distribution has two key features. First, we can only get back values between 0 and 1, because the probability of all other values is 0. Second, every value in the range 0 to 1 is equally probable. It's just as likely we'd get 0.25 as 0.33 or 0.793718. We say that Figure 2-6 is *uniform*, or *constant*, or *flat*, in the range 0 to 1, all of which tell us that all the values in that range are equally probable. We also say that it's *finite*, meaning that all the nonzero values are within some specific range (that is, we can say with certainty that 0 and 1 are the smallest and largest values it can return).

Library functions that create uniform distributions for us often let us choose to start and end the nonzero region where we like. Probably the most popular choice, after the default of 0 to 1, is the range –1 to 1. The library takes care of details like adjusting the height of the function so that the area is always 1.0 (recall that this is required if a graph is to represent a probability distribution).

The Normal Distribution

Another frequently used distribution is the *normal distribution*, also called the *Gaussian distribution*, or simply the *bell curve*. Unlike the uniform distribution, it's smooth and has no sharp corners or abrupt jumps.

Figure 2-7 shows a few typical normal distributions.

All four curves in Figure 2-7 have the same basic shape. The shapes only vary because we scaled the curve, moved it horizontally, or both. For these illustrations, we didn't scale the curve so that the area under it sums to 1.

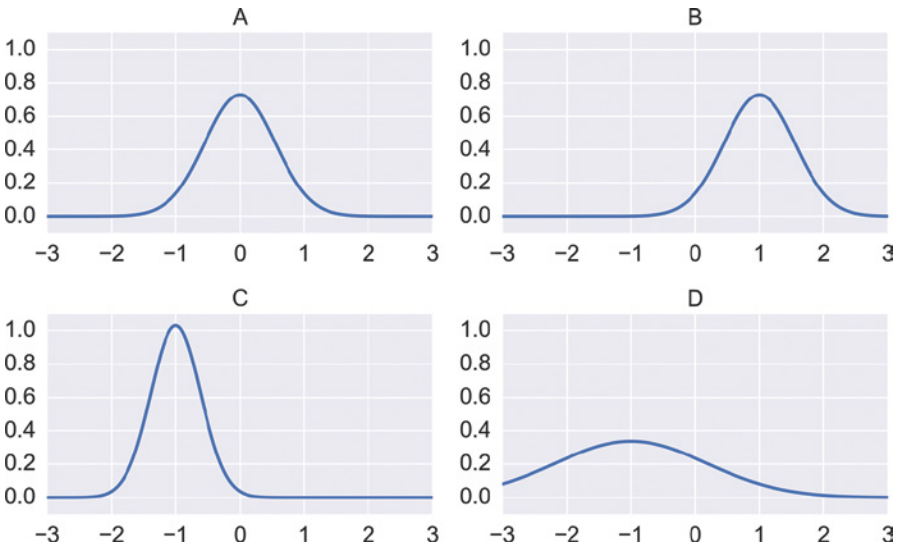


Figure 2-7: A few normal distributions

Figure 2-8 shows some representative samples that we'd get by drawing values from each distribution. They bunch up where the distribution's value is high (that is, getting a sample at that value has a high probability), and they are sparser where the distribution's value is low (where getting back a sample has a low probability). The vertical locations of the red dots representing the samples are jittered only to make the samples easier to see, and have no meaning.

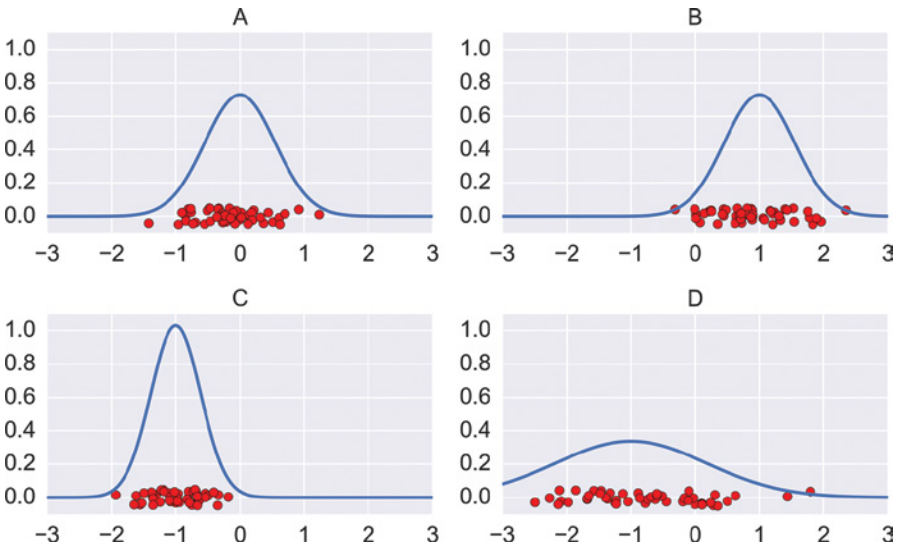


Figure 2-8: Each red circle shows the value of a sample resulting from drawing a value from its normal distribution.

The normal distribution is nearly 0 almost everywhere, except for the region where it rises up in a smooth bump. Though the values drop off ever closer to 0 to the sides of the bump, they never quite reach 0. So we say that the width of this distribution is *infinite*. In practice, we usually treat any value that's very nearly 0 as actually 0, giving us a finite distribution. A few other terms sometimes come up when people discuss normal distributions. The values produced by random variables from a normal distribution are sometimes called *normal deviates*, and are said to be *normally distributed*. We also say these values *fit*, or follow, a normal distribution.

Each normal distribution is defined by two numbers: the *mean* and the *standard deviation*. The mean tells us the location of the center of the bump. Figure 2-9 shows our four Gaussians from Figure 2-7, with their means. Here's one of the many nice properties of a normal distribution: its mean is also its median and its mode.

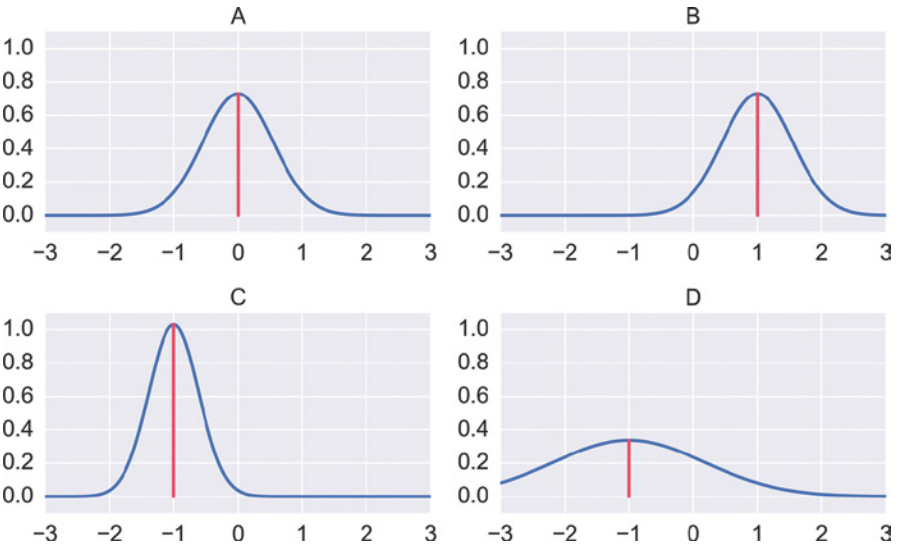


Figure 2-9: The mean of a normal distribution is the center of the bump, here shown with a red line.

The *standard deviation* is also a number, often represented by the lowercase Greek letter σ (sigma), which tells us the width of the bump. Imagine starting at the center of the bump and moving symmetrically outward until we're enclosing about 68 percent of the total area under the curve. The distance from the center of the bump to either of these ends is called *one standard deviation* for that curve. Figure 2-10 shows our four Gaussians, with the area inside one standard deviation shaded in green.

We can use the standard deviation to characterize a bump: when the standard deviation is small, it means the bump is narrow. As the standard deviation increases, the bump becomes more spread out horizontally.

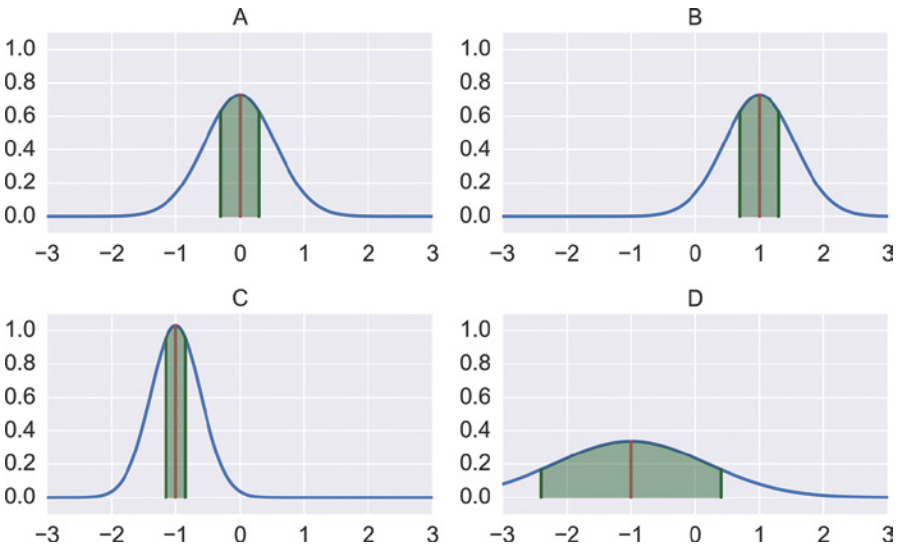


Figure 2-10: Some normal distributions with the area within one standard deviation shaded in green

If we go out symmetrically from the center by another standard deviation (that is, the same distance again), then we've enclosed about 95 percent of the area under the curve, as shown in Figure 2-11. And if we go out one more standard deviation, we've enclosed about 99.7 percent of the area under the curve, also shown in Figure 2-11. This property is sometimes called the *three-sigma rule*, because of the use of σ for the standard deviation. It also sometimes goes by the catchy name of the *68-95-99.7 rule*.

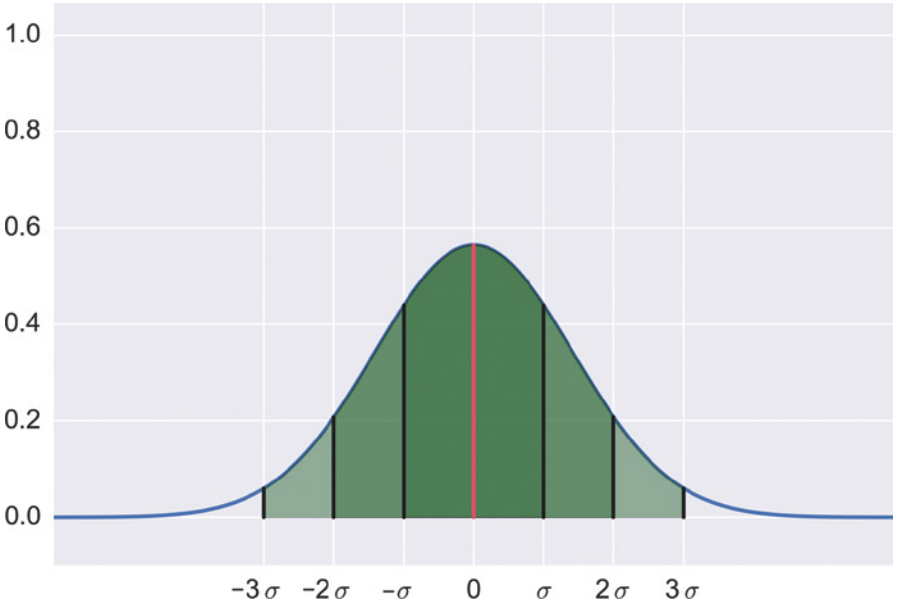


Figure 2-11: The three-sigma, or 68-95-99.7 rule

Suppose we look at 1,000 samples drawn from any normal distribution. We would find that about 680 of them are no more than one standard deviation from that distribution's mean, or in the range $-\sigma$ to σ; about 950 of them are within two standard deviations, or in the range -2σ to 2σ; and about 997 of them are within three standard deviations, or in the range -3σ to 3σ.

To summarize, the mean tells us where the center of the curve is, and the standard deviation tells us how spread out the curve is. The larger the standard deviation, the broader the curve, because that 68 percent cutoff is farther away.

Sometimes instead of the standard deviation, people use a different but related value called the *variance*. The variance is just the standard deviation multiplied by itself (that is, the standard deviation squared). This value is sometimes more convenient in calculations. The general interpretation is the same, though: curves with big variances are more spread out than curves with small variances.

The normal distribution appears frequently in machine learning and in other fields, because it naturally describes many real-world observations. If we measure the height of adult male horses in some region, or the size of sunflowers, or the lifespans of fruit flies, we'll find that these all tend to take on the shape of a normal distribution.

Discrete Distributions

Now let's look at two discrete distributions.

The Bernoulli Distribution

A useful, but special, discrete distribution is the *Bernoulli distribution* (pronounced ber-noo′-lee). This distribution returns just two possible values: 0 and 1. A common example of a Bernoulli distribution is the probability of getting heads and tails from flipping a coin. We usually use the letter p to describe the probability of getting back a 1 (let's say that means heads). If we ignore weird cases like when the coin lands sideways, the two probabilities of heads and tails must add up to 1, which means that the probability of getting back a 0 (or tails) is $1 - p$. Figure 2-12 shows this graphically for a fair coin and a weighted coin. Because we have just two values, we can draw the Bernoulli distribution as a bar chart, rather than the lines and curves we saw for the continuous distributions.

It may seem like overkill to use the language of distributions for such a simple situation, but the payoff is that we can use it with our library routines that produce values from distributions. We can hand our routine a uniform distribution, or a Gaussian, or a Bernoulli, and it will return a value drawn from that distribution according to its probabilities. This makes programming easier.

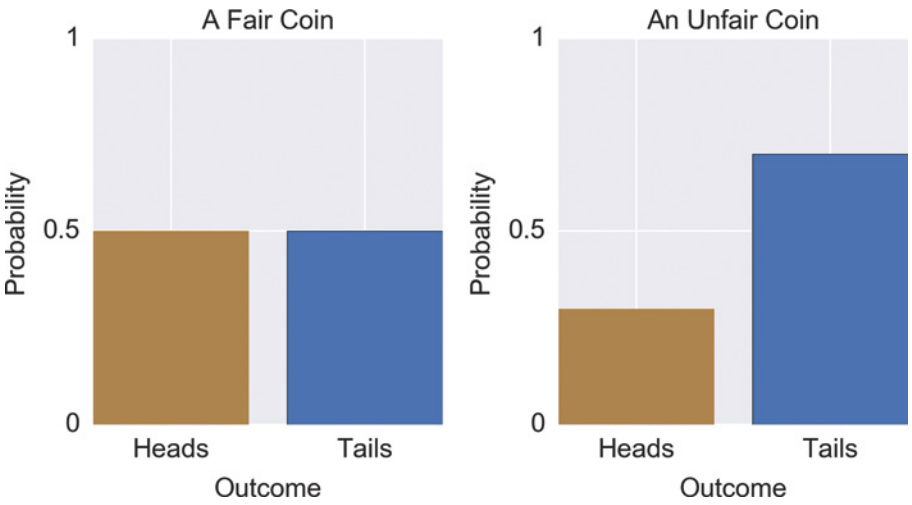


Figure 2-12: The Bernoulli distribution tells us the chance of drawing either a 0 or 1. Left: A fair coin. Right: An unfair coin.

The Multinoulli Distribution

The Bernoulli distribution only returns one of two possible values. But suppose we are running an experiment that can return any one of a larger number or possibilities. For instance, instead of flipping a coin that can come up either heads or tails, we might roll a 20-sided die that can come up with any of 20 values.

To simulate the result of rolling this die, our random variable could return an integer from 1 to 20. But in other situations, it's useful to have a list of possible values where all the entries have a corresponding probability of 0 except for the one entry we drew, which is set to 1. Such a list is useful when we build machine learning systems to classify inputs into different categories—for example, to describe which of 10 different animals appears in a photograph.

Let's suppose that we have a photo of an alligator, and that's entry five in our list. If our algorithm wasn't sure what the image was, we might get back something like the left of Figure 2-13, where three animals are identified as possibilities. We'd want the system to produce the output on the right, where every entry is 0 except for the alligator, which is 1.

This way of representing a single choice from a list is a key step in training classifiers with more than two possible classes. We'll return to this idea in Chapter 6, as a component of an idea called *cross entropy*.

Because each distribution in Figure 2-13 is a generalization of the two-outcome Bernoulli distribution into multiple outcomes, we could call it a "multiple-Bernoulli distribution," but instead we mush the words together into a portmanteau and call it a *multinoulli distribution* (or sometimes the less colorful *categorical distribution*).

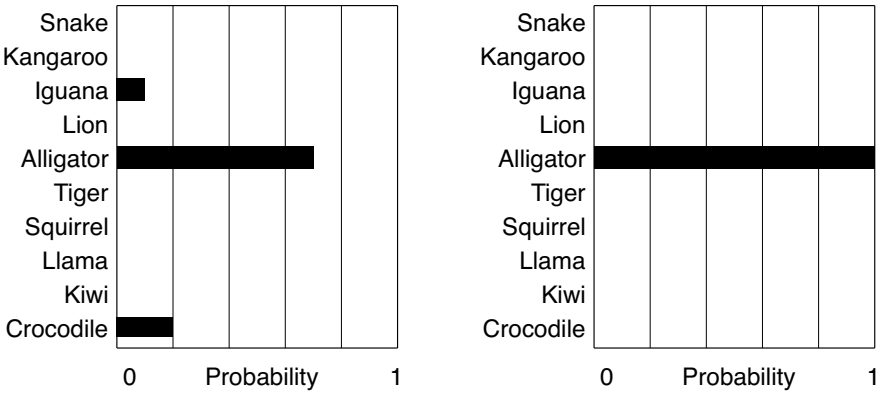


Figure 2-13: Left: Possible predicted probabilities for a picture of an alligator. Right: The probabilities we want.

Collections of Random Values

We've seen how to generate random values from a distribution. We draw a value from a random variable using the probabilities of that distribution to tell us which return values are more likely to be selected than others.

When we have lots of values drawn from one or more random variables, it's useful to characterize the collection so that we can speak of it as a group. Let's look at three such ideas that show up frequently in machine learning.

Expected Value

If we pick a value from any probability distribution, and then we pick another, and another, over time we build up a long list of values.

If these values are numbers, their mean is called the *expected value*. This is useful information for many applications. For a simple example, we might have a need for random numbers between –1 and 1, with a roughly equal number of positive and negative values. If the expected value of the random variable is 0, then we know we're getting a balanced set of values.

Note that the expected value might not be one of the values ever drawn from the distribution! For example, if the values 1, 3, 5, and 7 are the only ones available, and they are all equally likely, then the expected value of the random variable that we use to draw a value from this list would be $(1 + 3 + 5 + 7) / 4 = 4$, a value that we'd never get back from the distribution.

Dependence

The random variables we've seen so far have been completely disconnected from one another. When we draw a value from a distribution, it doesn't matter if we've drawn other values before. Each time we draw a new random variable, it's a whole new world.

We call these *independent* variables, because they don't depend on each other in any way. These are the easiest kind of random variable to work with, because we don't have to worry about managing how two or more random variables might influence one another.

By contrast, there are *dependent* variables, which do depend on each other. For example, suppose we have several distributions for the fur length of different animals: dogs, cats, hamsters, and so on. We might first pick an animal at random from a list of animals, and then use that to select the appropriate fur length distribution. We'd then draw a value from that distribution to find a value for the animal's fur. The choice of animal depends on nothing else, so it's an independent variable. But the length of the fur depends on the distribution we use, which in turn depends on which animal we chose, so fur length is a dependent variable.

Independent and Identically Distributed Variables

The math and algorithms of many machine-learning techniques are designed to work with multiple values that are drawn from random variables with the same distribution and are also independent of each other. That is, we draw values from the same distribution over and over, and there is no relationship between successive values. In fact, some algorithms require that we generate our random values this way, while others work best when we do.

This requirement is common enough that such variables have a special name: *i.i.d.*, which stands for *independent and identically distributed* (the acronym is unusual because it's usually written in lowercase, with periods between the letters). We might see, for example, the arguments for a library function described this way: "Make sure successive inputs are i.i.d."

The phrase *identically distributed* is just a compact way of saying "selected from the same distribution."

Sampling and Replacement

In machine learning it's often useful to build new datasets from existing ones by randomly selecting some of the elements of the existing set. We'll do just this in the next section, when we look for the mean value of a set of samples. Let's look at two ways of creating a list of selections, chosen from a starting pool of objects.

Selection with Replacement

Let's first look at an approach where we make a copy of each selected item, so the original stays in place, as in Figure 2-14. We call this approach *selection with replacement*, or *SWR*, because we can think of it as removing the object, making a copy for ourselves, and replacing the original.

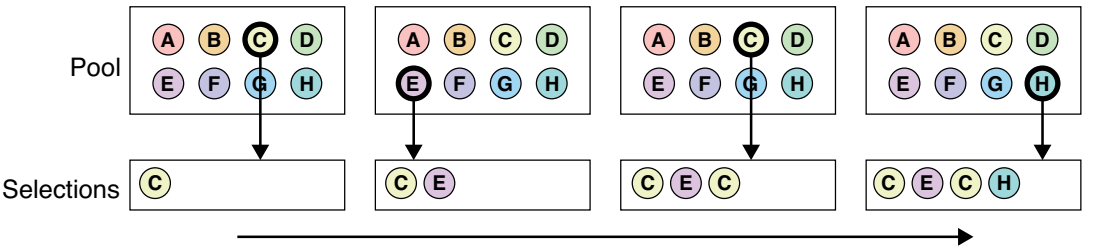


Figure 2-14: Selection with replacement

One implication of selection with replacement is that we might end up with the same object more than once. In an extreme case, our entire new dataset might be nothing more than multiple copies of the same object. A second implication is that we can make a new dataset that is smaller than the original, or the same size, or even much bigger. Since the original dataset is never altered, we can continue picking elements as long as we like.

A statistical implication of this process is that our selections are *independent* of one another. There is no history, so our selections are not at all affected by previous choices, nor do they influence future choices. To see this, note that the pool (or starting dataset) in Figure 2-14 always has eight objects, so the odds of picking each one are 1 in 8. In the figure, we first picked element C. Now our new dataset has element C inside of it, but we've replaced that element back into the pool after selecting it. When we look again at the pool, all eight items are still there, and if we choose again, each still has a 1 in 8 chance of being picked.

An everyday approximation of sampling with replacement is ordering at a well-stocked coffee shop. If we order a vanilla latte, it's not removed from the menu but remains available to the next customer.

Selection Without Replacement

The other way to randomly choose our new dataset is to remove our choice from the original dataset and place it in our new one. We don't make a copy, so the original dataset has just lost one element. This approach is called *selection without replacement*, or *SWOR*, and is shown in Figure 2-15.

An everyday example of sampling without replacement is playing a card game like poker. Each time a card is dealt, it's gone from the pack and cannot be dealt again (until the cards are recollected and shuffled).

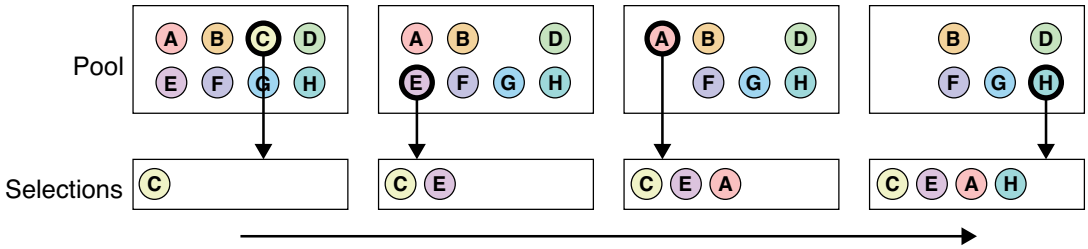


Figure 2-15: Selection without replacement

Let's compare the implications of SWOR with those of SWR. First, in SWOR, no object can be selected more than once, because we remove it from the original dataset. Second, our new dataset can be smaller than the original, or the same size, but it can never be larger. Third, our choices are now dependent. In Figure 2-15, each element originally had the same 1 in 8 chance of being picked the first time. When we selected item C, we did not replace it. When we go to make another selection, there are only seven elements available to us, each now with a 1 in 7 chance of being selected. The odds of selecting any one of those elements have gone up, simply because there are fewer elements competing for selection. If we select another item, the remaining elements each have a 1 in 6 chance of being picked, and so on. And after we've selected seven items, the last one is 100 percent sure to be selected the next time.

Continuing the comparison, suppose that we want to make a new dataset that's smaller than the original pool. We could build it with or without replacement. But sampling with replacement can generate many more possible new collections than sampling without. To see this, suppose we had just three objects in our original pool (let's call them A, B, and C), and we want a new collection of two objects. Sampling without replacement gives us only three possible new collections: (A,B), (A,C), and (B,C). Sampling with replacement gives us those three, and also (A,A), (B,B), and (C,C). Generally speaking, sampling with replacement always gives us a larger set of possible new collections.

Bootstrapping

Let's look at a useful application for the SWR and SWOR algorithms we just covered.

Sometimes we want to know some statistics about a dataset that's much too large for us to work with in practice. For example, suppose we want to know the mean height of all people alive in the world right now. There's just no practical way to measure everyone. Usually we try to answer this kind of question by extracting a representative piece of the dataset, and then measuring that. We might find the height of a few thousand people, and hope that the mean of those measurements is close to what we'd get if we were able to measure everyone.

Let's call every person in the world our *population*. Since that's too many people to work with, we'll gather a reasonably sized group of people that we hope are representative of the population. We call that smaller group a *sample set*. We'll build this sample set without replacement, so each time we select a value from the population (that is, a person's height), it's removed from the population, placed into the sample set, and cannot be chosen again.

We hope that by building our sample set carefully, we are making it a reasonable proxy for the whole population with respect to the properties we want to measure. Figure 2-16 shows the idea for a population of 21 circled numbers. The sample set contains 12 elements from the population.

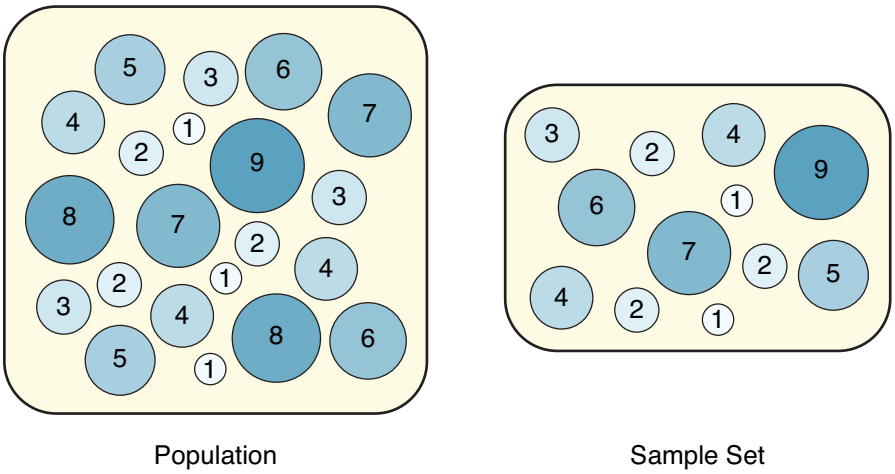


Figure 2-16: Creating a sample set from a population by sampling without replacement

Now we'll measure the mean of the sample set, and use that as our estimate of the mean of the population. In this little example, we can compute the mean of the population, which comes out to about 4.3. The mean of our sample set is about 3.8. This match isn't great, but it's not wildly wrong.

Most of the time we won't be able to measure the population (that's why we're building the sample set in the first place). By finding the mean of the sample set, we've come up with an approximation, but how good is it? Is this a number we ought to rely on as a good estimate for the whole population? It's hard to say. Things would be better if we could express our result in terms of a *confidence interval*. This lets us make a statement of the form, "We are 98 percent certain that the mean of the population is between 3.1 and 4.5." To make such a statement, we need to know the upper and lower bounds of the range (here, 3.1 and 4.5) and have a measure of how confident we are that the value is in that range (here, 98 percent). Typically, we pick the confidence we need for whatever task we have at hand, and from that, we find the lower and upper values of the corresponding range.

We'd like to be able to make this kind of statement about the mean, or any other statistical measure we're interested in. We can do this with the technique of *bootstrapping* (Efron and Tibshirani 1993; Teknomo 2015), which involves two basic steps. The first is the step we saw in Figure 2-16,

where we create a sample set from the original population using SWOR. The second step involves resampling that sample set to make new sets, this time using SWR. Each of these new sets is called a *bootstrap*. The bootstraps are the key to coming up with our confidence statement.

To create a bootstrap, we first decide how many elements we want to pick out of the starting sample set. We can pick any number up to the number of elements in the set, though we often use far fewer. Once we've picked that number, we randomly extract that many elements from the sample set with replacement, so it's possible that we'll pick the same element more than once. The process is illustrated in Figure 2-17.

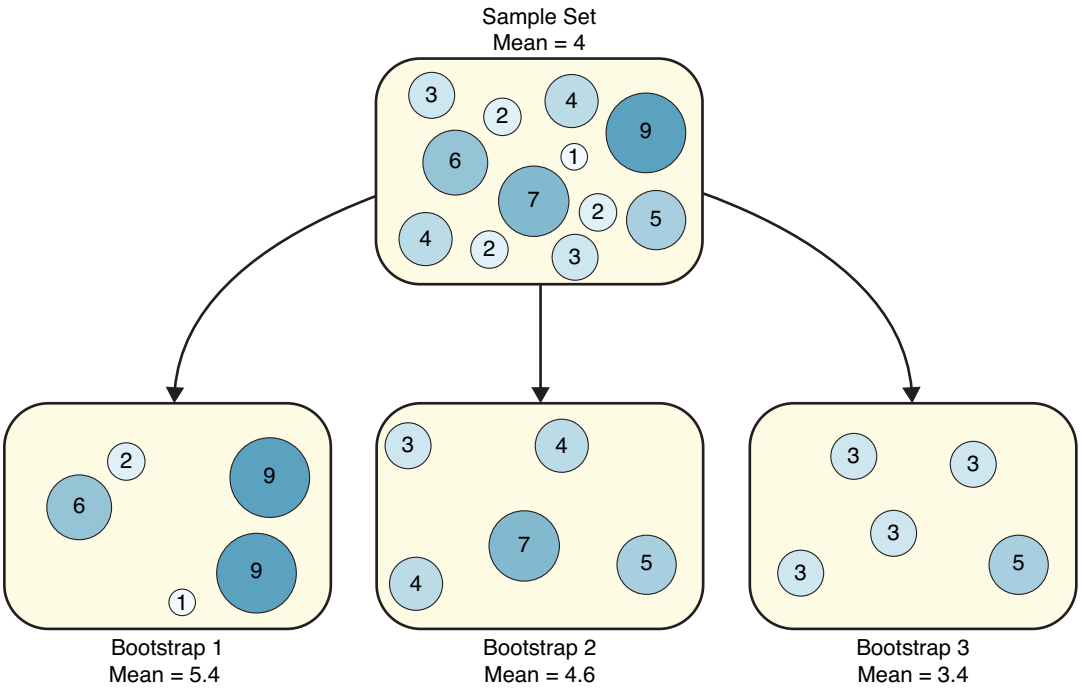


Figure 2-17: Creating three bootstraps using SWR and finding their means

To recap, we start with a population. We make a sample set from the population using sampling without replacement. Then we make bootstraps from that sample set using sampling *with* replacement. We need to select with replacement in this last step because we might want to build bootstraps that have the same size as the sample set. In our example, we might want our bootstraps to hold 12 values. If we didn't sample with replacement, then every bootstrap would be identical to the sample set.

If we really hope to find the average height of everyone in the world, we need a lot more than 21 measurements. Let's scale up the number of samples and zoom way down on their range. For convenience, let's focus on the size of two-month-old babies. They are typically about 50 centimeters long, so we created a simulated population of 5,000 measurements with lengths from 0 to 1,000 millimeters (that's 1 meter, or about 3.2 feet). From this

population, we drew 500 measurements at random to make a sample set, and then we created 1,000 bootstraps, each with 20 elements. Figure 2-18 shows the number of bootstraps that had a mean value in each of about 100 different bins across the range from 0 to 1,000 (there were almost no means below 200 or above 800). Graphs like this take the form of an approximate bell curve almost every time, because the nature of the bootstraps causes more of them to have a mean around the true mean, and fewer with mean values farther away.

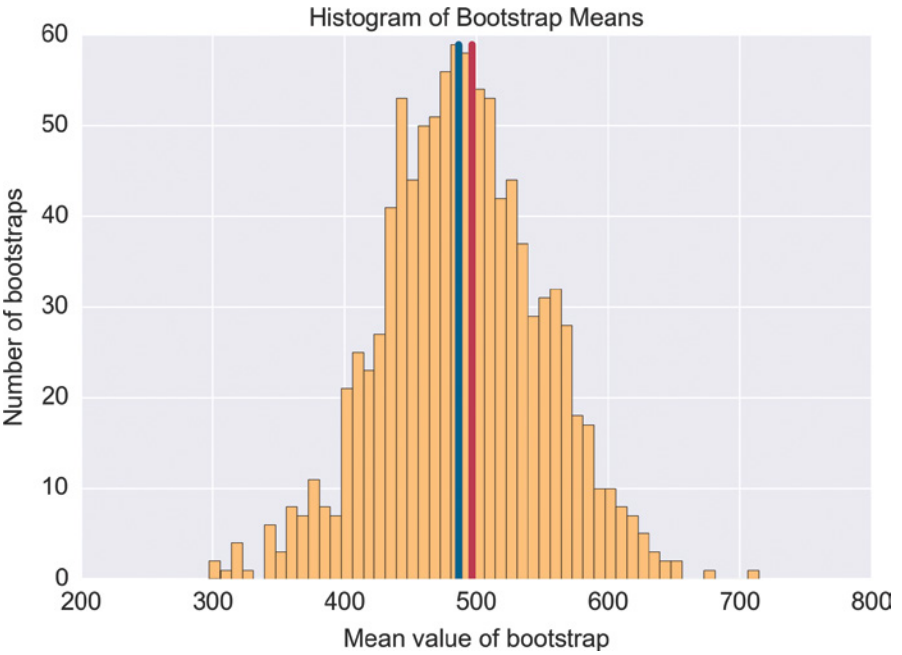


Figure 2-18: The histogram shows how many bootstraps have a mean of the given value. The blue bar at about 490 is the mean of the sample set. The red bar at about 500 is the mean of the population.

Since we created the data, we know the mean of the population is 500. The mean of our sample set is close to this, at about 490. The purpose of bootstrapping is to help us determine how much we should trust this value of 490. Without going into the math, the approximate bell curve of the mean values of the bootstraps tells us everything we need to know. Let's say we want to find the values that we're 80 percent confident brackets the mean of the population. Then we only need to slice off the lowest and highest 10 percent of the bootstrap values, leaving the middle 80 percent (Brownlee 2017). Figure 2-19 shows a box that does just this, enclosing the values that we are 80 percent confident contain the real value, which we know is 500. Reading from the graph, we could now say, "We are 80 percent confident that the mean of the original population is between about 410 and 560."

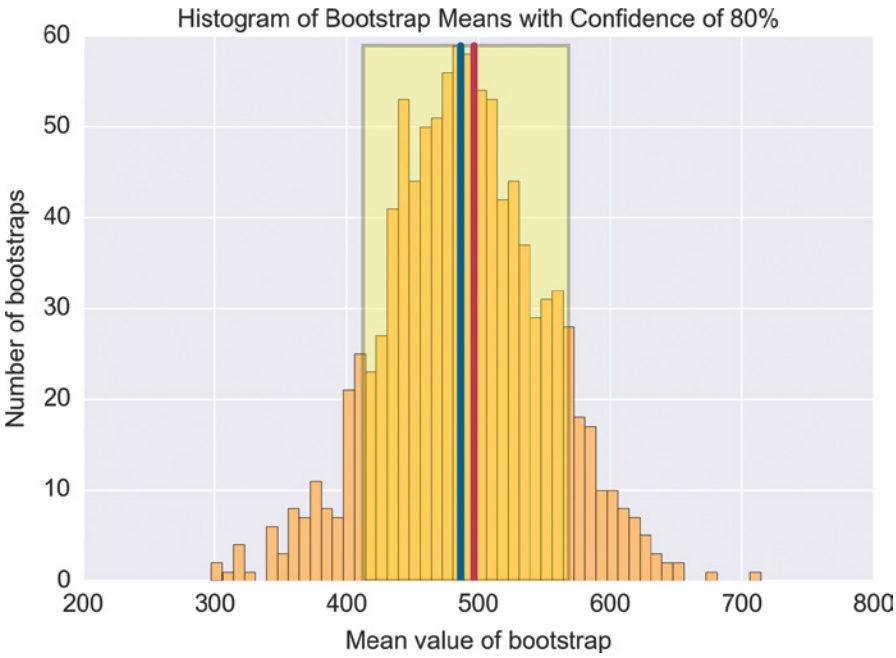


Figure 2-19: We are 80 percent confident that the box contains the population's mean.

Bootstrapping is appealing because often we can use small bootstraps, perhaps only 10 or 20 elements each, even with huge populations of millions of measurements. Because each bootstrap is small, it's typically fast to build and process. To compensate for their small size, we often create thousands of bootstraps. The more bootstraps we build, the more the results look like a Gaussian bump, and the more precise we can be with our confidence intervals.

Covariance and Correlation

Sometimes variables can be related to one another. For example, one variable might give us the temperature outside, and the other the chance of snow. When the temperature is very high, there's no chance of snow, so knowledge of one of the variables tells us something about the other. In this case, the relationship is *negative:* as the temperature goes up, the chance of snow goes down, and vice versa. On the other hand, our second variable might tell us the number of people we expect to find swimming in the local lake. The connection between the temperature and the number of people swimming is *positive,* because on warmer days we see more swimmers, and vice versa.

It's useful to be able to find these relationships, and measure their strength. For instance, suppose we're planning to teach an algorithm to extract information from a dataset. If we find that two of the values in the

data are strongly related (like temperature and chance of snow), we might be able to remove one of them from the data, since it's redundant. This can improve our training speed and can even improve our results.

In this section, we'll look at a measurement called *covariance,* developed by mathematicians to let us determine the strengths of these relationships. We'll also see a variation called *correlation,* which is often more useful because it doesn't depend on the sizes of the number involved.

Covariance

Suppose that we have two variables and we notice a specific numerical pattern involving them. When either variable's value increases, the other increases by a fixed multiple of that amount, and the same thing happens when either variable decreases. For example, suppose variable A goes up by 3, and variable B goes up by 6. Then later, B goes up by 4, and A goes up by 2. Then A decreases by 4, and B decreases by 8. In every example, B goes up or down by twice the amount A went up or down, so our *fixed multiple* is 2.

If we see such a relationship (for any multiple, not just 2), we say that the two variables *covary*. We measure the strength of the connection between the two variables, or the consistency with which they covary, with a number called the *covariance*. If we find that when one value increases or decreases the other does the same by a predictable amount, then the covariance is a positive number, and we say that the two variables are demonstrating *positive covariance*.

The classic way to talk about covariance is to draw points in 2D, as in Figure 2-20. Here we see two different sets of covariant points. Each point has coordinates x and y, but those are just stand-ins for whatever two variables we are interested in comparing. The more consistently the change in y tracks the change in x, the stronger the covariance.

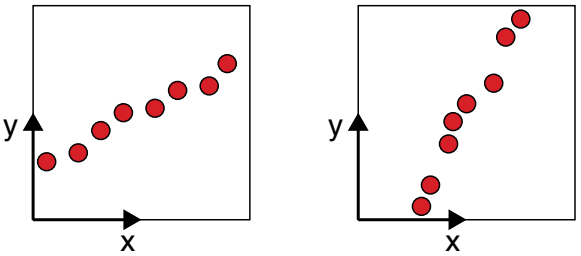


Figure 2-20: Each diagram shows a different set of points with positive covariance.

In the left-hand side of Figure 2-20, the change in y between each pair of horizontally neighboring points is roughly the same. This is positive covariance. On the right side, the change in y is a little more variable between each pair of points, indicating weaker positive covariance. A very strong positive covariance tells us that the two variables move together, so every time one of them changes by a given amount, the other changes by a consistent, predictable amount.

If one value decreases whenever the other increases, we say the variables have *negative covariance*. Figure 2-21 shows two different sets of negatively covariant points.

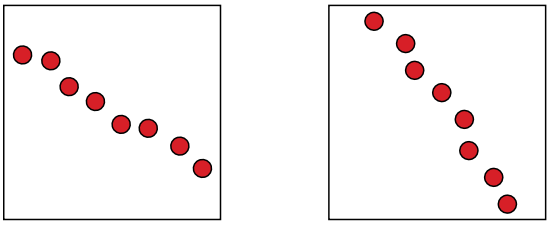


Figure 2-21: Each diagram shows a different set of points with negative covariance.

If the two variables have no such consistently matched motion, then the covariance is zero. Figure 2-22 shows some examples.

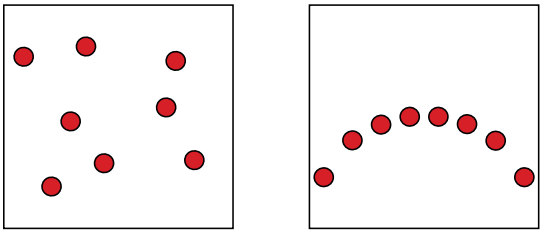


Figure 2-22: Each of these sets of data points has zero covariance.

Our idea of covariance only captures relationships between variables when their changes are multiples of each other. The graph on the right of Figure 2-22 shows that there can be a clear pattern among the data (here the dots form part of a circle), but the covariance is still zero because the relationships are so inconsistent.

Correlation

Covariance is a useful concept, but it has a problem. Because of the way it's defined mathematically, it doesn't take into account relationships between the units of the two variables, which makes it hard for us to compare the strengths of different covariances. For example, suppose we measured a dozen variables describing a guitar: the thickness of the wood, the length of the neck, the time that a note resonates, the tension on the strings, and so on. We might find the covariance between various pairs of these measurements, but we are not able to meaningfully compare the amount of covariance to find which pairs have the strongest and weakest relationships. Even the scale matters: if we find the covariance for a pair of measurements in centimeters and the covariance for another pair of measurements in inches, we can't compare those values to say which pair is more strongly covariant.

The *sign* of the covariance is all we learn: a positive value means a positive relationship, a negative value means a negative relationship, and zero means no relationship. Having only the sign is a problem, because we really want to compare different sets of variables. Then we can find out useful information such as which variables are the most and the least strongly positively and negatively correlated. We can use that information to then prune the size of our dataset, for example, by removing one of the measurements in one or more strongly related pairs.

To get a measure that lets us make these comparisons, we can compute a slightly different number called the *correlation coefficient*, or just the *correlation*. This value starts with the covariance but includes one extra step of computation. The result is a number that does not depend on the units that were chosen for the variables. We can think of the correlation as a scaled version of the covariance, always giving us back a value between –1 and 1. A value of +1 tells us we have a *perfect positive correlation*, while a value of –1 tells us we have a *perfect negative correlation*.

Perfect positive correlation is easy to spot: all the dots fall along a straight line that moves northeast-southwest, as in Figure 2-23.

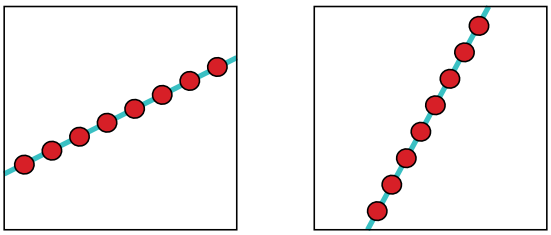


Figure 2-23: Plots showing perfect positive correlation, or a correlation of +1

What kind of relationship between points gives us a positive correlation, but somewhere in the range between 0 and 1? It's one where the y value continues to increase with x, but the proportion won't be constant. We might not be able to predict how much it changes, but we know that increases in x cause increases in y, and decreases in x cause decreases in y. Figure 2-24 shows dot diagrams for some of positive values of correlation between 0 and 1. The closer the dots are to falling on a straight line, the closer the correlation value is to 1. We say that if the value is near zero the correlation is *weak* (or *low*), if it's around 0.5 it's *moderate*, and if it's near 1 it's *strong* (or *high*).

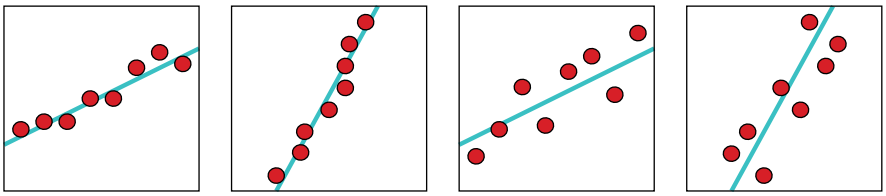


Figure 2-24: Examples of decreasing positive correlation from left to right

Let's now look at a correlation value of zero. A zero correlation means that there is no relationship between a change in one variable and a change in the other. We can't predict what's going to happen. Recall that the correlation is just a scaled version of the covariance, so when the covariance is zero, so is the correlation. Figure 2-25 shows some data with zero correlation.

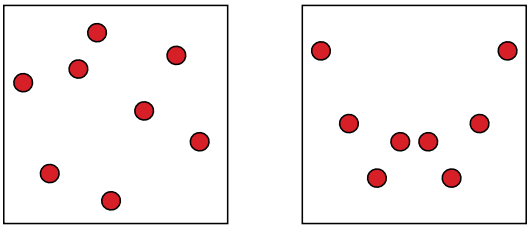


Figure 2-25: These patterns have zero correlation.

Negative correlations are just like positive ones, only the variables move in opposite directions: as x increases, y decreases. Some examples of negative correlations are shown in Figure 2-26. Just like with positive correlation, if the value is near zero the correlation is *weak* (or *low*), if it's around –0.5 it's *moderate*, and if it's near –1 it's strong (or *high*).

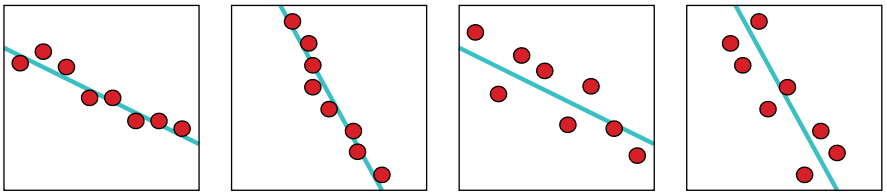


Figure 2-26: Examples of decreasing negative correlation, left to right

Finally, Figure 2-27 shows perfect negative correlation, or a correlation of –1.

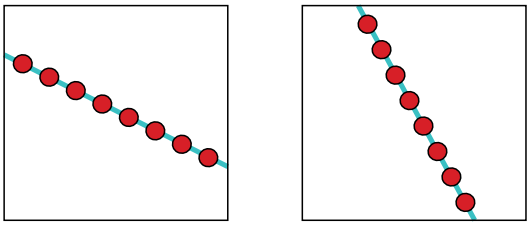


Figure 2-27: These patterns have a perfect negative correlation, or a correlation of –1.

A few other terms are worth mentioning because they pop up in documentation and literature from time to time. Our preceding discussion of two variables is usually called *simple correlation*. We can find the relationship between more variables, however, and this is called *multiple correlation*. If we have a bunch of variables but we're only studying how two of them affect each other, that's called *partial correlation*.

When two variables have a perfect positive or negative correlation (that is, a value of +1 and –1), we say that the variables are *linearly correlated*, because (as we've seen) the points lie on a line. Variables described by any other values of the correlation are said to be *nonlinearly correlated*.

Figure 2-28 summarizes the meanings of different values of linear correlation.

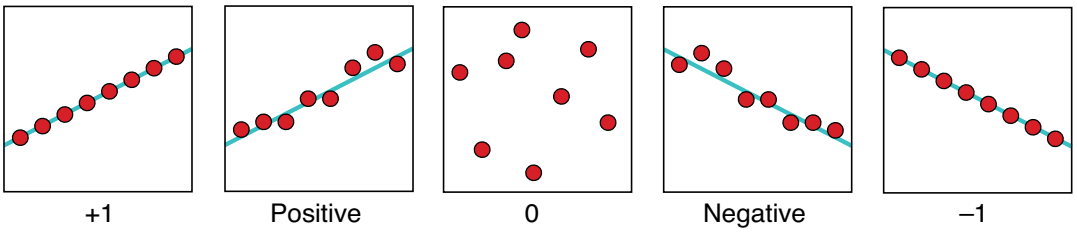


Figure 2-28: Summarizing the meanings of different values of linear correlation

Statistics Don't Tell Us Everything

The statistics we've seen in this chapter tell us a lot about a set of data. But we shouldn't assume that the statistics tell us everything. A famous example of how we can be fooled by statistics is composed of four different sets of 2D points. These sets look nothing like one another, yet they all have the same mean, variance, correlation, and straight-line fit. The data is known as *Anscombe's quartet*, after the mathematician who invented these values (Anscombe 1973). The values of the four datasets are widely available online (Wikipedia 2017a).

Figure 2-29 shows the four datasets in this quartet, along with the straight line that best fits each set.

The amazing thing about these four different sets of data is that they share many of the same statistics. The mean of the x values in each dataset is 9.0. The mean of the y values in each dataset is 7.5. The standard deviation of each set of x values is 3.16, and the standard deviation of each set of y values is 1.94. The correlation between x and y in each dataset is 0.82. And the best straight line through each dataset has a Y axis intercept of 3 and a slope of 0.5.

In other words, all seven of these statistical measures have almost the same values for all four sets of points (some of the statistics differ from one another when we look farther out into more digits). If we just went by the statistics, we'd assume that these four datasets were identical.

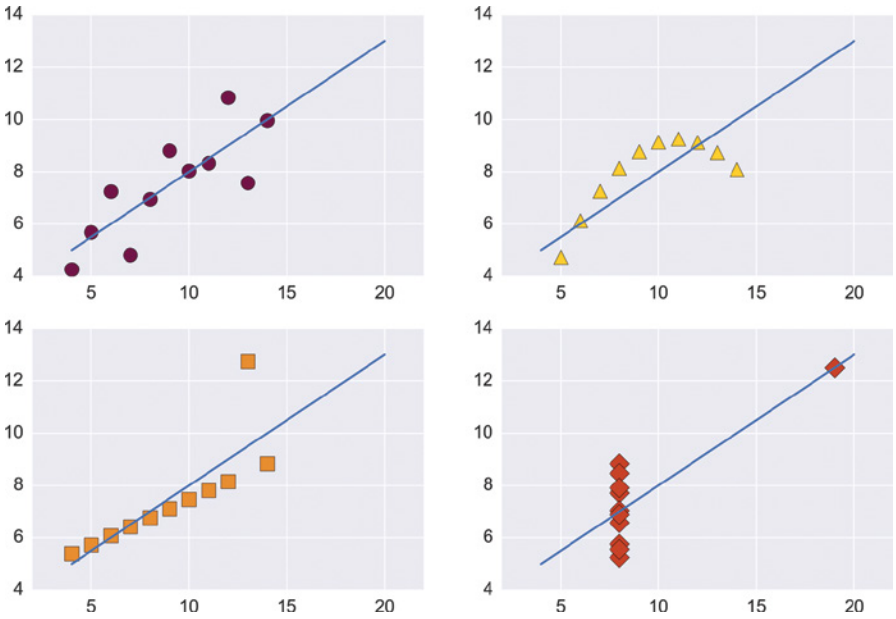


Figure 2-29: The four datasets in Anscombe's quartet and the straight lines that fit them best

Figure 2-30 superimposes all four sets of points, and their best straight-line approximations. All four lines are the same, so we only see one line in the plot.

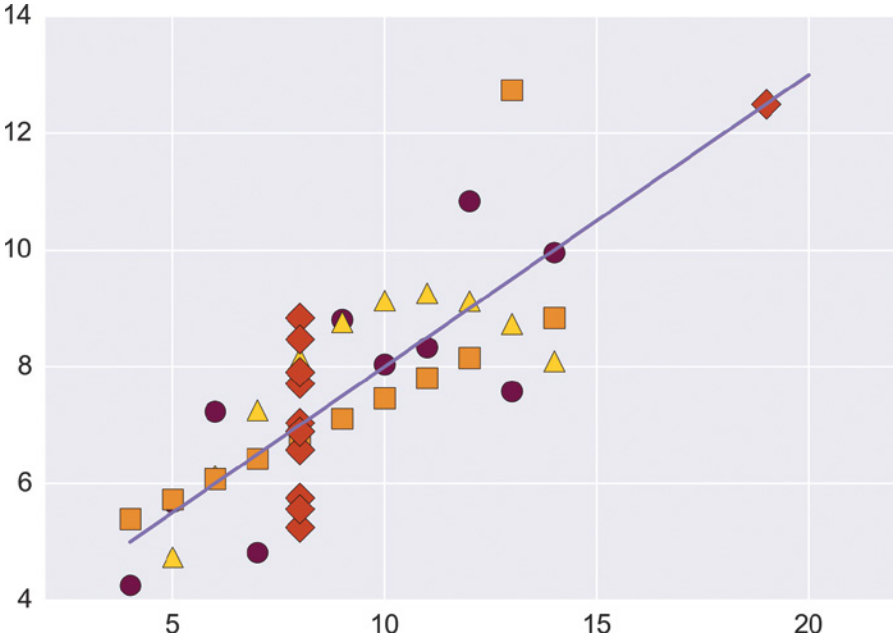


Figure 2-30: The four datasets of Anscombe's quartet and their best straight-line fits, superimposed

These four sets of data, while famous, are not special. If we want to make more sets of different data that have identical (or near-identical) statistics, we can make as many as we want (Matejka and Fitzmaurice 2017). The moral is that we shouldn't assume that statistics tell us the whole story about any set of data.

Any time we work with a new set of data, it's always worth spending time to get to know it. This can include computing statistics, but the investigative process usually also includes drawing plots and other visualizations. Generally speaking, the better we understand our data, the better we can design and train algorithms to learn from that data.

High-Dimensional Spaces

Let's visit one more topic dealing with numbers. It's more of a concept than a statistical tool, but it influences how we think about our data when we do statistics, or machine learning, or almost anything else with large datasets.

In machine learning, we often bundle up many numbers into a single *sample*, or piece of data. For example, we might describe a piece of fruit by its weight, color, and size. We call each number a *feature* of the sample. A photograph would be described as a sample whose features are the numbers that describe the color of each pixel.

We often talk about how each sample is a point in some enormous *space*. If a sample has two features, we can plot the sample as a point, or dot, on a page by associating one feature with the X axis, and the other with the Y axis. If the sample has three features, we can place a dot in 3D space. But we often have samples with far more features. For example, a grayscale photograph that is 1,000 pixels wide by 1,000 pixels high is described by 1,000 × 1,000 pixel values. That's a million numbers. We can't draw a picture of a dot in a space with a million dimensions, and we can't even imagine what such a space might look like, but we can reason about it by analogy with the 2D and 3D spaces we're familiar with. This is an important mental tool for working with real data, so let's get a feeling for the spaces occupied by samples with huge numbers of features.

The general idea is that each dimension, or axis, of a space corresponds to a single feature in our sample. It's useful to think of all the features (that is, all the numbers) in our sample as making up a list. If we have a piece of data that has just one feature (say, a temperature), then we can represent that feature with a list that is only one number long. Visually, we need only show the length of a line to show the size of that measurement, as in Figure 2-31. We call this line a *one-dimensional space*, because from any point on the line, we can only move in one dimension, or direction. In Figure 2-31, that one choice is horizontally.



Figure 2-31: A piece of data with a single value requires only one axis, or dimension, to plot its value. Left: The X axis. Right: Some pieces of data represented by either dots on the X axis, or line segments of different lengths.

If we have two pieces of information in a sample, say the temperature and wind speed, then we need a list that is two items long. To draw it, we need two dimensions, one for each measurement. Graphically, we usually use two perpendicular axes, as in Figure 2-32. The location of a point is given by moving along the X axis by an amount given by the first measurement, and then along the Y axis by an amount given by the second measurement. We say that this is a *two-dimensional space*.

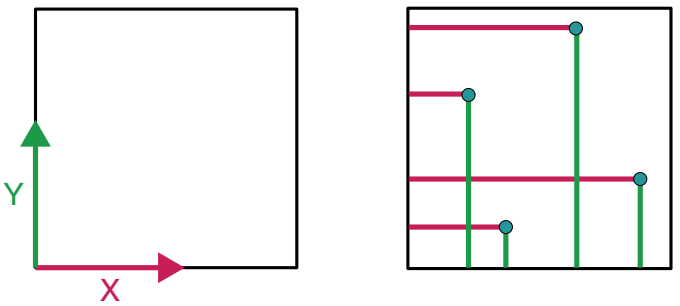


Figure 2-32: If our data has two values, we need two dimensions, or axes, to plot that data.

If we have three values in our sample, then we use a list of three values. As before, each value has a corresponding dimension in the space we're going to plot it. These three dimensions can be represented using three axes, as in Figure 2-33. We call this a *three-dimensional space*.

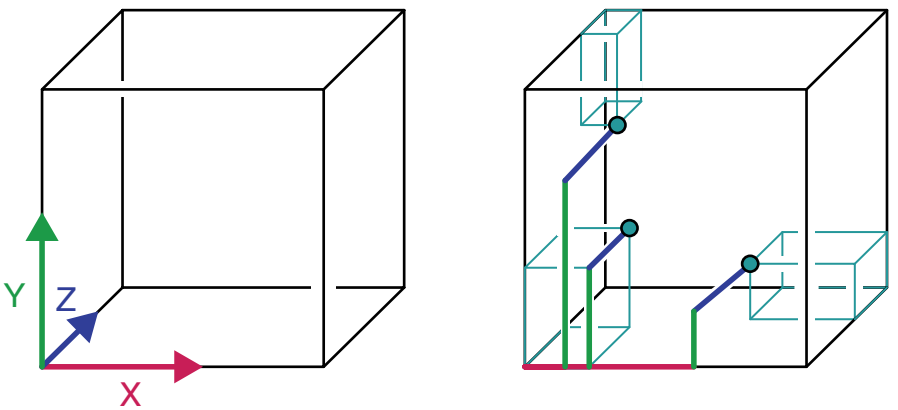


Figure 2-33: When each piece of data has three values, we need three dimensions, or axes, to draw it.

What if we have four measurements? Despite some valiant efforts, there's no generally accepted way to draw a four-dimensional space, particularly on a two-dimensional page (Banchoff 1990; Norton 2014; ten Bosch 2020). And once we start getting up to five, ten, or a million dimensions, drawing a picture of the space is pretty much a lost cause.

It might seem that these high-dimensional spaces are esoteric and rare, but in fact they're common and we see them every day. As we saw, a grayscale picture that's 1,000 pixels on a side has a million values, corresponding to 1,000,000 dimensions. A color picture of the same size has 3,000,000 values, so it's a point (or a dot) in a space of three million dimensions. There's no way we can draw a picture with that many dimensions. There's no way we can even picture one in our minds. Yet our machine learning algorithms can handle such a space as easily as if it had two or three dimensions. The mathematics and algorithms don't care how many dimensions there are.

The key thing to keep in mind is that each piece of data can be interpreted as a single point in some vast space. Just as a two-dimensional (2D) point uses two numbers to tell us where it is on the plane, a 750,000-dimensional point uses 750,000 numbers to tell us where it's located in that enormous space. We often name spaces so that we can keep track of what they describe, so we might say that our image is represented by a single point in a *picture space*.

We call spaces that have lots of dimensions *high-dimensional spaces*. There's no formal agreement on just when "high" begins, but the phrase is often used for spaces that have more than the three dimensions we can reasonably draw. Certainly, dozens or hundreds of dimensions would qualify as high for most people.

One of the great strengths of the algorithms we'll be using in this book is that they can handle data with any number of dimensions. Computations take more time when more data is involved, but in theory, we can handle data with 2,000 dimensions the same way as data with 2 dimensions (in practice, we usually tune our algorithms and data structures to be most efficient with the dimensionality of the dataset they'll be working with).

We'll frequently work with data that can be thought of as points in abstract, high-dimensional spaces. Rather than dive into the math, we'll rely on an intuitive generalization of the ideas we've just seen, thinking of our spaces as giant (and unvisualizable) analogies of our line, square, and cube, where each piece of data is represented by a point in some vast, abstract space where each direction, or dimension, corresponds to a single value in the sample. We need to be careful about relying on our intuition too much, though. In Chapter 7, we'll see that high-dimensional spaces don't always behave like the 2D and 3D spaces we're used to.

Summary

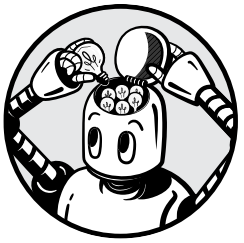
We often need to characterize collections of numbers. The field of statistics is devoted to finding useful ways to describe such collections. In this chapter, we looked at basic statistical measurements that will be useful to us throughout the book. We saw that a convenient way to control the kinds of numbers we need in machine learning is to use a distribution, and we saw some useful distributions.

We saw that we can choose elements from a population with or without replacement, giving us different kinds of collections. We can use the statistics of many such collections, or bootstraps, to estimate the statistics of the starting population. We looked at the ideas of covariance and correlation, which give us a way to measure the amount by which a change in one variable predicts the change in another. And we saw that we can think of lists of numbers as points in spaces of any number of dimensions.

In the next chapter, we'll turn to the ideas of probability, in which we take random events and try to describe how likely they are to happen, and how likely one event is to be followed by another, or occur at the same time as another.

3

MEASURING PERFORMANCE



As we build systems to predict, classify, and otherwise find patterns in our data, we'll need some way to discuss how well they're doing their job. We use a variety of numerical measurements for just this purpose, which we collectively call *performance metrics*. They've been designed to enable us to carefully describe what the system is doing right, and more importantly, when the system gets the wrong answers, specifically how those answers are wrong. These tools are the keys to interpreting any system's results.

Our metrics are based on *probability*, or how likely it is that we'll see different types of results. So we'll begin with a light discussion of probability, focusing on just the most important ideas. Then we'll apply it to build our performance metrics.

Probability is an enormous subject, with many deep specialties. Since our focus is on using machine learning tools sensibly, we only need command of a few basic terms and topics: different kinds of probability, how to

measure correctness, and a particular way of organizing probabilities called the *confusion matrix*. With a command of these basic ideas, we'll be able to prepare our data to get the best performance out of the tools we'll be using later. Broader and deeper discussions on all the topics we'll cover here, as well as many other topics in this field, may be found in many references (Jaynes 2003; Walpole et al. 2011; Kunin et al. 2020).

Different Types of Probability

There are many types of probability. We'll discuss a few of them here, beginning with a metaphor.

Dart Throwing

Dart throwing is the classic metaphor for discussing basic probability. The fundamental idea is that we're in a room with a bunch of darts in our hand, facing a wall. Instead of hanging a cork target, we've painted the wall with some blobs of different colors and sizes. We'll throw our darts at the wall, and we'll track which colored region each one lands in (the background counts as a region as well). The idea is illustrated in Figure 3-1.



Figure 3-1: Throwing darts at a wall. The wall is covered in blobs of paint of different colors.

We're going to assume from now on that our darts will always strike the wall somewhere (rather than going into the floor or ceiling, for instance). So the probability of each dart striking the wall *somewhere* is 100 percent. We'll use both floating-point (or real) numbers and percentages for probabilities, so a probability of 1.0 would be a percentage of 100 percent, a probability of 0.75 would be a percentage of 75 percent, and so on.

Let's look more closely at our dart-throwing scenario. In the real world, we're more likely to hit the part of the wall that's directly in front of us,

rather than, say, something well off to the side. But for the purpose of this discussion, we're going to assume that the probability of our hitting the wall at any point is the same *everywhere*. That is, *every point on the wall has the same chance of being hit by a dart*. Using the language of Chapter 2, we could also say that the probability of striking any given point is given by a uniform distribution.

The heart of the rest of the discussion will be based on comparing the areas of the various regions, and our chances of striking each of those areas. Remember that the background counts as a region (in Figure 3-1, it's the white region).

Here's an example. Figure 3-2 shows a red square on the wall. When we throw a dart, we know it will hit the wall somewhere, with a probability of 1.

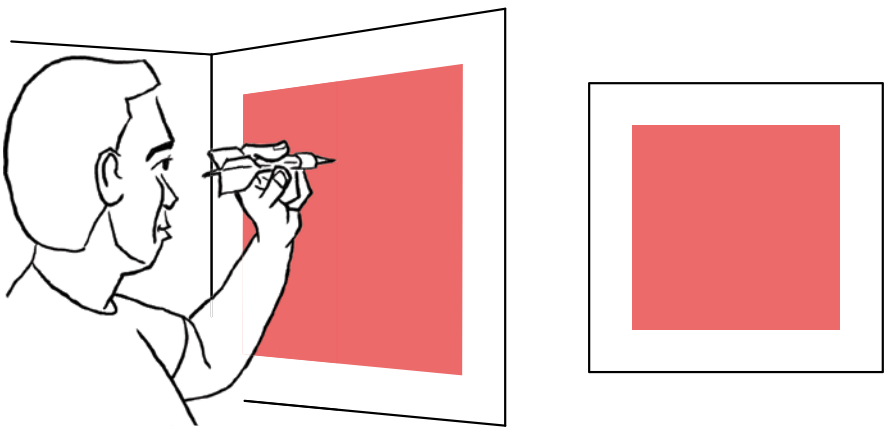


Figure 3-2: We're guaranteed to hit the wall. What's the probability that we'll hit the red square?

What's the probability of hitting the red square? In this figure, the square covers half of the wall's total area. Since our rule is that every point on the wall has an equal likelihood of being hit, when we throw our dart, we have a 50 percent chance, or a probability of 0.5, of the dart landing in the red square. The probability is just the ratio of the areas. The larger our square, the more area it encloses, and so the more likely it is that we'll land inside of it.

We can illustrate this with a little picture that draws the ratios of the areas. Figure 3-3 shows the ratio for our square with respect to the wall. This kind of diagram, where we draw a "fraction" composed of one shape above the other, gives us a visual way to track which areas we're talking about and get an intuitive feel for their relative sizes.

Figure 3-3 shows the relative areas accurately, so the area of the red square is really half the area of the white box under it. Using the full-size shapes can make for awkward diagrams when one of the shapes is much larger than the other, so sometimes we'll scale down regions to make the resulting figure fit the page better. That's okay, because the ratio of the areas won't change. Remember that the purpose of these ratios of shapes is to illustrate the relative area of one shape compared to the area of another.

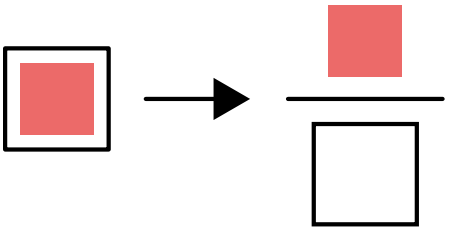


Figure 3-3: The probability of hitting the square in Figure 3-2 is given by the ratio of the area of the red square to the area of the wall, here shown as a symbolic fraction.

Simple Probability

When we talk about the probability of something happening, we refer to that something as an *event*. We often refer to events with capital letters, such as A, B, C, and so on. The phrase "the probability of event A happening" simply means the probability that A happens. To save some space, rather than write "the probability of event A happening," or more succinctly "the probability of A," we usually write P(A) (some authors use a lowercase p, writing p(A)).

Let's say A is the event in which we throw a dart and hit the red square from Figure 3-2. We can represent P(A) with a ratio, as we did earlier. Figure 3-4 shows this graphically.

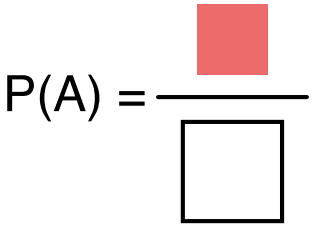


Figure 3-4: We'll say that hitting the square with our dart is event A. The probability of event A occurring is given by the symbolic ratio of areas in Figure 3-3. We write this probability as P(A).

Here, P(A) is the area of the square divided by the area of the wall, so P(A) is 1/2. This ratio is the probability that, when throwing a dart, we'll hit the square rather than the rest of the wall. We call P(A) a *simple probability*.

Conditional Probability

Let's now talk about probabilities involving two events. Either of these events might happen, or both of them, or neither of them.

For example, we might ask for the probability that a house contains a piano, and the probability that there's a dog inside. There's probably no relationship between these two qualities (or events). We say that two events that are not related to one another in any way are *independent*.

Many types of events are not independent, but have at least some kind of connection. We call these *dependent*. When events are dependent, we might want to find their relationship. That is, we'd like to find the probability of one specific event, when we already know that another specific event has happened (or is happening). For example, suppose we pass a house and hear a dog barking inside. Then we might ask, "What is the probability that there's a dog's chew toy in the house, *given* that we know there's a dog inside?" In other words, we know that one event has happened, and we want to know the probability of the other.

Let's make this a bit more abstract, and discuss two events called A and B. Suppose that we know that B has happened, or equivalently, that B is true. Knowing this, we can ask what's the probability that A is *also* true? We write this probability as P(A|B). The vertical bar represents the word *given*, so we'd say this out loud as "the probability that A is true, given that B is true," or more simply, "the probability of A given B." This is called the *conditional probability* of A given B, since it only applies to the situation, or condition, that B is true. We can also talk about P(B|A), which is the probability that B is true, given that A is true.

We can illustrate this with our picture diagrams. The left diagram in Figure 3-5 shows our wall, with two overlapping blobs labeled A and B. P(A|B) is the probability that our dart landed in blob A, given that we already know it landed in blob B. In the symbolic ratio on the right of Figure 3-5, the top shape is the region that is common to both A and B. That is, it's their overlap, or the area where the dart can land in A, given that we know it landed in B.

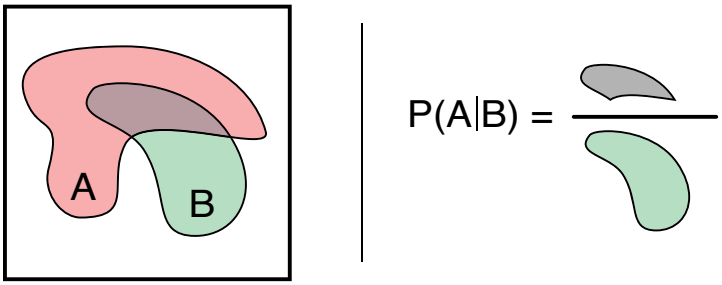


Figure 3-5: Left: The two blobs painted on the wall. Right: The probability of being in A given that the dart is already in B is the ratio of the area of A overlapping B, divided by the area of B.

P(A|B) is a positive number that we can estimate by using our darts. We can estimate P(A|B) by counting all the darts that land in the overlap of A and B, and dividing that number by how many land in any part of B.

Let's see this in action. In Figure 3-6 we've thrown a number of darts at the wall containing the blobs of Figure 3-5. We placed the points to get good coverage over the whole area, with no two points too close to one another. Dart tips are too hard to see, so we show the location of each dart's impact by a black circle, where the center of the circle shows where the dart struck.

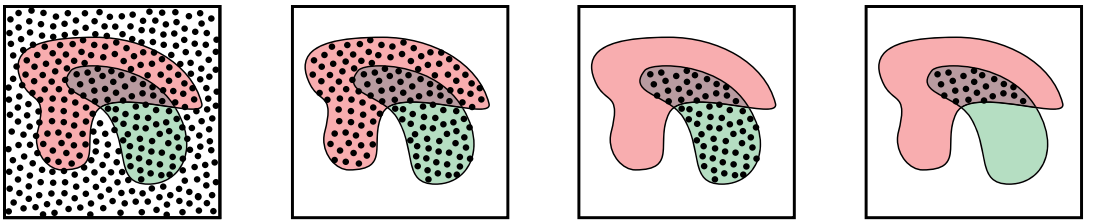


Figure 3-6: Throwing darts at the wall to find P(A|B). (a) Darts striking the wall. (b) All the darts in either A or B. (c) The darts only in B. (d) The darts that are in the overlap of A and B.

In Figure 3-6(a) we show all the darts. In Figure 3-6(b) we've isolated just the darts that landed in either A or B (remember it's only the center of each black circle that counts). In Figure 3-6(c) we see the 66 darts that have landed in region B, and in Figure 3-6(d) we see the 23 darts that are in both A and B. The ratio of 23/66 (about 0.35) estimates the probability that a dart landing in B will also land in A. So P(A|B) is about 0.35. That is, if a dart lands in B, then about 35 percent of the time, it will also be in A.

Note that this process doesn't depend on the absolute area of the colored blobs, such as a number in square inches. It's just the *relative* size of one area with respect to another, which is the only measure we really care about (if the wall doubled in size and so did the colored regions, the probability of landing in each one wouldn't change).

The bigger the overlap of A and B, the more likely the dart is to land in both. If A surrounds B, as in Figure 3-7, then we *must* have landed in A given that we landed in B. In this case, the overlap of A and B (shown in gray) is the region of B itself. Thus the ratio of the overlap's area to B's area is 100 percent, or P(A|B) = 1.

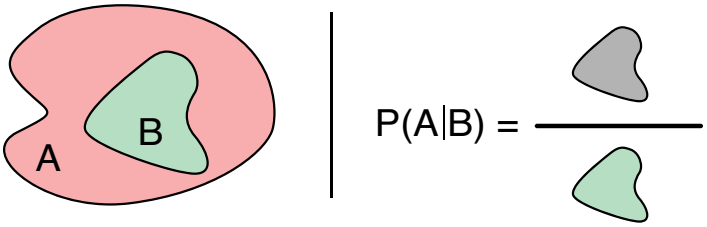


Figure 3-7: Left: Two new blobs on the wall. Right: The probability of landing in A given that we're in B is 1, because A encloses B, and thus their overlap is the same as B.

On the other hand, if A and B don't overlap at all, as in Figure 3-8, then the probability of the dart being in A given that it landed in B is 0 percent, or P(A|B) = 0.

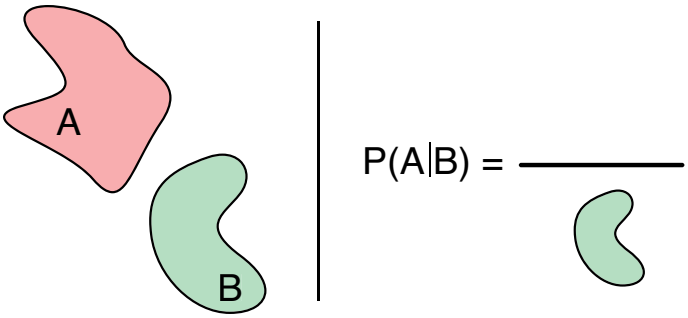


Figure 3-8: Left: Another two new blobs on the wall. Right: The probability of landing in A given that we're in B is 0 (or, equivalently, 0 percent), because there's no overlap between A and B.

The symbolic ratio in Figure 3-8 shows that the area of overlap is 0, and 0 divided by anything is still 0.

For fun, let's flip this around the other way, and ask about P(B|A), or the probability that we're in blob B *given that we're in blob A*. Using the same blobs as in Figure 3-5, the result is shown in Figure 3-9.

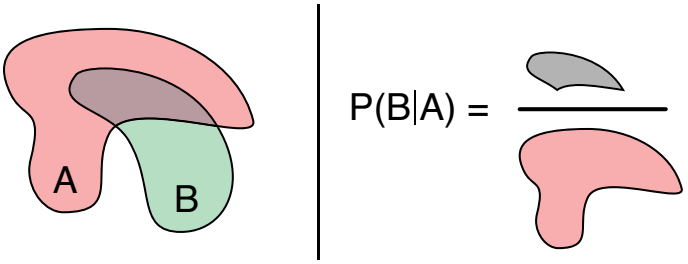


Figure 3-9: The conditional probability P(B|A) is the probability we landed in B, given that we landed in A.

The logic is the same as before. The area of overlap divided by the area of A tells us how much of B appears in A. The more they overlap, the more likely it is that a dart landing in A will also land in B. Let's assign a number to P(B|A). Referring back to Figure 3-6, we see that 104 darts land in A, and 23 in B, so P(B|A) is 23/104 or about 0.22.

Note that the order is important. We can see from Figure 3-5 and Figure 3-9 that P(A|B) does not have the same value as P(B|A). Given the sizes of A, B, and their overlap, the chance of landing in A given that we landed in B is greater than the chance of landing in B given that we landed in A. That is, P(A|B) is about 0.35, but P(B|A) is about 0.22.

Joint Probability

In the last section, we saw a way to express the probability of one event happening, given that another event had already occurred. It would also be

helpful to know the probability of both things happening at once. In the language of our blobs, what's the chance that a dart thrown at the wall will land in *both* blob A and blob B? We write the probability of both A and B happening as P(A,B), where we think of the comma as meaning the word *and*. Thus we read P(A,B) out loud as "the probability of A and B."

We call P(A,B) the *joint probability* of A and B. Using our blobs, we can find this joint probability P(A,B) by comparing the area of the overlap of blobs A and B to the area of the wall. After all, we're asking for the chance that our dart lands in both A and B, meaning inside their overlap, compared to the chance it could land anywhere on the wall. Figure 3-10 shows this idea.

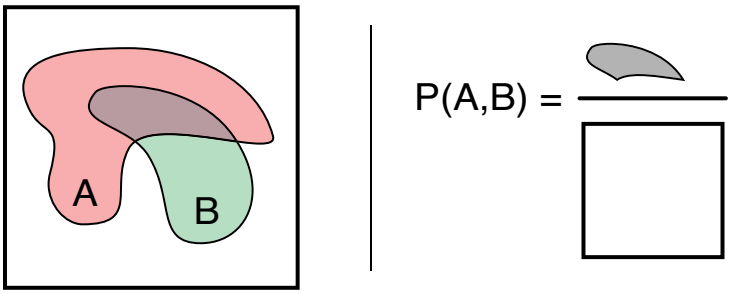


Figure 3-10: The probability that both A and B will occur is called their joint probability, written P(A,B).

There's another way to look at the joint probability that's a little more subtle, but powerful. It's so useful that it will lead to the heart of Chapter 4. This alternative view of the joint probability combines a simple probability with a conditional probability.

Suppose we know the simple probability of hitting B, or P(B). And suppose we also know the conditional probability P(A|B), or the probability of hitting A, knowing that we hit B. We can combine these into a chain of reasoning: given the probability of hitting B, we'll combine that with the probability of hitting A given that we hit B, to get the probability of hitting both A and B at the same time.

Let's see the chain of reasoning with an example. Suppose that blob B covers half of the wall, so P(B) = 1/2. Further, suppose that blob A covers a third of blob B, so P(A|B) = 1/3. Then half of our darts thrown at the wall will land in B, and a third of those will fall in A. Since half of the darts fall in B, and a third of those will also fall in A, the total number that land in both B and in A is 1/2 × 1/3, or 1/6.

This example shows us the general rule: to find P(A,B) we multiply P(A|B) and P(B). This is really quite remarkable: we just found the joint probability P(A,B) using only the conditional probability P(A|B) and the simple probability P(B)! We write this as P(A,B) = P(A|B) × P(B). In practice, we usually leave off the explicit multiplication sign, writing just P(A,B) = P(A|B) P(B).

Figure 3-11 shows what we just did using our little area diagrams.

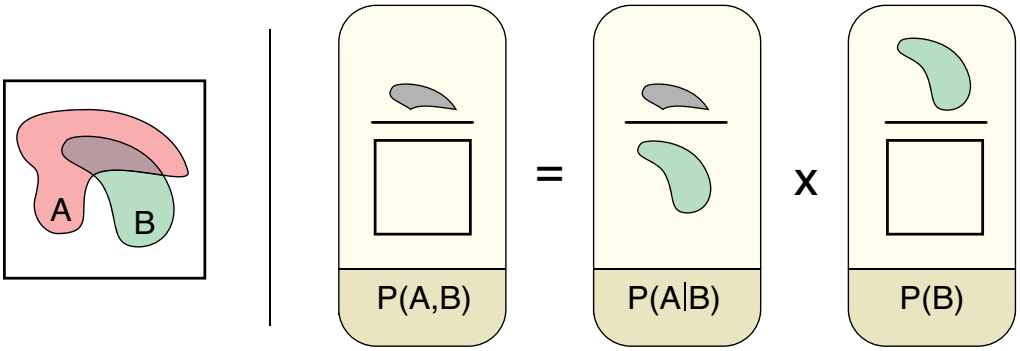


Figure 3-11: Another way to think about the joint probability P(A,B)

Consider the right side of Figure 3-11 and think of the little symbolic ratios as actual fractions. Then the green blobs of area B cancel each other, and we're left with the gray area over the square, showing that the left and right sides of our little equation are, indeed, equal.

We can do this the other way around, too, using event A rather than B. We start with P(B|A) to learn the probability of landing in B given that we landed in A, and then we multiply that by the probability of landing in A, or P(A). The result is P(A,B) = P(B|A) P(A). Graphically, this follows the same pattern as Figure 3-11, only now it's the A blobs that cancel each other.

In symbols, P(B,A) = P(A,B), since both refer to the probability of landing in A *and* B simultaneously. Unlike conditional probability, in joint probability, the order of naming A and B doesn't matter.

These ideas can be a little challenging to get used to, but mastering them will pay off in Chapter 4. It may help to make up a few little scenarios and play with them, imagining different blobs and how they overlap, or even thinking of A and B as actual situations. For instance, imagine an ice cream shop where people can buy different flavors of ice cream, in either a waffle cone or cup. We might say V is true if someone orders vanilla ice cream, and W is true if a person orders their ice cream in a waffle cone. Then P(V) is how likely a random customer will order vanilla, and P(W) is how likely an independently chosen customer will ask for a waffle cone. P(V|W) tells us how likely it is that someone who got a waffle cone ordered vanilla, and P(W|V) tells us how likely it is that someone who ordered vanilla got it in a waffle cone. And P(V,W) tells us how likely it is that a randomly chosen customer got vanilla ice cream in a waffle cone.

Marginal Probability

Another term used for simple probability is *marginal probability*, and understanding where this term comes from will help us understand how we can calculate simple probabilities for multiple events.

Let's start with the word *marginal*, which can seem pretty strange in this context. After all, what does a margin have to do with probability? The legend behind the word *marginal* is that it comes from books that contained tables of precomputed probabilities. The idea is that we (or the printer) would sum up the totals in each row of these tables, and write those totals in the margin of the page (Glen 2014).

Let's illustrate this idea by returning to our ice cream shop. In Figure 3-12 we show some recent purchases made by our customers. Our shop is brand new and serves only vanilla and chocolate, in either a waffle cone or cup. Based on the purchases of the 150 people who came in yesterday, we can ask the probability of someone buying a cup versus a waffle cone, or vanilla versus chocolate. We find those values by adding up the numbers in each row or column (giving us the number in the margin) and dividing by the total number of customers.

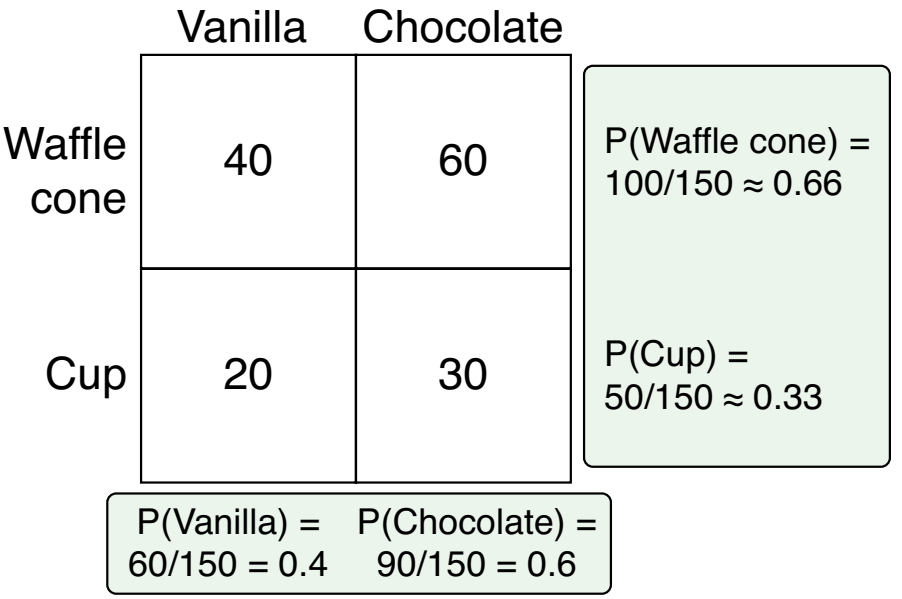


Figure 3-12: Finding marginal probabilities for 150 recent visitors at an ice cream shop. The values in the green boxes (located in the margins of the grid) are the marginal probabilities.

Note that the probabilities of someone buying a cup *or* waffle cone add up to 1, since every customer buys one or the other. Similarly, everyone buys either vanilla or chocolate, so those probabilities also add up to 1. In general, all the probabilities for the various outcomes of any event will always add up to 1, because it's 100 percent sure that *one* of those choices will occur.

Measuring Correctness

Let's now move from probability to our first performance measure: given an imperfect algorithm, how likely is it to produce the correct answer? This

is a key question in machine learning, because we will almost always work with systems that fall short of being perfectly accurate. So it's important to understand what kinds of errors they make.

Let's consider a simple classifier with just two classes. We can ask it for the probability that a piece of data is in some specific class (the two classes are then *in category* and *out of category*). For instance, we might ask for the probability that a photograph is of a dog, or the probability that a hurricane will hit land, or how likely it is that our high-tech enclosures are strong enough to hold our genetically engineered super-dinosaurs (spoiler: not very).

Naturally, we'd like our classifier to make accurate decisions. The trick is to define what we mean by *accurate*. Just counting the number of incorrect results is the easiest way to measure something that we might call accuracy, but it's not very illuminating. The reason is that there is more than one way to be wrong. If we want to use our mistakes to improve our performance, then we need to identify the different ways our predictions can be wrong and consider how much trouble each kind of error causes us. This kind of analysis applies far beyond just machine learning. The following ideas can help diagnose and solve all kinds of problems where we're making decisions on the basis of labels we've assigned.

Before we dig in, we'll note that some of the terms we'll be using here, such as *precision*, *recall*, and *accuracy*, are used casually in popular and informal writing. But in technical discussions (like in this book), these words have precise definitions and have specific meanings. Unfortunately, not all authors use the same definitions for these terms, which can cause all kinds of confusion. In this book, we'll stick to the way they're usually used when discussing probability and machine learning, and we'll define them carefully when we come to them later in this chapter. But be aware that these terms appear in lots of places with different meanings or are just left as vague concepts. It's unfortunate when words get overloaded this way, but it happens.

Classifying Samples

Let's narrow our language to the task at hand. We want to know if a given piece of data, or sample, is, or isn't, in a given class. For now, think of this in yes/no question form: Is this sample in the class? There are no "maybe" answers allowed.

If the answer is "yes," we call the sample *positive*. If the answer is "no," we call the sample *negative*. We'll discuss accuracy by comparing the answers we get from our classifier against the real, or correct, labels that we've assigned beforehand. The choice of positive or negative that we've manually assigned to the sample is called its *ground truth* or *actual value*. We'll say that the value that comes back from our classifier is the *predicted value*. In a perfect world, the predicted value would always match the ground truth. In the real world, there are often errors, and our goal here is to characterize those errors.

We'll illustrate our discussion with two-dimensional (2D) data. That is, every sample, or data point, has two values. These might be a person's height and weight, or a weather measurement of humidity and wind speed, or a musical note's frequency and volume. Then we can plot each piece of data on a 2D grid with the X axis corresponding to one measurement and the Y axis to the other.

Our samples will each belong to one of two classes. Let's call them *positive* and *negative*. To identify a sample's correct classification, or its ground truth, we'll use color and shape cues, as in Figure 3-13.

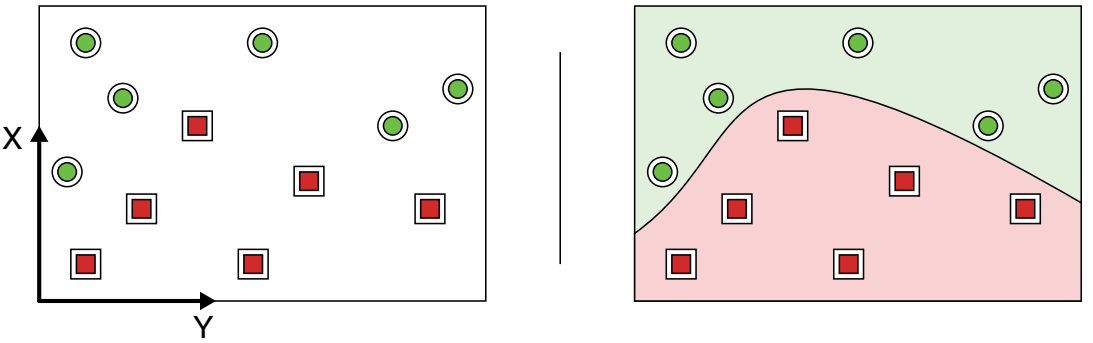


Figure 3-13: Two-dimensional data belonging to two different classes

We'll show the results of our predictions by drawing a *boundary*, or curve, through the collection of points. The boundary may be smooth or twisty. We can think of it as a kind of summary of the classifier's decision-making process. All points in one side of the curve will be predicted to be of one class, while all those on the other side will be predicted to be in the other class. In the right diagram of Figure 3-13, the classifier has done a perfect job of predicting the ground truth of each sample. That's a rare thing.

We sometimes say that the boundary has a positive side and a negative side. This matches up to our class if we think of the classifier as answering the question, "Does this sample belong to the class?" If the answer is positive, then the prediction is "yes," otherwise the prediction is "no." It's often helpful to color in the regions on either side of the boundary, as we've done in Figure 3-13, to make it easy to see which side holds the predictions of *positive*, and which holds the predictions of *negative*.

For our dataset, we'll use a set of 20 samples, shown in Figure 3-14. The samples with a ground truth (or manual label) of *positive* are shown as green circles, while those with a ground truth (or manual label) of *negative* are drawn as red squares. So the color and shape of each sample corresponds to its ground truth, and the background color shows the value assigned by the classifier.

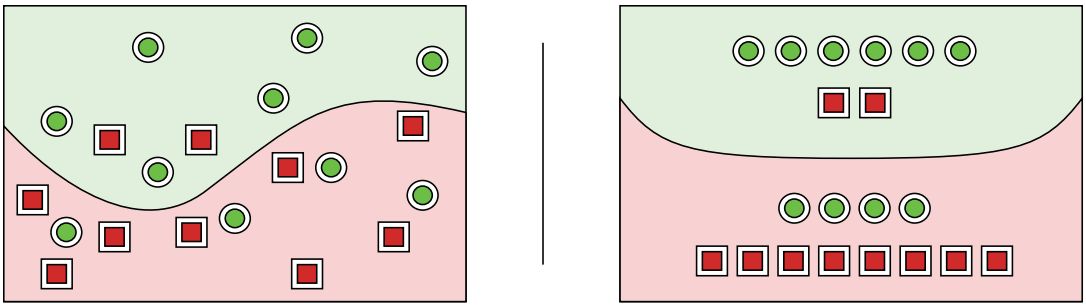


Figure 3-14: Left: The classifier's curve does an okay job of classifying the data, but it makes some mistakes. Right: A schematic version of the same diagram. The curved boundary reminds us that the actual boundary is rarely a straight line.

The job of the classifier is to try to find a boundary so that all the positive samples land on one side, and all the negative samples land on the other. To see how well the classifier's prediction of each sample matches its ground truth, we can just look to see if that sample ended up on the correct side of the classifier's boundary curve. That curve splits the space into two regions. We've used light green to show the positive region, and light red for negative, so every point in the light-green region is predicted, or classified, as positive, and every point in the light-red region is classified as negative.

In a perfect world, all the green circles (the ones with a positive ground truth) would be on the green side of the boundary curve (showing that the classifier predicted them as positive), and all the red squares would be on the red side. But as we can see in the figure, this classifier has made some mistakes. On the left of Figure 3-14 we plotted each piece of data using its two values, along with the boundary curve (and regions) that characterize the classifier's decisions. But we don't really care in this discussion about the specific locations of the points or the shape of the curve. Our interest is in how many points were correctly and incorrectly classified and thus landed on the right and wrong side of the boundary. So in the figure on the right, we've cleaned up the geometry to make it easier to count the samples at a glance.

This diagram represents what typically happens when we run a classifier on a real dataset. Some data is classified correctly, and some isn't. If our classifier isn't performing well enough for us, we'll need to take some sort of action—perhaps by modifying the classifier or even throwing it out and making a new one—so it's important to be able to usefully characterize how well it's doing.

Let's find some ways to do that. We'd like to characterize the errors in Figure 3-14 in a way that tells us something about the nature of the classifier's performance, or how well its predictions matched our given labels. It would be nice to know something more than just "right" and "wrong"—we'd like to know the nature of the mistakes, because some mistakes might matter to us a lot, while others might not matter much at all.

The Confusion Matrix

To characterize the classifier's answers, we can make a little table that has two columns, one for each predicted class, and two rows, one for each actual, or ground truth, class. That gives us a 2 by 2 grid, referred to as a *confusion matrix*. The name refers to how the grid, or matrix, shows us the ways in which our classifier was mistaken, or confused, about its predictions. The classifier's output is repeated in Figure 3-15, along with its confusion matrix.

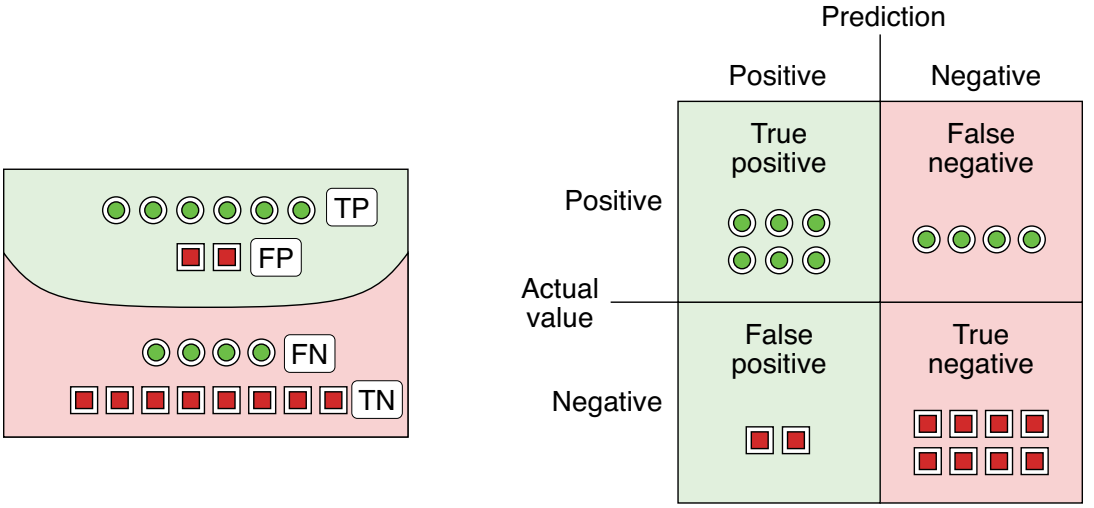


Figure 3-15: We can summarize what went where in Figure 3-14 (repeated here on the left, with labels) into a confusion matrix, which tells us how many samples landed in each of the four classes.

As Figure 3-15 shows, each of the four cells in the table has a conventional name, which describes a specific combination of the predicted and actual values. The six positive green circles were correctly predicted as positive, so they go into the *true positive* category. In other words, they were predicted to be positive, and they actually were positive, so the prediction of positive was correct, or true. The four green circles that were incorrectly classified as negative go into the *false negative* category, because they were incorrectly, or falsely, labeled as negative. The eight red negative squares were correctly classified as negative, so they all go into the *true negative* category. Finally, the two red squares that were incorrectly predicted to be positive go into *false positive*, because they were incorrectly, or falsely, predicted to be positive.

We can write this more concisely using two-letter abbreviations for the four classes and a number describing how many samples fell into each category. Figure 3-16 shows the form that the confusion matrix is usually shown in.

Unfortunately, there is no universal agreement on where the various labels go in confusion matrix diagrams. Some authors put predictions on the left and actual values on top, and some place positive and negative in the opposite locations than shown here. When we encounter a confusion

matrix, it's important to look at the labels and make sure we know what each box represents.

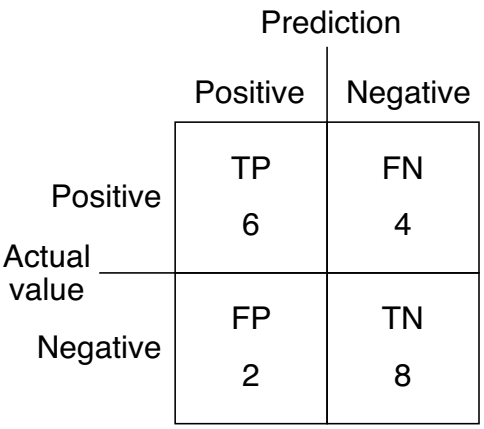


Figure 3-16: The confusion matrix of Figure 3-15 in numerical form

Characterizing Incorrect Predictions

We mentioned earlier that some errors might matter more to us than others. Let's see why that might be.

Suppose that we work for a company that makes toy figurines in the likeness of popular TV characters. Our toys are a hit right now, so our production line is running at full capacity. Our job is to take the manufactured figurines, box them, and ship them off to retail stores.

Suddenly, one day we're told that our company has lost the rights to sell a particular character named Glasses McGlassface. If we accidentally ship any of those figurines, we'll get sued, so it's important to make sure that none of them leave our factory. Unfortunately, the machines are still cranking them out, and if we stop the production line to update the machines, we'll fall way behind on our orders. We decide the better approach is to keep making the forbidden figurines, but spot them after they've been made and throw them into a bin for recycling. So our goal is to identify each Glasses McGlassface and throw it in the bin, making sure none of them get out the door.

Figure 3-17 shows the situation.

We need to work fast, so we might make some mistakes. In Figure 3-17 we see one figurine that we incorrectly recycled. That is, when answering the question, "Is this Glasses McGlassface?" we incorrectly said, "yes." Using our language from the last section, this doll is a false positive. How big a problem is that?

In this case, it's not a big deal (as long as we don't do it too often). Our goal is to make sure that every Glasses McGlassface is correctly identified and removed. Missing even one would cost us a lot. But a false positive costs us only a little, since we'll melt down the plastic and reuse it. So in this situation, false positives, while not desirable, are tolerable.

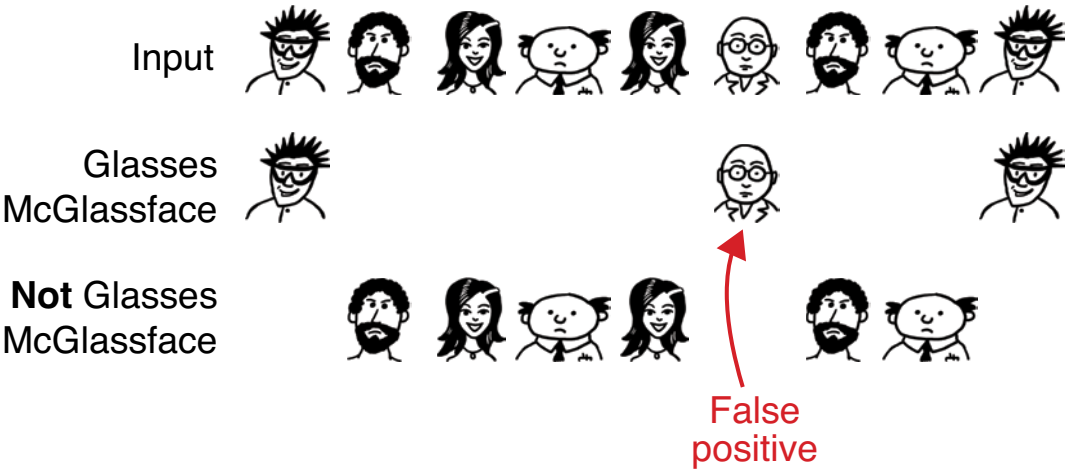


Figure 3-17: Glasses McGlassface is the first character on the top row. We want to remove any doll that could be that character. Our selections are in the middle row.

Suppose we've later noticed that some figurines are not having their eyes painted on properly. Giving a child a toy without eyes could be traumatic, so we definitely want to catch them all. As before, we'll look at every toy, this time asking, "Are the eyes present?" If not, we throw the figurine into a bin for recycling. Figure 3-18 shows the idea.

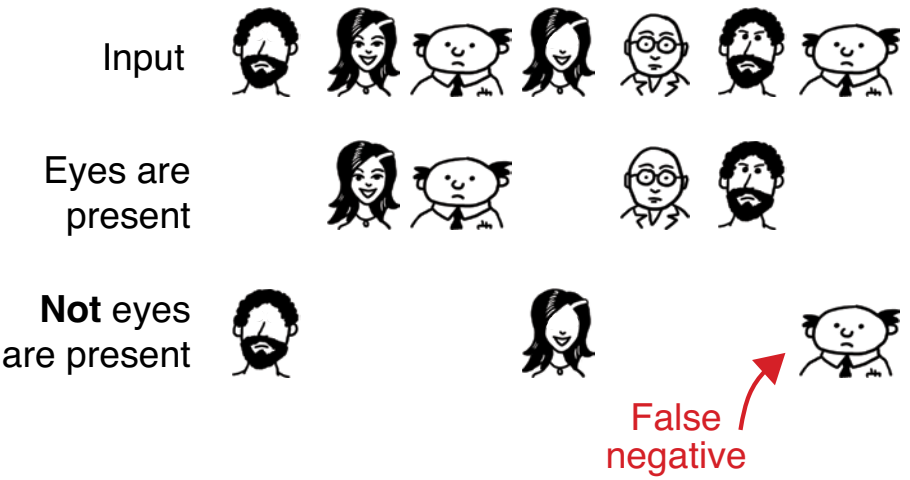


Figure 3-18: A new group of toys. Now we're looking for any with mispainted eyes. Our selections are in the bottom row.

Here we have a false negative: the doll has its eyes painted in, but we said it didn't. In this situation, a few false negatives aren't so bad. As long as we're sure to remove every doll that is missing its eyes, it's okay if we remove a few with their eyes present.

To sum up, true positives and true negatives are the easy cases to understand. How we should respond to false positives and false negatives is dependent on our situation and our goals. It's important to know what our question is, and what our policy is, so we can work out how we want to respond to these different types of errors.

Measuring Correct and Incorrect

Let's return to our overview of true and false positives and negatives, as summarized in a confusion matrix. Looking at a confusion matrix can be, well, confusing, so people have created a variety of terms to help us talk about how well our classifier is performing.

We'll illustrate these terms using a medical diagnosis scenario, where *positive* means someone has a particular condition, and *negative* means they're healthy. Suppose that we're public health workers who have come to a town that's experiencing an outbreak of a terrible but completely imaginary disease called *morbus pollicus (MP)*. Anyone who has MP needs to have their thumbs surgically removed right away, or the disease will kill them within hours. It's therefore critical that we correctly diagnose everyone with MP. But we definitely don't want to make any incorrect diagnoses that lead to removing anyone's thumbs if their life is not in danger—thumbs are important!

Let's imagine that we have a laboratory test for detecting MP. The lab test is flawless, so it always gives us the correct answer: a positive diagnosis means the person has MP, and a negative diagnosis means they do not. Using this test, we've checked every person in town, and we now know whether or not they have MP. But our lab test is slow, and expensive. We're worried about future outbreaks, so based on what we've just learned, we develop a fast, cheap, and portable field test that will predict immediately if someone does or does not have MP.

Unfortunately, our field test is not perfectly reliable, and sometimes makes incorrect diagnoses. Although we know our field test is flawed, when we're in the middle of an outbreak it may be the only tool we have. So we want to characterize how often the field test is correct and how often it's wrong, and when it's wrong, we want to characterize the ways it's wrong.

To work this out, we need data. We've just heard of another town where a few people have reported MP. We'll check every person in town with both tests: our perfect (but slow and expensive) lab test, and our imperfect (but quick and cheap) field test. In other words, the lab test gives us the ground truth for each person, and the field test gives us a prediction. The lab test is too expensive to always run both tests on every person, but we can afford it this once.

By comparing the field test predictions with the lab test label, we'll know all four quadrants of the confusion matrix for our field test:

True Positive: the person has MP, and our field test correctly says that they have it.

True Negative: the person does *not* have MP, and our field test agrees.

False Positive: the person does *not* have MP, but our field test says that they do.

False Negative: the person has MP, but our field test says they don't.

Both true positive and true negative are correct answers, while false negative and false positive are incorrect. A false positive means we'd operate without cause, and a false negative would leave someone at risk of dying.

If we build a confusion matrix for our field test by attaching numbers to each of the four cells, we can use those values to determine how well our field test is performing. We will be able to characterize its performance with a few well-known statistics. The *accuracy* will tell us how often the field test gives us a correct answer, the *precision* will tell us something about false positives, and the *recall* will tell us about false negatives. These values are the standard way that people talk about the quality of a test like this, so let's look at those values now. Then we'll come back to our confusion matrix for the field test, compute these values, and see how they help us interpret the test's predictions.

Accuracy

Each of the terms we'll discuss in this section is built from the four values in the confusion matrix. To make things a bit easier to discuss, we'll use the common abbreviations: TP for true positive, FP for false positive, TN for true negative, and FN for false negative.

Our first term to characterize the quality of a classifier is *accuracy*. The accuracy of the predictions made for any collection of samples is a number from 0 to 1. It's a measure of the percentage of samples that were assigned to the correct category. So it's just the sum of the two "correct" values, TP and TN, divided by the total number of samples measured. Figure 3-19 shows the idea graphically. In this figure, as in the ones to come, the samples we're counting for any given computation will be shown, and the samples that don't contribute to that value will be omitted.

We want the accuracy to be 1.0, but usually it will be less than that. In Figure 3-19, we have an accuracy of 0.7, or 70 percent, which isn't great. The accuracy doesn't tell us in what way the predictions are wrong, but it does give us a broad feeling for how much of the time we get the right result. Accuracy is a rough measurement.

Let's now look at two other measures that provide more specific characterizations of our predictions.

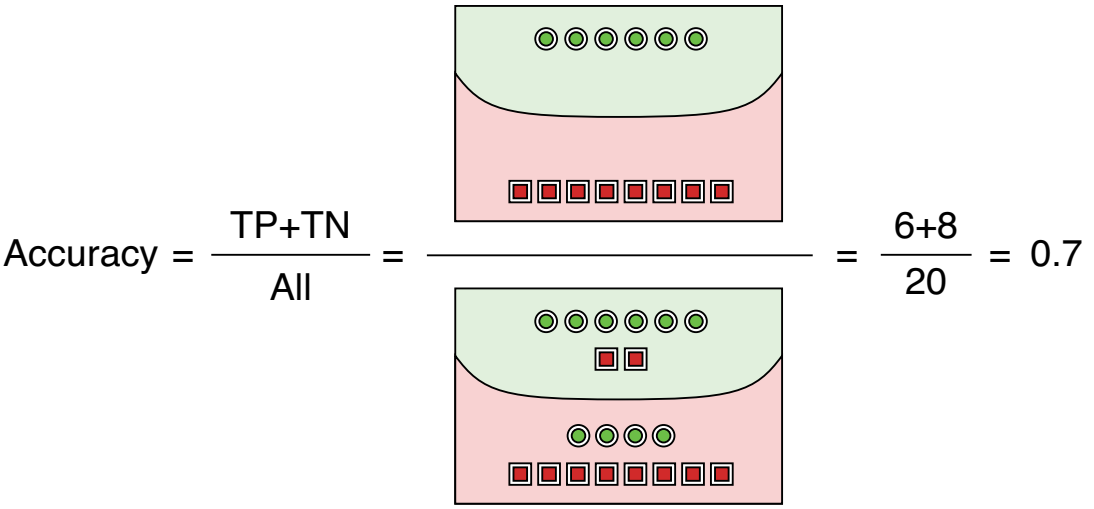


Figure 3-19: Accuracy is a number from 0 to 1 that tells us how often our prediction is correct.

Precision

Precision (also called *positive predictive value*) tells us the percentage of our samples that were properly labeled positive, relative to all the samples we labeled as positive. Numerically, it's the value of TP relative to TP + FP. In other words, precision tells us what percentage of our positive predictions were correct.

If the precision is 1.0, then every sample that really is positive was correctly predicted as positive. As the percentage falls, it carries with it our confidence in these predictions. For example, if the precision is 0.8, then we can only be 80 percent sure that any given sample that's labeled positive has the correct label. Figure 3-20 shows the idea visually.

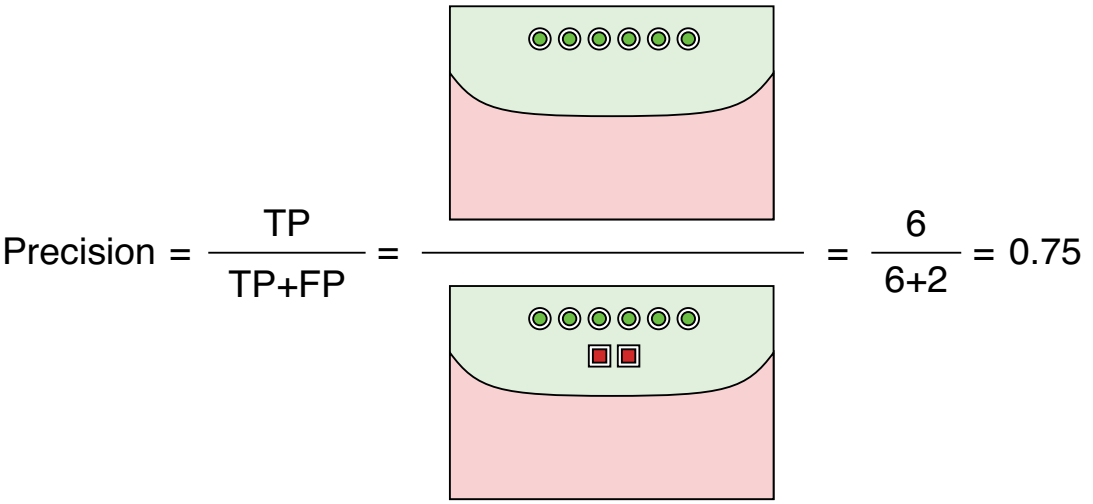


Figure 3-20: The value of precision is the total number of positive samples that really are positive, divided by the total number of samples that we labeled as positive.

When the precision is less than 1.0, it means we labeled some samples as positive when we shouldn't have. In our healthcare example from before with our imaginary disease, a precision value of less than 1.0 means that we'd perform some unnecessary operations. An important quality of precision is that it doesn't tell us if we actually found all the positive objects: that is, all the people who had MP. Precision ignores all samples except those labeled as positive.

Recall

Our third measure is *recall* (also called *sensitivity*, *hit rate*, or *true positive rate*). This tells us the percentage of the samples we correctly predicted to be positive, relative to all the samples that really were positive. That is, it tells us the percentage of positive samples that we correctly predicted.

When recall is 1.0, then we correctly predicted every positive event. The more that recall drops below that number, the more positive events we missed. Figure 3-21 shows this idea visually.

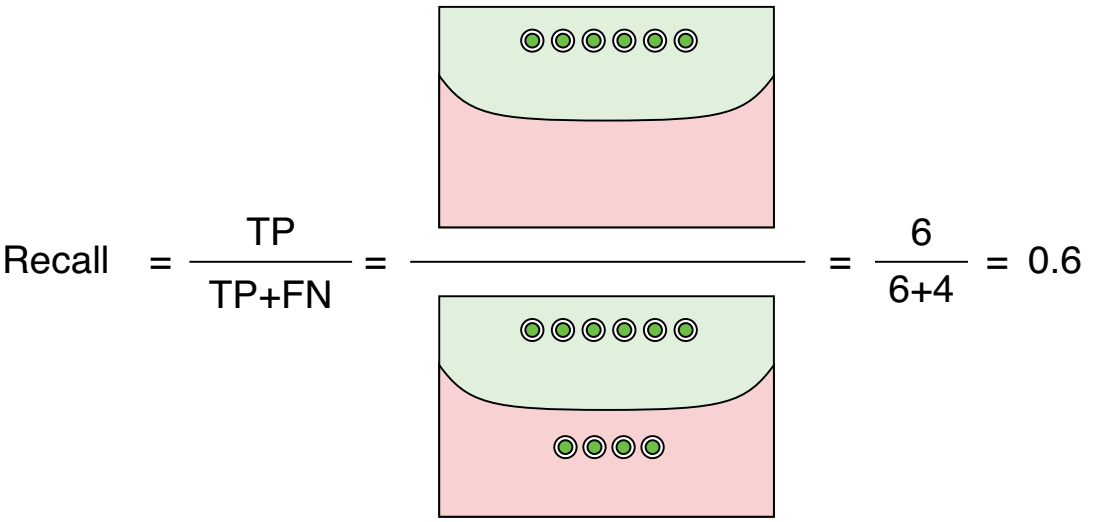


Figure 3-21: The value of recall is the total number of correctly labeled positive samples, divided by the total number of samples that should have been labeled as positive.

When recall is less than 1.0, it means that we missed some positive answers. In our healthcare example, it means we would misdiagnose some people with MP as not having the disease. The result is that we wouldn't operate on those people, even though they're infected and in danger.

Precision-Recall Tradeoff

When we're categorizing data into two classes, and we can't eliminate false positives and false negatives, there's a tradeoff between precision and recall: as one goes up, the other goes down. That's because as we reduce the number of false positives (and therefore increase precision), we necessarily also increase the number of false negatives (and therefore reduce recall), and vice versa. Let's see how this comes about.

Figure 3-22 shows 20 pieces of data. They start out as negatives (red squares) at the far left, and gradually become positives (green circles) as we move right. We'll draw a boundary line vertically somewhere, predicting everything to its left as negative, and everything to its right as positive. We want all the red squares to be predicted as negative, and all the green circles to be positive. Because they're mixed up, there's no boundary that separates the two groups perfectly.

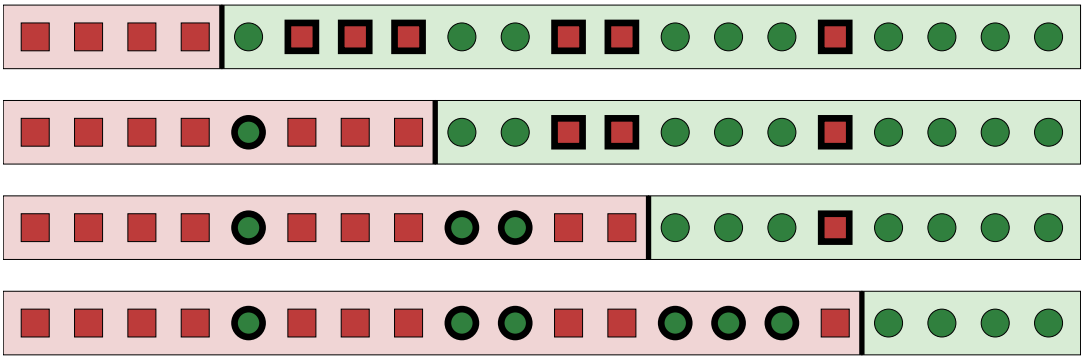


Figure 3-22: As we move the boundary line to the right, from top to bottom, we decrease the number of false positives (red squares with a heavy border), but increase the number of false negatives (green circles with a heavy border).

In the top row of Figure 3-22, the boundary is near the left end. All the green circles are correctly marked positive, but many of the red squares are false positives (shown with a thick outline). As we move the boundary to the right in lower rows, we reduce the number of false positives, but we increase the number of false negatives, because now we're predicting more green circles to be negative.

Let's increase the dataset size to 5,000 elements. The data will be like Figure 3-22, so each entry will be positive with a probability given by its distance from the left end. The leftmost graph of Figure 3-23 shows the number of true positives and true negatives as we move the decision boundary from the far left to the far right. The middle graph shows the number of false positives and false negatives, and the rightmost graph shows the resulting accuracy.

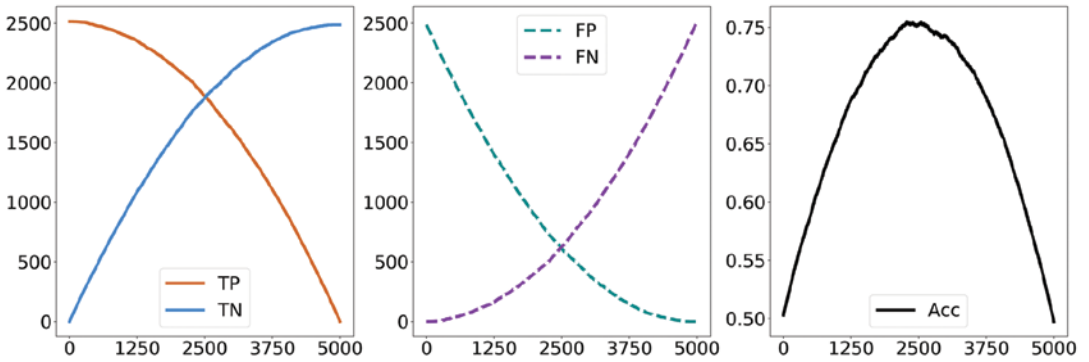


Figure 3-23: Left: The number of true positives and true negatives as we move the boundary. Middle: The number of false positives and false negatives. Right: The accuracy.

To find precision and recall, we'll gather together TP and FP in the left graph of Figure 3-24, and TP and FN in the middle. At the right, we show the result of combining these pairs with TP, following the earlier definitions to compute the precision and recall for each position of the boundary.

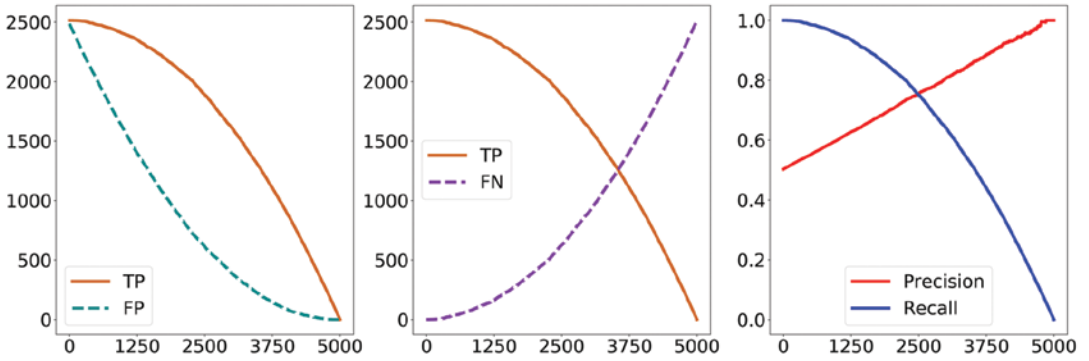


Figure 3-24: TP and FP, TP and FN, and precision and recall as we move the boundary from far left to right

Notice that as we increase precision, we decrease recall, and vice versa. That's the precision-recall tradeoff.

In this example, the precision follows a straight line, whereas the recall is a curve. To get a feeling for why, consider that the sum TP + FP of the curves shown at the left of Figure 3-24 would be a diagonal line from northwest to southeast, whereas the sum TP + FN of the curves in the middle of the figure would be a horizontal line. Dividing the TP curve by these two differently oriented lines gives us the different shapes of the precision and recall curves.

For other kinds of datasets, all of these curves would look different, but the precision-recall tradeoff would remain: the better the precision, the worse the recall, and vice versa.

Misleading Measures

Accuracy is a common measure, but in machine learning precision and recall appear more frequently because they're useful for characterizing the performance of a classifier and comparing it against others. But both precision and recall can be misleading if taken all by themselves, because extreme conditions can give us a great value for either measure, whereas overall performance is lousy.

These misleading results can come from many sources. Perhaps the most common, and difficult to catch, is when we're not careful enough about what we ask the computer to do for us. For example, our organization might want us to produce a classifier that delivers extremely high precision or recall. That may sound desirable, but let's see why it could be a mistake.

To see the problem, consider what might happen if we ask for one of the two extremes of *perfect precision* and *perfect recall*. We'll invent lousy boundary curves to demonstrate the issues, but keep in mind that these can come out naturally from an algorithm tasked to produce perfect precision or recall.

One way to create a boundary curve with perfect precision is to look through all of the samples and find the one we are most certain is really true. Then we draw the curve so that the point we selected is the only positive sample, and everything else is negative. Figure 3-25 shows the idea.

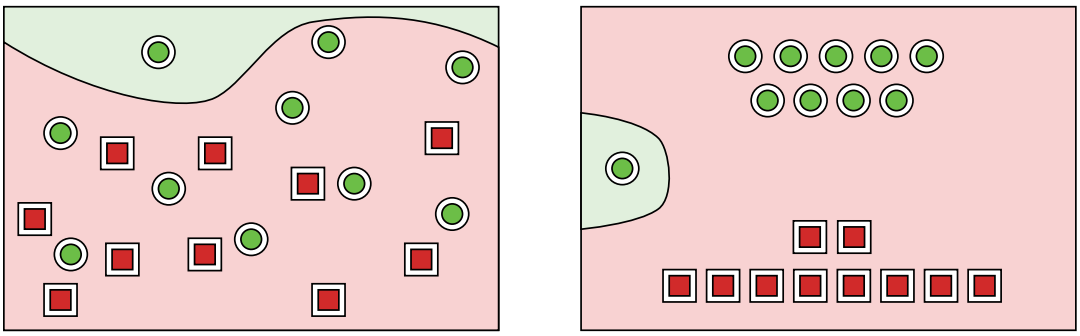


Figure 3-25: Left: This boundary curve gives us a perfect score for precision. Right: A schematic version of the figure on the left.

How does this give us perfect precision? Remember that precision is the number of true positives (here only 1) divided by the total number of points labeled positive (again, just 1). So we get the fraction 1/1, or 1, which is a perfect score. But the accuracy and recall are both pretty awful because we've also created lots of false negatives, as shown in Figure 3-26.

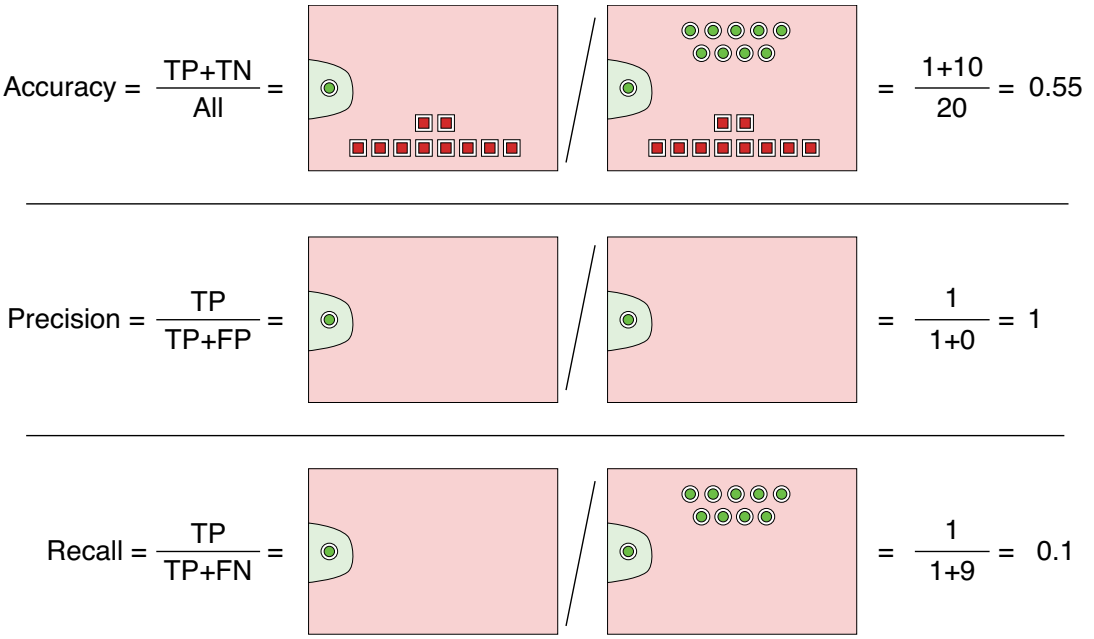


Figure 3-26: These figures all share the same boundary curve, which has labeled exactly one green circle as positive, and all the others as negative.

Let's do a similar trick with recall. To create a boundary curve with perfect recall is even easier. All we have to do is label everything as positive. Figure 3-27 shows the idea.

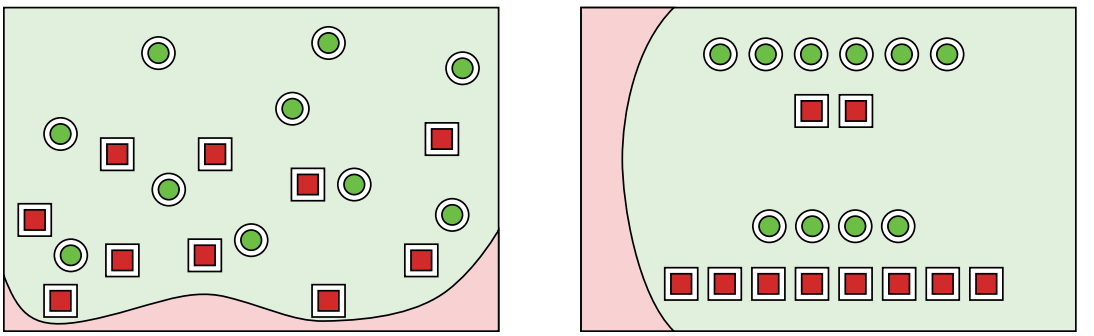


Figure 3-27: Left: This boundary curve gives us a perfect recall score. Right: A schematic version of the figure on the left.

We get perfect recall from this because recall is the number of correctly labeled true points (here, all 10 of them) divided by the total number of true points (again, 10). So 10/10 is 1, or a perfect score for recall. But of course, accuracy and precision are both poor, because every negative sample is now a false positive, as shown in Figure 3-28.

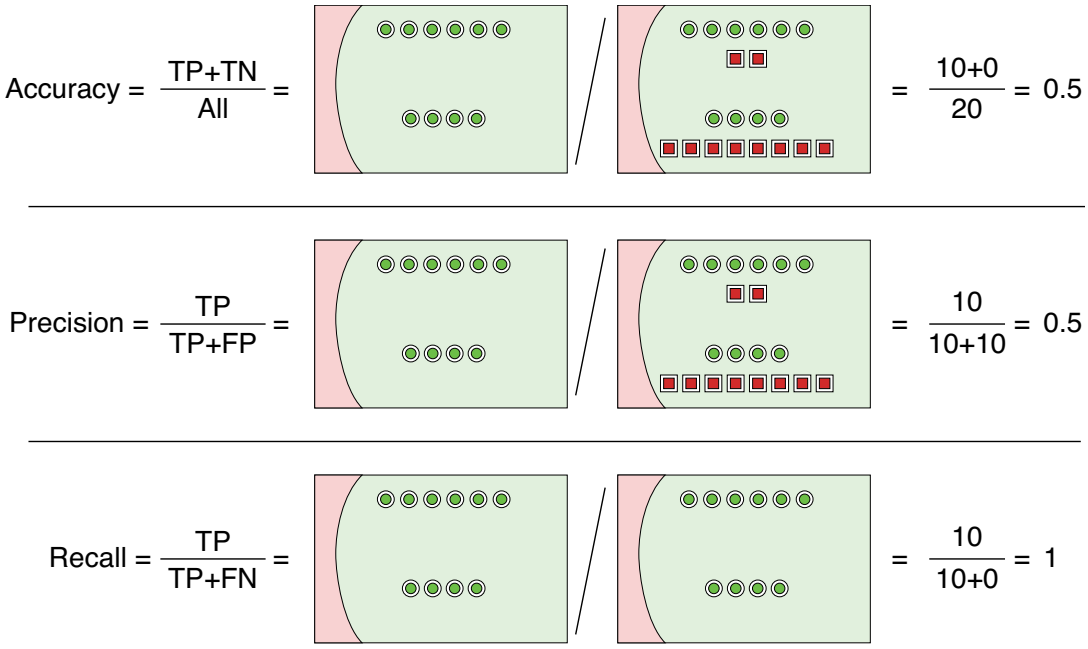


Figure 3-28: All of these figures share the same boundary curve. With this curve, every point is predicted to be positive. We get a perfect recall, because every positive point is correctly labeled. Unfortunately, accuracy and precision both have very low scores.

The moral of Figures 3-26 and 3-28 is that asking for perfect precision or perfect recall is unlikely to give us what we really want, which is perfect correctness. We want accuracy, precision, and recall to all be near 1, but if we're not careful, we can get a great score for just one of these measures by picking an extreme solution that performs poorly when we look at the results in just about any other way.

f1 Score

Looking at both precision and recall is informative, but they can be combined with a bit of mathematics into a single measure called the *f1 score*. This is a special type of "average" called a *harmonic mean*. It lets us look at a single number that combines both precision and recall (the formula appears later on in the last lines of Figure 3-30 and Figure 3-32).

Figure 3-29 shows the f1 score visually.

Generally speaking, the f1 score will be low when either precision or recall is low and will approach 1 when both measures also approach 1.

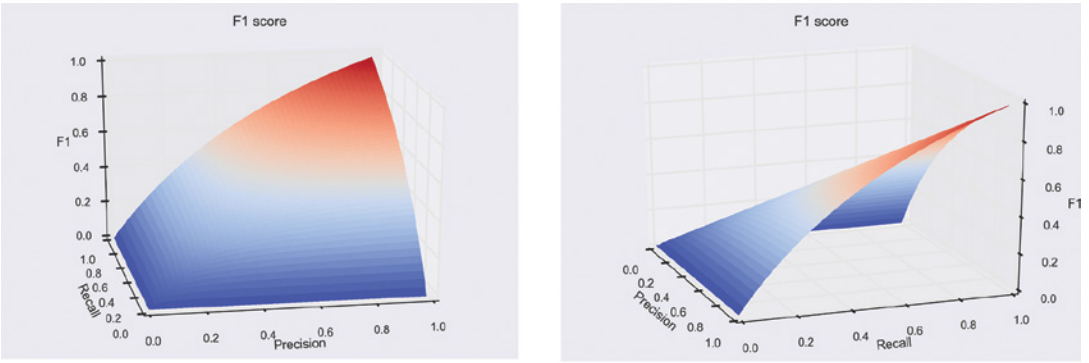


Figure 3-29: The f1 score is 0 when either precision or recall is also 0, and 1 when both are 1. In between it slowly rises as both measures increase.

When a system is working well, sometimes people just cite the f1 score as a shorthand way of showing that both precision and recall are high.

About These Terms

The terms *accuracy*, *precision*, and *recall* may not seem obviously connected to what they measure. Let's make those connections, which can help us remember what these terms mean.

Accuracy tells us what percentage of the samples we predicted correctly. If we predicted every label perfectly, accuracy would be 1. As the percentage of mistakes increases, accuracy drops toward 0. To characterize our mistakes, we want to know our rate of false positives and false negatives. This is what precision and recall are for.

Precision reveals our percentage of false positives, or how many samples we incorrectly predicted to be positive. So this measures the specificity, or precision, of our positive prediction. The larger the value of precision, the more confidence we have that a positive prediction is accurate. In terms of our medical example, if our test has high precision, then it's likely that a positive diagnosis means that person really has MP. But precision doesn't tell us how many infected people we improperly declared to be disease-free.

Recall reveals our percentage of false negatives. If we think of our system as finding, or recalling, just the positives from a set of data, this tells us how well we've done. The better our recall, the more confidence we have that we correctly retrieved all the positive samples. In our medical example, if our test has high recall, then we can feel confident that we've identified everyone with MP. But recall doesn't tell us how many healthy people we incorrectly identified as having MP.

Other Measures

We've seen the measures of accuracy, recall, precision, and f1. There are lots of other terms that are sometimes used in discussions of probability

and machine learning (Wikipedia 2020). We won't encounter most of these terms in this book, but we'll summarize them here to provide a one-stop reference that gathers all the definitions in one place.

Figure 3-30 provides this summary. Don't bother memorizing any unfamiliar terms and their meanings. The purpose of this table is to offer a convenient place to look these things up when needed.

Common Name	Other Names	Abbreviation	Definition	Interpretation
True Positive	Hit	TP	True sample labeled True	Correctly labeled True sample
True Negative	Rejection	TN	False sample labeled False	Correctly labeled False sample
False Positive	False Alarm, Type I Error	FP	False sample labeled True	Incorrectly labeled False sample
False Negative	Miss, Type II Error	FN	True sample labeled False	Incorrectly labeled True sample
Recall	True Positive Rate	TPR	TP/(TP+FN)	% of True samples correctly labeled
Specificity	True Negative Rate	SPC, TNR	TN/(TN+FP)	% of False samples correctly labeled
Precision	Positive Predictive Value	PPV	TP/(TP+FP)	% of samples labeled True that really are True
Negative Predictive Value		NPV	TN/(TN+FN)	% of samples labeled False that really are False
False Negative Rate		FNR	FN/(TP+FN)=1-TPR	% of True samples incorrectly labeled
False Positive Rate	Fall-out	FPR	FP/(FP+TN)=1-SPC	% of False samples incorrectly labeled
False Discovery Rate		FDR	FP/(TP+FP)=1-PPV	% of samples labeled True that are really False
True Discovery Rate		TDR	FN/(TN+FN)=1-NPV	% of samples labeled False that are really True
Accuracy		ACC	(TP+TN)/(TP+TN+FP+FN)	Percent of samples correctly labeled
f1 score		f1	(2*TP)/((2*TP)+FP+FN)	Approaches 1 as errors decline

Figure 3-30: Common confidence terms derived from the confusion matrix

This table is a lot to take in. We provide an alternative that presents the terms graphically, using our distribution of samples from Figure 3-14, repeated here as Figure 3-31.

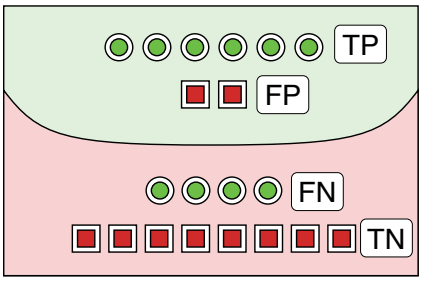


Figure 3-31: The data from Figure 3-14 labeled with the four classes of True Positive, False Positive, False Negative, and True Negative

Reading from top to bottom, we have six positive points correctly labeled (TP = 6), two negative points incorrectly labeled (FP = 2), four positive points incorrectly labeled (FN = 4), and eight negative points correctly labeled (TN = 8).

With these points, we can illustrate the measures of Figure 3-30 by combining these four numbers, or their pictures, in different ways. Figure 3-32 shows how we'd compute the measures using just the relevant pieces of the data.

Constructing a Confusion Matrix Correctly

Understanding a test (or classifier) from its statistical measures can be difficult. There's a lot to take in, and keeping everything straight and organized can be a challenge. It's important to rise to this challenge, because most real-world tests (in every field) are imperfect, as are most machine-learning systems. In general, they need to be understood in terms of their statistical performances.

The confusion matrix is a simple but powerful way to simplify and summarize our understanding. But we have to build and interpret it carefully, or we can too easily come to the wrong conclusions. To wrap up this chapter, let's look more closely at how to properly build and interpret a confusion matrix.

The plan will be to return to our imaginary disease of MP, attach some numbers to our confusion matrix, and ask some questions about the quality of our fast, but inaccurate, field test. Recall that we earlier said that we'd measure everyone in a town with our slow and expensive, but perfectly accurate, lab test (giving us the *ground truth*), as well as our faster, cheaper, and imperfect field test (giving us *predictions*).

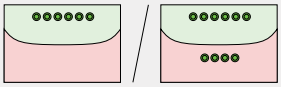
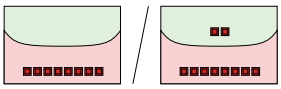
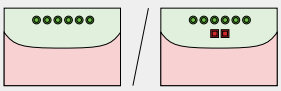
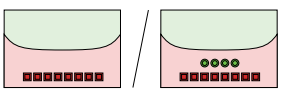
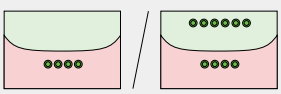
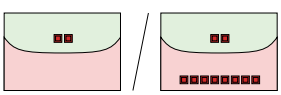
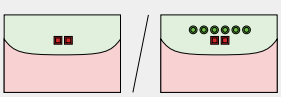
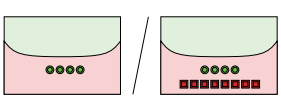
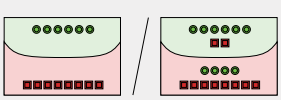
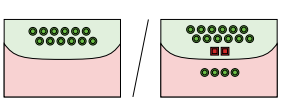
Recall	TPR	$\frac{TP}{TP+FN}$	 $= \frac{6}{6+4} = 6/10 = 0.6$
Specificity	TNR	$\frac{TN}{TN+FP}$	 $= \frac{8}{8+2} = 8/10 = 0.8$
Precision	PPV	$\frac{TP}{TP+FP}$	 $= \frac{6}{6+2} = 6/8 = 0.75$
Negative Predictive Value	NPV	$\frac{TN}{TN+FN}$	 $= \frac{8}{8+4} = 8/12 \approx 0.66$
False Negative Rate	FNR	$\frac{FN}{FN+TP}$	 $= \frac{4}{4+6} = 4/10 = 0.4$
False Positive Rate	FPR	$\frac{FP}{FP+TN}$	 $= \frac{2}{2+8} = 2/10 = 0.2$
False Discovery Rate	FDR	$\frac{FP}{TP+FP}$	 $= \frac{2}{2+6} = 2/8 = 0.25$
True Discovery Rate	TDR	$\frac{FN}{TN+FN}$	 $= \frac{4}{4+8} = 4/12 \approx 0.33$
Accuracy	ACC	$\frac{TP+TN}{TP+FP+TN+FN}$	 $= \frac{6+8}{6+2+4+8} = 14/20 = 0.7$
f1 Score	f1	$\frac{2\ TP}{2\ TP+FP+FN}$	 $= \frac{2*6}{(2*6)+2+4} = 12/18 \approx 0.66$

Figure 3-32: Our statistical measures of Figure 3-29 in visual form using the data of Figure 3-30

Let's suppose that these measurements show that the field test has a high true positive rate: we found that 99 percent of the time, someone with MP is correctly diagnosed. Since the TP rate is 0.99, the false negative (FN) rate, which contains all of the people with MP who we did *not* correctly diagnose, is 1 – 0.99 = 0.01.

The test does a bit worse for people who *don't* have MP. We'll suppose that the true negative (TN) rate is 0.98, so 98 times out of 100 when we predict that someone is not infected, they really aren't. But this means that the false positive (FP) rate is 1 – 0.98 = 0.02, so 2 people in 100 who don't have MP will get an incorrect positive diagnosis.

Let's suppose that we've just heard of an outbreak of MP in a new town of 10,000 people. From experience, given the amount of time that has passed, we expect that 1 percent of the population is already infected. *This is essential information.* We're not testing people blindly. We *already know* that there's only a 1 in 100 chance that someone has MP. It will be essential for us to include this information to correctly understand the results from our field test.

So we pack up our gear and head into town at top speed.

There's no time to send our results to the big and slow lab, so we get everyone to come down to city hall to get tested with our field test. Suppose someone comes up positive. What should they do? How likely is it that they have MP? Suppose instead the test is negative. What should those people do? How likely is it that they *don't* have it?

We can answer these questions by building a confusion matrix. If we jump into it, we might build a confusion matrix just by popping the values above into their corresponding boxes, as in Figure 3-33. But this is *not* the way to go! This matrix is incomplete and will lead us to the wrong answers to our questions.

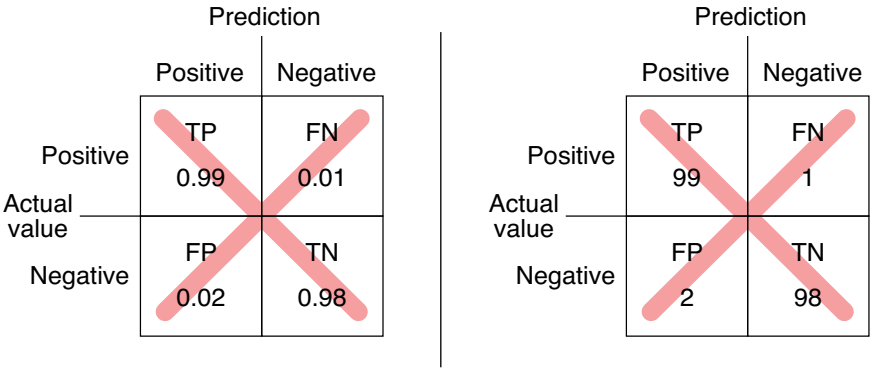


Figure 3-33: This is not the confusion matrix we're looking for. Left: The matrix using our measured values. Right: Multiplying each value by 100 to show them as percentages.

The problem is that we're ignoring a critical piece of information: only 1 percent of the people in town will have MP right now. The chart in Figure 3-33 doesn't include that knowledge and therefore isn't telling us what we need to know.

In Figure 3-34, we work out the proper matrix by considering the 10,000 people in town and analyzing what we expect from the test by using our knowledge of the infection rate and the test's measured performance.

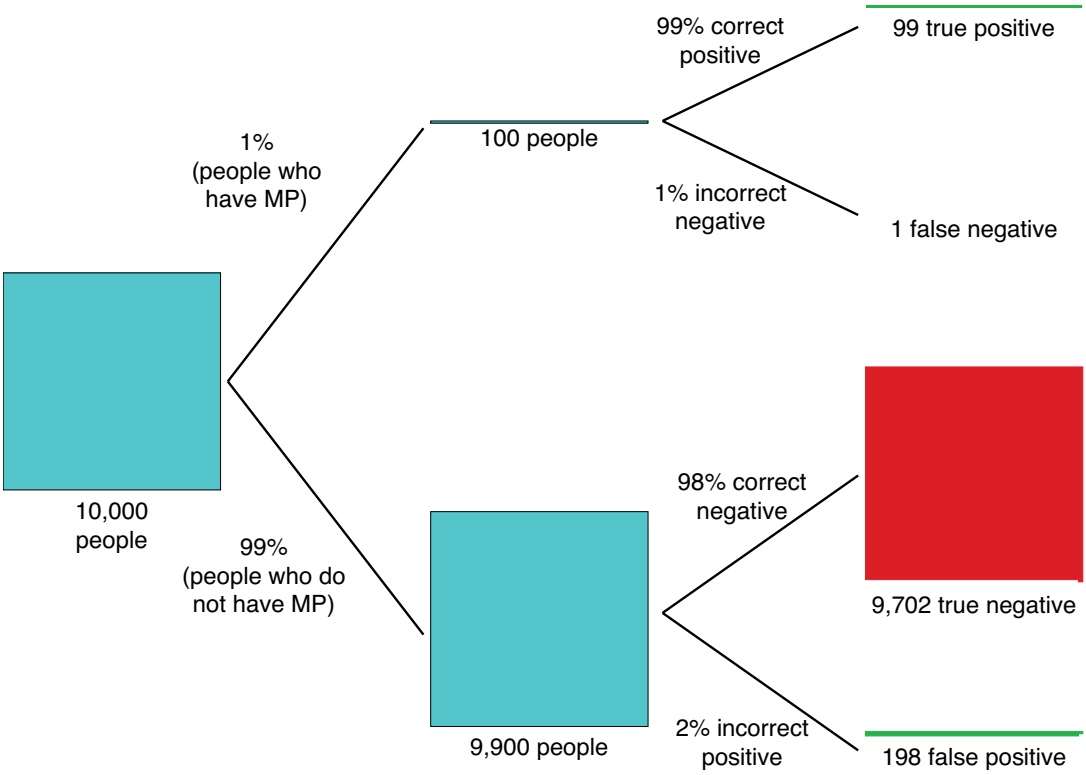


Figure 3-34: Working out the populations we expect from our infection rate and our test

Figure 3-34 forms the heart of the correct process, so let's walk through it. We start at the left with 10,000 people in town. Our essential starting information is that we already know from prior experience that 1 person out of 100, or 1 percent of the population, will be infected with MP. That's shown in the upper path, where we show the 1 percent of 10,000, or 100, people who have MP. Our test will correctly come up positive for 99 of them, and negative for only 1. Returning to our starting population, on the lower path we follow the 99 percent, or 9,900, people who are not infected. Our test will correctly identify 98 percent of them, or 9,702 people, as being negative. The remaining 2 percent of those 9,900, or 198 people, will get an incorrect positive result.

Figure 3-34 tells us the values that we *should* use to populate our confusion matrix, because they incorporate our knowledge of the 1 percent infection rate. From our 10,000 tests, we'll expect (on average) 99 true positives, 1 false negative, 9,702 true negatives, and 198 false positives. These values give us the proper confusion matrix in Figure 3-35.

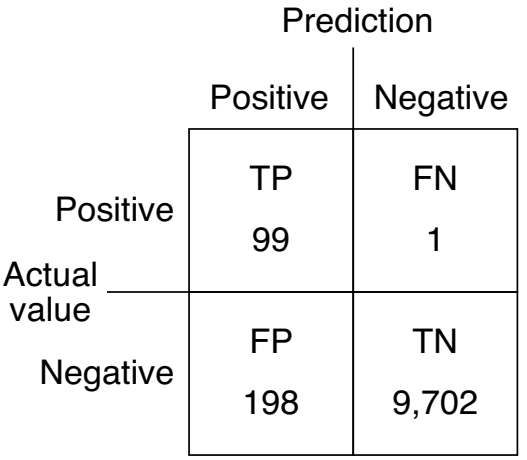


Figure 3-35: The proper confusion matrix for our MP test, incorporating our knowledge of the 1 percent infection rate

Compared to Figure 3-33, the TN rate has changed by a lot! Instead of a TN value of 98, we have 9,702. The value for FP has also gone through a huge change, from 2 to 198. This is important: 198 healthy people are going to get back results saying that they're infected.

Now that we have the right matrix, we're ready to answer our questions. Suppose someone gets a positive test result. What's the chance that they really do have MP? In statistical terms, what's the conditional probability that someone has MP, given that the test says they do? More simply, what percentage of the positive results we get back are true positives? That's just what precision measures. In this case, the precision is 99 / (99 + 198), or 0.33, or 33 percent.

Wait a second. Something seems strange. Our test has a 99 percent probability of correctly diagnosing MP, yet 2/3 of the times when it gives us a positive result, that person does *not* have the disease. More than half of our positive results are wrong!

That definitely seems weird.

And that's why we're going through this example. Understanding probabilities can be tricky. Here we have a test with a 99 percent true positive rate, which sounds pretty great. Yet the majority of our positive diagnoses are wrong.

This surprising result comes about because even though the chance of missing an infected person is very small, there's a huge number of healthy people being tested. So we get a whole lot of those rare incorrect positive diagnoses, and they add up fast. The result is that if someone gets a positive result, we should *not* operate right away. We should instead interpret this result as a signal to do the more expensive and accurate test.

Let's look at these numbers using our region diagrams. We'll have to distort the sizes of the areas in Figure 3-36 in order to make a diagram that we can interpret.

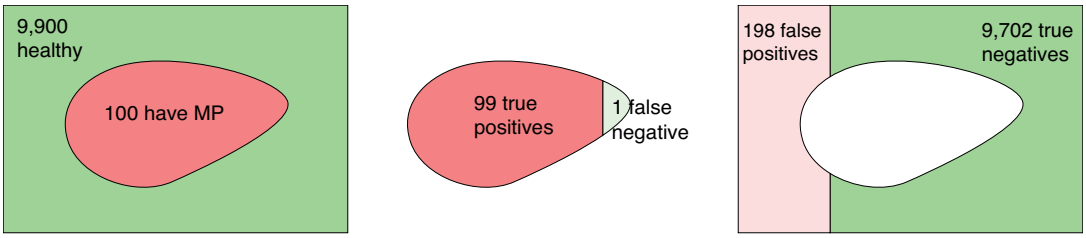


Figure 3-36: Left: The population contains 100 people with MP, and 9,900 without. Middle and Right: The results of our test. The sizes of the shapes are not to scale.

We saw earlier that the precision tells us the chance that someone who is diagnosed as positive really does have MP. This is illustrated at the far left of Figure 3-37. We can see that the field test incorrectly labels people without MP as positive, giving us a precision of 0.33. That tells us to be suspicious of positive results, because $1 - 0.33 \approx 0.66$, or 66 percent, of those results will be wrong.

What if someone gets a negative result? Are they really clear? That's the ratio of true negatives to the total number of negatives, or TN / (TN + FN), which Figure 3-29 gives the name of *negative predictive value*. In this case it's 9,702 / (9,702 + 1). That's well over 0.999, or 99.9 percent. So if someone gets back a negative result, there's only about 1 chance in 10,000 that the test was wrong and they do have MP. We can tell them that, and let them decide if they want the slower, more expensive test.

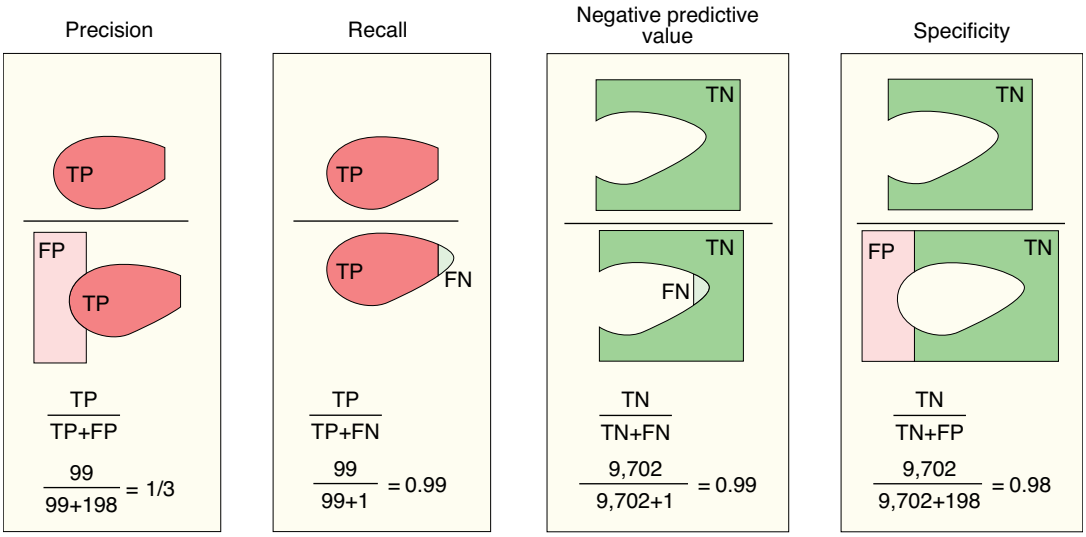


Figure 3-37: Four statistics describing our test for MP based on the results of Figure 3-36. Precision: What percentage of our positives are accurate? Recall: What is our percentage of finding all the positives? Negative Predictive Value: What percentage of our negatives are accurate? Specificity: What is our percentage of finding all the negatives? As before, the region sizes are not to scale.

We've found that the chance that a positive result means that someone actually does have MP is only about 33 percent. On the other hand, a negative result is 99.9 percent sure to be really negative.

Figure 3-37 shows a couple of other measurements. The recall tells us the percentage of people that are properly diagnosed as positive. Since we only missed one person out of 100, that value is 99 percent. The specificity tells us the percentage of people that are properly diagnosed as negative. Since we gave 198 incorrect negative diagnoses, that result is a little less than 1.

To summarize, out of 10,000 people in this town with a 1 percent infection rate, our test will only miss 1 case of MP. But we'll get nearly 200 incorrect positive diagnoses (that is, false positives), which can unduly scare and worry people. Some might even have the surgery right away, rather than wait for the slower test. In our desire to correctly find every person with MP, our test may have become overly zealous in telling people they're infected.

As we saw earlier, if we wanted to make a test that would never miss any person with MP, we could simply label everyone as positive, but that's not useful. The goal in real situations with imperfect systems is to balance the false negatives and false positives in a way that serves our purposes, while keeping those errors in mind.

Our example of MP was imaginary, but the real world is full of situations where people are making important decisions based on incorrect confusion matrices or poorly constructed questions. And some of those decisions are related to real and very serious health issues.

For example, many women have had needless mastectomies because their surgeons misunderstood the probabilities from a breast exam and gave their patients bad counseling (Levitin 2016). Recommending someone undergo an unnecessary surgery is a dangerous mistake. Men were also operated on without cause, because many were given bad advice based on their doctors misunderstanding the statistics of using elevated PSA levels as evidence for prostate cancer (Kirby 2011).

Probability and statistics can be subtle. It's essential that we go slow, think things through, and make sure that we're interpreting our data correctly.

Now we know that we shouldn't be fooled by hearing that some test is "99 percent accurate," or even that it "correctly identifies 99 percent of the positive cases." In our town where only 1 percent of the people are infected, using a test with an impressive 99 percent true positive rate, anyone with a positive diagnosis is more than likely to *not* really have the disease.

The moral is that statistical claims in any situation, from advertising to science, need to be looked at closely and placed into context. Often, terms like "precision" and "accuracy" are used colloquially or casually, which, at best, makes them difficult to interpret. Even when these terms are used in their technical sense, bare claims of accuracy and related measures can easily be misleading and can lead to poor decisions.

When it comes to probability, don't trust your gut. There are surprises and counterintuitive results that lie in wait all over the place. Go slow, gather all the data, and think it through.

Summary

We've seen a lot in this chapter! We covered some of the most important ideas in probability. We saw a term for how likely it is for some event A to happen, P(A); or for some event A to happen given that some other event B already happened, P(A|B); or for events A and B to happen together, P(A,B).

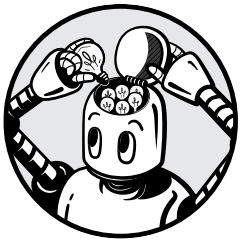
We then looked at a few performance measures that let us characterize how well a test is able to properly identify the positive and negative samples in a dataset. We saw that we can use these measures to help us interpret the results of any decision-making process. We organized those terms into a confusion matrix, which helps us make sense of all that information.

We saw that statistics can be misleading. If we're not careful, we can create tests (or classifiers) that seem to do a great job according to one set of measurements, but are lousy in other ways. It's important to go slow, consider all the data, think things through, and use language carefully when working with probability.

In Chapter 4, we'll apply some of these ideas to a method of reasoning about probabilities that is widely used in machine learning. This will give us another tool to help us later in designing learning algorithms that will learn and be able to usefully perform our desired tasks.

4

BAYES' RULE



In Chapter 3 we discussed some performance measures based on probability. As we dig a little deeper into probability and its use in machine learning, we find that there are two fundamentally different schools of thought about how to approach the subject.

The approach that's most commonly taught in schools is called the *frequentist method*. The other approach is called the *Bayesian method*, named for Thomas Bayes, who originally presented the idea in the 1700s. Although it's less well known, the Bayesian method is popular in machine learning. There are many reasons for this, but one of the most important is that it gives us a way to explicitly identify and use our expectations about the system we're measuring.

In this chapter we first look at the difference between frequentist and Bayesian methods. We then discuss the basics of Bayesian probability, covering enough of it to allow us to make sense of machine learning papers and documentation that are based on Bayesian ideas. We focus on Bayes' Rule, also called Bayes' Theorem, the cornerstone of Bayesian statistics. Even this

single rule is a substantial topic, so we only address it in its broadest terms (Kruschke 2014). We come back to comparing Bayesian and frequentist approaches at the end of the chapter.

Frequentist and Bayesian Probability

In mathematics, there's almost always more than one way to think about a problem, or even a whole field. Sometimes the differences between approaches are subtle, and sometimes they're dramatic. Probability definitely sits in the latter camp. There are at least two different philosophical approaches to probability, each with its strengths and weaknesses. The differences between the frequentist and Bayesian approaches have deep philosophical roots and are often expressed in the nuances of the mathematics and logic that are used to build their corresponding theories of probability (VanderPlas 2014). This makes it difficult to discuss the differences without getting into a wealth of detail. Despite being so different, the challenge of carefully describing the distinctions between these two approaches to probability has been called "especially slippery" (Genovese 2004).

Our approach here is to skip the complex arguments. Instead, we describe both approaches in general terms so that we can get a feeling for their different goals and processes without diving into the details. This helps set the stage conceptually for our discussion of Bayes' Rule, which is the foundation of the Bayesian approach.

The Frequentist Approach

Generally speaking, a *frequentist* is a person who distrusts any specific measurement or observation, considering it to be only an approximation of a true, underlying value. For instance, if we're frequentists, and we want to know the height of a mountain, we assume that each measurement we take is likely to be at least a little too big or too small. At the heart of this attitude is the belief that a true answer already exists, and that it's our job to find it. That is, the mountain has some exact, well-defined height, and if we work hard enough and take enough observations, we'll be able to discover that value.

To find this true value, we combine a large number of observations. Even though we consider each measurement to probably be inexact, we also expect each measurement to be an approximation of the real value. If we take a large number of measurements, we say that the value that comes up most *frequently* is the one that's most *probable*. This focus on the most-occurring value is what gives frequentism its name. The true value is found by combining a large number of measurements, with the most frequent values having the most influence (in some cases, we can merely take an average of all the measurements).

When probability is first discussed in schools, the frequentist approach is usually the one that's presented because it's easy to describe, and often fits well with common sense.

The Bayesian Approach

Using the same broad brush, a *Bayesian* is a person who trusts every observation as an accurate measure of *something*, though it might be a slightly different something each time. The Bayesian attitude is that there's no "true" value waiting to be found at the end of a process. Going back to our mountain example, a Bayesian would say that the true value of the height of the mountain is a meaningless idea. Instead, every measurement of the height of a mountain describes the distance from some point on the ground to some point near the top of the mountain, but they won't be the identical two points every time. So even though every measurement has a different value, each one is an accurate measurement of something we could call the height of the mountain. Each careful measurement is just as true as the others—there's not a single, definitive value out there, waiting for us to find it.

Instead, there's only a range of possible heights for the mountain, each described by a probability. As we take more observations, that range of possibilities generally becomes more narrow, but it never shrinks to a single value. We can never state the height of the mountain as a number, but only as a range, where each value has its own probability.

Frequentists vs. Bayesians

These two approaches to probability have led to an interesting social phenomenon. Some serious people working in probability believe that only the frequentist approach has any merit, and the Bayesian approach is a useless distraction. Other serious people believe exactly the other way around. Many people have less extreme, but still heartfelt, feelings on which approach should be considered the right way to think about probability. Of course, many people think that both approaches offer useful tools that are applicable in different situations. When we work with real data, our choice of how to think about probability can greatly influence what kinds of questions we can ask and answer (Stark and Freedman 2016).

A key feature of the Bayesian approach is that we explicitly identify our expectations before we start taking measurements. In our mountain example, we'd state up front about how high we expect the mountain to be. Some frequentists object to this, arguing that you should never enter an experiment with a preconceived expectation, or bias. Bayesians reply that bias is inevitable since it's baked into the design of every experiment, influencing what we choose to measure, and how. They argue that it's best to state those expectations clearly so that they can be examined and debated. Frequentists disagree and present counterarguments, Bayesians then present counter-counterarguments, and the debate goes on.

Let's look at the two techniques in action by flipping a coin and asking if the coin is fair, so heads and tails come up about equally often, or if it's weighted to come up one way or the other more frequently. Let's start by looking at how a frequentist would address the question, and then we'll see how a Bayesian would go about it.

Frequentist Coin Flipping

People often use flipping (or tossing) coins as an example when discussing probability (Cthaeh 2016a; Cthaeh 2016b). Coin flipping is popular because it's familiar to everyone, and each flip has only two possible outcomes: heads or tails (we'll ignore weird cases like the coin landing on its side). Having only two outcomes makes the math simple enough that we can often work things out by hand. Though we won't be doing any math beyond a few little examples, coin flipping is still a great way to see the underlying ideas, so we use that as our running example here.

We say that a *fair* coin is one that, on average, comes up heads half the time and tails the other half. A coin that isn't fair we call a *rigged*, *weighted*,or *unfair* coin. To describe a rigged coin, we refer to its tendency to come up heads, or its *bias*. A coin with a bias of 0.1 comes up heads about 10 percent of the time, and a bias of 0.8 tells us to expect heads about 80 percent of the time. If a coin has a bias of 0.5, it comes up heads and tails equally often, and it is indistinguishable from a fair coin. We actually might say that a coin with a bias of 0.5, or 1/2, is the definition of a fair coin.

So, by taking lots of measurements (flips) and combining their results (heads or tails), we can hope to find the true answer (the coin's bias). Figure 4-1 illustrates the frequentist's approach to finding the bias of three different coins, using one row per coin.

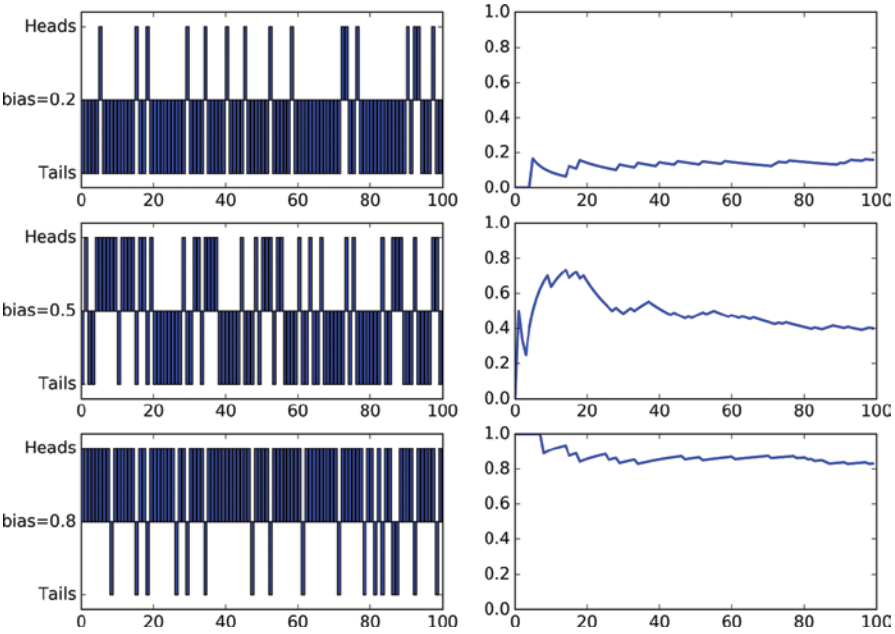


Figure 4-1: Top row: A coin with a bias of 0.2. Middle row: A coin with a bias of 0.5. Bottom row: A coin with a bias of 0.8.

On the left of each row in Figure 4-1 we show 100 consecutive flips of the coin. On the right, we show a frequentist's estimate of that coin's bias after each flip. This is found by dividing the number of heads we've found up until then by the total number of flips. As usual, the frequentist looks at each measurement (here whether the coin is heads or tails) and considers it to be just one little approximation to the truth. By combining all these approximations (here with just a running average) we can converge on the single "true" value giving the bias of the coin.

Bayesian Coin Flipping

Let's consider how we might estimate a coin's bias using a Bayesian point of view. To do so, let's use a slightly more complicated situation that highlights how a Bayesian asks and answers questions.

A Motivating Example

Let's suppose that we have a friend who's a deep-sea marine archaeologist. Her latest discovery is an ancient shipwreck that has, among its treasures, a box containing a marked board and a bag of two identical-looking coins. She thinks these were used for a betting game, and with her colleagues, she's even reconstructed some of the rules.

The key element is that only one of the two seemingly identical coins is fair. The other coin is rigged and will come up heads two-thirds of the time (that is, it has a bias of 2/3). The difference between a bias of 1/2 and a bias of 2/3 isn't big, but it's enough to build a game around. The rigged coin has been cleverly made so that we can't tell which coin is which by looking at them, or even by picking them up and casually feeling them. The game involves players flipping these coins and trying to figure out which coin is which, along with various forms of bluffing and betting along the way. When the game is over, the players figure out which coin is rigged and which is fair by spinning both coins on their edges. Because of its uneven weighting, the biased coin drops sooner than the fair coin.

Our archaeologist friend wants to explore the game further, but she needs to know the true identities of the coins. She asks us to help sort it out. She gives us two envelopes, marked Fair and Rigged, and our job is to put each coin in the appropriate envelope. We could use the spinning test to work out which coin is which, but let's do it with probabilities instead, so we can get some experience with thinking this way.

Let's start by picking a coin, flipping it once, seeing if it comes up heads or tails, and then finding out what we can do with that information. The first step is to select a coin. Since we can't tell the two coins apart by looking at them, we have a 50 percent chance of picking up the fair coin, and the same chance for selecting the rigged coin. This choice sets up our big question: *Did we choose the fair coin?* Once we know which coin we've got, we can put it in its corresponding envelope and put the other coin in the other envelope. Let's rephrase our question in terms of probabilities: *What is the*

probability that we picked the fair coin? If we can be sure we have the fair coin, or be sure that we don't, we'll know everything we need to know.

So, we've got a question and we've got a coin. Let's flip. Heads! The great thing about reasoning with probabilities is that with just this one flip, we can already make a valid, quantified statement about which coin we have.

Picturing the Coin Probabilities

To draw the various probabilities involved in the coming discussion, let's bring back our probability diagrams from Chapter 3. Let's imagine a square wall at which we'll throw darts, and that every dart has an equal probability of landing on every point on the wall. We can paint regions of the wall in different colors that correspond to different outcomes. For example, if one outcome has a 75 percent chance of happening, and the other has a 25 percent chance, we can paint three-fourths of the wall blue, and the remaining quarter pink. If we throw 100 darts, we'd expect about 75 of them to land in the blue region, and the rest to land in the pink region.

Our first step was to choose a coin. Since we can't tell one coin from the other, the odds of choosing the fair coin are 50:50. To represent this, we can imagine splitting the wall into two regions of equal size. Let's paint the fair region with flax (a kind of beige), and the rigged region with red, as in Figure 4-2. When we throw a dart at the wall, the odds of it landing in the fair region are 50:50, corresponding to selecting the fair coin.

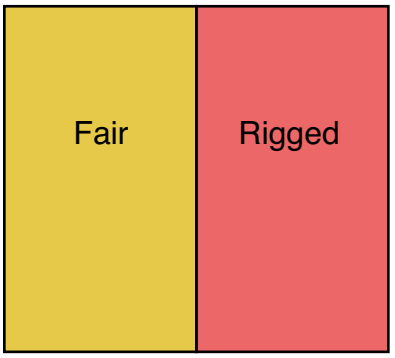


Figure 4-2: When we pick one of the two coins at random, it's the same as throwing a dart at a wall that's been painted with two equal areas, one for the fair coin and one for the rigged coin.

Let's paint the wall in a more informative way that tells us how likely we are to get heads or tails from our first flip. We know that the fair coin has a 50:50 chance of being heads or tails, so we can split the fair region into two equal pieces, one each for heads and tails, as in Figure 4-3.

In Figure 4-3 we also split the rigged side. Since we know from our friend that the rigged coin has a two-thirds chance of coming up heads, we assigned two-thirds of its area to heads and one-third to tails. Figure 4-3 summarizes everything we know about our system. It tells us the likelihood of picking either coin (corresponding to landing in the yellow or red zones),

and the likelihood of getting heads or tails in each situation (from the relative sizes of the heads and tails regions).

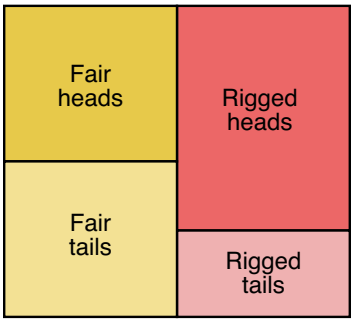


Figure 4-3: We can split up the fair and rigged regions of the wall into heads and tails for each coin using the information we already know about how they are likely to come up when flipped.

If we throw a dart at a wall painted like Figure 4-3, our dart will land in a region corresponding to one of the coins and either heads or tails. But since we've already flipped the coin and observed heads, we know we landed in either Fair heads or Rigged heads.

Remember our question: What is the probability that we picked the fair coin? We can tighten that up by using the information that we got back heads. We'll see later that the best way to phrase our question is in the form of a template that asks, "What is the probability that (something1) is true, given that (something2) is true?" In this case that becomes, "What is the probability that we have the fair coin, given that we saw heads?"

We can diagram this in pictures. It's the area of the Fair heads region compared to the total area that could have given us heads, which is the sum of Fair heads and Rigged heads. Figure 4-4 shows this ratio.

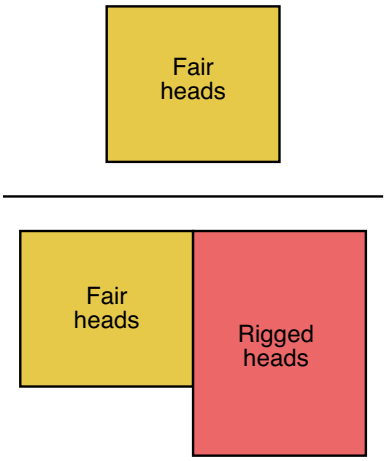


Figure 4-4: If the coin comes up heads, how likely is it that we had the fair coin? It's the size of the region where a fair coin gives us heads divided by all the areas combined that would give us heads.

Let's think about this picture for a moment. Since the Rigged heads area is larger than the Fair heads area, that makes it more likely that our result of "heads" came from landing in the rigged zone. In other words, now that we've seen that our coin came up heads, it's a little more likely that it's the rigged coin, just as a dart thrown at a wall painted as in Figure 4-3 is more likely to land in the Rigged heads region than the Fair heads region.

Later on, we're going to talk about "the ways something can happen," or "all of the ways that something can come about." This means if we're looking for some property to be true, we account for all of the possible events that give us that result. In this case, the bottom half of Figure 4-4 is the sum of all the ways we can get heads. In other words, we can receive heads from either the fair coin or from the rigged coin, so representing "all the ways to get heads" means combining these two possibilities.

Expressing Coin Flips as Probabilities

Let's rephrase Figure 4-4 using probability terms. The probability of getting the fair coin *and* getting heads is P(H,F) (or equivalently P(F,H)). The probability of getting the rigged coin *and* getting heads is P(H,R).

Now we can interpret the ratio of areas in Figure 4-4 as a probability statement. That diagram shows us the chance that our coin, which we know came up heads, is the fair coin. That's P(F|H), which stands for "the probability that we have the fair coin given that we observed heads." That is, this conditional probability is the answer to our question.

We can put this all together into Figure 4-5.

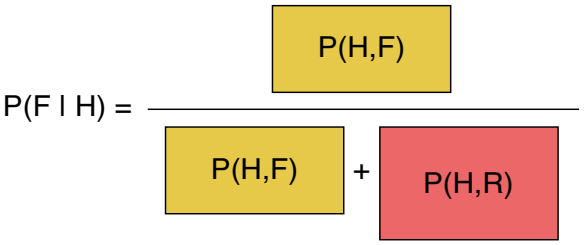


Figure 4-5: Translating Figure 4-4 into the language of probability

Can we plug numbers into this diagram and come up with an actual probability? Sure, in this case we can, because the situation was contrived to be simple. But in general, we won't know any of these joint probabilities, and they won't be easy to find out.

Not to worry. All the boxes on the right of Figure 4-5 are joint probabilities, and we saw in Chapter 3 that we can write any joint probability in two different and equivalent ways, each involving a simple probability and a conditional probability. Those terms are usually much easier for us to put numbers to. Those two approaches are repeated here as Figure 4-6 and Figure 4-7.

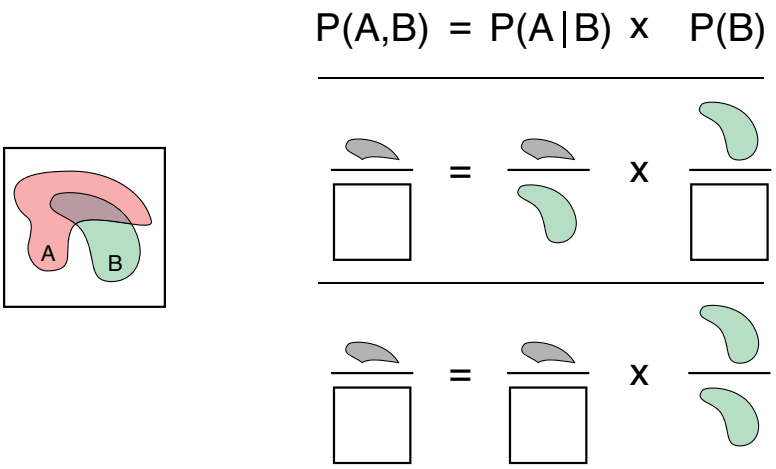


Figure 4-6: We can write the joint probability of two events A and B as the conditional probability P(A|B) times the probability of B, given by P(B).

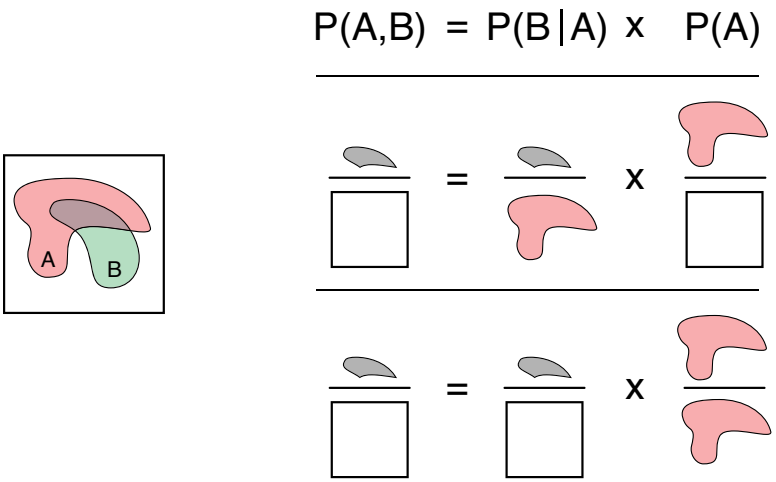


Figure 4-7: We can find the joint probability of two events A and B as the conditional probability P(B|A) times the probability of A given by P(A).

Let's write our expression for P(F|H) in Figure 4-5 without the colored boxes, and then replace P(H,F) with the expression in Figure 4-7, which tells us that we can find the joint probability of P(H,F), or landing in heads *and* using the fair coin, by multiplying the chance of getting heads from a fair coin, P(H|F), with the chance of having the fair coin in the first place, P(F). This change is shown Figure 4-8.

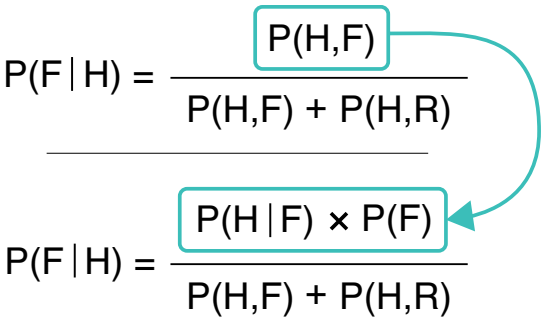


Figure 4-8: Our ratio of Figure 4-5 represents P(F|H), the chance that we have the fair coin given that it came up heads.

Let's do the same thing for the two other joint probabilities, replacing them by their expanded versions (the first of the two values is just P(H,F) again). Figure 4-9 shows the result.

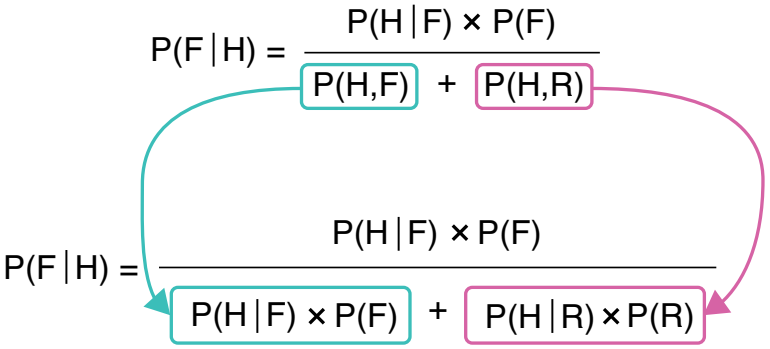


Figure 4-9: We can replace the other two joint probabilities in Figure 4-8 with their expanded versions as well.

Since we can generally find numbers for all of the symbolic expressions in this expanded version, this is a useful way to find P(F|H).

Let's use this expression to find the chance that we just flipped the fair coin. We need to assign a number to each term in Figure 4-9. P(F) is the probability that we picked the fair coin when we started. We've already seen that's P(F)=1/2. P(R) is the probability that we picked the rigged coin when we started, which is also 1/2. P(H|F) is the probability of getting heads given that we chose the fair coin. By definition, that's 1/2. P(H|R) is the probability of getting heads from the rigged coin. From what our archaeologist friend told us, that's 2/3.

So now we have all the numbers we need to work out the probability that we have the fair coin, given that we just flipped it and got heads. Figure 4-10 shows plugging in the numbers and cranking through the steps (following math convention, we perform multiplications before additions—this lets us leave out some distracting parentheses).

$$P(F \mid H) = \frac{P(H \mid F) \times P(F)}{P(H \mid F) \times P(F) + P(H \mid R) \times P(R)}$$

$$= \frac{\frac{1}{2} \times \frac{1}{2}}{\left(\frac{1}{2} \times \frac{1}{2}\right) + \left(\frac{2}{3} \times \frac{1}{2}\right)}$$

$$= \frac{\frac{1}{4}}{\frac{1}{4} + \frac{1}{3}} = \frac{\frac{3}{12}}{\frac{3}{12} + \frac{4}{12}} = \frac{3}{7} \approx 0.43$$

Figure 4-10: Finding the probability that we picked the fair coin, given that we just saw heads

The result is 3/7 or about 0.43. This is kind of remarkable. It tells us that after *just one flip of the coin*, we can already say in a principled way that there's only a 43 percent chance we have the fair coin, and therefore a 57 percent chance that we have the rigged coin. That's a 14 percent difference, from one flip!

As an aside, consider that a frequentist wouldn't dare to characterize the coin as fair or not after just one flip, while this Bayesian approach is already describing the coin with specific probabilities.

Returning to our first flip, let's suppose we received tails instead. Now we'd like to find P(F|T), or the probability that we have the fair coin, given that we saw it come up tails. Recall that the bias is the probability of coming up heads, so the probability of coming up tails is 1 – bias. For the fair coin, the chance of it coming up tails, or P(T|F), is (1 – (1/2)) = 1/2. For the rigged coin, we know from our friend that the bias is 2/3, so P(T|R) is (1 – (2/3)) = 1/3. The chances of picking the fair and rigged coins, given by P(F) and P(R), are each 1/2, just as before. Let's plug those values in and find P(F|T), the probability that we picked the fair coin given that it came up tails. Figure 4-11 shows the steps. The expression for P(F|T) is like the one for P(F|H) with the H's and T's reversed.

This is an even more dramatic answer, telling us it's 60 percent likely that this result of tails means that we're flipping the fair coin (and thus it's 40 percent likely that we're flipping the rigged coin). That's a huge boost of confidence from just one flip! Note that the results aren't symmetrical. If we get heads, we have a 43 percent probability of a fair coin, but if we get tails, we have a 60 percent probability of a fair coin.

We've seen that we can get a lot of information from one flip, but even 60 percent is far from being certain. Making more flips gives us a chance to find more refined probabilities, and we'll see how to do that later in this chapter.

$$P(F|T) = \frac{P(T|F) \times P(F)}{P(T|F) \times P(F) + P(T|R) \times P(R)}$$

$$= \frac{\frac{1}{2} \times \frac{1}{2}}{\left(\frac{1}{2} \times \frac{1}{2}\right) + \left(\frac{1}{3} \times \frac{1}{2}\right)}$$

$$= \frac{\frac{1}{4}}{\frac{1}{4} + \frac{1}{6}} = \frac{\frac{3}{12}}{\frac{3}{12} + \frac{2}{12}} = \frac{3}{5} = 0.6$$

Figure 4-11: What if we got tails on our coin flip?

Bayes' Rule

Let's find another way to write P(F|H). In Figure 4-5, Figure 4-8, and Figure 4-9, we saw several different ways to write the probability that we picked the fair coin given that we saw heads.

Let's go back to the version in Figure 4-8 (repeated at the top of Figure 4-12). Note that the bottom part of the ratio, P(H,F) + P(H,R), combines the probabilities for all the possible ways we could have gotten heads (after all, it has to come from either the fair or rigged coin). If we were dealing with, say, 20 coins, then we'd have to write a sum of 20 joint probabilities, which would make for a very messy expression. We usually use a shortcut, and write these combined probabilities as simply P(H), or "the probability of getting heads." This implicitly means the sum of all the ways we could have gotten heads. If we had 20 different coins, this would be the sum of the probability of each of those coins giving us heads. Figure 4-12 shows this abbreviated notation.

$$P(F|H) = \frac{P(H|F) \times P(F)}{P(H,F) + P(H,R)}$$

$$P(F|H) = \frac{P(H|F) \times P(F)}{P(H)}$$

Figure 4-12: The last line of Figure 4-8, but we've replaced the bottom part of the ratio with the symbol P(H)

Figure 4-13 shows this most recent version all by itself. This is the famous *Bayes' Rule* or *Bayes' Theorem* that we mentioned earlier in the chapter. Here we've adopted the mathematician's convention that two values placed side by side should be multiplied.

$$P(F|H) = \frac{P(H|F)\ P(F)}{P(H)}$$

Figure 4-13: Bayes' Rule, or Bayes' Theorem, as it's usually written

Expressing this in words, we want to find P(F|H), the probability that we have a fair coin given that we just flipped it and saw heads. To determine that, we combine three pieces of information. First, P(H|F), or the probability that, if we do indeed have a fair coin, it will come up heads. We multiply that by P(F), the probability that we have a fair coin. As we've seen, this multiplication is just a more convenient way to evaluate P(H,F), or the probability that our coin is fair *and* that it came up heads. Lastly, we divide everything by P(H), or the probability that our coin will come up heads, *taking into account both the fair and rigged coins.* This is how likely we will be to get heads using *either* of these coins.

Bayes' Theorem is usually written in the form of Figure 4-13, because it breaks things down into pieces that we can conveniently measure (the letters are often changed to better suit what's being discussed). We just replace each term by its corresponding value and out pops the conditional probability that, given heads, we picked the fair coin. Remember that P(H) stands for the sum of the joint probabilities, as we saw in Figure 4-12.

This is why the questions we ask of Bayes' Rule need to be in the form of a conditional probability: *What is the probability that (something1) is true, given that (something2) is true?* It's because that's what Bayes' Rule provides us with. If we can't express our problem in that form, then Bayes' Rule isn't the right tool for answering it.

Discussion of Bayes' Rule

Bayes' Rule can be hard to remember because there are lots of letters floating around, and each one has to go in the right place. But the nice thing is that we can quickly re-derive the rule perfectly any time we need it.

Let's write the joint probability of F and H in both forms (that is, P(F,H) and P(H,F)). We know that these are both the same thing: the probability of having a fair coin and getting heads. Replacing them with the expanded versions as we did in Figure 4-8 gives us the second line of Figure 4-14.

To get Bayes' Rule, just divide each side by P(H), as shown in the third line. The result is the last line, which is Bayes' Rule. This can be a handy way to re-create the rule if we need it and have forgotten it.

$$P(F,H) = P(H,F)$$

$$P(F|H)\ P(H) = P(H|F)\ P(F)$$

$$\frac{P(F|H)\ P(H)}{P(H)} = \frac{P(H|F)\ P(F)}{P(H)}$$

$$P(F|H) = \frac{P(H|F)\ P(F)}{P(H)}$$

Figure 4-14: How to rediscover Bayes' Rule if we forget, or a quick demonstration of why it's true

Each of the four terms in Bayes' Rule has a conventional name, summarized in Figure 4-15.

$$\underset{\text{Posterior}}{P(A|B)} = \frac{\underset{}{\underset{\text{Likelihood}}{P(B|A)} \times \underset{\text{Prior}}{P(A)}}}{\underset{\text{Evidence}}{P(B)}}$$

Figure 4-15: The four terms in Bayes' Rule, with their names

In Figure 4-15 we used the traditional letters A and B, which stand for any kinds of events and observations. With these letters, P(A) is our initial estimate for whether or not we have the fair coin. Because it's the probability we use for "we chose the fair coin" before, or prior to, flipping the coin, we call P(A) the *prior probability*, or just the *prior*.

P(B) tells us the probability of getting the result we did, which in this case was that the coin came up heads. We call P(B) the *evidence*. This word can be misleading, since sometimes this word refers to something like a fingerprint at a crime scene. In our context, *evidence* is the probability that event B could have come about *by any means at all*. Remember that the evidence is the sum of the probabilities for every coin we might have chosen to come up heads.

The conditional probability P(B|A) tells us the likelihood of getting heads, assuming we have a fair coin. We call P(B|A) the *likelihood*.

Finally, the result of Bayes' Rule tells us the probability that we picked the fair coin, given the observation of heads. Because P(A|B) is what we get at the end of the calculation, it's called the *posterior probability*, or just the *posterior*.

Earlier in this chapter we said that a virtue of the Bayesian approach is that it lets us explicitly identify our preconceptions and expectations. Now we can see that we do that by choosing our prior, P(A). In general, we know the likelihood P(B|A) and evidence P(B) from our experimental setup, but we'll have to guess at the prior P(A). This can be a problem if we run the experiment only once, because if our estimate of the prior is wrong, the posterior will also be wrong. We'll see later that if we can run an experiment multiple times (such as by flipping the coin more than once), then we can use Bayes' Rule after each flip to refine our initial prior into a better and better description of P(A), giving us a more accurate value for what we really care about, the posterior P(B|A).

We came up with a value for the prior pretty easily in our little coin-testing example, but in more complicated situations, choosing a good prior can be more complicated. Sometimes it comes down to a combination of experience, data, knowledge, and even just hunches about what the prior should be. Because there's some subjective, or personal, aspect to our choice, picking a prior by ourselves is called *subjective Bayes*. On the other hand, sometimes we can use a rule or an algorithm to pick the prior for us. If we do so, that's called *automatic Bayes* (Genovese 2004).

Bayes' Rule and Confusion Matrices

In Chapter 3 we looked at using the confusion matrix to help us properly understand the outcomes of a test. Let's look at this idea again, but this time using Bayes' Rule.

Rather than create some artificial, contrived example, let's use something realistic and everyday. You're the Captain of the Starship *Theseus*, on a mission into deep space to find rocky, uninhabited planets to mine for raw materials. You've just come across a promising rocky planet. It would be great to start mining it, but your orders are to never, ever mine a planet that has life on it. So the big question is this: Is there life on this planet?

In your experience, most of the life on these rocky worlds is just a little bacteria or bit of fungus, but life is life. As protocol dictates, you send down a probe to investigate. The probe lands and reports "no life."

Because no probe is perfect, we must now ask the question, "What is the probability that the planet contains life, given that the probe detected nothing?" This question is in perfect form for using Bayes' Rule. One condition (let's call it L) is "life is present," where a positive value means the planet has life on it, and a negative value means the planet doesn't have life (so we can start mining). The other condition (we'll call it D) is "detected life," where positive means the probe detected life, and negative means it didn't.

The situation we really want to avoid is mining on a planet that has life. That's a false negative: the probe reported negative, but it shouldn't have. This would be terrible, since we don't want to interfere with, much less destroy, life in any form. False positives are less worrisome. Those are planets that are barren, but the probe thought it found signs of life. The

only drawback there is that we fail to mine a planet we otherwise could have. There's a financial loss, but that's all.

The scientists who built our probes shared these same concerns, so they struggled hard to minimize the false negatives. They tried to keep down the false positives, too, but that wasn't as critical.

In practice, some planets with life might not have life everywhere, so it's possible that a probe could land in a life-free region of a populated planet, and detect nothing. For simplicity, let's not worry about such situations, and say that any incorrect results (that is, missing life that's there, or saying life is present when it's not) will be due to the probe, and not the planet.

The probes they sent us out with have the performance shown in Figure 4-16. To get these numbers, they sent their probes down onto 1,000 known planets of the type we'll be wanting to mine, 101 of which were known to contain life. These values turn into our prior: out of every 1,000 planets, we expect life on 101 of them.

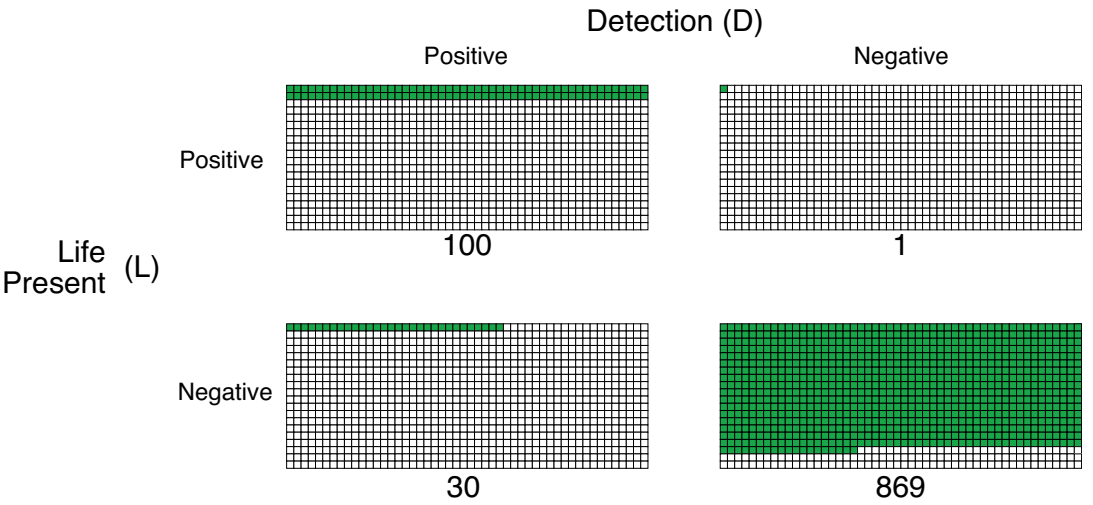


Figure 4-16: The performance of our probes

The probe correctly reported that it found life (that is, a true positive) 100 times out 1,000. In other words, of the 101 planets with life, the probe missed life signs (a false negative) only once.

Out of the 899 empty planets, the probe correctly reported there was no life (a true negative) 869 times. Finally, it incorrectly reported finding life on a barren planet (a false positive) 30 times. All told, these aren't bad numbers, since they're skewed in favor of protecting life.

Using the letter D for "detected life" (the probe's result), and the letter L for "life is present" (the reality on the ground), we can summarize these results in the confusion matrix of Figure 4-17. For the marginal probabilities, we write not-D for the probe result "not detected-life" (that is, the probe said there was no life), and not-L for "not life-is-present" (that is, there really is no life on the planet).

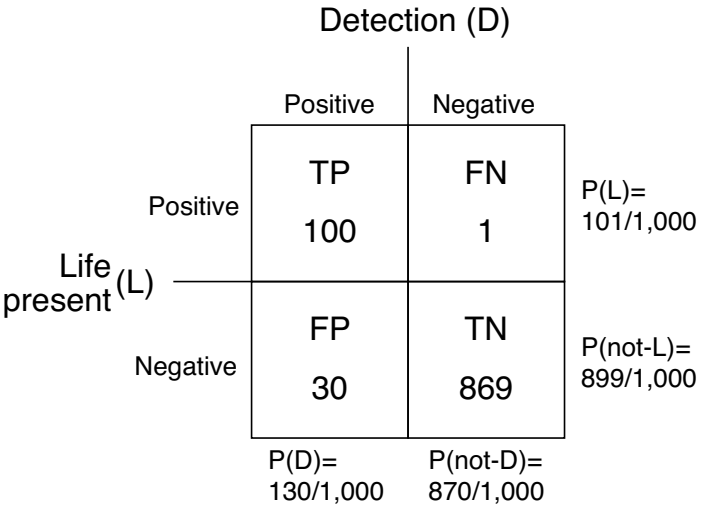


Figure 4-17: The confusion matrix that summarizes Figure 4-16, demonstrating the performance of our life-detecting probe. The four marginal probabilities are shown in the right and bottom margins.

Figure 4-18 gathers up the four marginal probabilities, plus two conditional probabilities that we'll be using.

P(D) = 130/1,000	P(not-D) = 870/1,000
P(L) = 101/1,000	P(not-L) = 899/1,000
P(D|L) = 100/101	P(not-D|L) = 1/101

Figure 4-18: Summaries of the four marginal probabilities and two conditional probabilities that we'll be using based on the data in Figure 4-17

To find P(D|L), or the probability that the probe reported life given that there really *is* life, we found the number of times the probe found life (100) and divided it by the number of planets where life was found (101). That is, we found TP / (TP + FN), which we saw in Chapter 3 is called the *recall*. The value of 100/101 is about 0.99.

To find P(not-D|L), we carried out the calculation the other way. It missed finding life once out of 101 planets. We found FN / (TP + FN), which we saw in Chapter 3 is called the *false negative rate*. It comes to 1/101, or about 0.01. (For more insight into the probe's behavior, we can also use the definitions in Chapter 3 to find the probe's accuracy as 969/1000, which is 0.969, and its precision as 100/130, which is about 0.77.)

Now we can answer our original question. The probability that there actually is life, given that our probe says there isn't, is P(L|not-D). Using Bayes' Rule, we plug in the numbers from either the previous paragraph or Figure 4-18, giving us Figure 4-19.

$$P(L \mid \text{not-D}) = \frac{P(\text{not-D} \mid L) \times P(L)}{P(\text{not-D})}$$

$$= \frac{\frac{1}{101} \times \frac{101}{1{,}000}}{\frac{870}{1{,}000}}$$

$$= \frac{\frac{1}{1{,}000}}{\frac{870}{1{,}000}} = \frac{1}{870} \approx 0.001$$

Figure 4-19: Working out the probability that a planet has life, given that the probe reported that no life was detected

This is reassuring. The probability that there's life on that planet, given that our probe said there wasn't, is about 1 in 1,000. That's a lot of confidence, but if want to be even more sure, we can send down more probes. We'll see later how each successive probe can increase our confidence about whether there really is or isn't any life down there.

Let's switch things up, and suppose that the probe instead came back with a positive report, telling us that it *did* detect life. That would be a financial loss for us, so we'd like to be sure. How confident can we be that there really is life on that planet? To find that, we just use Bayes' Rule again, but this time we work out P(L|D), the probability of life given that the probe detected life. Let's work though the numbers in Figure 4-20.

$$P(L \mid D) = \frac{P(D \mid L) \times P(L)}{P(D)}$$

$$= \frac{\frac{100}{101} \times \frac{101}{1{,}000}}{\frac{130}{1{,}000}}$$

$$= \frac{\frac{100}{1{,}000}}{\frac{130}{1{,}000}} = \frac{100}{130} \approx 0.77$$

Figure 4-20: Working out the probability that a planet has life, given that the probe reported it found signs of life

Wow. If the probe says it found life, we can be about 77 percent confident that there really is life, just from this one probe. This is nowhere near the level of confidence we got from the negative report, but that's because the probe was designed to have a greater chance of reporting a false positive than a false negative. Since we always want to err on the side of protecting life, these are good results overall.

As we mentioned earlier, we can send more probes to increase our confidence in either result, but we'll never get to absolute certainty either way. At some point, either on probe 1 or probe 10 or probe 10,000, we'll need to make a judgment call about whether to mine the planet or not.

Let's now see how sending more probes can help us increase our confidence.

Repeating Bayes' Rule

In the preceding sections we saw how to use Bayes' Rule to answer a question of the form *"What is the probability that (something1) is true, given that (something2) is true?"* We approached this question as a one-time event, plugging in what we know about the system and getting back a probability.

One event, or measurement, is not much to go on. Let's return to our coin game with the two coins. Recall that one coin is fair, and one is rigged to come up heads more than half the time. We chose one of the two coins at random, flipped it, found that it came up heads, and we produced a probability that we had the fair coin. And that was the end of it.

But we can keep going. In this section let's put Bayes' Rule in the heart of a loop, where each new piece of data gives us a new posterior that we then use as the prior for the next observation. Over time, if the data is consistent, the prior should home in on the underlying probabilities we're looking for.

Here's the basic intuition, before we get into the details. We usually know the likelihood, P(B|A), and the evidence, P(B), from our experimental setup, so those are settled. But we rarely know the prior, P(A). We need a value for this, so we think about the problem and take our best guess. Since this completes all the values needed by Bayes' Rule, we can plug them in and get the posterior, P(A|B).

Now comes the interesting step. The posterior tells us the probability of A given that event B happened, but *we know event B happened.* Whether it's a coin coming up heads, or a probe finding life on a planet, we know that B happened since we chose to compute P(A|B), rather than, say, the probability of A given that B did not happen. Since we know that B *has* happened, P(A|B) is just P(A).

Let's express this with another example to lock it down. Suppose event B is "the day is warm" and event A is "people are wearing sandals." Suppose that it's warm today. Then the probability that "people are wearing sandals, given that it's warm" is equal to just "people are wearing sandals," because we have already observed that it's warm.

In other words, the posterior, P(A|B), becomes P(A), which is our prior! That's the key insight. When we know that B has happened, then the output

of Bayes' Rule gives us a new estimate of P(A). So, Bayes' Rule gives us a way to change and improve our expectations, or beliefs, about the system based on what experiments tell us.

To summarize, we guess at P(A). Then we run an experiment. Critically, we then choose to compute either P(A|B) or P(A|not-B) based on whether we saw event B or not, as we saw in Figures 4-19 and 4-20. This choice of which version of Bayes' Rule to evaluate is the magic that makes the whole loop work. Our choice of P(A|B) or P(A|not-B), which is determined by which outcome we actually observed, adds new information into our process. That new information helps us refine our understanding of the system we're learning about. So, having made this choice, we plug the numbers into the appropriate form of Bayes' Rule, and produce a posterior. That becomes the new prior. With this new P(A), we run another experiment, using Bayes' Rule again, compute either P(A|B) or P(A|not-B) based on whether B happened or not, and update our expectations again by using that result as our new P(A), or prior, for the next experiment, and so on. Over time, our belief, or expectation, about the probability of A, or P(A), gets gradually refined from a guess to an experimentally supported value.

Let's package up this description into a loop, starting with a guess for the prior P(A) and then refining by running more experiments.

The Posterior-Prior Loop

In Figure 4-15 we gave names to the terms of Bayes' Rule. These aren't the only names that are used. We also refer to the events in Bayes' Rule (which we've been calling A and B) in terms of a hypothesis and an observation (sometimes abbreviated Hyp and Obs). Our *hypothesis* states something whose truth we want to discover (for example, "we have the fair coin"). The *observation* is the experimental result (for example, "we got heads"). Figure 4-21 shows Bayes' Rule with these labels.

$$P(\text{Hypothesis} \mid \text{Observation}) = \frac{P(\text{Observation} \mid \text{Hypothesis}) \times P(\text{Hypothesis})}{P(\text{Observation})}$$

Figure 4-21: Writing Bayes' Rule with descriptive labels for A and B

In our coin-flipping example, our hypothesis is "We have selected the fair coin." We ran an experiment and got an observation, which was "The coin came up heads." We combine our prior P(Hyp) with the likelihood P(Obs|Hyp) of the observation given that hypothesis to get the joint probability of both the observation and hypothesis being true. We then scale that by the evidence P(Obs), or the probability that the observation could have come about by any means. The result is the posterior P(Hyp|Obs), which tells us the probability that our hypothesis is true, given the observation.

As promised, let's now wrap this in a loop. We compute the posterior, and then (because we know that the Observation has happened) we can use that as the prior when we repeat the experiment. The result is a new

posterior, which we can use as our prior the next time, and so on. Each time around the loop, our prior gets a little more accurate at describing the system, thanks to the inclusion of the results of each previous experiment.

A drawing of this loop is shown in Figure 4-22.

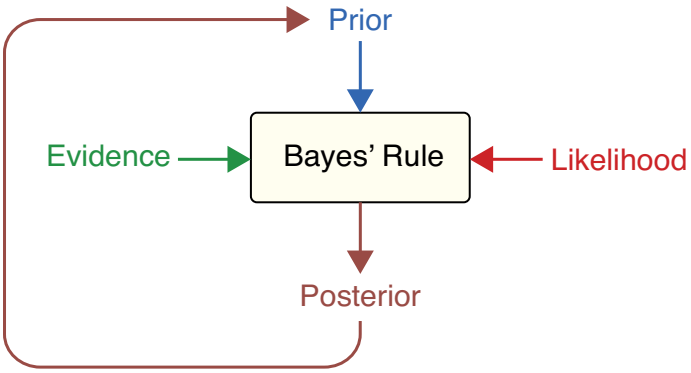


Figure 4-22: Each time we have a new observation, we combine the evidence, the likelihood of that observation, and the prior to compute a posterior. That posterior is then used as the prior when a new observation is evaluated.

To recap, we start with a prior. This comes from analysis, experience, data, an algorithm, or just guesswork. Then we run an experiment (or make an observation) and start the loop. We combine the evidence, the likelihood of that observation, and the prior to select one of the two forms of Bayes' Rule, with which we compute a posterior. This posterior becomes our new prior. Now, when another observation arrives, we enter the loop again, using our new prior.

The idea is that each time through the loop, our prior improves, from our initial guess toward a range of highly probable answers. The prior is improving because each time around the loop, the prior incorporates the latest observation, in addition to all the previous observations.

Let's see this loop in action using our coin-flipping example.

The Bayes Loop in Action

Recall our archaeologist friend and her two-coin problem. Let's generalize it so we can try out a number of variations and explore how to use the Bayes' Rule loop shown in Figure 4-22 to answer questions.

Rather than having a single bag with a fair coin and a rigged coin, let's suppose she found many such bags, where no two rigged coins had the same bias. Each bag is marked with the bias of its rigged coin (the bias is often written with the lowercase Greek θ [theta]).

She thinks that before players started a game, they'd agree on how biased they wanted the rigged coin to be. Then they'd pick the corresponding bag, and proceed as usual, picking out one of the two coins and then betting on which one had been picked.

Like them, we'll first select a bag, and then a coin from the bag. Then we'll determine the probability that we picked the fair coin. We can use the

repeated form of Bayes' Rule by flipping the coin many times, recording the heads and tails we get, and watching what Bayes' Rule does with the observation, or result, of each flip when producing the posterior.

Suppose we make 30 flips. Even with so little data, we might see unusual events. For instance, we might have the fair coin and still get 25 heads and 5 tails. It's very unlikely, but possible. It's more likely that we'd get those results from a rigged coin with a high bias. Let's see how Bayes' Rule helps us decide which coin we have, based on multiple flips.

Let's start by choosing the bag with a fair coin and a rigged coin with a bias of 0.2, which means we expect it to show 2 heads out of every 10 flips. Suppose we flip this coin 30 times, and only 20 percent (that is, 6) flips come up heads, so the other 24 flips come up tails. Do we have the fair coin or the rigged one? Since we'd expect 15 heads out of 30 flips from the fair coin, and 6 heads out of 30 flips from the rigged coin, getting 6 heads back seems like a good case for us having the rigged coin.

Figure 4-23 shows the result of Bayes' Rule after each flip. As before, the probability that we have the fair coin is shown in flax (or beige), while the probability of the rigged coin is shown in red. The two probabilities always add up to 1.

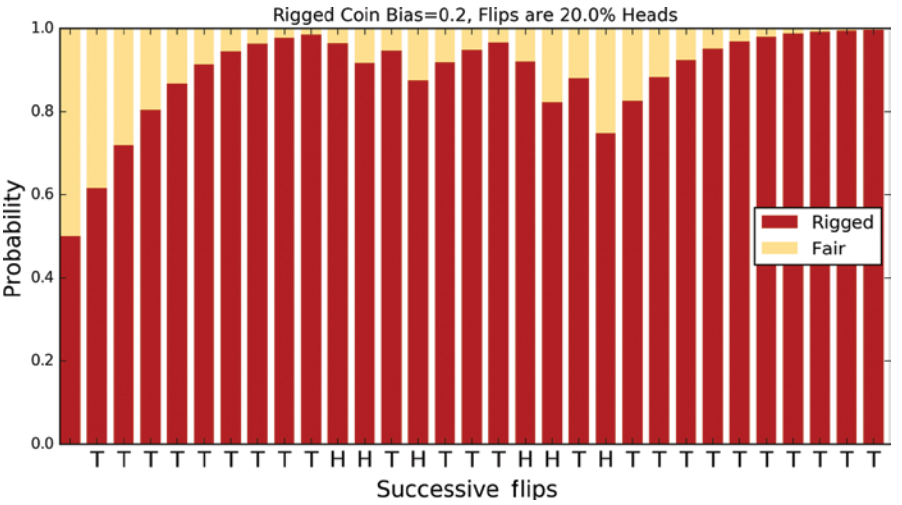


Figure 4-23: The probability that we have the fair coin is shown in flax (beige) through successive flips. The letter below each bar is the observation that produced that bar.

To understand what this chart is telling us, look first at the letters along the bottom. These are either "H" or "T," telling us the result of that flip. In this case, we have 24 tails, with 6 heads appearing here and there. Now consider the bars, starting at the left. The leftmost column shows which coin we think we have before we've flipped the coin at all, so the probabilities are both 0.5. After all, the chances are equal that we picked either coin, and we haven't flipped it to gain any data. The bar to the right of that shows the result of the posterior of Bayes' Rule after observing tails (T) on the first flip. Since the chance of getting tails from the fair coin is 0.5, but the chance

of tails from the rigged coin is 0.8, getting tails suggests that it's more likely we have the rigged coin. Continuing to the right, about 80 percent of the flips are tails. That's what we'd expect from the rigged coin, so its probability quickly approaches 1. Note that the probability of our having the rigged coin dips about 2/3 of the way through the run when we get a bunch of heads close to one another, but then it goes back up with each new tail.

The height of the flax, or beige, block in each bar is the value of P(F), the probability that we selected the fair coin. After each flip, we use Bayes' Rule to compute either P(F|H) or P(F|T) as appropriate. That becomes the new value of P(F), or our belief that we have the fair coin. We use that to compute the new posterior after the next flip. As we mentioned earlier, this choice is the key step that makes the whole loop work. After each experiment, we choose the version of Bayes' Rule that returns P(F|H) or F(F|T) depending on what we observe. That choice is what enables us to use the posterior as a new prior, because it reflects what actually happened.

Toward the end, the probability that we have the fair coin is nearly 0. It never gets to exactly 0, because we can never be absolutely sure that this isn't a fair coin with a wildly unusual flip pattern, so that option is always has at least a shred of probability.

In this example, the data we got back pretty clearly revealed that we had the rigged coin. Let's keep this coin and do another run. Suppose we get even fewer heads on the next run, perhaps only three in total, making the case for the rigged coin even stronger. Running this through the loop produces the results in Figure 4-24.

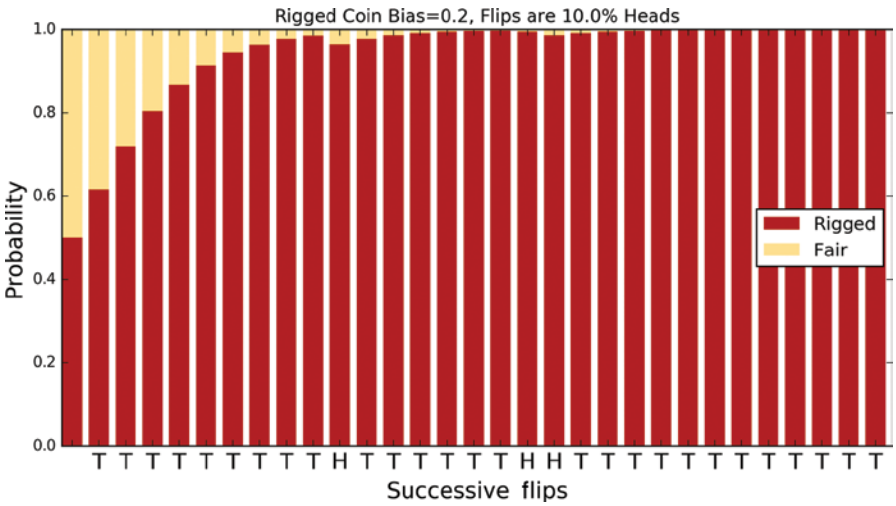


Figure 4-24: We use the same coin with a bias of 0.2, but this time we happened to get only three heads in our 30 flips.

We get to about 90 percent confidence that we've got the rigged coin after just four flips. After 30 flips, the probability of a fair coin is again almost, but never quite, 0.

Suppose we do another run of 30 flips, and this time we happen to get 24 heads. This doesn't match either coin very well. We'd expect the fair

coin to give us 15 heads, but we'd expect only 6 heads from the rigged coin. Given just these two choices, the fair coin seems more likely. Figure 4-25 shows our results from Bayes' Rule.

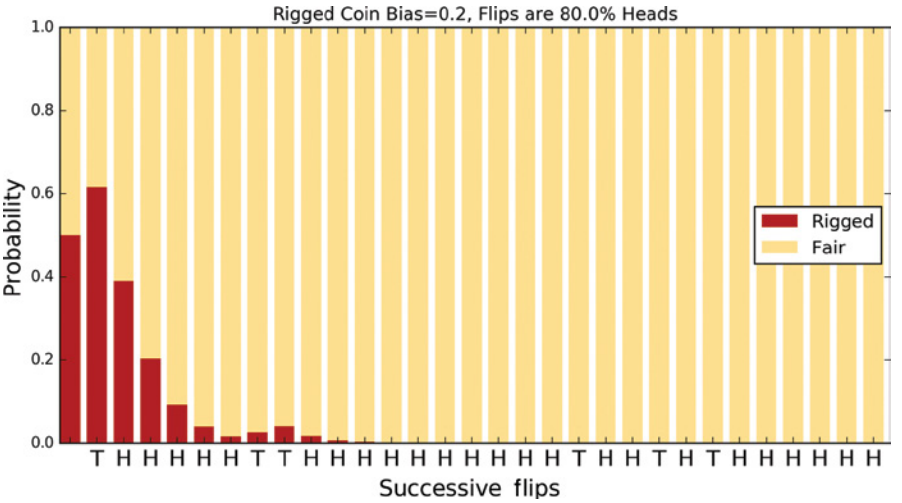


Figure 4-25: We use the same coin with a bias of 0.2 but this time we happened to get 24 heads in our 30 flips.

Even though the fair coin should only come up heads about half the time, the rigged coin would come up heads only 20 percent of the time. All of those heads are unlikely from either coin, but they're a lot *more* unlikely from the rigged coin, bolstering our confidence that we've got the fair coin.

We've seen three different flipping results for this coin, from a pattern of almost all tails to a pattern of almost all heads. Let's generalize these results by flipping 10 coins with different biases. Let's create 10 different flip patterns for each coin, with different ratios of heads to tails. We can use Bayes' Rule on each pattern for each coin, creating 100 scenarios. The results are in Figure 4-26, where each cell is a little bar chart like those in Figure 4-23 through Figure 4-25.

Let's start in the bottom left corner. At this location our value on the horizontal axis (labeled "Rigged coin bias") is around 0.05. That means we'd expect this coin to come up heads about 1 time in 20. Our value on the vertical axis (labeled "Flip sequence bias") is also about 0.05. This means we're going to create an artificial sequence of observations, like we did earlier, where there's a 1 in 20 chance that each observation will be heads. In this case, the number of heads in the pattern of 30 observations we created for this cell in the grid matches the number of heads we'd expect from the rigged coin, so our confidence that the coin is rigged (in red) grows quickly.

Let's move up three cells. Since we haven't moved horizontally, our horizontal axis value is still 0.05, so we're flipping a coin that should come up heads 1 time in 20. But now the vertical axis is about 0.35, so we're looking at a pattern where heads are significantly more frequent. With all of those

heads, it seems more likely that we're getting an unusual series of flips from a fair coin, than a *very* unusual series of flips from the rigged coin. Our confidence that the coin is fair grows stronger as the number of flips increases.

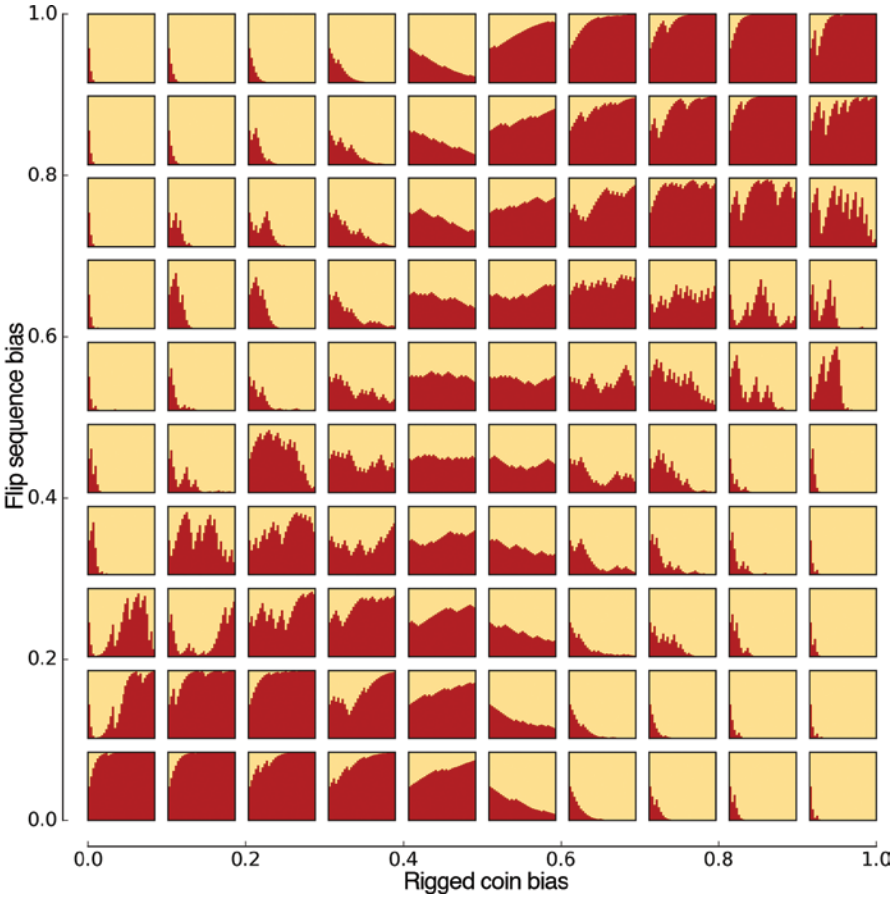


Figure 4-26: Flipping a coin 30 times, and using repeated Bayes' Rule applications to decide which coin we're flipping. Each square reports our results for just one run of 30 random flips. Each row uses the same sequence of heads and tails.

Each cell can be understood in the same way. We invent a pattern of 30 heads and tails, where the relative proportion of heads is given by the vertical location, and we ask whether that pattern is more likely to come from a fair coin, or a coin with a probability of heads given by its horizontal location.

In the middle of the grid, where both values are close to 0.5, it's almost impossible to tell. The rigged coin comes up heads almost as often as the fair coin, and the patterns of heads and tails are about evenly split, so we could be flipping either coin. The probabilities for both stick to around 0.5. But as we make patterns with fewer heads (the lower part of the figure) or more heads (the upper part), we can say how well that pattern matches

a rigged coin with a low probability of coming up heads (the left side) or a high probability (the right side).

A series of 30 flips is revealing, but we can still get unusual surprises (like 25 heads from a fair coin). If we increase the number of flips to 1,000 in each plot, as in Figure 4-27, such unusual sequences become rarer, and the patterns become clearer.

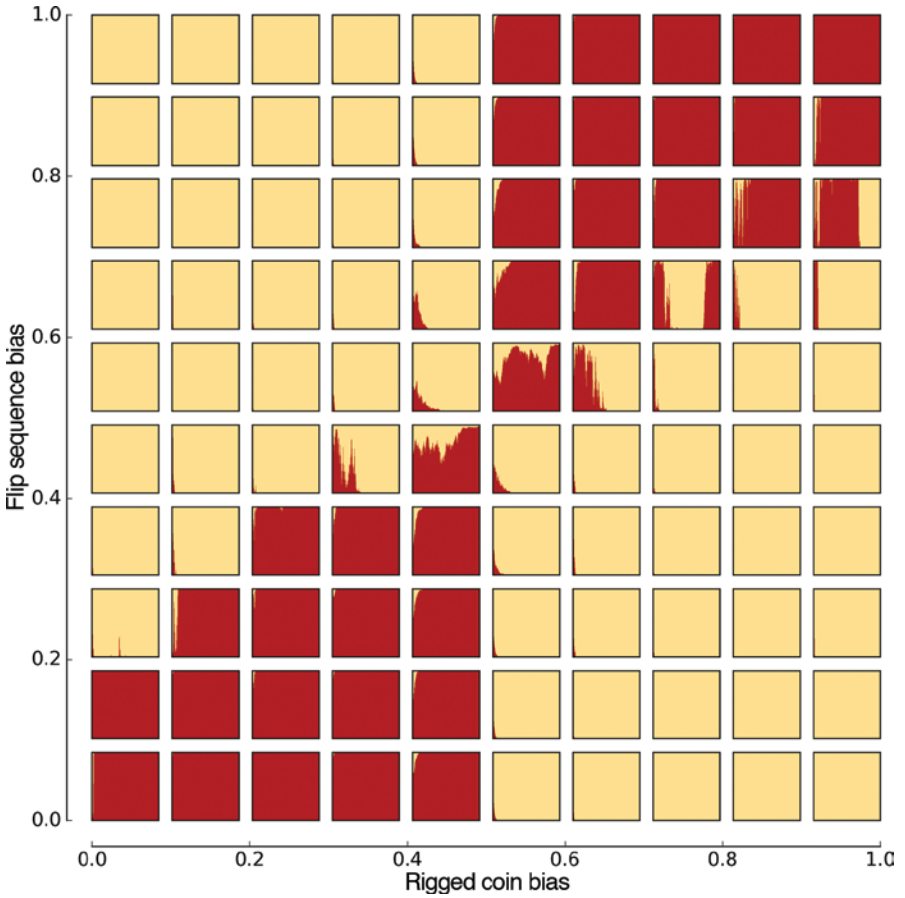


Figure 4-27: This chart has the same setup as Figure 4-26, but now we flip each coin 1,000 times.

In the lower-left and upper right, the pattern of flips more closely matches the bias of the rigged coin than the fair coin, and Bayes' Rule pushes the prior of "we selected the fair coin" toward 0. In the upper left and lower right, the flips more closely match the fair coin, and the prior moves toward 1.

The general lesson of Figure 4-26 and Figure 4-27 is that the more observations we make, the more certain we become that our hypothesis is either true or false. Each observation either increases or decreases our confidence. When the observations match up with our prior ("we have the fair

coin"), our confidence in that prior grows. When the observations contradict that prior, our confidence decreases, and since there's only one other alternative in this scenario ("we have the rigged coin"), that becomes more probable. Even when we have only a few observations, we can often gain a great deal of confidence early on.

Multiple Hypotheses

We've seen how to use Bayes' Rule to improve a hypothesis by combining it with an observation, perhaps repeatedly. But there's nothing limiting us to testing just a single hypothesis. We've been making multiple hypotheses all along, actually. In just the last section, we explicitly saw the two hypotheses "this coin is fair" and "this coin is rigged" being updated simultaneously. Since we knew the two probabilities had to add up to 1, knowing either one of them revealed the other, so we only had to keep track of one of them.

But we could explicitly calculate both probabilities if we wanted. We'd just use two copies of Bayes' Rule. Suppose that a flip comes up heads. Then we can independently compute the conditional probability that we have a fair coin, P(F|H), and the conditional probability that we have a rigged coin, P(R|H). This is shown in Figure 4-28.

$$\text{(a)} \quad P(F \mid H) = \frac{P(H \mid F)\ P(F)}{P(H)}$$

$$\text{(b)} \quad P(R \mid H) = \frac{P(H \mid R)\ P(R)}{P(H)}$$

$$\text{(c)} \quad P(H) = P(H \mid F)\ P(F) + P(H \mid R)\ P(R)$$

Figure 4-28: Calculating probabilities for two hypotheses. Line (c) explicitly shows how to compute P(H) in both cases.

From Figure 4-28, we can see that the probabilities of the fair coin and the rigged coin are just their share of the two ways the coin can come up heads. If we want to track multiple hypotheses at once, we can use multiple copies of Bayes' Rule in this way, updating them all after each new observation.

We can use this ability to help out our archaeologist friend again. She's just found a chest with pieces for a new game, and again, the game uses bags to hold pairs of coins. Like before, the rigged coins in the bags have different biases. Because an extremely biased coin (that is, one that comes

up heads much more often than tails, or vice versa) would be easier to spot, she thinks maybe there were levels of players, from newcomers to old hands. New players would play with coins that were highly biased, but as players became more skilled, they'd switch to rigged coins whose bias was closer and closer to 0.5. Those would be harder to detect, leading to longer games with riskier and more complicated betting.

Since our friend wants to know all about her discovery, she emptied out all the coins into one big box, and has asked us to find the bias of each coin. For the moment, let's suppose that there are only five possible bias values—0, 0.25, 0.5, 0.75, and 1 (recall that a bias of 0.5 corresponds to a fair coin)—so we'll create five hypotheses. We'll number them 0 through 4, corresponding to the different bias values. Hypothesis 0 states, "This is the coin with bias 0," Hypothesis 1 states, "This is the coin with bias 0.25," and so on, up to Hypothesis 4, which states, "This is the coin with bias 1."

Now we'll pick up a coin at random from the box, flip it repeatedly, and try to determine which of these hypotheses is most likely. To get started, we need to cook up a prior for each hypothesis. Remember that this is going to get updated every time through the loop, so we only need a good starting guess. Since we don't know anything about the coin we've selected, let's say the chance of having picked each one is the same, so all five prior values have a 1 in 5 chance of being right, or $1 / 5 = 0.2$.

Note that we could get fancier if we wanted. Each pair of coins has one that's fair and one that's rigged. Say we have 16 coins. If this is the case, then we have 8 fair coins, and 8 rigged coins (2 for each allowed value of the bias). The chance of picking a fair coin is then $8 / 16 = 0.5$, whereas the chance of picking each rigged coin is $2 / 16 = 0.125$. This is probably a better prior, since it uses more of the information we already know. Starting out with a better prior means that our loop will home in on the high-probability solutions more quickly. But one of the beauties of the Bayesian approach is that we can start with almost any prior that's even roughly close, and, ultimately, we get the same results. For simplicity, let's use the first prior, where every hypothesis has a value of 0.2.

The only thing left for us to specify is the likelihood for each coin. But we've already got those, because they *are* the bias. That is, if the coin has a bias of 0.2, then its likelihood of coming up heads is 0.2. That means the likelihood of tails is $1 - 0.2 = 0.8$.

Hypothesis 0, which says, "We have the coin with bias 0.0," has a likelihood of getting heads of 0, and tails of 1. Hypothesis 1, which says, "We have the coin with bias 0.2," has a likelihood of getting heads of 0.2 (or 20 percent), and a likelihood of tails of 0.8 (or 80 percent). Our likelihoods are plotted in Figure 4-29.

Since the coins themselves don't change as we flip them and gather observations, the likelihoods don't change, either. We'll reuse these same likelihoods over and over again, each time we evaluate Bayes' Rule after getting a new observation.

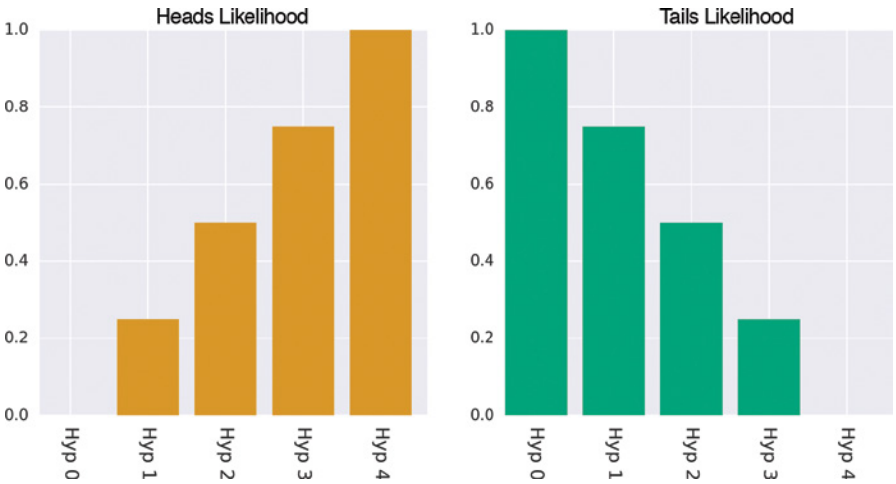


Figure 4-29: The likelihoods for getting heads or tails for each of our five hypotheses

Our goal is to flip our coin over and over and watch what happens to our five priors as they evolve. To show what's happening at each flip, let's draw the five prior values in red, and the five posterior values in blue. In Figure 4-30 we show the result of our first flip, which we'll suppose came up heads. The five red bars, representing the prior for each of our five hypotheses, are all at 0.2. Since the coin came up heads, we multiply each prior by its corresponding likelihood from the left side of Figure 4-29, which gives us our likelihoods. After dividing by the sum of all five probabilities for getting heads, we get the posterior, or the output of Bayes' Rule, shown in blue.

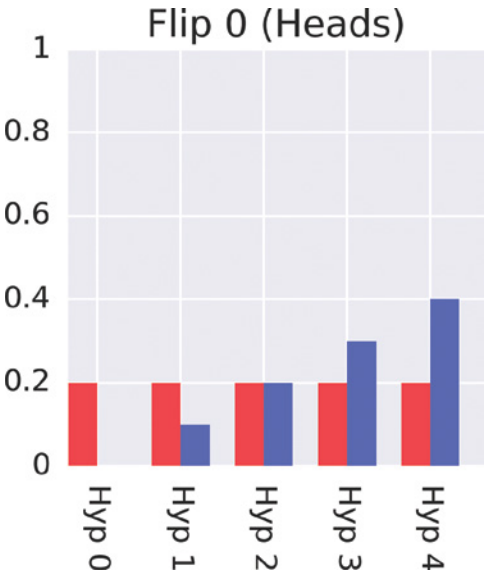


Figure 4-30: We're testing five hypotheses, which assert that our coin has a bias of 0, 0.25, 0.5, 0.75, and 1.0. We start with a prior of 0.2 (in red) for each hypothesis. After one flip of the coin, which came up heads, we compute the posterior (in blue).

In Figure 4-30, each pair of bars shows the prior and posterior value for a single hypothesis. Right away we've ruled out Hypothesis 0, because that says that the coin never comes up heads, and we just got heads. The hypothesis that says this coin always comes up heads is the strongest so far, because we just saw heads.

Let's now make a series of flips. We'll use a list of 100 flips that contain 30 percent heads. That is, the flips correspond to the results we'd get from a coin with a bias of 0.3. None of our five hypotheses matches this exactly, but Hypothesis 1 comes the closest, representing a coin with bias 0.25. Let's see how Bayes' Rule performs. Figure 4-31 shows the results for the first 10 flips in the top two rows, and then takes bigger jumps in the bottom row.

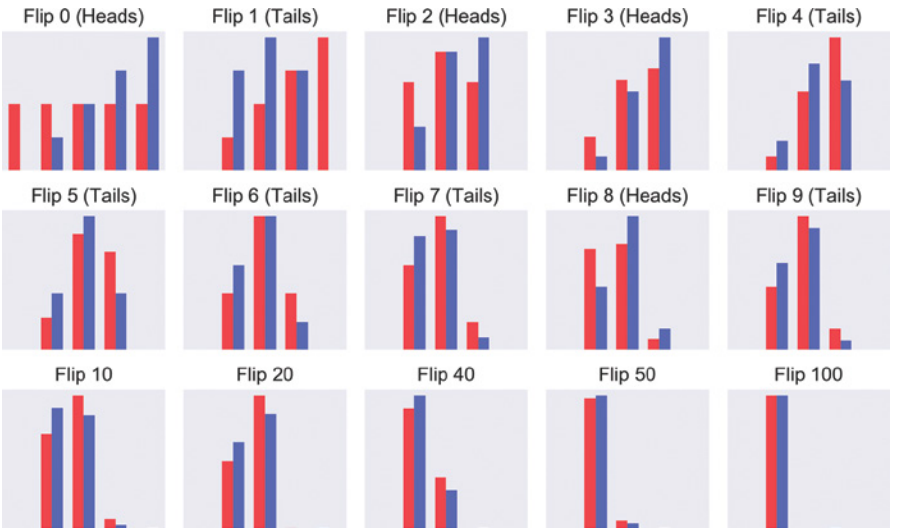


Figure 4-31: The evolution of our priors (red) and posteriors (blue) in response to a series of flips generated by a coin with bias 0.3. The number of the flip that was just evaluated is shown at the top of each graph.

As we can see in the first two lines, the posterior computed after each flip (in blue) becomes the prior for the next flip (in red). We can also see that after the first flip (heads), Hypothesis 0 at the far left dropped to a likelihood of 0, because that hypothesis was that we had a coin that would never come up heads. Then on the second flip, which happened to be tails, Hypothesis 4 went to 0, because that corresponded to a coin that always came up heads. That leaves just three hypotheses.

We can see how the three remaining options trade off the probabilities with each flip. As more flips come along, the number of heads comes closer to 30 percent, and Hypothesis 1 dominates. When we've reached 100 flips, the system has pretty much decided that Hypothesis 1 is the best one available, meaning that our coin is more likely to have a bias of 0.25 than any of the other choices.

If we can test 5 hypotheses, we can test 500. Figure 4-32 shows 500 hypotheses, each corresponding to a bias equally spaced from 0 to 1. We've

added a fourth row showing many more flips. We've eliminated the vertical bars in these charts so we can more clearly see the values of all 500 hypotheses.

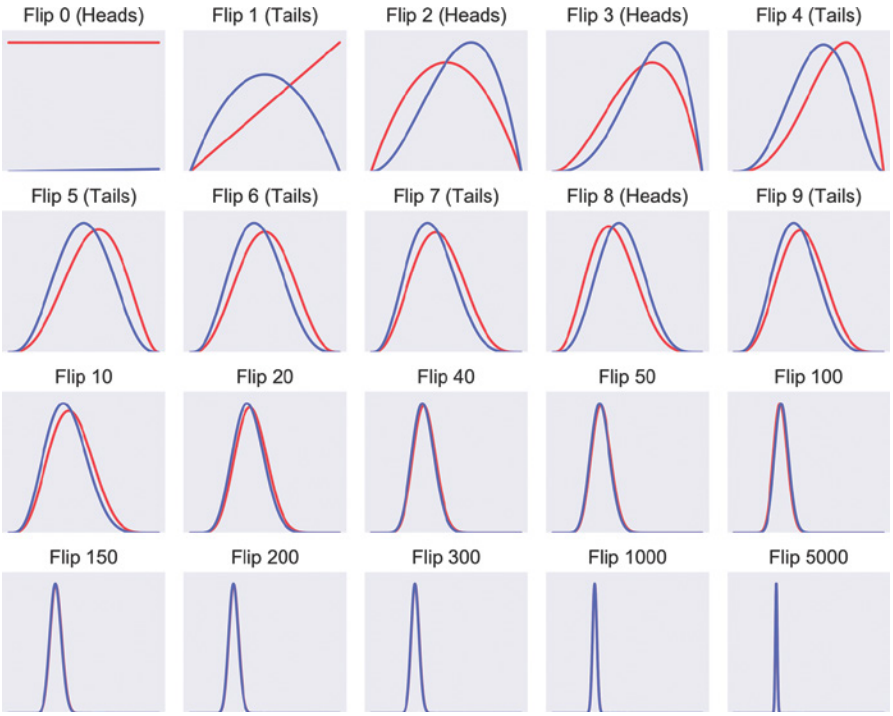


Figure 4-32: The same situation as Figure 4-31, but now we're evaluating 500 simultaneous hypotheses, each based on a coin with a slightly different bias

In this figure (as in the figures to come), we're reusing the identical series of flip results that we used in Figure 4-31. And as we'd expect, the winning hypothesis is the one predicting a bias of about 0.3. But another interesting thing is happening here: the posteriors are taking on the form of a Gaussian. Recall from Chapter 2 that a Gaussian curve is the famous bell curve that's flat except for a symmetrical bump. This is a typical feature for the priors that evolve from the mathematics of Bayes' Rule. It's just another of the many places in statistics and probability where a Gaussian curve often emerges from the data.

Notice that, as we said at the start of the chapter, Bayesian reasoning doesn't zero in on one correct answer. Rather, it gradually assigns more probability to a smaller range of answers. The thinking is that any value in that range has its own probability of being the answer we seek.

If Bayes' Rule seems to evolve so that the prior takes on the shape of a Gaussian, what would be the result if we *started* with priors that formed a Gaussian? Let's do just that, but let's make it even harder for the system by putting the mean of the prior's bump (that is, its center) out at around 0.8. That says that our belief is that the coin we're testing is most likely to have a bias of 0.8. This is far away from the value of 0.3 that we baked into our

sequence of heads and tails. The probability at 0.3 that describes our coin starts out with a value of merely 0.004, so we're asserting, through our prior, that the chance of this coin having a bias of 0.3 is 0.4 percent, or 4 out of 1,000. How does the system respond to a prior that is so wrong, that the correct answer has only this very slim chance?

Figure 4-33 shows the result.

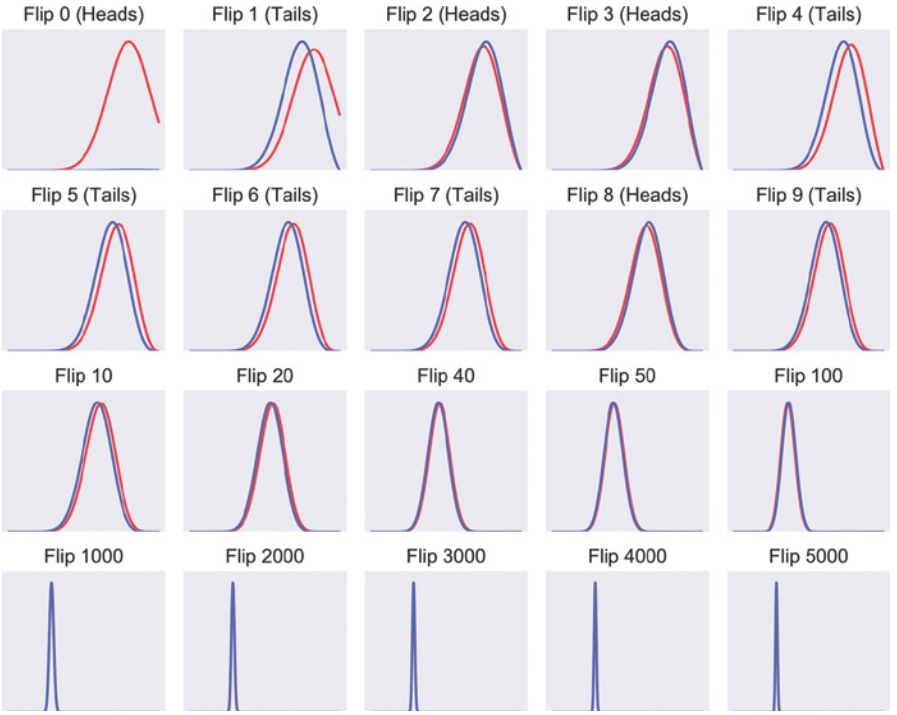


Figure 4-33: This figure has the same setup as Figure 4-32, only now we're starting with a set of priors formed in a Gaussian bump centered at 0.8.

Nice. Even with our poorly chosen prior, the system homed in on the proper bias of 0.3. It took a while, but it got there.

In Figure 4-34 we show the priors of Figure 4-33 at 10 steps along the way in evaluating the first 3,000 flips, laid on top of one another rather than in sequence.

Notice that we start out with a broad prior with a mean of 0.8, but as we flip more and gather more observations, the prior's mean moves toward 0.3. The width of the bump also narrows, telling us the system is deciding that bias values far from the mean are less likely. The number of flips for each curve were chosen by hand so that the curves are spaced out roughly equally. Notice that as the system grows confident, by producing a narrow prior, the curve changes more slowly. In other words, the more certain the results, the more observations we need to make a big change to the posterior.

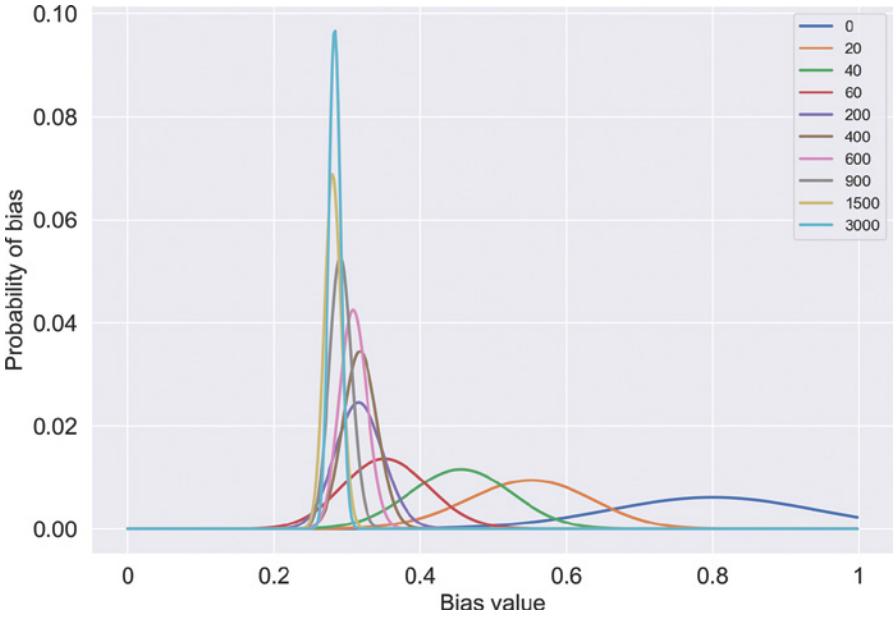


Figure 4-34: Some snapshots of the posteriors from the first 3,000 flips from Figure 4-33, overlaid on one another. The different colors show how many flips have elapsed, as given by the legend in the upper right. We can see the system giving more and more weight to the priors near 0.3, while reducing the probabilities elsewhere. The heights change in order to keep the area under each curve at 1.0.

We won't get into the details, but with some math, we can carry our increasing number of possible biases (and thus the number of hypotheses) to its logical extreme, replacing our lists of values with continuous curves, like those suggested by Figure 4-34. The advantage is that we can then get as precise as we like, finding a bias for any value rather than just the closest one in a list.

Summary

There are two broad camps in the field of probability: the frequentists and the Bayesians. The frequentist approach imagines that anything we choose to measure has an accurate, or true value. Each measurement is therefore only an approximation of that value. The Bayesian approach says that there is no single true value, only a range of possible values, each with its own probability. Each measurement is an accurate measure of something, but perhaps not what we want to measure.

We spent most of this chapter working with the Bayesian approach. Bayesian probability is popular in deep learning, because it's well suited to the nature of the kinds of problems we face and the kinds of questions we want to answer. The language of Bayesian probability is found in many of the papers, books, and documents of deep learning systems. At its core, it

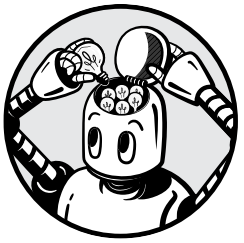
presents us with a set of tools for describing a measurement, not by looking for a single true number, but by finding a range of possible values for that measurement, each with its own probability.

For example, if a deep learning system is helping someone write a text message by offering shortcuts for the next word, it usually shows several high-probability guesses, rather than a single best next word.

In the next chapter, we'll look at some of the properties of curves and surfaces, which we'll use to understand the types of errors our learning systems can make (and later, how to correct those errors).

5

CURVES AND SURFACES



In machine learning, we frequently work with various kinds of curves and surfaces. Two of the most important properties of these objects are called the *derivative* and the *gradient*. They describe the shape of a curve or surface, and thus which directions to move in order to climb uphill or slide downhill. These ideas are at the heart of how deep systems learn. Knowing about the derivative and gradient is key to understanding backpropagation (the topic of Chapter 14), and thus knowing how to build and train successful networks.

As usual, we'll skip the equations, and instead focus on building intuition for what these two terms describe. You can find mathematical depth and rigor on everything we touch on here in most books on modern multivariable calculus and in more approachable form on many online websites (Apostol 1991; Berkey 1992; 3Blue 2020).

The Nature of Functions

As just mentioned, in machine learning, we often deal with various kinds of curves. Most often, these are plots of mathematical *functions*. We usually think of functions in terms of an input and an output. When we're dealing with a curve in two dimensions (2D), the input is expressed by selecting a location on the horizontal axis of a graph. The output is the value of the curve directly above that point. In this scenario, we provide one number as input, and get back one number as output.

When we have two inputs, we move into the world of three dimensions. Here, our function is a surface, like a sheet fluttering in the wind. Our input is a point on the ground below the sheet, and the output is the height of the sheet directly above that point. In this situation, we provide two numbers as input (to identify a point on the ground) and again get back a single output.

These ideas can be generalized, so functions can accept any number of input values, also called *arguments*, and can provide multiple output values, sometimes called *returned values*, or simply *returns*. We can think of a function as a machine that converts inputs to outputs: one or more numbers go in, and one or more numbers come out. As long as we don't deliberately introduce randomness, the system is *deterministic*: every time we give a particular function the same inputs, we get back the same outputs.

In this book we're going to use curves and surfaces in a few ways. One of the most important ways, and the focus of this chapter, is to determine how to move along them in order to get back larger or smaller outputs. The technique we use for that process requires that our functions satisfy a few conditions. We'll illustrate those conditions with curves, but the ideas extend to surfaces and more complex shapes as well.

We want our curves to be *continuous*, meaning that we can draw them with a single stroke of a pen or pencil, without ever lifting it from the page. We also want our curves to be *smooth*, so that they have no sharp corners (called *cusps*). Figure 5-1 shows a curve that has both of these forbidden features.

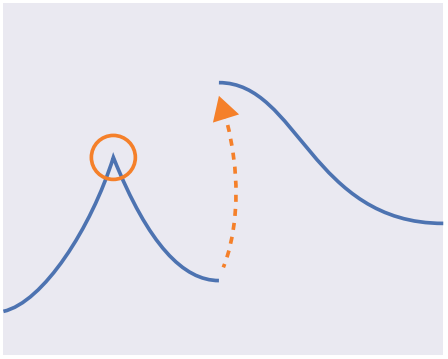


Figure 5-1: The circle encloses a cusp, and the dashed arrow shows a discontinuity, or jump.

We also want our curves to be *single-valued*. In 2D, this means that for each horizontal position on the page, if we draw a vertical line at that point, the line crosses the curve only once, so only a single value corresponds to that horizontal position. In other words, if we follow the curve with our eyes from left to right (or right to left), it never reverses direction on itself. A curve that violates this condition is shown in Figure 5-2.

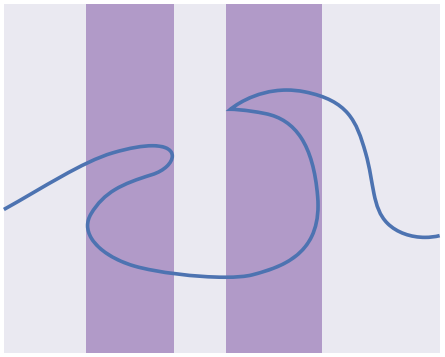


Figure 5-2: Inside the purple zones, the curve has multiple values in the vertical direction.

From now on, let's assume that all of our curves meet these rules (that is, they're smooth, continuous, and single-valued). This is a safe assumption because we're usually going to deliberately choose curves that have these properties.

The Derivative

One of the most important aspects of a curve is called its *derivative*. The derivative tells us a lot about the shape of a curve at any point along it. In this section, we look at some core ideas that lead us to the derivative.

Maximums and Minimums

A vital part of training in deep learning involves minimizing the system's error. We usually do this by imagining the error as a curve and then searching for the smallest value of that curve.

The more general problem is finding the smallest or largest value of a curve anywhere along its entire length, as illustrated in Figure 5-3. If these are the largest and smallest values for the whole curve (and not just the part we happen to be looking at), we call these points the *global minimum* and *global maximum*.

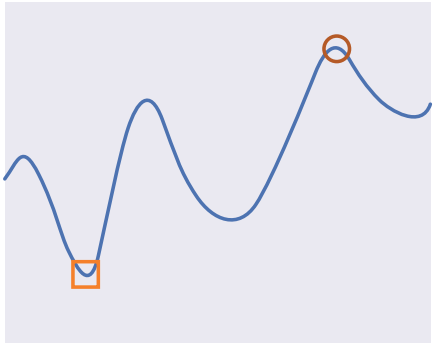


Figure 5-3: The global maximum (brown circle), and the global minimum (orange square) of a curve

Sometimes we want only these largest and smallest values, but other times we want to know *where* on the curve these points are located. Sometimes finding these values can be difficult. For example, if the curve goes on forever in both directions, how can we be sure we found the very smallest or largest values? Or if the curve repeats, as it does in Figure 5-4, which of the high (or low) points should we pick as the location of *the* global maximum or minimum?

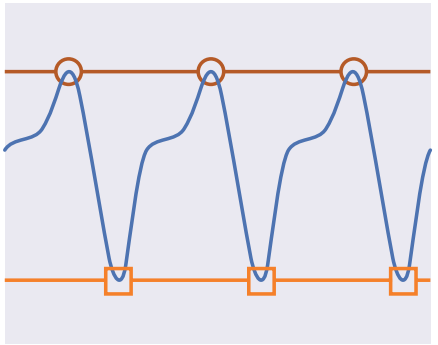


Figure 5-4: When a curve repeats forever, we can have infinitely many points that we could use as the location of the maximum (brown circles) or minimum (orange squares).

To get around these problems, let's think of maximum and minimum values in the *neighborhood* of a given point. To describe this, consider the following little thought experiment. Starting from some point on the curve, let's travel to the left until the curve changes direction. If the values start increasing as we move left, we continue as long as they increase, but as soon as they start to decrease, we stop. We follow the same logic if the values are decreasing as we move to the left, stopping when they start to increase. We do the same thought experiment again, starting at the same point, but this time, we move to the right. This gives us three interesting points: our starting point, and the two points where we stopped when moving left and right.

The smallest value out of these three points is the *local minimum* for our starting point, and the largest value of the three points is the *local maximum* for our starting point. Figure 5-5 shows the idea.

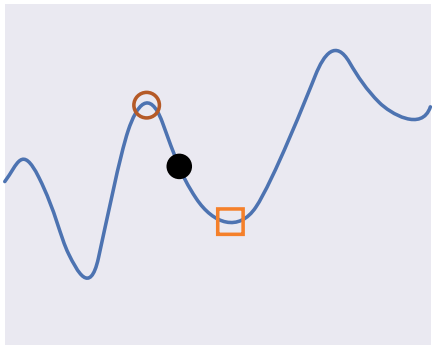


Figure 5-5: For the point in black, the brown circle and orange box respectively show that point's local maximum and minimum.

In Figure 5-5 we moved left until we got the point we then marked with a circle, and we moved right until we reached the point we marked with a square. The local maximum is given by the largest value of these three points, which, in this case, is the center of the brown circle. The local minimum is given by the smallest value of these three points, which, in this case, is the center of the orange square.

If the curve zooms off to positive or negative infinity, things get more complicated. In this book, we always assume that we can find a local minimum and maximum for any point on any curve we want.

Note that there is only one global maximum and only one global minimum for any given curve, but there can be many local maximums and minimums (sometimes called *maxima* and *minima*) for any given curve or surface, since they depend on the point we're considering. Figure 5-6 shows this idea visually.

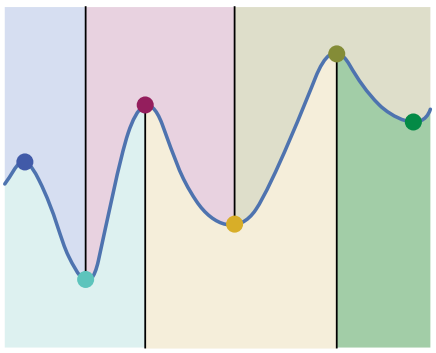


Figure 5-6: The influences of these local maximums and minimums are shown by their corresponding colored region.

Tangent Lines

The next step in our road to the derivative involves an idea called a *tangent line*. To illustrate the idea, we've marked up a two-dimensional curve in Figure 5-7.

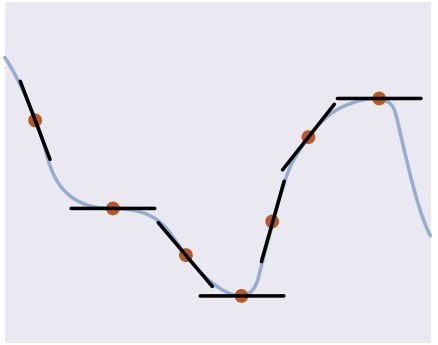


Figure 5-7: Some points on this curve are marked with dots. The tangent line at each of those points is drawn in black.

At each point on the curve, we can draw a line whose slope is given by the shape of the curve at that point. This is the tangent line. We can think of this as a line that just grazes the curve at that point. If we imagine ourselves traveling along the curve, the tangent line tells us where we're looking (as well as where we'd be looking if we had eyes in the back of our heads). Tangent lines are useful to us because they are horizontal at every local maximum and local minimum. One way to find a curve's maximum and minimum values is to find points on the curve where the tangent is horizontal (as Figure 5-7 shows, the tangent is also horizontal where the curve is horizontally flat, but we'll ignore that for now).

Here's one way to find the tangent line. Let's pick a point, which we'll call the *target point*. We can move an equal distance along the curve to the left and right of the target point, draw dots there, and draw a line connecting those two dots, as in Figure 5-8.

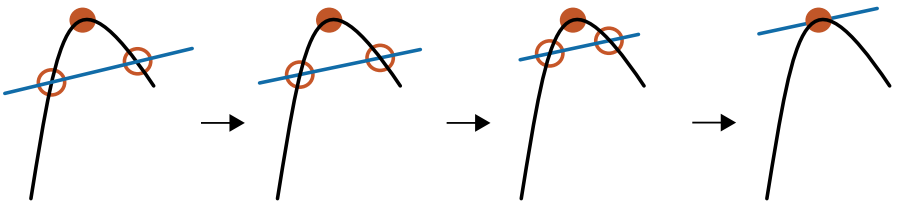


Figure 5-8: To find the tangent line at a given point, we can look at a pair of points at equal distances along the curve around that point and draw a line between them.

Now let's pull the two dots in toward the target point at the same speed, keeping each on the curve. At the very last instant before they merge, the line that passes through them is the tangent line. We say that this line is

tangent to the curve, meaning that it just touches it. It's the best straight line that describes the curve at that point. The ancient Greeks called the tangent line the *kissing line*.

We can measure the *slope* of the tangent line we constructed in Figure 5-8. The slope is just a single number that tells us the angle that the line forms with respect to a horizontal line. A horizontal line has a slope of 0. If we rotate the line counterclockwise, the value takes on increasingly positive values. If we rotate the line counterclockwise, the slope takes on increasingly negative values. When a line becomes exactly vertical, its slope is said to be infinite.

And now we've come to the derivative! It's just another name for the slope. Every point on a curve has its own derivative, because every point's tangent line has its own slope.

Figure 5-9 shows why we created the rules before that said our curves need to be continuous, smooth, and single-valued. Those rules guarantee that we can always find a tangent line, and thus a derivative, for every point on the curve.

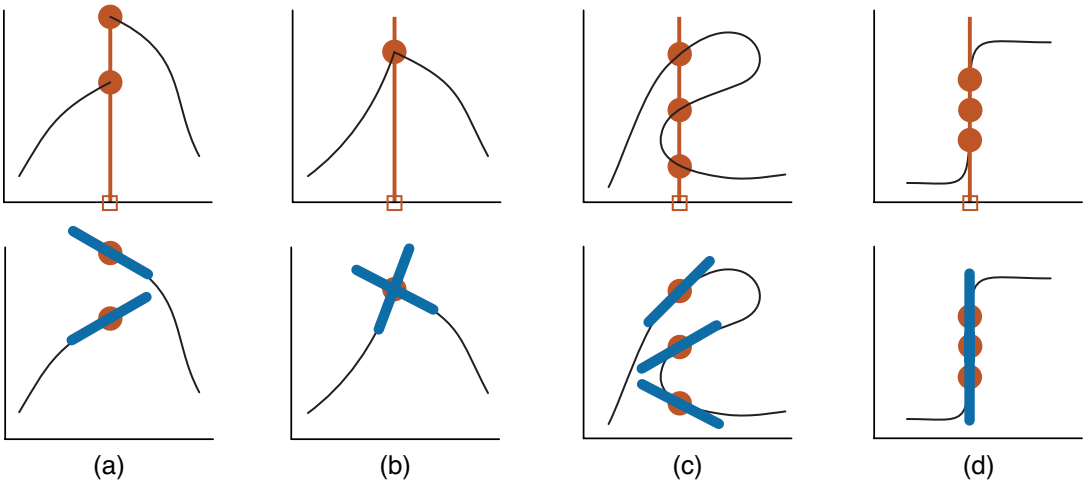


Figure 5-9: Top row: Curves with issues. Bottom row: Problems finding the derivative, shown in blue.

In Figure 5-9(a), the curve isn't continuous, so the two different curve ends have different derivatives above our chosen point (marked with a square). The problem is that we don't know which derivative to pick, so we avoid the question by not allowing discontinuities in the first place. In Figure 5-9(b), the curve isn't smooth, so the slopes are different as we arrive at the cusp from the left and right. Again, we don't know which one to pick, so we won't work with curves that have cusps. In Figure 5-9(c), the curve isn't single-valued. We have more than one point on the curve to choose from, each with its own derivative, and once again we don't know which one to pick. Figure 5-9(d) shows that if a curve ever becomes perfectly vertical, that also violates our single-value rule. Worse, the tangent line is perfectly vertical, which means it has an infinite slope. Handling infinite values can make

simple algorithms messy and complicated. So, we sidestep this problem, just like the others, and say that we won't use curves that can become vertical, and thus we never need to worry about infinite derivatives. By requiring our curves to be continuous, smooth, and single-valued, we can be sure that they can never create one of these situations.

We said before that a curve is a graphical version of a function: we provide an input value, conventionally along the horizontal X axis, and then look up (or down) to find the y value of the curve at that x. That y value is the output of the function, as shown in Figure 5-10.

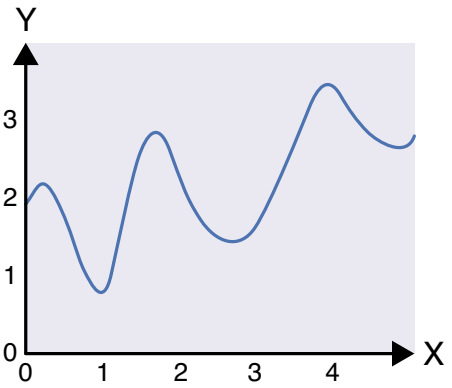


Figure 5-10: A curve in two dimensions. Values of x increase as we move right, and values of y increase as we move up.

As we move to the right from some point (that is, as x increases), we can ask if the curve is giving us values of y that are increasing, decreasing, or not changing at all. We say that if y increases as x increases, the tangent line has a *positive slope*. If y decreases with an increasing x, we say the tangent line has a *negative slope*. The more extreme the slope (that is, the closer it gets to vertical), the more positive or negative it becomes. This is just another way to state the relationship of the angle of the slope relative to a horizontal line. Figure 5-11 shows the idea.

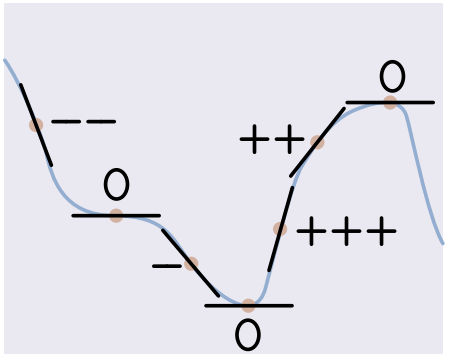


Figure 5-11: Marking the tangent lines from Figure 5-7 by whether they have a positive slope (+), negative slope (–), or are flat (0)

Notice that there are points in Figure 5-11 that aren't hills and valleys but still have a slope of 0. We only find slopes of 0 at the tops of hills, the bottoms of valleys, and plateaus like these.

Finding Minimums and Maximums with Derivatives

Let's see how to use the derivative to drive an algorithm that finds a local minimum or maximum at a point.

Given a point on a curve, we first find its derivative. If we want to move along the curve so that the y values increase, we move in the direction of the *sign* of the derivative. That is, if the derivative is positive, then moving in the positive direction along the X axis, or to the right, takes us to larger values. In the same way, if the derivative is negative, then to find smaller values of y, we move left. Figure 5-12 shows the idea.

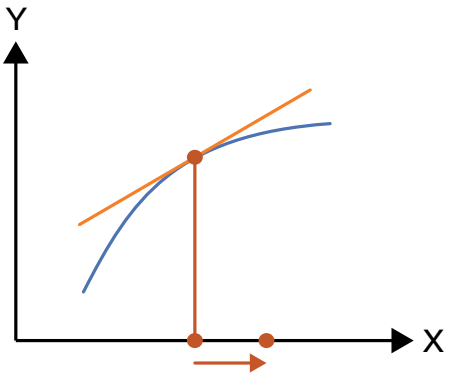


Figure 5-12: The derivative at a point tells us which way to move to find larger or smaller values of the curve.

We can gather up both cases and say that to find the local maximum near some point, we find the derivative at that point and take a small step along the X axis in the direction of the sign of the derivative. Then we find the derivative there and take another small step. We repeat this process over and over until we reach a point where the derivative is 0. Figure 5-13 shows this in action, starting from the rightmost point.

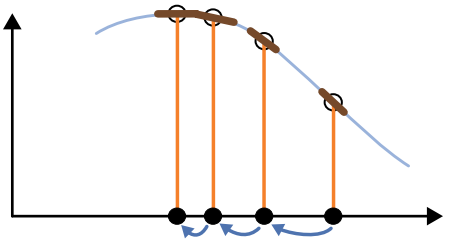


Figure 5-13: Using the derivative to find the local maximum at a point

At our starting point, the rightmost, we get a somewhat large, negative derivative, so we take a big step to the left. The second derivative is a bit smaller (that is, the slope is still negative, but a little less so), so we take a smaller step to the left. A third, smaller step takes us to the local maximum, where the tangent line is horizontal, so the derivative is 0. To make this algorithm practical, we'd have to address some details, such as the size of the steps we take and how to avoid overshooting the maximum, but right now we're just after the conceptual picture.

To find a local minimum, we do the same thing, but we move along X in the direction given by the *opposite* of the derivative's sign, as in Figure 5-14. Here we start at the leftmost point and find it has a negative derivative, so we keep moving right until we find a derivative of 0.

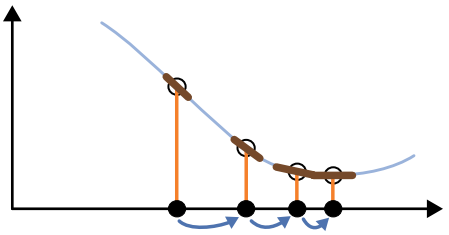


Figure 5-14: Using the derivative to find the local minimum at a point

Finding local maximums and minimums is a core numerical technique used throughout machine learning, and it relies on our being able to find the derivative at every point on the curve we're following. Our three curve conditions of smoothness, continuity, and being single-valued were chosen specifically so that we can always find a single, finite derivative at every point on our curve, which means that we can rely on this curve-following technique to find local minimums and maximums.

In machine learning, most of our curves obey these rules most of the time. If we happen to be using a curve that doesn't, and we can't compute the tangent or derivative at some point, there are mathematical techniques that usually (though not always) automatically finesse the problem so we can carry on.

We mentioned earlier that the derivative is also 0 where the curve itself flattens out. This can trick our algorithm into thinking it's found a maximum or minimum. In Chapter 15, we'll see a technique called *momentum* that can help us avoid getting fooled in this way and continue on our search for a real maximum or minimum.

The Gradient

The *gradient* is the generalization of the derivative into three dimensions, or four dimensions, or *any* number of dimensions beyond that. With the gradient, we can find the minimums and maximums for surfaces in these higher dimensional spaces. Let's see how this works.

Water, Gravity, and the Gradient

Imagine that we're in a big room, and above us is a billowing sheet of fabric that rises and falls without any creases or tears, as in Figure 5-15.

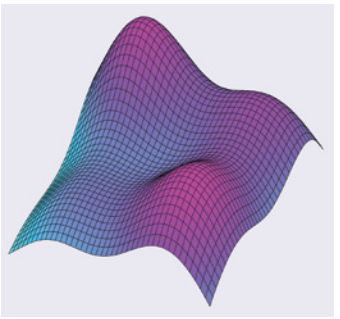


Figure 5-15: A sheet of smooth fabric without creases or tears

The surface of this fabric naturally satisfies the rules we required of our curves before: it's both smooth and continuous because it's a single piece of fabric, and it's single-valued because the fabric never curls over on itself (like a crashing wave). In other words, from any point on the floor below it, there is just one piece of the surface above it, and we can measure its height above the floor.

Now let's imagine that we can freeze the fabric at a particular moment. If we climb up onto the fabric and walk around on it, it will feel like we're hiking on a landscape of mountains, plateaus, and valleys.

Suppose that the fabric is dense enough that water can't pass through it. As we stand in one spot, let's pour some water onto the fabric at our feet. The water, naturally, flows downhill. In fact, the water follows the path that takes it downhill in the fastest possible way, because it's being pulled downward by gravity. At every point, it effectively searches the local neighborhood and moves in the direction that takes it downhill the fastest, as shown in Figure 5-16.

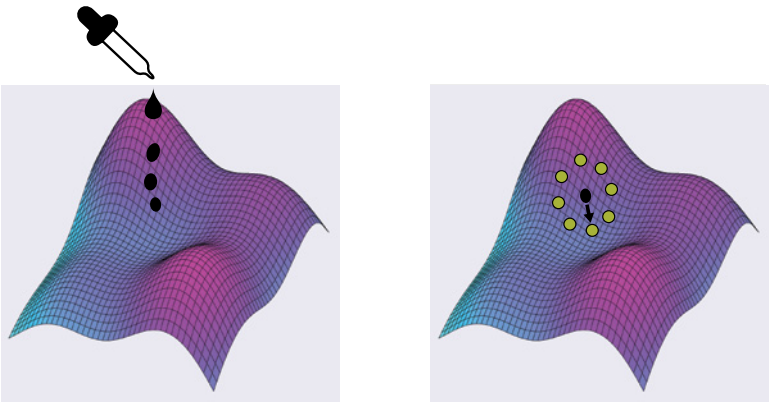


Figure 5-16: Left: Dripping water onto the surface. Right: A drop of water exploring multiple points in its local neighborhood (yellow) to find the one that is the most downhill.

Out of all the ways to move, the water always follows the steepest route downhill. The direction followed by the water is called the direction of *maximum descent*. The opposite direction, in which we climb upward as fast as possible, is the direction of *maximum ascent*.

The direction of maximum ascent is the same as the gradient. If we want to descend, we follow the *negative of the gradient*, or just the *negative gradient*. A hiker trying to reach the highest mountaintop as quickly as possible follows the gradient. A stream of flowing water flowing downhill as quickly as possible follows the negative gradient.

Now that we know the direction of maximum ascent, we can also find its *magnitude*, or strength, or size. That's simply how quickly we're going uphill. If we're going up a gentle slope, the magnitude of our ascent is a small number. If we're climbing up a steep grade, it's a bigger number.

Finding Maximums and Minimums with Gradients

We can use the gradient to find the local maximum in three dimensions (3D), just as we used the derivative in two dimensions (2D). In other words, if we're on a landscape and we want to climb to the highest peak around, we need only follow the gradient by always moving in the direction of the gradient associated with the point under our feet as we climb.

If we instead want to descend to the lowest point around, we can follow the negative gradient and always walk in the direction exactly opposite the gradient associated with each point under our feet as we descend. Essentially, we're acting like a drop of water, moving downhill in the fastest way possible. Figure 5-17 shows this step-by-step process in action.

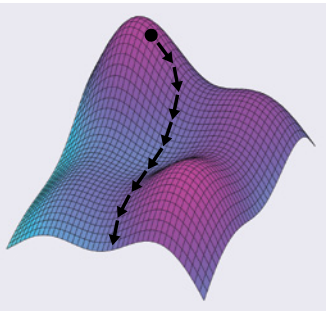


Figure 5-17: To get downhill, we can repeatedly find the negative gradient and take a small step in that direction.

Suppose that we're at the very top of a hill, as in Figure 5-18. This is a local maximum (and maybe the global maximum). Here, there is no uphill direction to go in. If we were to zoom in on the very top of the hill, we'd find the nearby surface is flat. Because there's no way to go up, our maximum rate of ascent is 0, and the magnitude of the gradient is 0. There's no gradient at all! We sometimes say the gradient has *vanished*, or that we have a *zero gradient*.

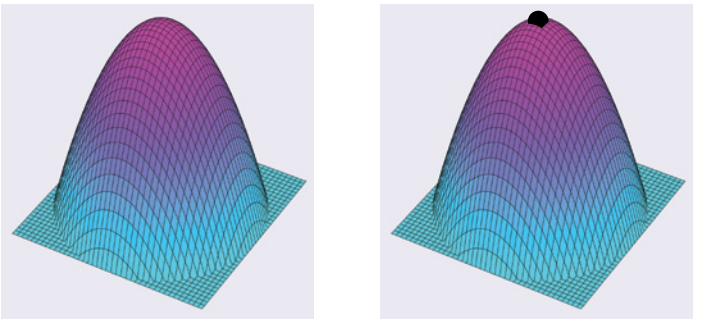


Figure 5-18: At the very top of a hill, there is no uphill. Left: The hill. Right: Our location at the very top of the hill.

When the gradient vanishes, as at the top of a hill, the negative gradient goes away, too.

What if we're at the bottom of a bowl-shaped valley, as in Figure 5-19? This is a local minimum (and maybe the global minimum).

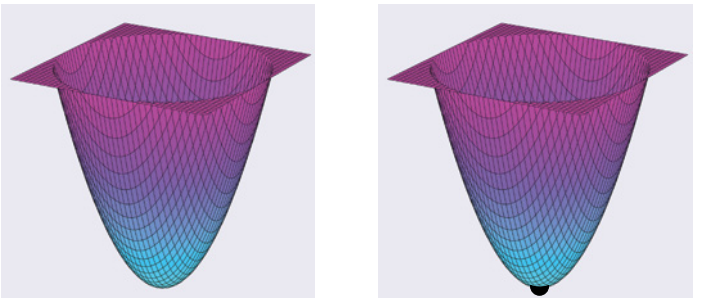


Figure 5-19: At the very bottom of a bowl, every move we make is uphill. Left: A bowl. Right: A point at the bottom of the bowl.

At the very bottom of the bowl, every direction seems to go up. But if we zoom way in, we'd see that the bottom of the bowl is flat. Again, the gradient has vanished.

What if we're not on a hilltop or in a valley or on the side of a slope but just on a flat plain, or *plateau*, as in Figure 5-20?

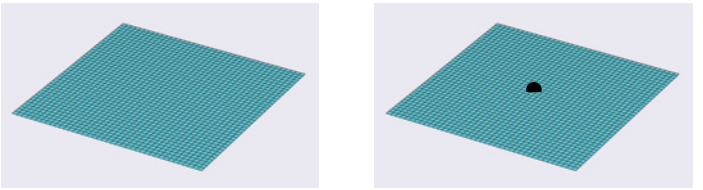


Figure 5-20: A flat surface, plain, or plateau. Left: The plateau. Right: The point on the plain is mainly on the plane. This point has no gradient.

Just like being on the hilltop, there's nowhere to go up or down. When we're on a plateau, we again have no gradient at all.

Saddle Points

So far, we've seen local minimums, maximums, and flat regions, just as we saw in 2D. But in 3D, there's a completely new type of feature. In one direction, we're in the bottom of a valley, while in the other direction, we're at the top of a hill. In the local neighborhood of such a point, the surface looks like the saddle that horse riders use. Naturally enough, this kind of shape is called a *saddle*. An example saddle is shown in Figure 5-21.

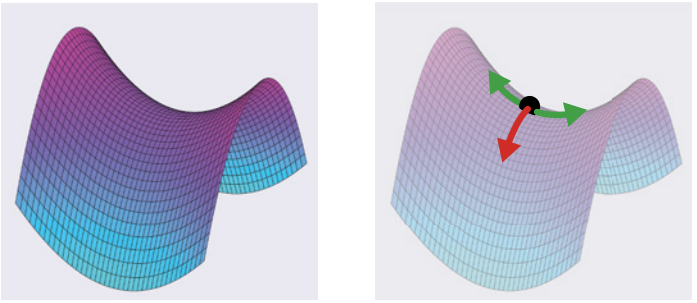


Figure 5-21: A saddle goes upward in one direction and downward in another. Left: A saddle. Right: A point on the saddle.

If we're in the middle of the saddle, as in Figure 5-21, then it's like being at a hilltop and valley at the same time. And just like those places, the local neighborhood looks like a plateau, so there's no gradient. But if we move just a little bit in one direction or another, we'll find a little bit of curvature, and then the gradient reemerges to show us the direction of maximum ascent from that spot.

When we train a deep learning algorithm, we usually want to find the least amount of error. Thinking of the error as a surface, the best scenario is when we can find the bottom of a bowl. But if we find ourselves at the top of a hill, or on a saddle, or on a plateau, we say that we've become *stuck* at these places. We know we're not at a minimum, but the gradient has vanished, so we have no idea which way to go in order to move downward.

Happily, modern algorithms offer a variety of automatic techniques to get us unstuck. But sometimes they fail, and unless we can introduce a major change, such as providing additional training data, our algorithm stays stuck, unable to move to a lower value of the surface. In practical terms, this means that the algorithm simply stops learning, and its output stops improving.

We'll see later that we can watch our learning progress by measuring its error. If the error stops improving before our results are acceptable, we can change the algorithm just a little so that it takes a different path when learning, and sidesteps that particular spot of zero gradient.

Summary

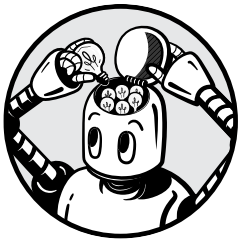
In this chapter, we looked at some ways to find the minimums and maximums of curves. When we train a deep learning system, we adjust it to minimize the system's overall error. If we think of the error as a surface in a many-dimensional space, we're looking for a minimum on that surface. To find that minimum error, we find the steepest downhill direction, given by the negative gradient. We then change the network so that the error moves in that direction. In essence, the gradient tells us how to change the network so that the overall error of the system is reduced.

In later chapters, we'll see how we actually use this idea in practice to teach our deep learning systems to get better and better at their jobs.

For now, let's turn to a little bit of information theory, which will help us better understand the nature of errors and how to interpret them.

6

INFORMATION THEORY



In this chapter we look at the basics of *information theory*. This is a relatively new field of study, introduced to the world in 1948 in a groundbreaking paper, which laid the foundation for technologies from modern computers and satellites to cell phones and the internet (Shannon 1948). The goal of the original theory was to find the most efficient way to communicate a message electronically. But the ideas of that theory are deep, broad, and profound. They give us tools for measuring how much we know about anything by converting it to a digital form that we can study and manipulate.

Terms and ideas from information theory form part of the bedrock of deep learning. For example, the measurements provided by information theory are useful when we evaluate the performance of deep networks. In this chapter, we take a fast tour through some of the basics of information theory, while staying free of abstract mathematical notation.

Let's begin with the word *information*, one of those words that has both an everyday meaning and a specialized, scientific meaning. In this case, the meanings share a lot conceptual overlap, but while the popular meaning is broad and open to personal interpretation, the scientific meaning is precise and defined mathematically. Let's start out by building up to the scientific definition of information, and ultimately work our way up to an important measurement that lets us compare two probability distributions.

Surprise and Context

When we receive a communication of any kind, something moved from one place to another, whether it was an electrical pulse, some photons of light, or the sound of someone's voice. Speaking broadly, we could say that a *sender* somehow transfers some kind of communication to a *receiver*. Let's introduce some more specialized vocabulary.

Understanding Surprise

In this chapter, we sometimes use the term *surprise* to represent how unexpected a sender's communication is to a receiver. Surprise isn't a formal term. In fact, one of our goals in this chapter is to find more formal names for surprise and attach specific meanings and measures to them.

Let's suppose that we're on the receiving end of a message. We want to describe how surprised we are by the communication we receive. Being able to do so is useful because, as we'll see, the greater the surprise, the greater the amount of information that was delivered.

Suppose we get an unexpected text message from an unknown number. We open it up and the first word is Thanks. How surprised are we? Surely we are at least a little surprised, because so far, we don't know who the message is from or what it's about. But receiving a text thanking us for something does happen, so it's not unheard of.

Let's make up an imaginary and completely subjective surprise scale, where 0 means something is completely expected, and 100 means it's a total surprise, as in Figure 6-1.

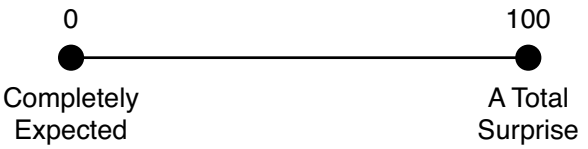


Figure 6-1: The surprise scale, expressed as a value from 0 to 100

On this scale, the word Thanks at the start of an unexpected text message might rank a 20. Now suppose that the first word in our message isn't Thanks, but instead is Hippopotamus. Unless we're working with those animals or are otherwise involved with them, that's likely to be a rather surprising

first word of a message. Let's rank this word at an 80 on the surprise scale, as in Figure 6-2.

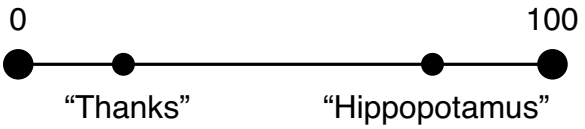


Figure 6-2: Placing messages on our surprise scale

Although hippopotamus might be a big surprise at the start of a message, it might not be surprising later on. The difference is context.

Unpacking Context

For our purposes, we can think of *context* as the environment of the message. Since we're focusing on the meaning of each message, rather than the physical way it's communicated, the context represents the shared knowledge between the sender and receiver, which gives the message meaning.

When the message is a piece of language, this shared knowledge must include the words used, since a message of Kxnfq rnggw would carry no meaning. We can extend that shared knowledge to include grammar, current interpretations of emoticons and abbreviations, shared cultural influences, and so on. This is all called *global context*. It's the general knowledge that we bring to any message, even before we've read it. In terms of our Bayes' Rule discussion of Chapter 4, some of this global context is captured in our *prior*, since that is how we represent our understanding of the environment and what we expect to learn from it.

In contrast to the global context, there is also *local context*. That's the environment composed of the elements of the message itself. In a text message, the local context for any given word is the other words in that message.

Let's imagine that we're reading a message for the first time, so each word's local context is made up only of the words that preceded it. We can use the context to get a handle on surprise. If Hippopotamus is the first word of our message, then there is no local context yet, only the global. And if we don't work with hippopotamuses on a regular basis, that word is likely very surprising. But if the message begins with, Let's go down to the river area at the zoo and maybe see a big gray, then in that context, the word hippopotamus isn't very surprising.

We can describe the amount of surprise carried by a specific word in our global context by assigning it a surprise value, as we did in Figure 6-1. Suppose that we assign a surprise value to every word in the dictionary (a tedious job, but certainly possible). If we scale these numbers so that they all add up to 1, we've created a probability mass function (or pmf), as we discussed in Chapter 2. That means we can draw a random variable from that pmf to get a word, with the most surprising words coming along more frequently than the less surprising words. A more common approach is to

set up the pmf to represent how common a word is, which is roughly the opposite of surprise. With that setup, we'd expect to draw the least surprising, or more common, words more frequently than uncommon words.

We'll use this idea later in the chapter to devise a scheme for transmitting the content of a message in an efficient manner.

Measuring Information

In this chapter, we're going to talk quite a lot about *bits*. In popular language, a bit is usually thought of as a little package of data, often labeled either 0 or 1. For instance, when we talk about internet speed in "bits per second," we might picture the bits as leaves flowing down a river, and we count them as they go by.

This is a convenient idea, but in technical language, a bit is not a thing, like a leaf, but a unit, like a gallon or a gram. That is, it isn't a piece of stuff but a way to talk about how much stuff we have. A bit is a container that holds just enough storage for what we currently think is the fundamental, indivisible, smallest possible chunk of information.

Speaking of bits as units in this way is technically correct, but it's inconvenient. And most of the time, we can speak casually without any confusion, like when we say, "My net connection is 8,000 bits per second," rather than, "My net connection is able to transmit 8,000 bits worth of information per second." We'll use the more casual language in most of this book, but it's worthwhile to know the technical definition, because it does pop up from time to time in papers and documentation where the distinction is important.

We can measure the amount of information in a text message with a formula that tells us how many bits are needed to represent that message. We won't get into the math, but we'll describe what's going on. The formula takes two inputs. The first is the text of the message. The second is a pmf that describes the surprise inherent in each word the message can contain (let's just call this a *probability distribution* for the rest of this chapter). When we take the text of the message and the probability distribution together, we can produce a number that tells us how many bits of information the message carries.

The formula was designed so that the values it produces for each word (or, more generally, each *event*) have four key properties. We'll illustrate each one using a context in which we work in an office, and not on a river.

1. Likely events have low information. Stapler has low information.
2. Unlikely events have high information. Crocodile has high information.
3. Likely events have less information than unlikely events. Stapler conveys less information than crocodile.
4. Finally, the total information due to two *unrelated* events is the sum of their individual information values found separately.

The first three properties relate single objects to their information. The oddball in the group is property 4, so let's look at it more carefully.

In normal conversation, it's rare for two consecutive words to be completely unrelated. But suppose someone asked us for a "kumquat daffodil." Those words are just about completely unrelated, so property 4 says that we could find the information in that phrase by adding the information communicated by each word independently.

In normal conversation, the words that lead up to any given word often narrow the possibilities of what it could be. If someone says, "Today I ate a big," then words like "sandwich" and "pizza" arriving next carry less surprise than "bathtub" or "sailboat." When words are expected, they produce less surprise than when they're not. By contrast, suppose we're sending a device's serial number, which is essentially an arbitrary sequence of letters and perhaps numbers, like "C02NV91EFY14." If the characters really have no relation to each other, then adding the surprise due to each character gives us the overall surprise in the entire message representing the serial number.

By combining the surprise of two unrelated words into the sum of their individual surprise values, we go from measuring the surprise, or information, in each of those words to the surprise in their combination. We can keep combining words this way into ever-larger groups until we've considered the entire message. Though we haven't gone into the math, we have reached a formal definition of *information*: it's a number produced from a formula that uses one or more events (such as words), and a probability distribution to describe how surprising each event would be to us. From those two inputs the algorithm provides a number for each event, and guarantees that those numbers satisfy the four properties we just listed. We call each word's number its *entropy*, telling us how many bits are needed to communicate it.

Adaptive Codes

The amount of information carried by each event is influenced by the size of the probability function we hand to our formula. In other words, the number of possible words we might communicate affects the amount of information carried by each word we send.

Suppose we want to transmit the contents of a book from one place to another. We might list all the unique words in that book and then assign a number to each word, starting perhaps with 0 for the, then 1 for and, and so on. Then, if our recipient also has a copy of that word list, we can send the book just by sending the number for each word, starting with the first word in the book. The Dr. Seuss book *Green Eggs and Ham* contains only 50 different words (Seuss 1960). To represent a number between 0 and 49, we need six bits of information per word. By contrast, Robert Louis Stevenson's book *Treasure Island* contains about 10,700 unique words (Stevenson 1883). We'd have to use 14 bits per word to uniquely identify each word in that book.

Although we could use one giant word list of all English words to send these books, it's more efficient to tailor our list to each book's individual vocabulary, including only the words we actually need. In other words, we can improve our efficiency by *adapting* our transmission of information to what's being communicated.

Let's take that idea and run with it.

Speaking Morse

A great example of adaptation is Morse code. In Morse code, each typographical character has an associated pattern of dots and dashes, separated by spaces, as shown in Figure 6-3.

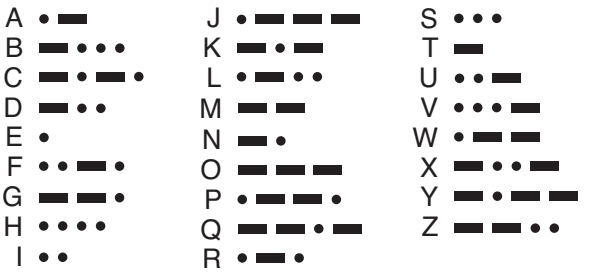


Figure 6-3: Each character in Morse code has an associated pattern of dots, dashes, and spaces.

Morse code is traditionally sent by using a telegraph key to enable or disable transmission of a clear tone. A dot is a short burst of sound. The length of time we hold down the key to send a dot is represented by a unit called the *dit*. A dash is held for the duration of three dits. We leave one dit of silence between symbols, a silence of three dits between letters, and a silence of seven dits between words. These are of course ideal measures. In practice, many people can recognize the personal rhythm, called the *fist*, of each of their friends and colleagues (Longden 1987).

Morse code contains three types of symbols: dots, dashes, and dot-sized spaces. Let's suppose we want to send the message "nice dog" in Morse code. Figure 6-4 shows the sequence of short tones (dots), long tones (dashes), and dot-sized spaces.



Figure 6-4: The three symbols of Morse code: dots (solid circles), dashes (solid boxes), and silent spaces (empty circles).

We typically talk about Morse code strictly in terms of dots and dashes, which are called the *symbols*. The assigned set of symbols for any letter is that letter's *pattern*. The length of time it takes to send a message depends

on the specific patterns assigned to the letters that make up the message's content. For example, even though the letters Q and H both have four symbols, Q requires 13 dits to send (3 for each of the 3 dashes, 1 for the dot, and 1 for each of the 3 spaces), while we need only 7 dits to send the letter H (4 dots, and 1 for each of the 3 spaces).

Let's compare the patterns of the different characters. When we look at Figure 6-3, it might not be clear to us that there's any principle behind how the various patterns are assigned. But a beautiful idea is there waiting to be uncovered. Figure 6-5 shows a list of the 26 Roman letters, sorted by their typical frequency in English (Wikipedia 2020). The most frequently used letter, E, leads the list.

E T A O I N S H R D L C U M W F G Y P B V K J X Q Z

Figure 6-5: The Roman letters sorted by their frequency of use in English

Now look back at the patterns in Figure 6-3. The most frequent letter, E, is just a single dot. The next most frequent letter, T, is just a single dash. Those are the only two possible patterns with just one symbol, so now we move on to two symbols. The letter A is a dot followed by a dash. O is next, and it breaks the pattern because it's too long: three dashes. Let's come back to that later. Returning to our list, the I is two dots, the N is a dash and a dot. The last two-letter pattern is M, with two dashes, but that's pretty far down the list from where we've gotten so far. Why is O too long and M too short? Morse code is almost following our letter-frequency table, but not quite.

The explanation starts with Samuel Morse, who only defined patterns for the numbers 0 through 9 in his original code. Letters and punctuation were added to the code by Alfred Vail, who designed those patterns in about 1844 (Bellizzi 11). Vail didn't have an easy way to look up letter frequencies, but he knew he should follow them, according to Vail's assistant, William Baxter. Baxter said,

> His general plan was to employ the simplest and shortest combinations to represent the most frequently recurring letters of the English alphabet, and the remainder for the more infrequent ones. For instance, he found upon investigation that the letter e occurs much more frequently than any other letter, and accordingly he assigned to it the shortest symbol, a single dot (•). On the other hand, j, which occurs infrequently, is expressed by dash-dot-dash-dot (– • – •) (Pope 1887)[1]

1 The quote refers to the letter J having the pattern dash-dot-dash-dot, but Figure 6-3 associates that pattern with the letter C. The quote refers to J's pattern in an early version of the code called American Morse, now rarely used (see *https://en.wikipedia.org/wiki/American_Morse_code*).

Vail figured that he could estimate the letter frequency table for English text by visiting his local newspaper in Morristown, New Jersey, where they were still setting stories by hand. In those days, typesetters built up a page one letter at a time. For each letter, they would choose an appropriate *slug*, or a metal bar with a letter embossed on one end, and place it into a large tray. Vail reasoned that the most popular characters would have the greatest number of slugs on hand, so he counted up the number of slugs in each letter's bin. Those popularity counts were his proxy for letter frequency in English (McEwen 1997). Given how small this sample was, he did a pretty great job, despite imperfections like apparently thinking that M was more frequent than O.

To see how well our frequency chart (and Morse code) lines up with some actual text, Figure 6-6 shows the frequencies for the letters from *Treasure Island* (Stevenson 1883). For this chart, we counted only the letters, which we turned into lowercase before counting. We also excluded numbers, spaces, and punctuation.

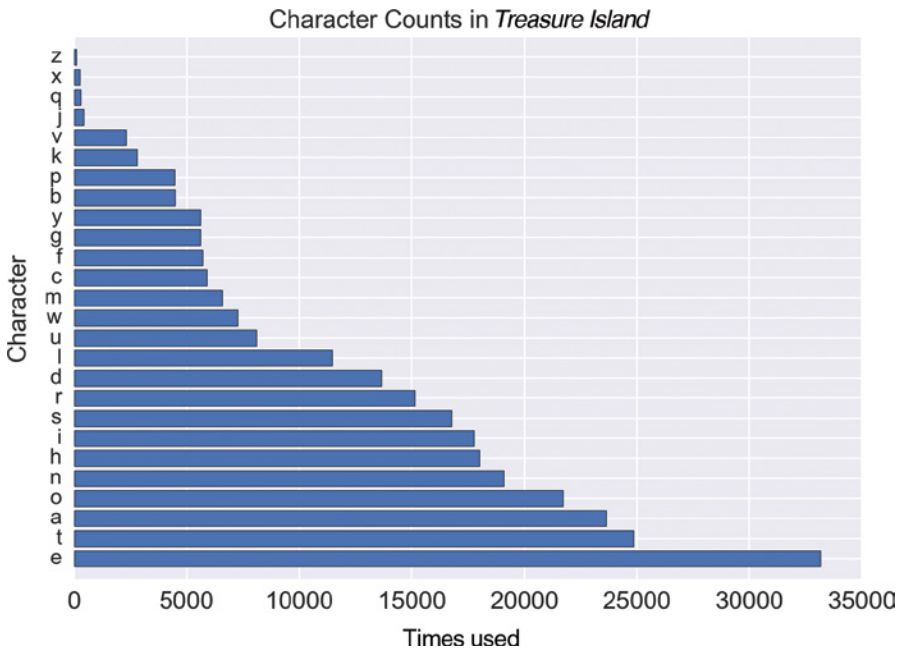


Figure 6-6: The number of times each letter appears in Treasure Island *by Robert Louis Stevenson. Uppercase letters were counted as lowercase.*

The order of the characters in Figure 6-6 isn't a perfect match to our letter frequency chart in Figure 6-5, but it's close. Figure 6-6 looks like a probability distribution over the letters A through Z. To make it an *actual* probability distribution, we have to scale it so that the sum of all the entries is 1. The result is shown in Figure 6-7.

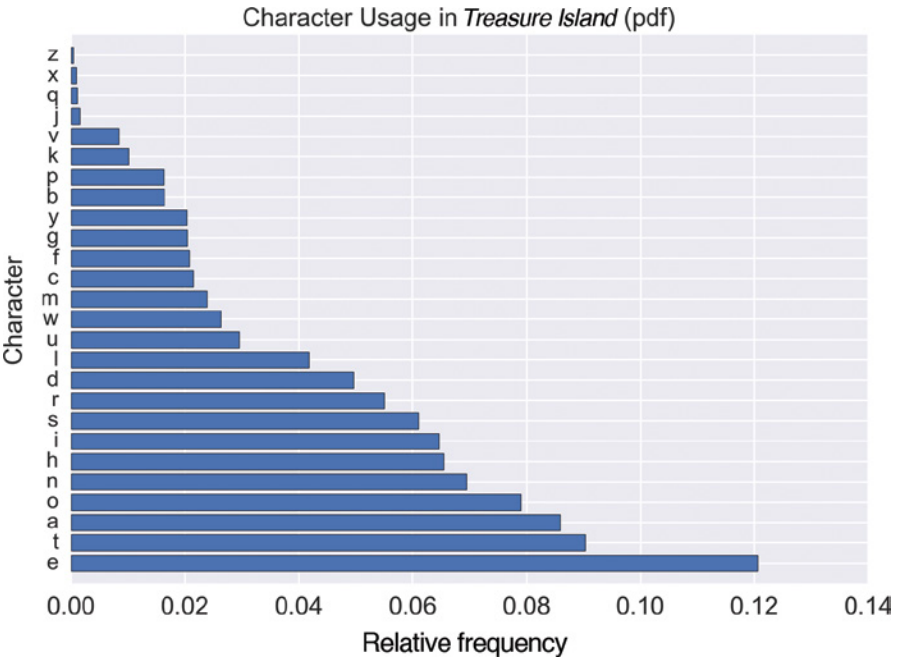


Figure 6-7: The probability distribution function (pdf) for characters in Treasure Island

Now let's use our probability distribution of letters to improve the efficiency of sending *Treasure Island* via Morse code.

Customizing Morse Code

To motivate our improvements to sending *Treasure Island* via Morse code, let's first take a step backward, and start with an imaginary version of Morse code where Mr. Vail didn't bother to journey down to the newspaper office. Instead, let's say he wanted to assign the same number of dot-and-dash symbols to each character. With four symbols he could only label 16 characters, but with five symbols he could label 32 characters.

Figure 6-8 shows how we might arbitrarily assign such a five-symbol pattern to each character. To keep things simple, we made the timing of every dot and dash the same by using different tones for the two symbols. So every dot (shown here as a black dot) is a high tone lasting one dit, and every dash (shown as a red square) is a low tone lasting one dit. The result is that every character takes nine dits of time to send (five for the dots and dashes, now high and low tones, and four for the silences between them). This is an example of a *constant-length code*, also called *a fixed-length code*.

In Figure 6-8 we didn't create a character for the space, following in the footsteps of the original Morse code, where it was assumed that we could figure out where the spaces ought to go by looking at the message. Sticking to that spirit, we'll ignore space characters for the rest of this discussion.

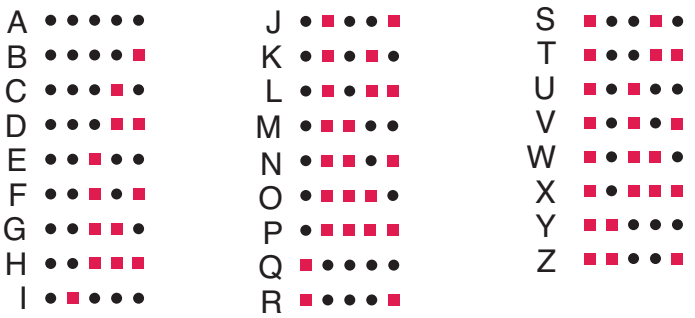


Figure 6-8: Assigning five symbols to each character gives us a constant-length code. Black circles are high tones; red squares are low tones. They all last one dit.

The first two words in the text of *Treasure Island* are the name "Squire Trelawney." Since every character in our two-tone version of Morse code requires 9 dits, this phrase of 15 letters (remember that we're ignoring the space) requires 9 × 15 = 135 dits of time to send. Adding in the 14 silences between letters, which take 3 × 14 = 42 bits, we find the fixed-length message takes 135 + 42 = 177 dits of time, as shown in Figure 6-9.

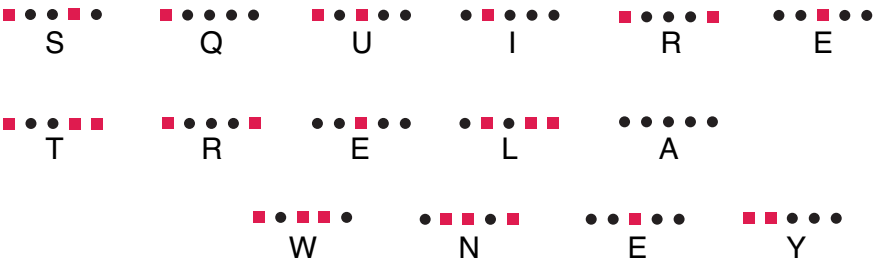


Figure 6-9: The first two words of Treasure Island*, using our constant-length code*

Now compare this to actual Morse code where, for the most part, the most common letters have fewer symbols than the uncommon letters. Figure 6-10 shows this. We'll continue sending dots and dashes using different tones that last one dit each.

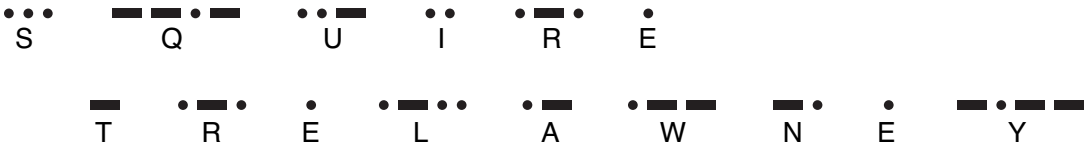


Figure 6-10: The first two words of Treasure Island*, using Morse code*

If we count up the elements (remembering that dots and dashes now take just one dit each), we find that the Figure 6-10 version requires 101 dits of time, about half as long as the fixed-length code (101 / 177 ≈ 0.57). That savings comes from adapting our code to the content we are sending. We call any code that tries to improve efficiency by matching up short patterns

with high-probability events a *variable-bitrate code*, or more simply, an *adaptive code*. Even in this simple example, our adaptive code is almost twice as efficient as the constant-length code, cutting our communication time nearly in half.

Let's look at the whole text of *Treasure Island*, which contains about 338,000 characters (excluding spaces, punctuation, etc.). The adaptive code would take only about 42 percent of the time required by the fixed-length code. We can send the book in less than half the time required by a non-adaptive code.

We can do even better if, instead of using standard Morse code, which is adapted to English writing in general, we tune the distribution of symbols to more closely match their actual percentages in the text of the specific book we're sending. Of course, we'd have to share our clever encoding with our recipient, but if we're sending a long message, that extra piece of communication is dwarfed by the message itself. Let's take that step, and imagine a custom Treasure Island code that is perfectly adapted to the contents of *Treasure Island* specifically. We should expect even more savings.

Let's rephrase this using the language of probability. An adaptive code creates a pattern for each value in a probability distribution. The value with the highest probability receives the shortest possible code. Then we work our way through the values, from the highest probability to the lowest, assigning patterns that are always as short as possible without repeating. That means each new pattern is as long as, or longer than, the pattern assigned to the previous value. That's just what Mr. Vail did in 1844, guided by the number of letters he found in the typesetter's bins of his local newspaper.

Now we can look at any message we want to communicate, identify each character, and compare it to the probability distribution that tells us how likely that character was in the first place. This tells us how much information, in bits, is carried by that character. Thanks to the fourth property in our description of the formula for computing information, the total number of bits required to represent the message (ignoring context for the moment), is just the sum of the individual numbers of bits required by each character.

We can also perform this process for our message before we send it. That tells us just how much information we're about to communicate to our recipient.

Entropy

We've discussed *surprise*, which refers to things we didn't expect. A related idea is *uncertainty*, which refers to those times when we know all the things that might happen, but we're not sure which one will actually occur. For instance, when we roll a fair six-sided die, we know that each of the six faces has an equal probability of coming up, but until we roll it and look, we are uncertain which face will be on top. A more formal term for this uncertainty is *entropy*.

We can assign a number to the uncertainty, or entropy, of an outcome. This number often depends on how many outcomes are possible. For example, flipping a coin can have only 2 outcomes, but rolling a six-sided die can have 6 outcomes, and picking a letter from the alphabet can have 26 outcomes. The uncertainty of these three results, or their entropy, is a number that increases in size, from the coin to the die to the alphabet, because the number of outcomes is increasing in each case. That makes each specific result more uncertain.

In those three examples, the probability of each outcome is the same (1/2 for each side of the coin, 1/6 for each die face, and 1/26 for each letter). But what if the probabilities of the outcomes are different? The formula for computing the entropy explicitly takes these different probabilities into account. Essentially, it considers all the possible outcomes in a distribution and puts a number to the uncertainty describing which outcome is actually produced when we sample the distribution.

It turns out that the uncertainty of a specific event occurring is the same as the number of bits required to send a message with a perfectly adapted code. Conceptually, a text message is a set of words drawn from a vocabulary, which is no different than the values of a die rolled multiple times. We use the term *entropy* for both values: the uncertainty of an event, or the number of bits required to communicate that event (and remove the uncertainty).

Entropy is useful in machine learning because it lets us compare two probability distributions. This is a key step in learning. For example, consider a classifier. We might have a picture that we have manually decided is 80 percent likely to be a dog, but 10 percent likely to be a wolf, 3 percent likely to be a fox, and a few other smaller probabilities for other animals. We'd like the system's predictions to match those labels. In other words, we want to compare our manual distribution with the system's predicted distribution and use any differences to improve our system. We can invent lots of ways to compare distributions, but the one that works best in both theory and practice is based on entropy. Let's build our way up to that comparison, starting with finding the entropy for a single distribution.

Consider a distribution made up of words. If only one word is in our distribution, there is no uncertainty about what word we'll get when we sample the distribution, and therefore the entropy is 0. If there are lots of words but they all have a probability of 0, except for a single word with a probability of 1, there's still no uncertainty, so the entropy is again 0. When all of the words have the same probabilities, we have the maximum uncertainty, since no choice is any more probable than any other. In this case, our uncertainty, or entropy, is at a maximum. Though it might be convenient to say that maximum entropy should be 1, or 100, the actual value is calculated by the formula. What we do know is that no other probability distribution will give us a larger entropy.

In the next section, we'll see how to apply entropy to pairs of distributions.

WHERE THE TERM ENTROPY COMES FROM

The term *entropy* has been used for decades in the field of thermodynamics, a branch of physics that is concerned with heat and temperature. In that field, entropy refers to "disorder." Let's see how the physics version of entropy compares to the information theory version.

In thermodynamics, we often think of whatever we're studying as a *system*. This is just the collection of things we care about. Let's imagine a system many people look forward to each morning. It consists of two mixed parts: coffee and milk. The physicist might ask, "Where is the coffee located in this this mixture?" The information theorist might ask, "If I spoon up some liquid, will I get coffee or milk?"

When we begin, the coffee is in a cup and the milk is in a small pitcher. To the physicist, this system has high order, or low disorder, since everything is in its own place. Because it has low disorder, it has low entropy. To the information theorist, dipping our spoon into the coffee cup is guaranteed to come up with coffee, so there is no uncertainty. Low uncertainty means low entropy.

Let's pour the milk into the coffee, and stir lightly. Now when the physicist asks, "Where is the coffee in this mixture?" the answer is harder to state. The coffee is all mixed up with the milk. There is a lot of disorder here, so there's high entropy. When the information theorist dips in a spoon, and we haven't stirred thoroughly, there's no way to predict what ratio of coffee and milk will come up. There's a lot of uncertainty, and thus again high entropy.

Disorder, uncertainty, and entropy are all different ways of referring to the same idea (Serrano 2017).

Cross Entropy

When we're training a deep learning system, we'll often want to have a measure that tells us to what degree two probability distributions are the same or different. The value we usually use is a quantity called the *cross entropy*, and it too is just a number. Recall that the entropy tells us how many bits we need to send a message using a code that is perfectly tuned to that message. The cross entropy tells us how many bits we need if we use some other, less perfect code. Generally, this is larger than the number of bits the perfect code needs (if the alternative code happens to be exactly as efficient as the ideal code, the cross entropy has its minimum value of 0). The cross entropy is a measurement that lets us compare two probability distributions numerically. Identical distributions have a cross entropy of 0, while increasingly different pairs of distributions have increasingly larger values of cross entropy.

To get a feeling for the idea, let's look at two novels, and build up a word-based adaptive code for each. Though our goal is to compare probability distributions, and we're here talking about codes, it's conceptually easy to go back and forth. Recall that by construction, smaller codes correspond to words with higher probabilities, while larger codes correspond to words with lower probabilities.

Two Adaptive Codes

The novels *Treasure Island* and *The Adventures of Huckleberry Finn*, by Mark Twain, were both written in English at about the same time (Stevenson 1883; Twain 1885). *Treasure Island* has the larger vocabulary, using about 10,700 unique words, compared to about 7,400 unique words in *Huckleberry Finn*. Of course, they use very different sets of words, but there's lots of overlap. Let's look at the 25 most popular words in *Treasure Island*, shown in Figure 6-11. For the purposes of counting words, we first converted all uppercase letters to lowercase. The single-letter pronoun "I" therefore appears in the charts as the lower-case "i."

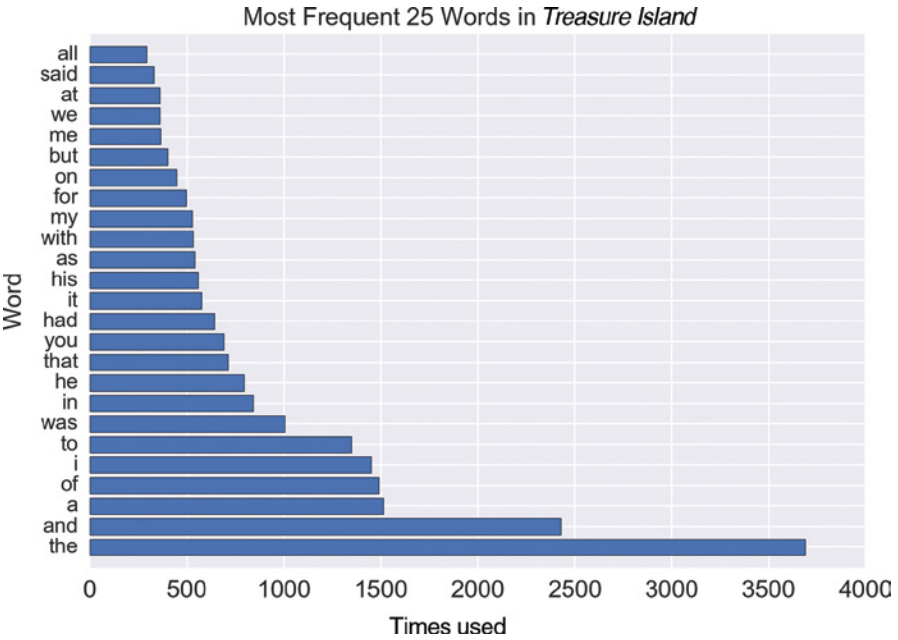


Figure 6-11: The 25 most popular words in Treasure Island*, sorted by number of appearances*

Let's compare these to the 25 most popular words in *Huckleberry Finn*, shown in Figure 6-12.

Perhaps unsurprisingly, the most popular dozen words in both books are almost the same (though in different orders), but then things begin to diverge.

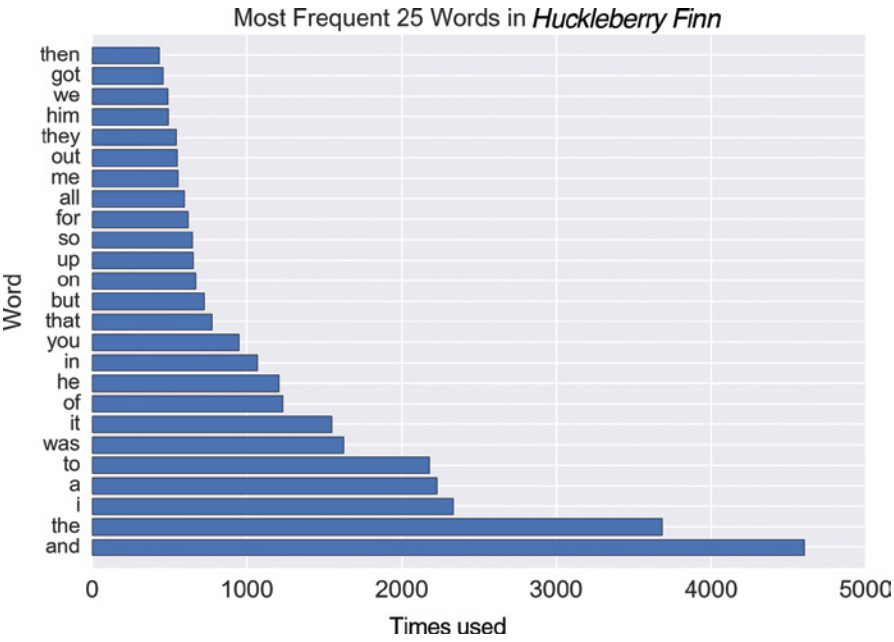


Figure 6-12: The 25 most popular words in Huckleberry Finn*, sorted by number of appearances*

Let's suppose we want to transmit the text of both books, word by word. We could go to the English dictionary and assign every word a number starting with 1, then 2, then 3, and so on. But we know from our earlier Morse code example that we can send information more efficiently by using a code that's adapted to the material being sent. Let's create that kind of code, where the more frequently a word appears, the smaller its code number. So super-frequent words like the and and can be sent with short codes, while the rare words have longer codes that require us to send more bits (in *Treasure Island* about 2,780 words appear only once; in *Huckleberry Finn* about 2,280 words appear only once).

The vocabularies of the two books mostly overlap, but each book has words that don't appear in the other. For instance, the word yonder appears 20 times *Huckleberry Finn*, but not even once in *Treasure Island*. And schooner is in *Treasure Island* 28 times, but it's nowhere to be found in *Huckleberry Finn*.

Because we want to be able to send either book with either code, let's unify their vocabularies. For each word in *Huckleberry Finn* that isn't in *Treasure Island*, we add one instance of that word when we make the Treasure Island code. We do the same thing for *Huckleberry Finn*. For example, we tack on one instance of yonder to the end of the book when we make the Treasure Island code so that we can use that code to send *Huckleberry Finn* if we wanted to.

Let's start with the words in *Treasure Island*. We'll make an adaptive code for this text, starting with a tiny code for the and working our way up to huge codes for one-time-only words like wretchedness. Now we can send the whole book using that code and save time compared to any other code.

Now we'll do the same thing for *Huckleberry Finn*, and make a code specifically for this text, giving the shortest code to and and leaving the big codes for one-time-only words like dangerous (shocking, but true: dangerous appears only once in *Huckleberry Finn*!). The Huckleberry Finn code now lets us send the contents of this book more quickly than any other code.

Note that these two codes are different. We'd expect that, because the two books have different vocabularies, and cover significantly different subject matter.

Using the Codes

Now we have two codes, each of which can transmit either book. The Treasure Island code is tuned to how many times each word appears in *Treasure Island*, and the Huckleberry Finn code is tuned to *Huckleberry Finn*.

The *compression ratio* tells us how much savings we get from using an adaptive code versus a fixed-length code. If the ratio is exactly 1, then our adaptive code uses exactly as many bits as a nonadaptive code. If the ratio is 0.75, then the adaptive code sends only 3/4 the number of bits needed by the nonadaptive code. The smaller the compression ratio, the more bits we're saving (some authors define this ratio with the numbers in the other order, so the larger the ratio, the better the compression).

Let's try sending our two books word by word. The top bar of Figure 6-13 shows the compression ratio that we get from sending *Huckleberry Finn* with the code we built for it. We used an adaptive code called a *Huffman code*, but the results would be similar for most adaptive codes (Huffman 1952; Ferrier 2020).

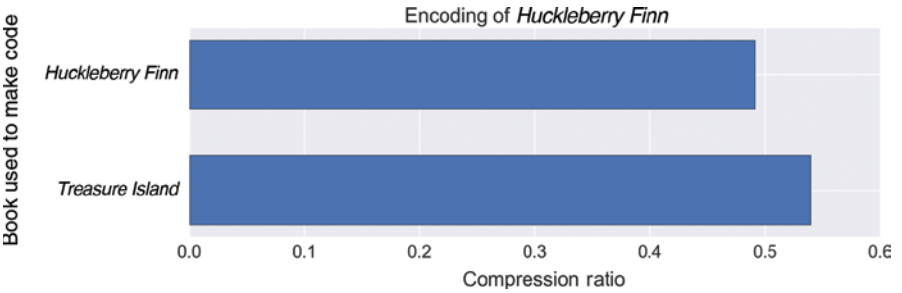


Figure 6-13: Top: The compression ratio from sending Huckleberry Finn *using the code built from that book. Bottom: The compression from using the code built from* Treasure Island.

This is pretty great. The adaptive code got a compression ratio of a little less than 0.5, meaning that to send *Huckleberry Finn* using this code would require a little less than half the number of bits required by a fixed-length

code. If we send *Huckleberry Finn* using the code built from *Treasure Island*, we should expect that the compression won't be as good, because our numbers in that code are not matched to the word frequencies we're encoding. The bottom bar of Figure 6-13 shows this result, with a compression ratio of around 0.54. That's still pretty great, but not quite as efficient.

Let's flip the situation around and see how *Treasure Island* does with a code built for it, and one built for *Huckleberry Finn*. The results are shown in Figure 6-14.

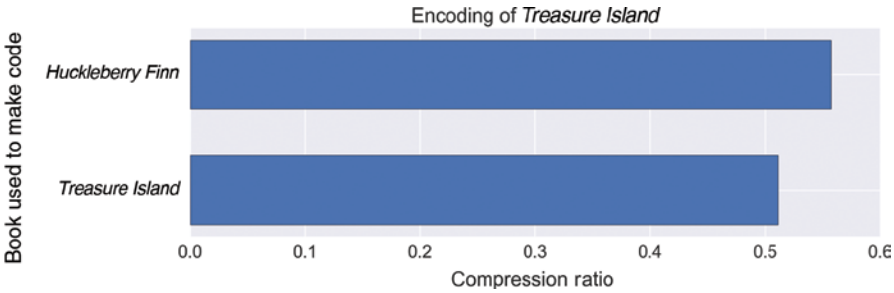


Figure 6-14: Top: The compression ratio from sending Treasure Island *using the code built from* Huckleberry Finn. *Bottom: The compression ratio for sending* Treasure Island, *using the code for* Treasure Island.

This time we find that *Treasure Island* compressed better than *Huckleberry Finn*, which makes sense because we used a code tuned to its word usage. In general, the fastest way to send any message is with a code that was built for the contents of that message. No other code can do better, and most will do worse.

We've seen that using the Treasure Island code to send *Huckleberry Finn* gives us worse compression. In other words, it requires more bits to send this book with a code that is imperfect for this message. This is because each code is based on its corresponding probability distribution, and those distributions are different.

The quantity we use to measure the difference between two probability distributions is *cross entropy*.

Note that the situation is not symmetrical. If we want to send words from *Treasure Island* using the Huckleberry Finn code, the cross entropy will be different from sending *Huckleberry Finn* with the Treasure Island code. We sometimes say that the cross entropy function is *asymmetrical* in its arguments, meaning that their order matters.

One way to conceptualize this is to picture that our space of probability distributions is like the ocean, with currents flowing in different directions in different places. The effort required to swim from some point A to another point B, sometimes fighting the currents and sometimes getting carried along by them, is generally different than the effort required to swim from B to A. In this metaphor, the cross entropy is measuring the amount of work, not the actual distance between the points. But as A and B

get closer together, the work involved in swimming between them, in either direction, goes down.

Cross Entropy in Practice

Let's see cross entropy in action. We'll use it just as we do when we're training a photo classifier and need to compare two probability distributions. The first is the label that we manually created to describe what's in the photo. The second is the set of probabilities that the system computes when we show it that photo. Our goal is to train the system so that its outputs match our labels. To do that, we need to know when the system gets it wrong and put a number to how wrong it is. That's the cross entropy we get by comparing the label and the predictions. The larger the cross entropy, the larger the error.

In Figure 6-15 we have the output of an imaginary classifier that's predicting the probabilities for a picture of a dog. In most real situations, all of the label values would be 0 except for the entry for dog, which would be 1. Here we've assigned arbitrary probabilities to each of the six labels to better show how the system tries to match the label distribution (we can imagine that the picture is blurry, so we're not sure ourselves what animal it shows).

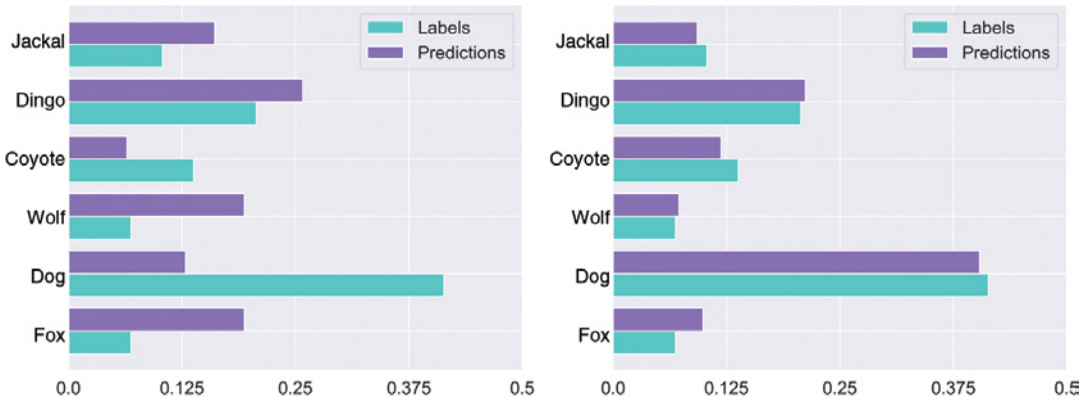


Figure 6-15: Classifying a picture of a dog. Left: At the start of training. Right: After much training. The cross entropy is lower when the match is better.

The figure at the left comes from the start of training. The system's predictions are a pretty poor match to our manual labels. If we run these numbers through the cross entropy formula, we get a cross entropy of about 1.9. On the right, we see the results after some training. Now the two distributions are much closer, and the cross entropy has dropped to about 1.6.

Most deep learning libraries offer built-in routines that compute the cross entropy for us in a single step. In Figure 6-15 we had six categories. When there are only two categories, we can use a routine that's specialized for that case. It's often called the *binary cross entropy* function.

Kullback–Leibler Divergence

Cross entropy is a great measure for comparing two distributions. By minimizing the cross entropy, we minimize the error between the classifier's outputs and our label, improving our system.

We can make things just a little simpler conceptually with one more step. Let's think of our word distributions as codes again. Recall that the entropy tells us how many bits are required to send a message with a perfect, tuned code. And the cross entropy tells us how many bits are required to send that message with an imperfect code. If we subtract the entropy from the cross entropy, we get the number of additional bits required by the imperfect code. The smaller we can get this number, the fewer additional bits we need, and the more the corresponding probability distributions are the same.

This extra number of bits required by an imperfect code (that is, the increase in entropy) goes by a large number of formidable names. The most popular is the *Kullback–Leibler divergence* or just *KL divergence*, named for the scientists who presented a formula for computing this value. Less frequently, it's also referred to as *discrimination information*, *information divergence*, *directed divergence*, *relative entropy*, and *KLIC* (for *Kullback–Leibler information criterion*).

Like the cross entropy, the KL divergence is asymmetrical: the order of the arguments matters. The KL divergence for sending *Treasure Island* with the Huckleberry Finn code is written KL(Treasure Island||Huckleberry Finn). The two bars in the middle can be thought of as a single separator, like the more frequently seen comma. We can think of them as representing the phrase "sent using the code for." If we run through the math, this value is about 0.287. We can think of this as telling us that we're "paying" around 0.3 extra bits per word because we're using the wrong code (Kurt 2017). The KL divergence for sending *Huckleberry Finn* with the Treasure Island code, or KL(Huckleberry Finn||Treasure Island), is much higher, at about 0.5.

The KL divergence tells us the number of additional bits we need in order to send our message with an imperfect code. Another way to think about this is that the KL divergence describes how much more information we need to turn our imperfectly adapted code into a perfect one. We can imagine this as a step of Bayes' Rule, where we go from an approximate prior (the imperfect code) to a better posterior (the adapted code). In this case, the KL divergence is telling us just how much we learn from that idealized step of Bayes' Rule (Thomas 2017).

We can train our systems either by minimizing the KL divergence, or the cross entropy, choosing whichever is more convenient. The KL divergence has nice mathematical properties and shows up in many mathematical and algorithmic discussions and even deep learning documentation. But in practice, the cross entropy is almost always faster to compute. Since minimizing either one has the same effect of improving our system, we usually see KL divergence in technical discussions, and cross entropy in deep learning programs.

Summary

In this chapter we looked at some of the basic ideas behind information theory, and how we can use them to train a deep learning system. We use these ideas in machine learning by translating our codes into probability distributions. That just means identifying the code elements with the smallest code numbers as the most frequent elements, and as the size of the number goes up, the frequency goes down. Interpreted this way, we can calculate the cross entropy of a classifier by comparing the list of predicted probabilities it produces in response to an input with the list of probabilities we assigned by hand. Our goal in training is to make the two distributions as similar as possible, which we can also state as trying to minimize the cross entropy.

This wraps up the first part of the book. We've covered some fundamental ideas that have value far beyond deep learning. Statistics, probability, Bayes' Rule, curves, and information theory all can help us make sense of a wide variety of problems and even things that come up in everyday life. They can help us improve our reasoning about events that happen in the world, and thus help us understand the past and prepare for the future.

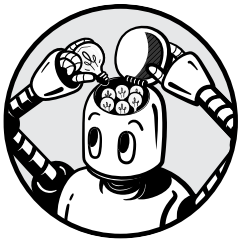
With these fundamentals in our pocket, we'll now turn to the basic tools of machine learning.

PART II

BASIC MACHINE LEARNING

7

CLASSIFICATION



An important application of machine learning involves looking at a set of inputs, comparing each one to a list of possible classes (or categories), and assigning each input to its most probable class. This process is called *classification* or *categorization*, and we say that it's carried out by a *classifier*. We can use classes for tasks as diverse as identifying the words someone has spoken into their cell phone, what animals are visible in a photograph, or whether a piece of fruit is ripe or not.

In this chapter, we look at the basic ideas behind classification. We don't consider specific classification algorithms here because we will get into those in Chapter 11. Our goal now is just to become familiar with the principles. We also look at *clustering*, which is a way to automatically group together samples that don't have labels. We wrap up by looking at how spaces of four and more dimensions can often foil our intuition, leading

to potential problems when we're training a deep learning system. Things can work in unexpected and surprising ways in spaces of more than three dimensions.

Two-Dimensional Binary Classification

Classification is a big topic. Let's start with a high-level overview of the process and then dig into some specifics.

A popular way to train a classifier uses *supervised learning*. We begin by gathering a collection of *samples*, or pieces of data, that we want to classify. We call this the *training set*. We also prepare a list of *classes,* or categories, such as what animals might be in a photo, or what genre of music should be assigned to an audio sample. Finally, we manually consider each example in the training set, and determine which class it should be assigned to. This is called that sample's *label*.

Then we provide the computer with each sample, one at a time, but we don't give it the label. The computer processes the sample and comes up with its own *prediction* of what class it should be assigned to. Now we compare the computer's prediction with our label. If the classifier's prediction doesn't match our label, we modify the classifier a little so that it's more likely to predict the correct class if it sees this sample again. We call this *training*, and say that the system is *learning*. We do this over and over, often with thousands or even millions of samples, recycled over and over. Our goal is to gradually improve the algorithm until its predictions match our labels so often that we feel it's ready to be released to the world, where we expect it will be able to correctly classify new samples it's never seen before. At that point we test it with new data to see how well it really works, and if it's ready to be used for real.

Let's look at this process a little more closely.

To get started, let's assume that our input data belongs to only two different classes. Using only two classes simplifies our discussion of classification, without missing any key points. Because there are only two possible labels (or classes) for every input, we call this *binary classification*.

Another way to make things easy is to use two-dimensional (2D) data, so every input sample is represented by exactly two numbers. This is just complicated enough to be interesting, but still easy to draw, because we can show each sample as a point on the plane. In practical terms, it means that we have a bunch of points, or dots, on the page. We can use color and shape coding to show each sample's label and the computer's prediction. Our goal is to develop an algorithm that can predict every label accurately. When it does, we can turn the algorithm loose on new data that doesn't have labels and rely on it to tell us which inputs belong to which class.

We call this a *2D binary classification system*, where "2D" refers to the two dimensions of the point data, and "binary" refers to the two classes.

The first set of techniques that we're going to look at are collectively called *boundary methods*. The idea behind these methods is that we can look at the input samples drawn on the plane and find a line or curve that

divides up the space so that all the samples with one label are on one side of the curve (or boundary), and all those with the other label are on the other side. We'll see that some boundaries are better than others when it comes to predicting future data.

Let's make things concrete using chicken eggs. Suppose that we're farmers with a lot of egg-laying hens. Each of these eggs might be fertilized and growing a new chick, or unfertilized. Let's suppose that if we carefully measure some characteristics of each egg (say, its weight and length), we can tell if it's fertilized or not. (This is completely imaginary, because eggs don't work that way! But let's pretend they do.) We bundle the two *features* of weight and length together to make a *sample*. Then we hand the sample to the classifier, which assigns it either the class "fertilized" or "unfertilized."

Because each egg that we use for training needs a label, or a known correct answer, we use a technique called *candling* to decide if an egg is fertilized or not (Nebraska 2017). Someone skilled at candling is called a *candler*. Candling involves holding up the egg in front of a bright light source. Originally candlers used a candle, but now they use any strong light source. By interpreting the fuzzy dark shadows cast by the egg's contents onto the eggshell, a skilled candler can tell if that egg is fertilized or not. Our goal is to get the classifier to give us the same results as the labels determined by a skilled candler.

To summarize, we want our *classifier* (the computer) to consider each *sample* (an egg) and use its *features* (weight and length) to make a *prediction* (fertilized or unfertilized). Let's start with a bunch of *training data* that gives the weight and length of some eggs. We can plot this data on a grid with weight on one axis and length on the other. Figure 7-1 shows our starting data. Fertilized eggs are shown with a red circle and unfertilized eggs with a blue box. With this data, we can draw a straight line between the two groups of eggs. Everything on one side of the line is fertilized, and everything on the other is unfertilized.

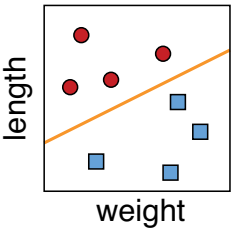


Figure 7-1: Classifying eggs. The red circles are fertilized. The blue squares are unfertilized. Each egg is plotted as a point given by its two dimensions of weight and length. The orange line separates the two clusters.

We're done with our classifier! When we get new eggs (without known labels), we can just look to see which side of the line they lie on when plotted. The eggs that are on the fertilized side get assigned the class "fertilized," and those on the unfertilized side are assigned the class "unfertilized." Figure 7-2 shows the idea.

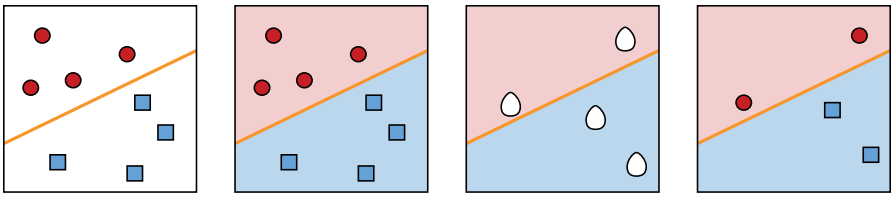


Figure 7-2: Classifying eggs. Left: The boundary. Second from left: Showing the two regions, or domains, made by the boundary. Third from left: Four new samples to be classified. Far right: The classes assigned to the new samples.

Let's say that this works out great for a few seasons, and then we buy a whole lot of chickens of a new breed. Just in case their eggs are different than the ones we've had before, we candle a day's worth of new eggs from both breeds manually to determine if they're fertilized or not, and then we plot the results as before. Figure 7-3 shows our new data.

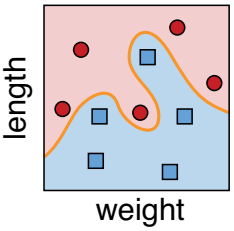


Figure 7-3: When we add some new types of chickens to our flock, determining which eggs are fertilized based just on the weight and length may get trickier.

The two groups are still distinct, which is great, but now they're separated by a squiggly curve rather than a straight line. That's no problem, because we can use the curve just like the straight line from before. When we have additional eggs to classify, each egg gets placed on this diagram. If it's in the red zone it's predicted to be fertilized, and if it's in the blue zone it's predicted to be unfertilized. When we can split things up this nicely, we call the sections into which we chop up the plane *decision regions*, or *domains*, and the lines or curves between them *decision boundaries*.

Let's suppose that word gets out and people love the eggs from our farm, so the next year we buy yet another group of chickens of a third variety. As before, we manually candle a bunch of eggs and plot the data, this time getting the diagram shown in Figure 7-4.

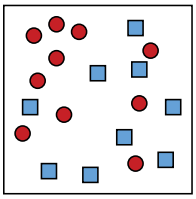


Figure 7-4: Our new purchase of chickens has made it much harder to distinguish the fertilized eggs from the unfertilized eggs.

We still have a mostly red region and a mostly blue region, but we don't have a clear way to draw a line or curve to separate them. Let's take a more general approach and, rather than predict a single class with absolute certainty, let's assign each possible class its own *probability*.

Figure 7-5 uses color to represent the probability that a point in our grid has a particular class. For each point, if it's bright red, then we're very sure that egg is fertilized, while diminishing intensities of red correspond to diminishing probabilities of fertility. The same interpretation holds for the unfertilized eggs, shown in blue.

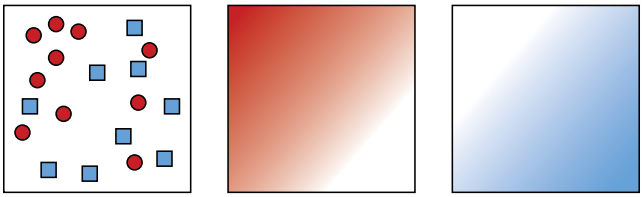


Figure 7-5: Given the overlapping results shown at the far left, we can give every point a probability of being fertilized, as shown in the center where brighter red means the egg is more likely to be fertilized. The image at the far right shows a probability for the egg being unfertilized.

An egg that lands solidly in the dark-red region is probably fertilized, and an egg in the dark-blue region is probably unfertilized. In other places, the right class isn't so clear. How we proceed depends on our farm's policy. We can use our ideas of accuracy, precision, and recall from Chapter 3 to shape that policy and tell us what kind of curve to draw to separate the classes. For example, let's say that "fertilized" corresponds to "positive." If we want to be very sure we catch all the fertilized eggs and don't mind some false positives, we might draw a boundary as in the center of Figure 7-6.

On the other hand, if we want to find all the unfertilized eggs, and don't mind false negatives, we might draw the boundary as in the right of Figure 7-6.

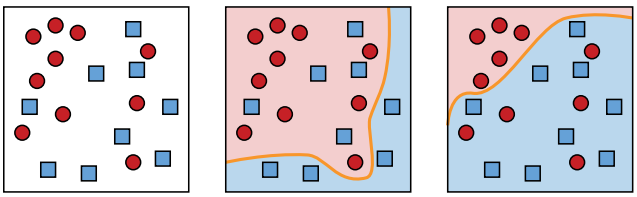


Figure 7-6: Given the results at the far left, we may choose a policy shown in the center, which accepts some false positives (unfertilized eggs classified as fertilized). Or we may prefer to correctly classify all unfertilized eggs, with the policy shown at the right.

When the decision regions have sharp boundaries and don't overlap, as in the center and right images in Figure 7-6, the probabilities for each sample are easy: the class where the sample falls is a certainty with probability 1,

and the other classes have probability 0. But in the more frequent case, when the regions are fuzzy or overlap, as in Figure 7-5, both classes might have nonzero probabilities.

In practice, eventually we always have to convert probabilities into a decision: is this egg fertilized or not? Our final decision is influenced by the computer's prediction, but ultimately, we also need to account for human factors and what the decision means for us.

2D Multiclass Classification

Our eggs are selling great, but there's a problem. We've only been distinguishing between fertilized and unfertilized eggs. As we learn more about eggs, we discover that there are two different ways an egg ends up as unfertilized. An egg that is unfertilized because it was never fertilized is called a *yolker*. These are good for eating. Fertilized eggs we can sell to other farmers are called *winners*. But in some fertilized eggs the developing embryo stopped growing for some reason and died. Such an egg is called a *quitter* (Arcuri 2016). We don't want to sell quitters, because they can burst unexpectedly and spread harmful bacteria. We want to identify the quitters and dispose of them.

Now we have three classes of egg: winners (viable fertilized eggs), yolkers (safe unfertilized eggs), and quitters (unsafe fertilized eggs). As before, let's pretend that we can tell these three kinds of eggs apart just on the basis of their weight and length. Figure 7-7 shows a set of measured eggs, along with the classes we manually assigned to them by candling the eggs.

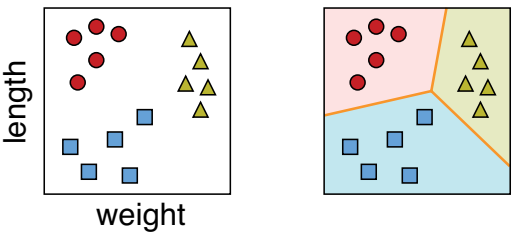


Figure 7-7: Left: Three classes of eggs. The red circles are fertilized. The blue squares are unfertilized but edible yolkers. The yellow triangles are quitters that we want to remove from our incubators. Right: Possible boundaries and regions for each class.

The job of assigning one of these three classes to new inputs is called *multiclass classification*. When we have multiple classes, we again find the boundaries between the regions associated with different classes. When a trained multiclass classifier has been released to the world and receives a new sample, it determines which region the sample falls into, and then assigns that sample the class corresponding to that region.

In our example, we might add in more features (or dimensions) to each sample, such as egg color, average circumference, and the time of day the

egg was laid. This would give us a total of five numbers per egg. Five dimensions is a weird place to think about, and we definitely can't draw a useful picture of it. But we can reason by analogy to the situation we can picture. In 2D, our data points tended to clump together in their locations, allowing us to draw boundary lines (and curves) between them. In higher dimensions, the same thing is true for the most part (we discuss this idea more near the end of the chapter). Just as we broke up our 2D square into several smaller 2D shapes, each one for a different class, we can break up our 5D space into multiple, smaller 5D shapes. These 5D regions also each define a different class.

The math doesn't care how many dimensions we have, and the algorithms we build upon the math don't care, either. That's not to say that we, as people, don't care, because typically, as the number of dimensions goes up, the running time and memory consumption of our algorithms goes up as well. We'll come back to some more issues involved in working with high-dimensional data at the end of the chapter.

Multiclass Classification

Binary classifiers are generally simpler and faster than multiclass classifiers. But in the real world, most data have multiple classes. Happily, instead of building a complicated multiclass classifier, we can build a whole bunch of binary classifiers and combine their results to produce a multiclass answer. Let's look at two popular methods for this technique.

One-Versus-Rest

Our first technique goes by several names: *one-versus-rest (OvR)*, *one-versus-all (OvA)*, *one-against-all (OAA)*, or the *binary relevance* method. Let's suppose that we have five classes for our data, named with the letters A through E. Instead of building one complicated classifier that assigns one of these five labels, let's instead build five simpler, binary classifiers and name them A through E for the class each one focuses on. Classifier A tells us whether a given piece of data does or does not belong to class A. Because it's a binary classifier, it has one decision boundary that splits the space into two regions: class A and everything else. We can now see where the name "one-versus-rest" comes from. In this classifier, class A is the one, and classes B through E are the rest.

Our second classifier, named Classifier B, is another binary classifier. This time it tells us whether a sample is, or is not, in class B. In the same way, Classifier C tells us whether a sample is or is not in class C, and Classifiers D and E do the same thing for classes D and E. Figure 7-8 summarizes the idea. Here we used an algorithm that takes into account all of the data when it builds the boundary for each classifier.

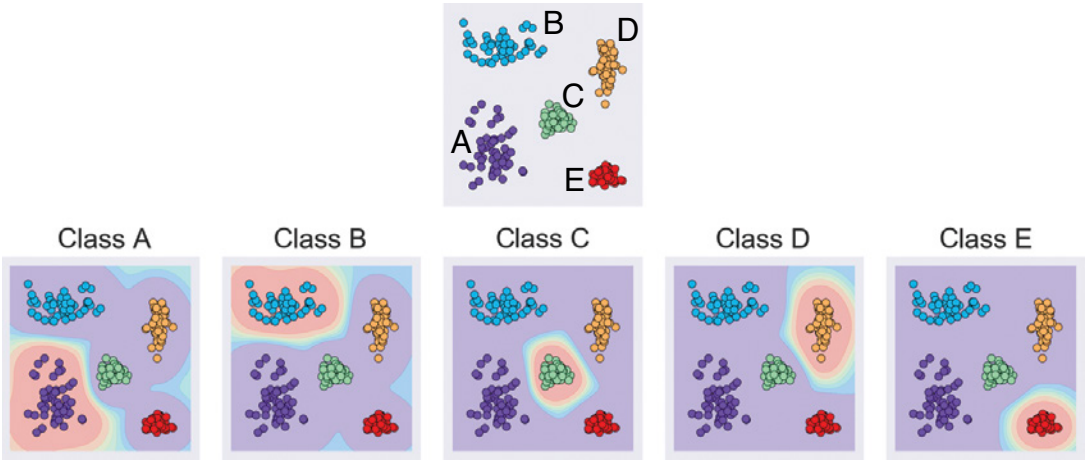


Figure 7-8: One-versus-rest classification. Top row: Samples from five different classes. Bottom row: The decision regions for five different binary classifiers. The colors from purple to pink show increasing probability that a point belongs to that class.

Note that some locations in the 2D space can belong to more than one class. For example, points in the upper-right corner have nonzero probabilities from classes A, B, and D.

To classify an example, we run it through each of our five binary classifiers in turn, getting back the probability that the point belongs to each class. We then find the class with the largest probability, and that's the class the point is assigned to. Figure 7-9 shows this in action.

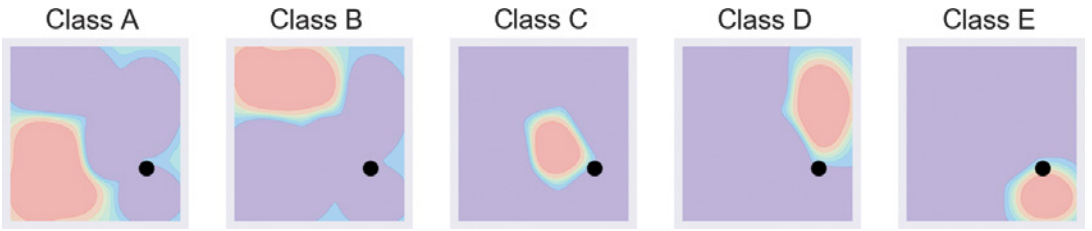


Figure 7-9: Classifying a sample using one-versus-rest. The new sample is the black dot.

In Figure 7-9, the first four classifiers all return low probabilities. The classifier for class E assigns the point a larger probability than the others, so the point is predicted to be from class E.

The appeals of this approach are its conceptual simplicity and its speed. The downside is that we have to teach (that is, learn boundaries for) five classifiers instead of just one, and we have to then classify every input sample five times to find which class it belongs to. When we have large numbers of classes with complex boundaries, the time required to run the sample through lots of binary classifiers can add up. On the other hand, we can run all five classifiers in parallel, so if we have the right equipment, we can

get our answers in the same amount of time it takes for any one of them. For any application, we need to balance the tradeoffs depending on our time, budget, and hardware constraints.

One-Versus-One

Our second approach that uses binary classifiers for multiple classes is called *one-versus-one (OvO)*, and it uses even more binary classifiers than OvR. The general idea is to look at every pair of classes in our data and build a classifier for just those two classes. Because the number of possible pairings goes up quickly as the number of classes increases, the number of classifiers in this method also grows quickly with the number of classes. To keep things manageable, let's work with only four classes this time, as shown in Figure 7-10.

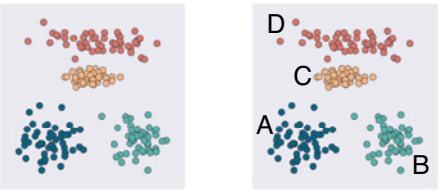


Figure 7-10: Left: Four classes of points for demonstrating OvO classification. Right: Names for the clusters.

We start with a binary classifier that is trained with data that is only from classes A and B. For the purposes of training this classifier, we simply leave out all the samples that are not labeled A or B, as if they don't even exist. This classifier finds a boundary that separates the classes A and B. Now every time we feed a new sample to this classifier, it tells us whether it belongs to class A or B. Since these are the only two options available to this classifier, it classifies every point in our dataset as either A or B, even when it's neither one. We'll soon see why this is okay.

Next, we make a classifier trained with only data from classes A and C and another for classes A and D. The top row of Figure 7-11 shows this graphically. Now we keep going for all the other pairings, building binary classifiers that are trained with data only from classes B and C, and B and D, as in the second row of Figure 7-11. Finally, we get to the last pairing of classes C and D, in the bottom row of Figure 7-11. The result is that we have six binary classifiers, each of which tells us which of two specific classes the data most belongs to.

To classify a new example, we run it through all six classifiers, and then we select the label that comes up the most often. In other words, each classifier votes for one of two classes, and we declare the winner to be the class with the most votes. Figure 7-12 shows one-versus-one in action.

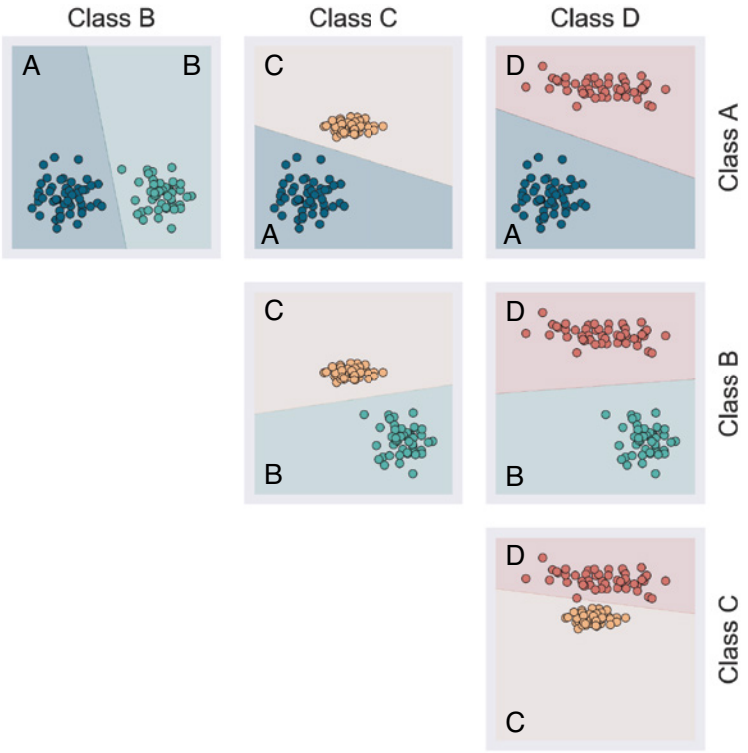


Figure 7-11: Building the six binary classifiers we use for performing OvO classification on four classes. Top row: Left to right, binary classifiers for classes A and B, A and C, and A and D. Second row: Left to right, binary classifiers for classes B and C, and B and D. Bottom row: A binary classifier for classes C and D.

In this example, class A got three votes, B got one, C got two, and D got none. The winner is A, so that's the predicted class for this sample.

One-versus-one usually requires far more classifiers than one-versus-rest, but it's sometimes appealing because it provides a clearer understanding of how the sample was evaluated with respect to each possible pair of classes. This can make a system more transparent, or explainable, when we want to know how it came up with its final answer. When there's a lot of messy overlap between multiple classes, it can be easier for us humans to understand the final results using one-versus-one.

The cost of this clarity is significant. The number of classifiers that we need for one-versus-one grows very fast as the number of classes increases. We've seen that with 4 classes we'd need 6 classifiers. Beyond that, Figure 7-13 shows just how quickly the number of binary classifiers we need grows with the number of classes. With 5 classes we'd need 10, with 20 classes we'd need 190, and to handle 30 classes we'd need 435 classifiers! Past about 46 classes we need more than 1,000.

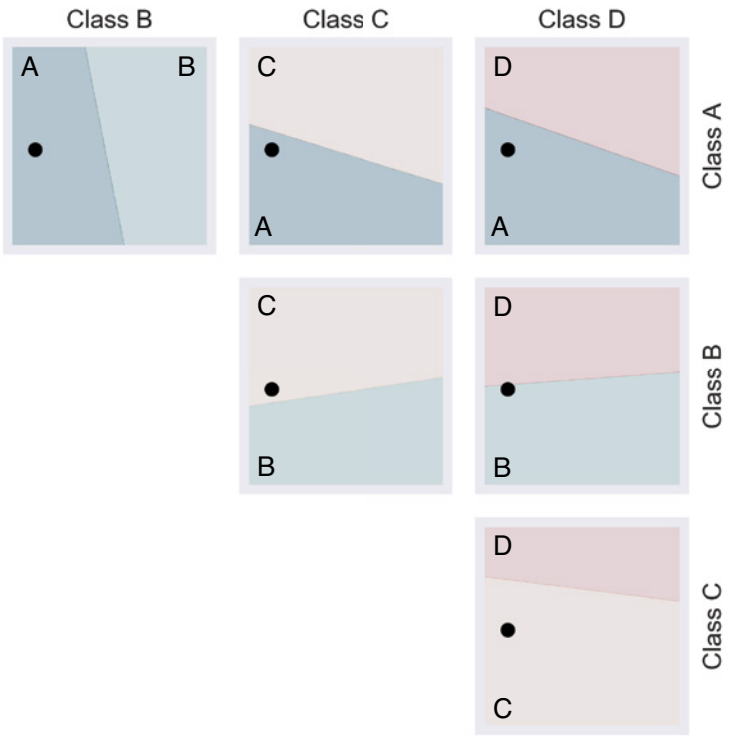


Figure 7-12: OvO in action, classifying a new sample shown as a black dot. Top row: Left to right, votes are A, A, and A. Second row: Left to right, votes are C and B. Bottom row: Vote is class C.

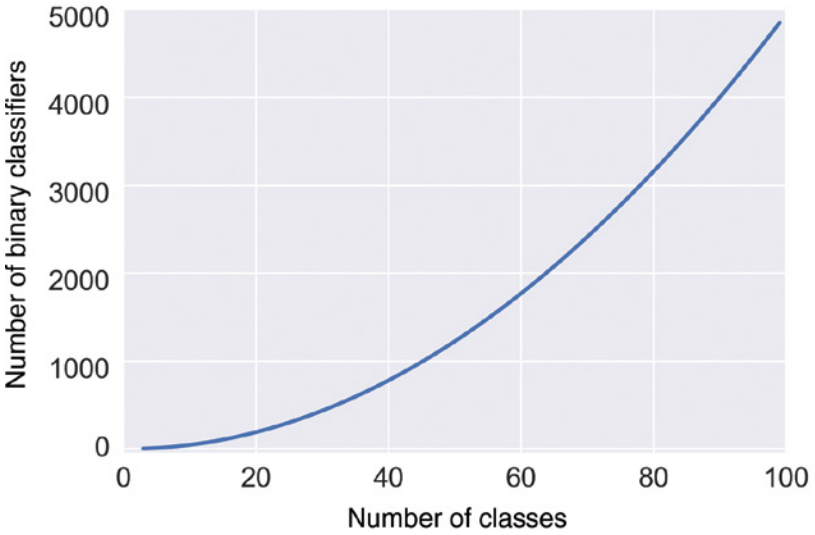


Figure 7-13: As we increase the number of classes, the number of binary classifiers we need for OvO grows very quickly.

Each of these binary classifiers has to be trained, and then we need to run every new sample through every classifier, which is going to consume a lot of computer memory and time. At some point, it becomes more efficient to use a single, complex, multiclass classifier.

Clustering

We've seen that one way to classify new samples is to break up the space into different regions and then test a point against each region. A different way to approach the problem is to group the training set data itself into *clusters*, or similar chunks. Let's suppose that our data has associated labels. How can we use those to make clusters?

In the left image of Figure 7-14 we have data with five different labels, shown by color. For these nicely separated groups, we can make clusters just by drawing a curve around each set of points, as in the middle image. If we extend those curves outward until they hit one another so that each point in the grid is colored by the cluster that it's closest to, we can cover the entire plane as in the right image.

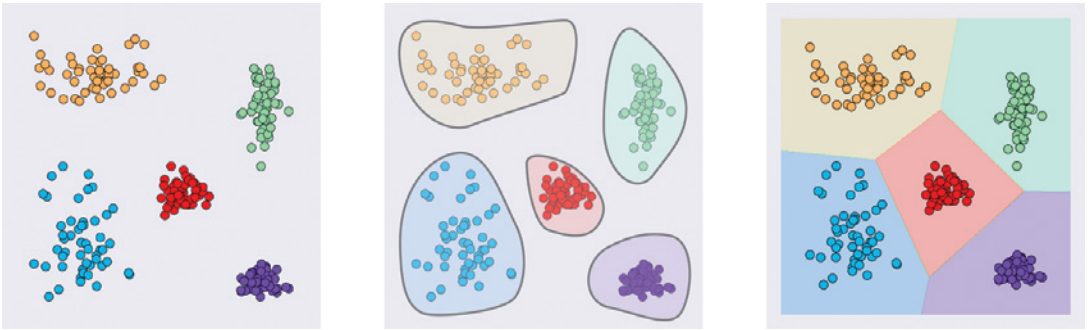


Figure 7-14: Growing clusters. Left: Starting data with five classes. Middle: Identifying the five groups. Right: Growing the groups outward so that every point has been assigned to one class.

This scheme required that our input data had labels. What if we don't have labels? If we can somehow automatically group our unlabeled data into clusters, we can still apply the technique we just described.

Recall that problems that involve data without labels fall into the type of learning we call *unsupervised learning*.

When we use an algorithm to automatically derive clusters from unlabeled data, we have to tell it how many clusters we want it to find. This "number of clusters" value is often represented with the letter k (this is an arbitrary letter and doesn't stand for anything in particular). We say that k is a *hyperparameter*, or a value that we choose before training our system. Our chosen value of k tells the algorithm how many regions to build (that is, how many classes to break up our data into). Because the algorithm uses the geometric means, or averages, of groups of points to develop the clusters, the algorithm is called *k-means clustering*.

The freedom to choose the value of k is a blessing and a curse. The upside of having this choice is that if we know in advance how many clusters there ought to be, we can say so, and the algorithm produces what we want. Keep in mind that the computer doesn't know where we think the cluster boundaries ought to go, so although it chops things up into k pieces, they may not be the pieces we're expecting. But if our data is well separated, so samples are clumped together with big spaces between the clumps, we usually get what we expect. The more the cluster boundaries get fuzzy, or overlap, the more things can potentially surprise us.

The downside of specifying k upfront is that we may not have any idea of how many clusters best describe our data. If we pick too few clusters, then we don't separate our data into the most useful distinct classes. But if we pick too many clusters, then we end up with similar pieces of data going into different classes.

To see this in action, consider the data in Figure 7-15. There are 200 unlabeled points, deliberately bunched up into five groups.

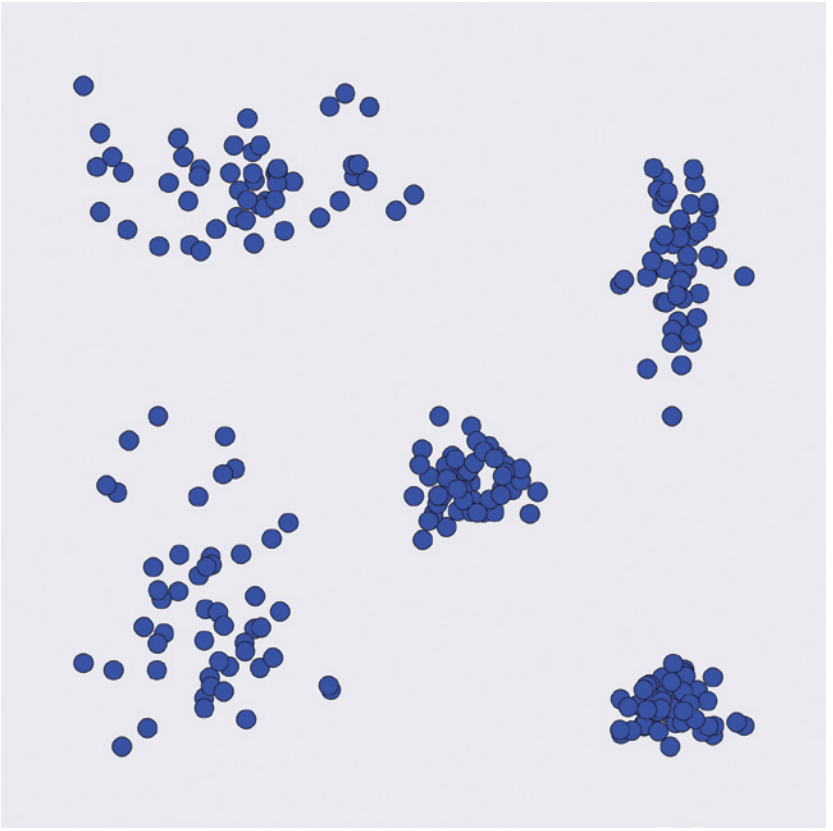


Figure 7-15: A set of 200 unlabeled points. They seem to visually fall into five groups.

Figure 7-16 shows how k-means clustering splits up this set of points for different values of k. Remember, we give the algorithm the value of k as an argument before it begins working.

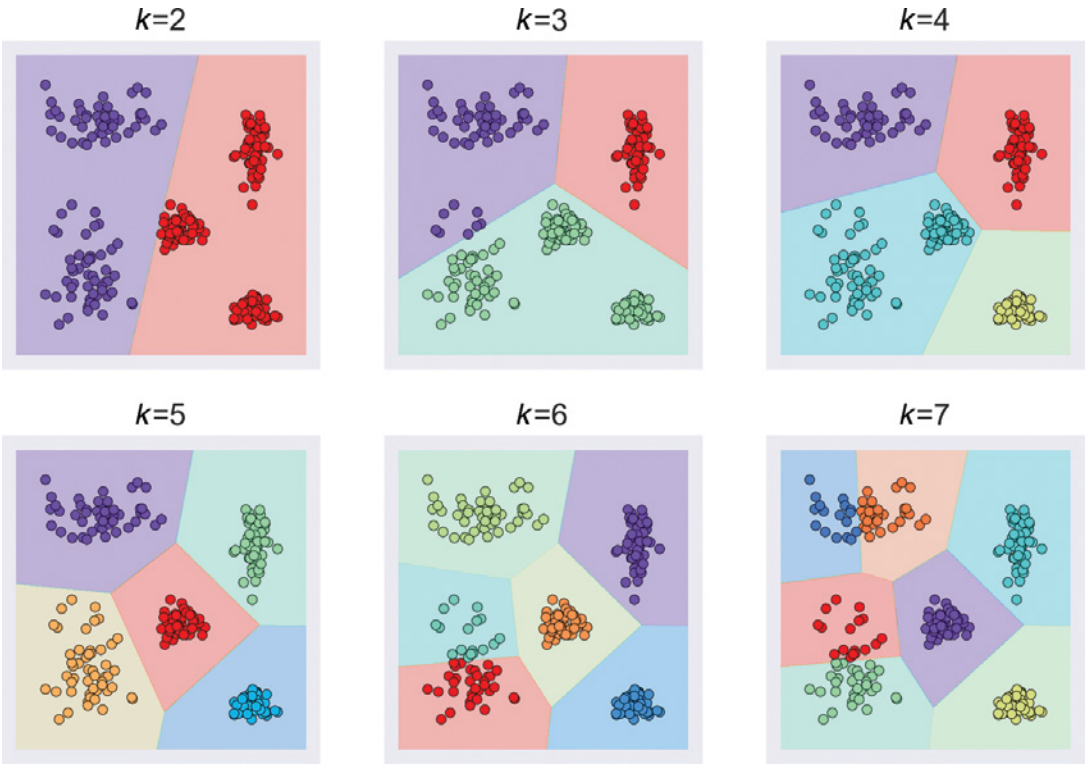


Figure 7-16: The result of automatically clustering our data in Figure 7-15 for values of k *from 2 to 7*

It's no surprise that $k = 5$ is doing the best job on this data, but we're using a cooked example in which the boundaries are easy to see. With more complicated data, particularly if it has more than two or three dimensions, it can be all but impossible for us to easily identify the most useful number of clusters beforehand.

All is not lost, though. A popular option is to train our clustering model several times, each time using a different value for k. By measuring the quality of the results, this *hyperparameter tuning* lets us automatically search for a good value of k, evaluating the predictions of each choice and reporting the value of k that performed the best. The downside, of course, is that this takes computational resources and time. This is why it's so useful to *preview* our data with some kind of visualization tool prior to clustering. If we can pick the best value of k right away, or even come up with a range of likely values, it can save us the time and effort of evaluating values of k that won't do a good job.

The Curse of Dimensionality

We've been using examples of data with two features, because two dimensions are easy to draw on the page. But in practice, our data can have any number of features or dimensions. It might seem that the more features we

have, the better our classifiers will be. It makes sense that the more features the classifiers have to work with, the better they should be able to find the boundaries (or clusters) in the data.

That's true to a point. Past that point, adding more features to our data actually makes things *worse*. In our egg-classifying example, we could add in more features to each sample, like the temperature at the time the egg was laid, the age of the hen, the number of other eggs in the nest at the time, the humidity, and so on. But, as we'll see, adding more features often makes it harder, not easier, for the system to accurately classify the inputs.

This counterintuitive idea shows up so frequently that it has earned a special name: *the curse of dimensionality* (Bellman 1957). This phrase has come to mean different things in different fields. We're using it in the sense that applies to machine learning (Hughes 1968). Let's see how this curse comes about, and what it tells us about training.

Dimensionality and Density

One way to picture the curse of dimensionality is to think about how a classifier goes about finding a boundary curve or surface. If there are only a few points, then the classifier can invent a huge number of curves or surfaces that all divide the data. In order to pick the boundary that will do the best job on future data, we'd want more training samples. Then the classifier can choose the boundary that best separates that denser collection. Figure 7-17 shows the idea visually.

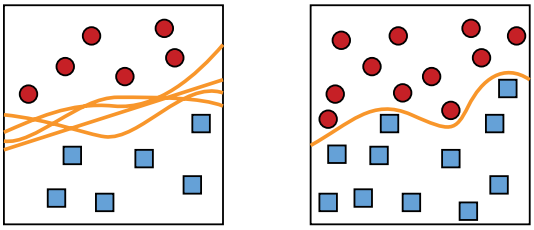


Figure 7-17: To find the best boundary curve, we need a good density of samples. Left: We have very few samples, so we can construct lots of different boundary curves. Right: With a higher density of samples, we can find a good boundary curve.

As Figure 7-17 shows, finding a good curve requires a dense collection of points. But here's the key insight: as we add dimensions (or features) to our samples, the number of samples we need in order to maintain a reasonable density in the sample space explodes. If we can't keep up with the demand, a classifier does its best, but it doesn't have enough information to draw a good boundary. It is stuck in the situation of the left diagram in Figure 7-17, where it's just guessing at the best boundary, which may lead to poor results on future data.

Let's look at the problem of loss of density using our egg example. To keep things simple, let's use a convention that all the features we may measure for our eggs (their volume, length, etc.) lie in the range 0 to 1. Let's start with a dataset that contains 10 samples, each with one feature (the

egg's weight). Since we have one dimension describing each egg, we can plot it on a one-dimensional line segment from 0 to 1. Because we want to see how well our samples cover every part of this line, let's break it up into pieces, or bins, and see how many samples fall into each bin. The bins are just conceptual devices to help us estimate density. Figure 7-18 shows how one set of data might fall on an interval [0,1] with 5 bins.

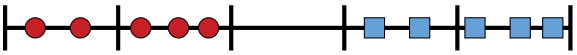


Figure 7-18: Our 10 pieces of data have one dimension each.

There's nothing special about the choices of 10 samples and 5 bins. We just chose them because it makes the pictures easy to draw. The core of our discussion is unchanged if we pick 300 eggs or 1,700 bins.

The *density* of our space is the number of samples divided by the number of bins. It gives us a rough way to measure how well our data is filling up the space of possible values. In other words, do we have examples to learn from for most values of the input? We can see problems start to creep in if we have too many empty bins. In this case, the density is 10 / 5 = 2, telling us each bin (on average) has 2 samples. Looking at Figure 7-18, we see that this is a pretty good estimate for the average number of samples per bin. In one dimension, for this data, a density of 2 lets us find a good boundary.

Let's now include the weight in each egg's description. Because we now have two dimensions, we can pull our line segment of Figure 7-18 upward to make a 2D square, as in Figure 7-19.

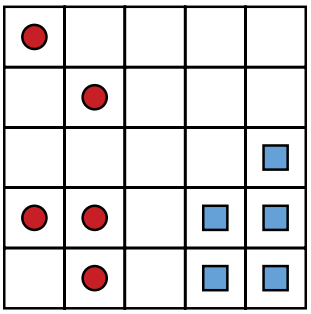


Figure 7-19: Our 10 samples are now each described by two measurements, or dimensions.

Breaking up each of the sides into 5 segments, as before, we have 25 bins inside the square. But we still have only 10 samples. That means most of the regions won't have any data. The density now is 10 / (5 × 5) = 10 / 25 = 0.4, a big drop from the density of 2 we had in one dimension. As a result, we can draw lots of different boundary curves to split the data of Figure 7-19.

Now let's add a third dimension, such as the temperature at the time the egg was laid (scaled to a value from 0 to 1). To represent this third dimension, we push our square back into the page to form a 3D cube, as in Figure 7-20.

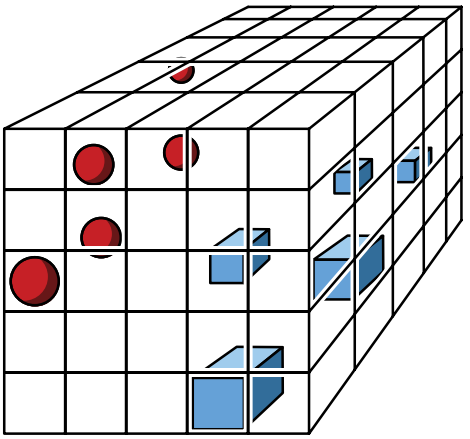


Figure 7-20: Now our 10 pieces of data are represented by three measurements, so we draw them in a 3D space.

By splitting each axis into 5 pieces, we now have 125 little cubes, but we still have only 10 samples. Our density has dropped to $10 / (5 \times 5 \times 5) = 10 / 125 = 0.08$. In other words, the average cell contains 0.08 samples. We've gone from a density of 2, to 0.4, to 0.08. No matter where the data is located in 3D, the vast majority of the space is empty.

Any classifier that splits this 3D data into two pieces with a boundary surface is going to have to make a big guess. The issue isn't that it's hard to separate the data, but rather that it's too easy. It's not clear how to best separate the data so that our system will *generalize*, that is, correctly classify points we get in the future. The classifier is going to have to classify lots of empty boxes as belonging to either one class or the other, and it just doesn't have enough information to do that in a principled way.

In other words, who knows where new samples are going to end up once our system is deployed? At this point, nobody. Figure 7-21 shows one guess at a boundary surface, but as we saw in Figure 7-17, we can fit all kinds of planes and curvy sheets through the big open spaces between the two sets of samples. Most of them are probably not going to generalize very well. Maybe this one is too close to the red balls. Or maybe it's not close enough. Or maybe it should be a curved surface rather than a plane.

The expected low quality of this boundary surface when we use it to predict the classes for new data is not the fault of the classifier. This plane is a perfectly good boundary surface, given the data available to the classifier. The problem is that because the density of the samples is so low, the classifier just doesn't have enough information to do a good job. The density

drops like a rock with each added dimension, and as we add yet more features, the density continues to plummet.

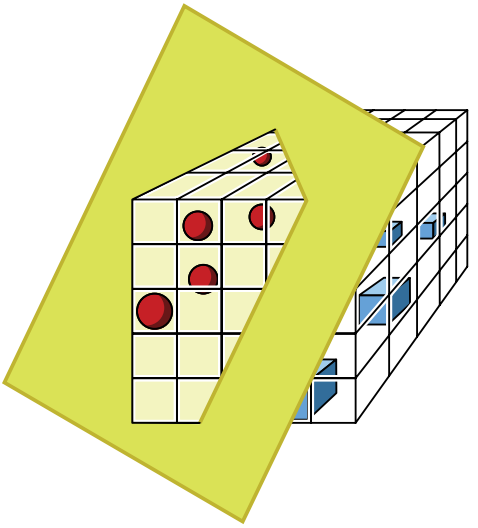


Figure 7-21: Passing a boundary surface through our cube, separating the red and blue samples

It's hard to draw pictures for spaces with more than three dimensions, but we can calculate their densities. Figure 7-22 shows a plot of the density of a space for our 10 samples as the number of dimensions goes up. Each curve corresponds to a different number of bins for each axis. Notice that as the number of dimensions goes up, the density drops to 0 no matter how many bins we use.

If we have fewer bins, we have a better chance of filling each one, but before long, the number of bins becomes irrelevant. As the number of dimensions goes up, our density always heads to 0. This means our classifier ends up guessing at where the boundary should be. Including more features with our egg data improves the classifier for a while, because the boundary is better able to follow where the data is located. But eventually we need enormous amounts of data to keep up with the density demands made by those new features.

There are some special cases where the lack of density resulting from new features won't cause problems. If our new features are redundant, then our existing boundaries are already fine and don't need to change. Or if the ideal boundary is simple, like a plane, then increasing the number of dimensions doesn't change that boundary's shape. But in the general case, new features add refinement and details to our boundary. Because adding new dimensions to accommodate those features causes the density to drop toward 0, that boundary becomes harder to find, so the computer ends up essentially guessing at its shape and location.

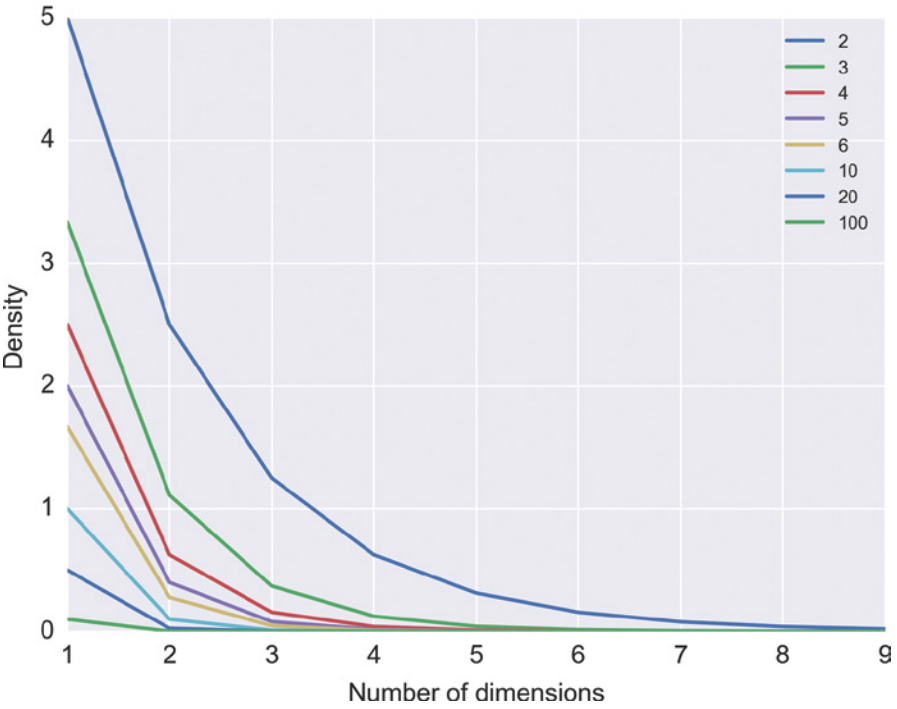


Figure 7-22: This is how the density of our points drops off as the number of dimensions increases for a fixed number of samples. Each of the colored curves shows a different number of bins along each axis.

The curse of dimensionality is a serious problem, and it could have made all attempts at classification fruitless. After all, without a good boundary, a classifier can't do a good job of classifying. What saved the day is the *blessing of non-uniformity* (Domingos 2012), which we prefer to think of as the *blessing of structure*. This is the name for the observation that, in practice, the features that we typically measure, even in very high-dimensional spaces, tend not to spread around uniformly in the space of the samples. That is, they're not distributed equally across the line, square, and cube we've seen, or the higher-dimensional versions of those shapes we can't draw. Instead, they're often clumped in small regions, or spread out on much simpler, lower-dimensional surfaces (such as a bumpy sheet or a curve). That means our training data and all future data will generally land in the same regions. Those regions will be dense, while the great majority of the rest of the space will be empty. This is good news, because it says that it doesn't matter how we draw the boundary surface in those big empty regions, because no data will appear there. Having good density where the samples themselves actually fall is what we want and what we usually find.

Let's see this structuring in action. In our cube of Figure 7-20, instead of having the samples spread around more or less uniformly throughout

the cube, we might find that the samples from each class are located on the same horizontal plane. That means that any roughly horizontal boundary surface that splits the groups probably does a good job on new data, too, as long as those new values also tend to fall into those horizontal planes. Figure 7-23 shows the idea.

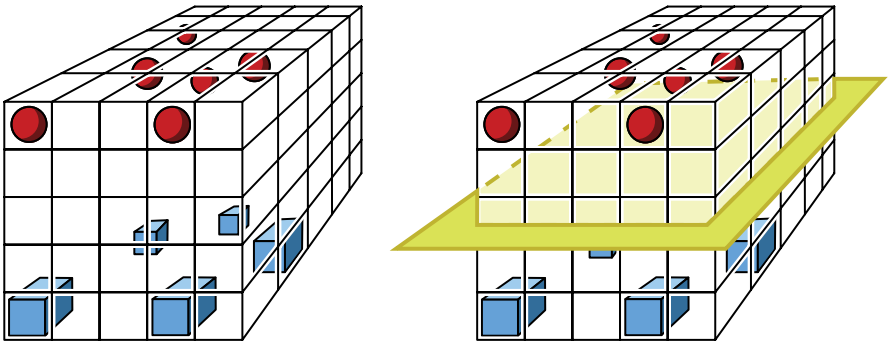


Figure 7-23: In practice, our data often has some structure in the sample space. Left: Each group of samples is mostly in the same horizontal slice of the cube. Right: A boundary plane passed between the two sets of points.

Most of the cube in Figure 7-23 is empty, and thus has low density, but the parts we're interested in have high densities, so we can find a sensible boundary. Although the curse of dimensionality dooms us to have a low density in our overall space, even with enormous amounts of data, the blessing of structure says that we usually get reasonably high density where we need it. On the right side of Figure 7-23, we show a boundary plane across the middle of the cube. This does the job of separating the classes, but since they're so well clumped, and since the space between them is empty, almost any boundary surface that splits the groups would probably do a good job of generalizing.

Note that both the curse and blessing are empirical observations, and not hard facts we can always rely on. So the best solution for this important practical problem is usually to fill out the sample space with as much data as we can get. In Chapter 10, we'll see some ways to reduce the number of features in our data if it seems to be producing bad boundaries.

The curse of dimensionality is one reason why machine learning systems are famous for requiring enormous amounts of data when they're being trained. If the samples have lots of features (or dimensions), we need lots and lots of samples to get enough density to make good class predictions.

Suppose we want enough points to get a specific density, say 0.1 or 0.5. How many points do we need as the number of dimensions increases? Figure 7-24 shows that the number of points needed explodes in a hurry.

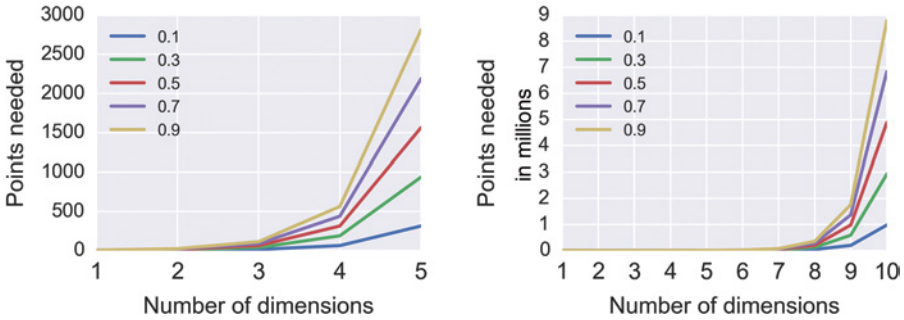


Figure 7-24: The number of points we need to achieve different densities, assuming we have five divisions on each axis

Speaking generally, if the number of dimensions is low, and we have lots of points, then we probably have enough density for the classifier to find a surface that has a good chance of generalizing well to new data. The values for *low* and *lots* in that sentence depend on the algorithm we're using and what our data looks like. There's no hard and fast rule for predicting these values; we generally take a guess, see what performance we get, and then make adjustments. Generally speaking, when it comes to training data, more really is more. Get all the data you ethically can.

High-Dimensional Weirdness

Since the samples in real-world data usually have many features, we often work in spaces with many dimensions. We saw earlier that if the data is locally structured, we are often okay: our ignorance of the boundary in the big empty regions doesn't work against us, since no input data lands there. But what if our data isn't structured or clumped?

It's tempting to look at pictures like Figures 7-19 and 7-20 and let our intuition guide our choices in designing a learning system, thinking that spaces of many dimensions are like the spaces we're used to, only bigger. This is very much not true! There's a technical term for the characteristics of high-dimensional spaces: *weird*. Things simply work out in ways that we don't expect. Let's look at two cautionary tales about the weirdness of geometry in higher dimensions in order to train our intuitions not to jump to generalizations from the low-dimensional spaces we're familiar with. This will help us stay on our toes when designing learning systems.

The Volume of a Sphere Inside a Cube

A famous example of the weirdness of high-dimensional spaces involves the volume of a sphere inside a cube (Spruyt 2014). The setup is simple: take a sphere and put it into a cube, then measure how much of the cube is occupied by the sphere. Let's first do this in one, two, and three dimensions. Then we'll keep going to higher dimensions and track the percentage of the cube occupied by the sphere as the number of dimensions goes up.

Figure 7-25 gets us started in the 1D, 2D, and 3D cases.

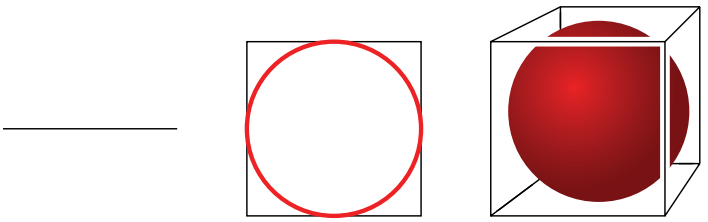


Figure 7-25: Spheres in cubes. Left: A 1D "cube" is a line segment, and the "sphere" covers it completely. Middle: A 2D "cube" is a square, and the "sphere" is a circle that touches the edges. Right: A 3D cube surrounds a sphere, which touches the center of each face.

In one dimension, our "cube" is just a line segment, and the sphere is a line segment that covers the whole thing. The ratio of the contents of the sphere to the contents of the "cube" is 1:1. In 2D, our "cube" is a square, and the sphere is a circle that just touches the center of each of the four sides. The area of the circle divided by the area of the box is about 0.8. In 3D, our cube is a normal 3D cube, and the sphere fits inside, just touching the center of each of the six faces. The volume of the sphere divided by the volume of the cube is about 0.5.

The amount of space taken up by the sphere relative to the box enclosing it is dropping. If we work out the math and calculate the volume of the sphere to the volume of its cube for higher dimensions (where they're called a *hypersphere* and *hypercube*), we get the results in Figure 7-26.

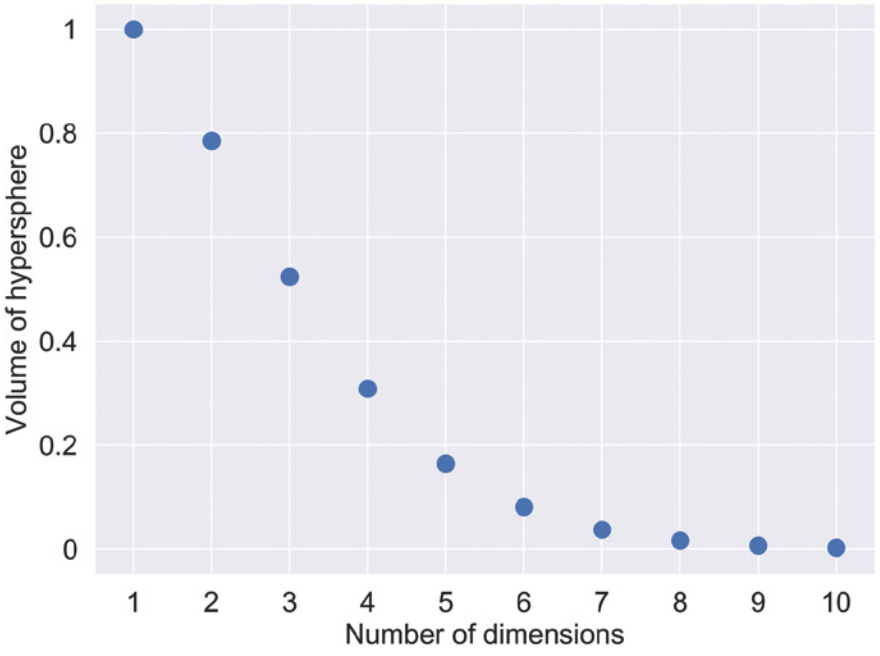


Figure 7-26: The amount of a box occupied by the largest sphere that fits into that box for different numbers of dimensions

The amount of volume taken up by the hypersphere drops down toward 0. By the time we reach 10 dimensions, the largest sphere we can fit into its enclosing box takes up almost none of the box's volume!

This is not what most of us would expect based on our experience in the 3D world. We've found that if we take a hypercube (of many dimensions) and place inside of it the largest possible hypersphere (of the same number of dimensions) that will fit, the volume of that hypersphere is just about 0.

There's no trick to this, and nothing's going wrong. When we work out the math, this is what happens. We saw the pattern starting in the first three dimensions, but we can't really draw the rest, so it's hard to picture how in the heck this can be going on. But it does happen this way, because higher dimensions are weird.

Packing Hyperspheres into Hypercubes

To make sure our intuitions get the message, let's look at another weird result from packing hyperspheres into hypercubes. Let's suppose we want to transport some particularly fancy oranges and make sure they're not damaged in any way. Each orange's shape is close to spherical, so we decide to ship them in cubical boxes protected by air-filled balloons. We put one balloon in each corner of the box, so that each balloon touches both the orange and the sides of the box. The balloons and orange are all perfect spheres. What's the biggest orange we can put into a cube of a given size?

We want to answer this question for cubes (and balloons and oranges) with any number of dimensions, so let's start with just two dimensions. Our box is then a 2D square of size 4 by 4, and the four balloons are each a circle of radius 1, placed in the corners, as in Figure 7-27.

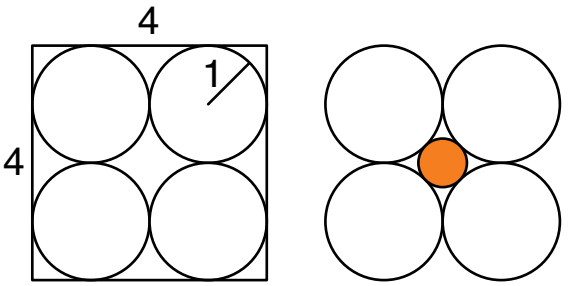


Figure 7-27: Shipping a circular 2D orange in a square box, surrounded by circular balloons in each corner. Left: The four balloons each have a radius of 1, so they fit perfectly into a square box of side 4. Right: The orange sitting inside the balloons.

Our orange in this 2D diagram is also a circle. In Figure 7-27 we show the biggest orange we can fit. A little geometry tells us that the radius of this orange circle is about 0.4.

Now let's move to 3D, so we have a cube (again, 4 units on a side). We can now fit eight spherical balloons, each of radius 1, into the corners, as in Figure 7-28. As always, the orange goes into the space in the middle of the balloons.

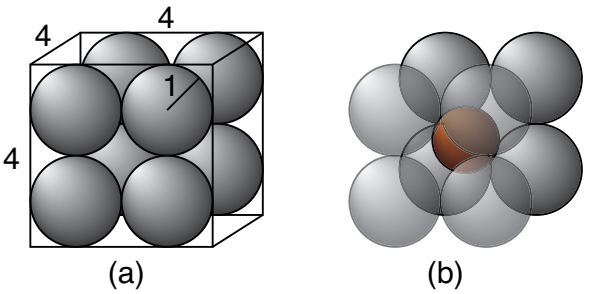


Figure 7-28: Shipping a spherical orange in a cubical box surrounded by spherical balloons in each corner. Left: The box is 4 by 4 by 4, and the eight balloons each have a radius of 1. Right: The orange sits in the middle of the balloons.

Another bit of geometry (this time in 3D) tells us that this orange has a radius of about 0.7. This is larger than the radius of the orange in the 2D case, because in 3D, there's more room for the orange in the central gap between spheres.

Let's take this scenario into 4, 5, 6, and even more dimensions, where we have hypercubes, hyperspheres, and *hyperoranges*. For any number of dimensions, our hypercubes are always 4 units on every side, there is always a hypersphere balloon in every corner of the hypercube, and these balloons always have a radius of 1.

We can write a formula that tells us the radius of the biggest hyperorange we can fit for this scenario in any number of dimensions (Numberphile 2017). Figure 7-29 plots this radius for different numbers of dimensions.

We can see from Figure 7-29 that in four dimensions, the biggest hyperorange we can ship has a radius of exactly 1. That means that it's as large as the hyperballoons around it. That's hard to picture, but it gets much stranger.

Figure 7-29 also tells us that in nine dimensions, the hyperorange has a radius of 2, so its diameter is 4. That means the hyperorange is as large as the box itself, like the 3D sphere in Figure 7-25. That's despite being surrounded by 512 hyperspheres of radius 1, each in one of the 512 corners of this 9D hypercube. If the orange is as large as the box, how are the balloons protecting anything?

But things get much crazier. At 10 dimensions and higher, the hyperorange has a radius that's *more* than 2. The hyperorange is now *bigger* than the hypercube that was meant to protect it. It seems to be extending beyond the sides of the cube, even though we constructed it to fit both inside the box and the protective balloons that are still in every corner. It's hard for those of us with 3D brains to picture 10 dimensions (or more), but the equations check out: the orange is simultaneously inside the box and extending beyond the box.

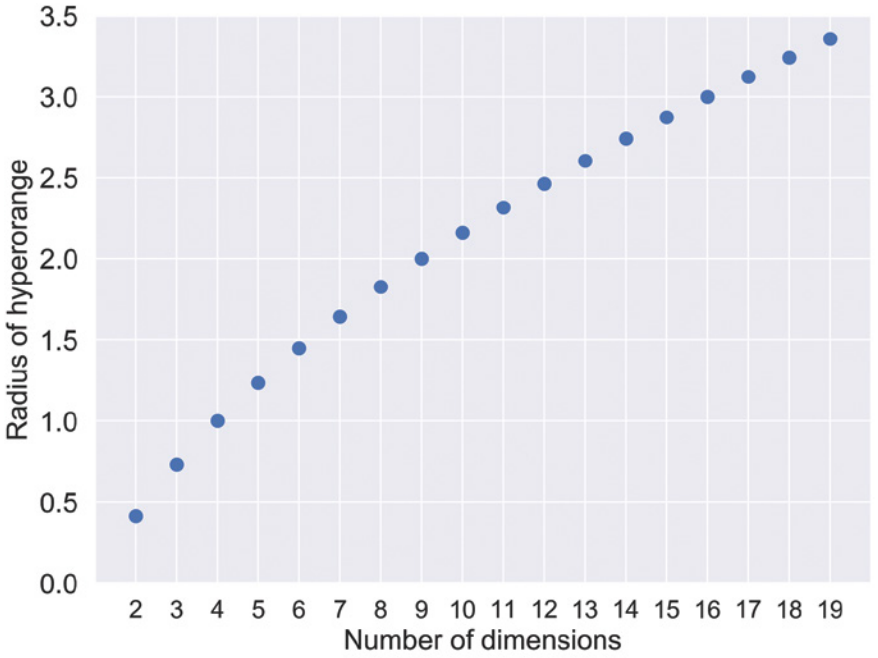


Figure 7-29: The radius of a hyperorange in a hypercube with sides 4, surrounded by hyperspheres of radius 1 in each corner of the cube

The moral here is that our intuition can fail us when we get into spaces of many dimensions (Aggarwal 2001). This is important because we routinely work with data that has tens or hundreds of features.

Any time we work with data that has more than three features, we've entered the world of higher dimensions, and we should not reason by analogy with what we know from our experience with two and three dimensions. We need to keep our eyes open and rely on math and logic, rather than intuition and analogy.

In practice, that means keeping a close eye on how a deep learning system is behaving during training when we have multidimensional data, and we should always be alert to seeing it act strangely. The techniques of Chapter 10 can reduce the number of features in our data, which may help. Throughout the book we'll see other ways to improve a system that isn't learning well, whether because of high-dimensional weirdness or other causes.

Summary

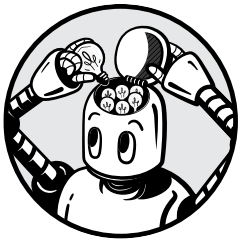
In this chapter we looked at the mechanics of classifying. We saw that classifiers can break up the space of data into domains that are separated by boundaries. A new piece of data is classified by identifying which domain it

falls into. The domains may be fuzzy, representing probabilities, so the result of the classifier is a probability for each class. We also looked at an algorithm for clustering. Lastly, we saw that our intuition often gets it wrong when we work in spaces of more than three dimensions. Things frequently just don't work the way we expect. Those higher-dimensional spaces are weird, and full of surprises. We learned that when working with multidimensional data, we should proceed carefully and monitor our system as it learns. We should never rely on guesswork based on our experience in 3D!

In the next chapter, we'll look at how to efficiently train a learning system, even when we don't have a lot of data.

8

TRAINING AND TESTING



In this chapter we'll look at *training*, the process of taking a system that's been initialized with default or random values and gradually improving it so that it's tuned to the data we want to understand. When we're done training, we can estimate how well our system will evaluate new data it hasn't seen before, a process known as *testing*.

We illustrate the ideas in this chapter using a supervised classifier, which we teach with labeled data. Most of the techniques we discuss are general and can be applied to almost all types of learners.

Training

When we train a classifier with supervised learning, every sample has an associated label describing the class we've manually assigned to it. The collection of all the samples we're going to learn from, along with their labels, is called the *training set*. We're going to present each of the samples in our training set to the classifier, one at a time. For each sample, we give the system the sample's features and ask it to predict its class.

If the prediction is correct (that is, it matches the label we assigned), then we move on to the next sample. If the prediction is wrong, we feed the classifier's output and the correct label back to the classifier. Using algorithms that we'll see in later chapters, we modify the classifier's internal parameters so that it's more likely to predict the correct label if it sees this sample again.

Figure 8-1 shows this idea visually. We use the classifier to get a prediction and compare that to the label. If they disagree, we update the classifier. Then we move on to the next sample.

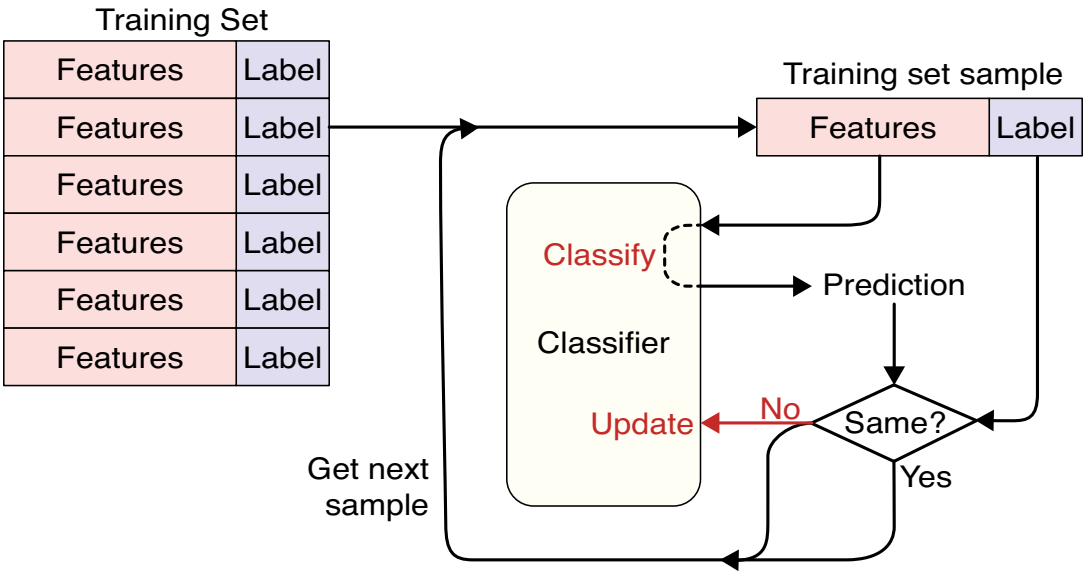


Figure 8-1: A block diagram of training a classifier

As our training process runs through this loop, one sample at a time, the classifier's internal variables are nudged toward values that do an increasingly good job of predicting labels. Each time we run through the entire training set, we say that we've trained for one *epoch*. We usually run the system through many epochs so the system sees every sample many times. Typically, we keep training as long as the system is still learning and improving its performance on the training data, but we might stop if we run out of time, or if we run into problems like those we discuss later in this chapter and in Chapter 9.

Let's now look at how to measure the classifier's accuracy at predicting correct labels.

Testing the Performance

We begin with a system whose parameters are initialized with random numbers. Then we teach it using the samples in the training data. Once the system has been *released*, or *deployed*, into the real world, it encounters new, *real-world data* (or *deployment data*, *release data*, or *user data*). We'd like to know how well our classifier will perform on real-world data before we deploy it. We may not need perfect accuracy, but we usually want the system to meet or exceed some quality threshold that we already have in mind. How can we estimate the quality of our system's predictions before it's released?

We need our system to do really well on the training data, but if we judge the system's accuracy based on just this data, we'll usually be misled. This is an important principle in practice, so let's look at it more detail.

Suppose we're going to use our supervised classifier to process pictures of dogs. For every image, it will assign a label identifying that dog's breed. Our goal is to put the system online so people can drag a picture of their dog onto their browser, and have it come back either with the dog's breed, or the catch-all "mixed breed."

To train our system, let's collect 1,000 photos of different purebred dogs, each labeled by an expert. Using Figure 8-1, we can show the system all 1,000 pictures, and then we show it all of them again, and again, over and over, one epoch after another. When doing so, we usually scramble the order of the images in each epoch so they don't always arrive in an identical sequence. If our system is designed well, it gradually starts producing more and more accurate results, until it's perhaps correctly identifying the breed of the dog in 99 percent of these training pictures.

This does *not* mean that our system is going to be 99 percent correct when we put it up on the web. The problem is that the system might be exploiting subtle relationships in the training data that aren't true for data in general. For example, suppose our images of poodles look like Figure 8-2.

When we assembled our training set, we didn't notice that all of the poodles had a little bob at the end of their tails and that none of the other dogs did. But the system noticed. That little idiosyncrasy in the data gave the system a way to easily classify poodles: instead of looking at the size of the dog's legs, the shape of its nose, and other features, the system could just look for the bob on the end of the tail. Using that rule, it would correctly classify all of our training images of poodles. We sometimes say that the system is doing what we asked for ("identify poodles"), but not what we want ("based on most of the features of the dog in the picture, determine if it's a poodle"). We often say that the system has learned to *cheat*, though that might be unfair. What it learned was a shortcut that gave us the results we asked for.

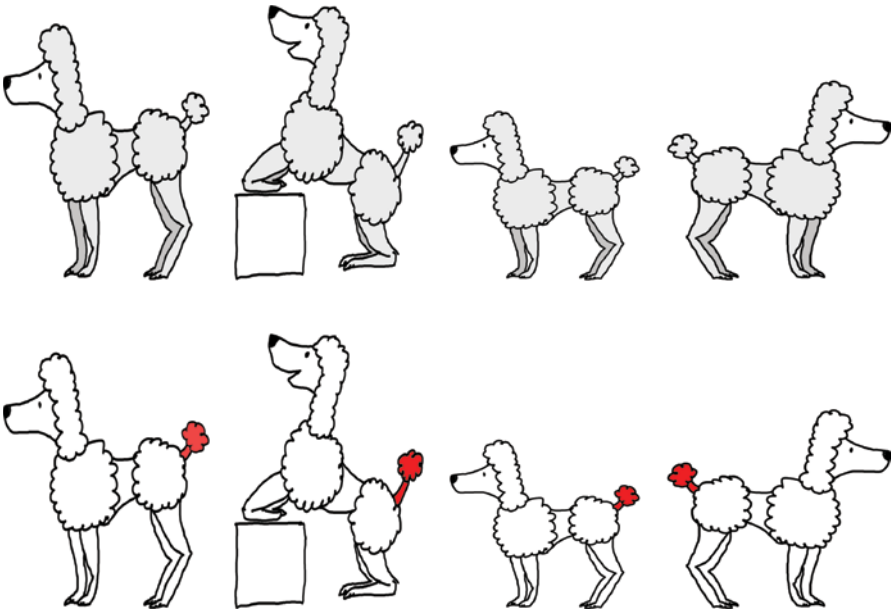


Figure 8-2: Training data for a system to identify dog breeds from pictures. Top row: Our input poodle images. Bottom row: The feature that our system learned to identify a photo as a poodle is highlighted in red.

For another example, suppose that all the pictures of Yorkshire terriers (or Yorkies) in our training data were taken when the dogs were sitting on a couch, as in Figure 8-3. We hadn't noticed this, nor another important fact: none of the other pictures had couches in them. The system may learn that if there's a couch in the image, it can immediately classify the image as a picture of a Yorkshire terrier. This rule works perfectly for our training data.

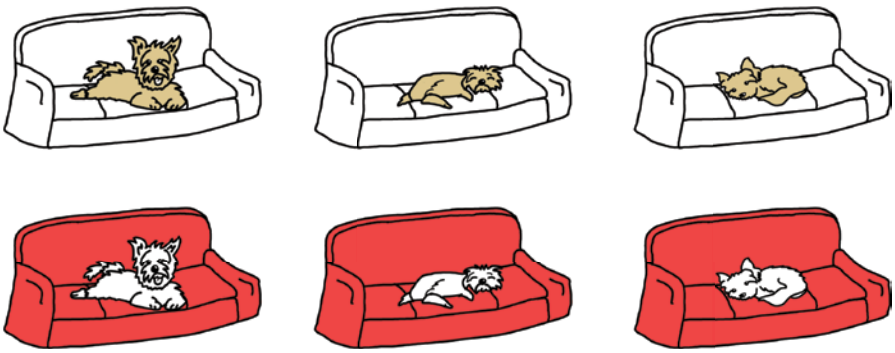


Figure 8-3: Top row: Three Yorkies on couches. Bottom row: Our system has learned to recognize the couch, shown in red.

Suppose we then deploy our system and someone submits a picture of a Great Dane standing in front of a holiday decoration of big white balls on a string, or their Siberian husky on a couch, as in Figure 8-4. Our system notices the white ball at the end of the Great Dane's tail and says that it's a

poodle, and it sees the couch, ignores the dog, and reports that the husky is a Yorkie.

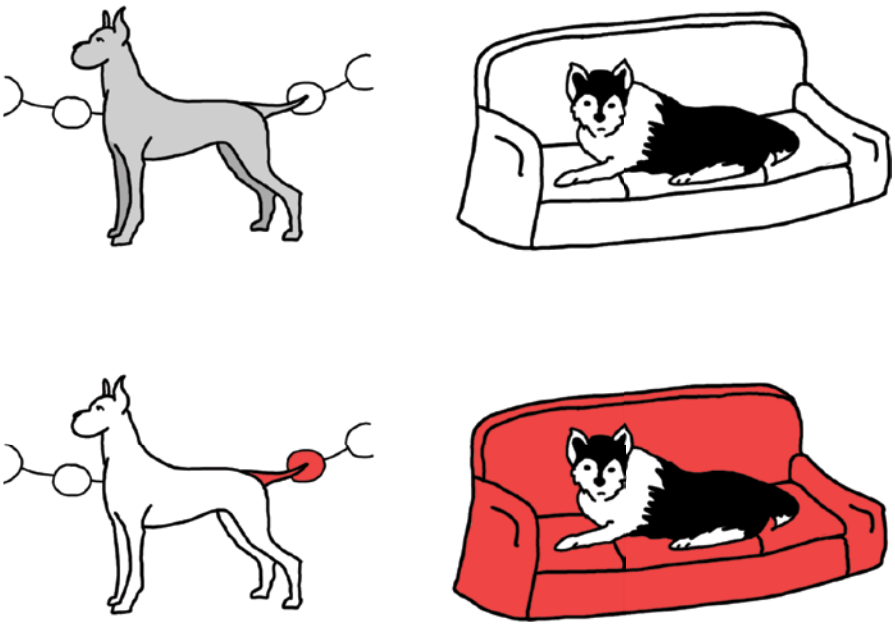


Figure 8-4: Top: A Great Dane is standing in front of a holiday display of white balls on a string, and a Siberian husky is lying on a couch. Bottom: The system sees the white ball on the end of the Great Dane's tail and tells us that the dog is a poodle, and it notices the couch and classifies the dog on it as a Yorkie.

This isn't just a theoretical concern. A famous example of this phenomenon describes a meeting in the 1960s where a presenter was demonstrating an early machine-learning system (Muehlhauser 2011). The details of the data are murky, but it seems that they had photos of stands of trees with a camouflaged tank in their midst, and stands of trees with no tank. The presenter claimed the system could pick out the image with the tank without fail. That would have been an incredible feat for the time.

At the end of the talk, an audience member stood up and observed that the photos with the tanks in them were all taken on sunny days, whereas the photos without the tanks were all taken on cloudy days. It seemed likely that the system had merely distinguished bright skies from dark skies, so the impressive (and accurate) results had nothing to do with tanks at all.

This is why looking at the performance on the training data isn't good enough to predict performance in the real world. The system might learn about some weird idiosyncrasies in the training data and then use that as a rule, only to be foiled by new data that doesn't happen to have those quirks. This is known formally as *overfitting*, though we often refer casually to it simply as *cheating*. We'll look at overfitting more closely in Chapter 9.

We've seen that we need some measure other than performance on the training set to predict how well our system is going to do if we deploy it. It

would be great if there was an algorithm or formula that would take our trained classifier and tell us how good it is, but there isn't. There's no way for us to know how our system will perform without trying it out and seeing. Like natural scientists who must run experiments to see what actually happens in the real world, we also must run experiments to see how well our systems perform.

Test Data

The best way anyone has found to determine how well a system will do on new, unseen data is to give it new, unseen data and see how well it does. There's no shortcut to this kind of experimental verification.

We call this unseen set of data points, or samples, the *test data* or a *test set*. Like the training data, we hope that the test data is representative of the data that we're going to see once our system has been released. The typical process is to train the system using the training data until it's doing as well as we think it can do. Then we evaluate it on the test data, and that tells us how well it's likely to do in the real world.

If the system's performance on the test data isn't good enough, we need to improve it. Since training on more data is almost always a good way to improve performance, it's a usually good idea to gather more data and train again. Another benefit of getting more data is that we can diversify our training set. For example, we might find dogs other than poodles with bobs on their tail, or we might find dogs other than Yorkies on couches. Then our classifier would have to find other ways to identify those dogs, and we'd avoid making mistakes due to overfitting.

An *essential* rule of the training and testing process is that *we never learn from the test data*. As tempting as it might be for us to put the test data into the training set so that the system has even more examples to learn from, doing so ruins the test data's value as an objective way to measure the accuracy of our system. The problem with learning from the test data is that it just becomes part of the training set. That means we're right back where we were before: the system can all too easily key in on idiosyncrasies in the test data. If we then use the test data to see how well the classifier works, it might predict the correct label for each sample, but it could be cheating. If we learn from the test data, it loses its special and valuable quality as a way to measure the performance of the system on new data.

For this reason, we split the test data off from the training data before we even begin to train, and we hold it aside. We only come back to the test data when training is over, and then we use it just one time to evaluate the quality of our system. If the system doesn't do well enough on the test set, we can't just train some more and then test again. Think of the test set as being like the final exam questions in a class: once they've been seen, they can't be used again. If our system doesn't perform well on the test data, we must start all over again with a system initialized with random values. Then we can train with more data, or for a longer time period. When training is done, we can use the test set again, because this newly trained system has never seen it before. If it again doesn't perform well enough, we must start our training all over.

This is important enough to repeat: we must never let the system see the test data in any way prior to its single use when training has completed.

The problem of accidentally learning from the test data has its own name: *data leakage*, also called *data contamination*, or *contaminated data*. We have to constantly look out for this, because as our training procedures and classifiers become more sophisticated, data leakage can sneak in wearing different (and hard-to-notice) disguises. Data leakage can be avoided by practicing *data hygiene*: always make sure the test data is kept separate and that it is only used once, when training has been completed.

We often create the test data by splitting our original data collection into two pieces: the training set and the test set. We commonly set up this split to give about 75 percent of the samples to the training set. Often samples are chosen randomly for each set, but more sophisticated algorithms can try to make sure that each collection is a good approximation of the complete input data. Most machine-learning libraries offer routines to perform this splitting process for us.

Figure 8-5 shows the idea.

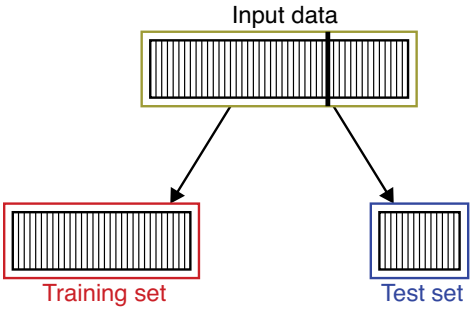


Figure 8-5: Splitting our input examples into a training set and a test set. The split is often about 75:25 or 70:30.

Validation Data

In our discussion up to this point, we trained the system for a while, then we stopped and evaluated its performance using the test set. If the performance wasn't good enough, we started training all over again.

There's nothing wrong with that strategy except that it is a slow way to work. In practice, we often want a rough estimate of the system's performance as we go along, so we can stop training when we think the system is going to give us the performance we want from the test set.

To make this estimate, we split the input data into three sets, rather than the two we've seen so far. We call this new set the *validation data*, or *validation set*. The validation data is yet another chunk of data that's meant to be a good proxy of the real-world data we'll see when we deploy the system. We typically make these three sets by assigning about 60 percent of the original data to the training set, 20 percent to the validation set, and the remaining 20 percent to the test set. Figure 8-6 shows the idea.

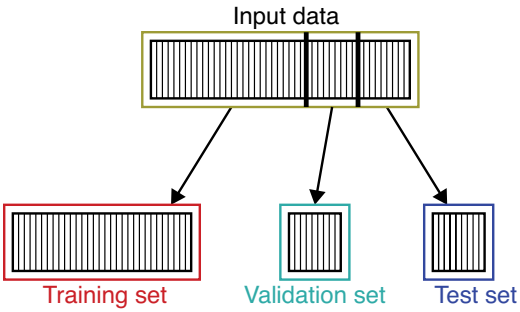


Figure 8-6: Splitting our input data into a training set, a validation set, and a test set

Our new process will be to train the system for an epoch, running through the entire training set, and then we estimate its performance by asking it to make predictions for the validation set. We do this after every epoch, so we're reusing the validation set. This causes data leakage, but we only use the validation data for informal estimates. We use the system's performance on the validation set to get a general sense of how well it's learning over time. When we think the system is doing well enough to deploy, we use the one-time test set to get a reliable performance estimate.

The validation set is also helpful when we use automated search techniques to try out many values of hyperparameters. Recall that hyperparameters are variables that we set before we run our system to control how it operates, such as how much it should update its internal values after an error, or even how complex our classifier should be. For each variation, we train on the training set and evaluate the system's performance on the validation set. As we mentioned, we don't learn from the validation set, but we do use it repeatedly. The results from the validation set are just an estimate of how well the system is doing, so we can decide when to stop training. When we think that performance is up to par, we break out the test set and use it once in order to get a reliable estimate of the system's accuracy.

This gives us a convenient way to repeatedly try out different hyperparameters and then choose the best ones based on how they do on the validation set.

This approach to trying out different sets of hyperparameters is based on running a loop. Let's look at a simplified version of that loop now.

To run our loop, we select a set of hyperparameters, train our system, and then evaluate its performance with the validation set. This estimates how well the system trained with those hyperparameters predicts new data. Next, we set that system aside and create a new system, initialized, as always, with random values. We apply the next set of hyperparameters, train, and use the validation set to evaluate this system's performance. We repeat this process over and over, once for each set of hyperparameters. When we've run through all sets of hyperparameters, we select the system that seemed to provide the most accurate results, run the test set through it, and discover how good its predictions really are.

Figure 8-7 shows this whole process graphically.

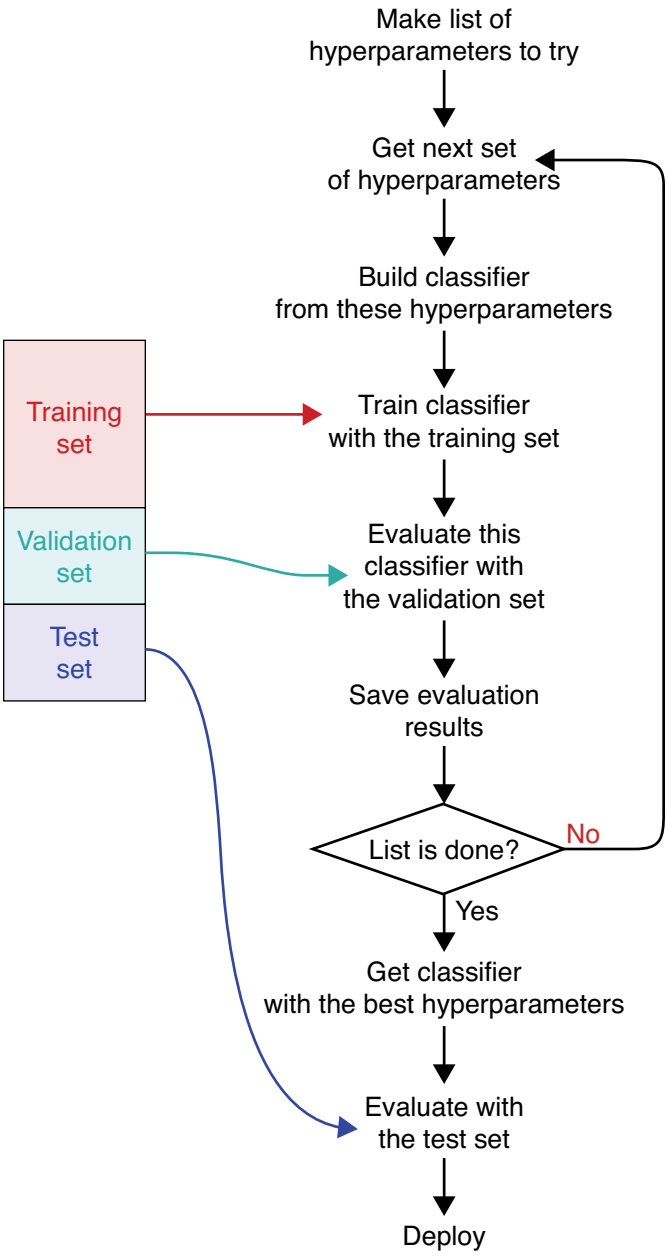


Figure 8-7: We use the validation set when we try out lots of different hyperparameter sets. Note that we still keep a separate test set, which we use just before deployment.

When the loop is done, we may be tempted to use the results from the validation data as our final evaluation of the system. After all, the classifier didn't learn from that data, since it was only used for testing. It may seem

that we can save ourselves the trouble of making a separate test set and then run it through the system to get a performance estimate.

But that would be working with leaked data, which would distort our conclusions. The source of this leakage is a bit sneaky and subtle, like many data contamination issues. The problem is that although the classifier didn't learn from the validation data, our whole training and evaluation system did, because it used that data to pick the best hyperparameters for the classifier. In other words, even though the classifier didn't explicitly learn from the validation data, that data influenced our choice of classifier. We chose a classifier that did best on the validation data, so we *already know* that it's going to do a good job with it. In other words, our knowledge of the classifier's performance on the validation data "leaked" into our selection process.

If this seems subtle or tricky, it is. This sort of thing is easy to overlook or miss, which is why we have to be vigilant about data contamination. Otherwise we risk thinking our system is better than it really is, and thus we deploy a system that isn't good enough for our intended use. To get a good estimate for how our system performs on brand-new data that it has never seen before, there's no shortcut: we need to test it on brand-new data that it has never seen before. That's why we always save the test set for the very end.

Cross-Validation

In the last section, we took almost half our training data and set it aside for validation and testing. That's fine when we have lots and lots of data. But what if our sample set is small and we can't get more data? Maybe we're working with photos of Pluto and its moons taken by the New Horizons spacecraft during its 2015 flyby, and we want to build a classifier we can install on future spacecraft to identify what kind of terrain they're looking at. Our dataset is limited and it's not going to get bigger: there are no new close-up photos of Pluto coming anytime soon. Every image that we have is precious, and we want to learn all we can from every photo we have. Setting some images aside just to determine how good our classifier is would be a huge price to pay.

If we're willing to accept an estimate of the system's performance, rather than a reliable measure of it, then we don't have to set aside a test set. We can indeed train on every piece of input data and still predict our performance on new data. The catch is that we're only going to get back an estimate of the system's accuracy, so it won't be as reliable a measure as we'd get by using a real test set, but when samples are precious, that tradeoff can be worth it.

The technique that does this job is called *cross-validation* or *rotation validation*. There are different types of cross-validation algorithms, but they all share the same basic structure (Schneider 1997). We're going to look at a version that doesn't require us to create a dedicated test set.

The core idea is that we can run a loop that repeatedly trains the same system from scratch and then tests it. Each time through we'll split the

entire input data into a one-time training set and a one-time validation set. The key thing is that we'll construct these sets differently each time through the loop. This lets us use all of our data for training (though, as we'll see, not all at the same time).

We begin by building a fresh instance of our classifier. We split the input data into a temporary training set and temporary validation set. We train our system on the temporary training set and evaluate it with the temporary test set. This gives us a score for the classifier's performance. Now we go through the loop again, but this time, we split the training data into different temporary training and test sets. When we've done this for every iteration through the loop, the average of all the scores is our estimate for the overall performance of our classifier.

A visual summary of cross-validation is shown in Figure 8-8.

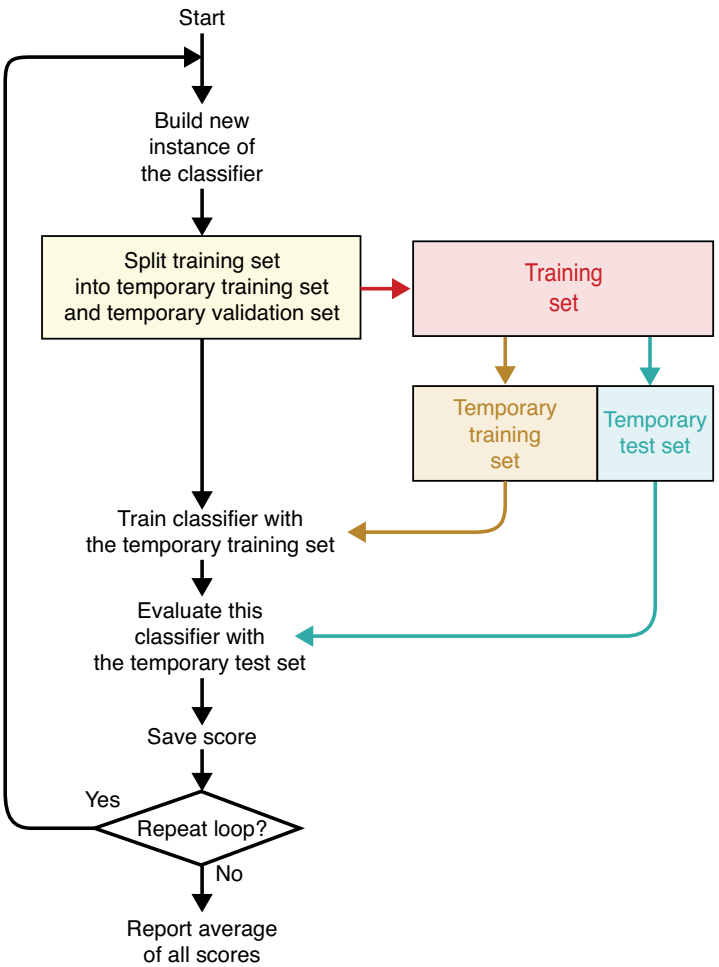


Figure 8-8: Using cross-validation to evaluate our system's performance

By using cross-validation, we get to train with all of our training data (though not all of it on every pass through the loop), yet we still get an objective measurement of the system's quality from a held-out test set. This algorithm doesn't have data leakage issues because each time through the loop, we create a new classifier, and the temporary test set for that classifier contains data that is brand-new and unseen with respect to *that specific* classifier, so it's fair to use it to evaluate that classifier's performance. The penalty for this technique is that our final estimate of the system's accuracy is not as reliable as what we'd get from a held-out test set.

A variety of different algorithms are available for constructing the temporary training and validation sets. Let's look at a popular approach.

k-Fold Cross-Validation

Perhaps the most popular way to build the temporary datasets for cross-validation is called *k-fold cross-validation*. The letter *k* here is not the first letter of a word. Instead, it stands for an integer (for example, we might run "2-fold cross-validation" or "5-fold cross-validation"). Typically, the value of *k* is the number of times we want to go through the loop of Figure 8-8.

The algorithm starts before the cross-validation loop begins. We take our training data and split it up into a series of equal-sized groups. Every sample is placed into exactly one group, and all groups are the same size (well, we allow one smaller group at the end if we can't split the input into equal-sized pieces).

It would have been great if these groups were each called something like "group," or "equal-sized piece," but the word that's come to describe this idea is *fold*. This word is used here in an unusual sense to mean the section of a page *between* creases (or ends). To picture this, imagine writing all the samples in the training set on a long piece of paper and then folding that up into a fixed number of equal pieces. Each time we bend the paper we're making a crease, and the material between the creases is called a fold. Figure 8-9 shows the idea.

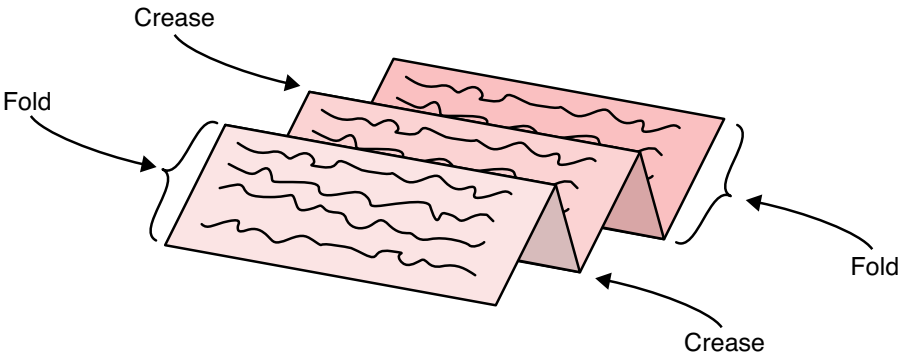


Figure 8-9: Creating the folds for k-fold cross-validation. Here we have four creases and five folds.

Let's build equal-sized folds from our training data. We can flatten out Figure 8-9 to create the more typical picture of five folds, shown in Figure 8-10.

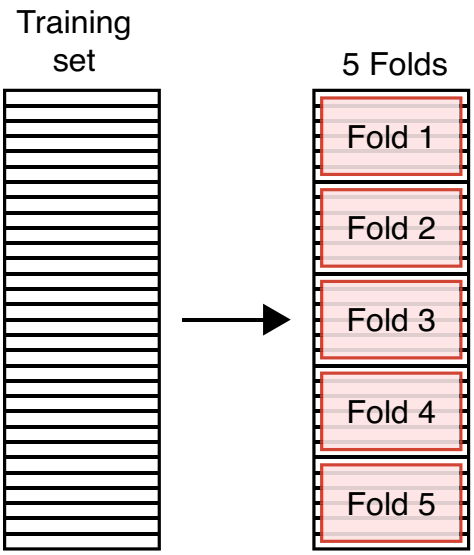


Figure 8-10: Splitting our training set into five equally sized folds, named Fold 1 through Fold 5

Let's use these five folds to see how the loop proceeds. The first time through the loop, we treat the samples in folds 2 through 5 as our temporary training set, and the samples in fold 1 as our temporary test set. That is, we train the classifier with the samples in folds 2 through 5 and then evaluate it with the samples in fold 1.

The next time through the loop, starting with a fresh classifier initialized with random numbers, we use the samples in folds 1, 3, 4, and 5 for our temporary training set, and the samples in fold 2 for our temporary test set. We train and test as usual with these two sets, and continue with the remaining folds. Figure 8-11 shows the idea visually.

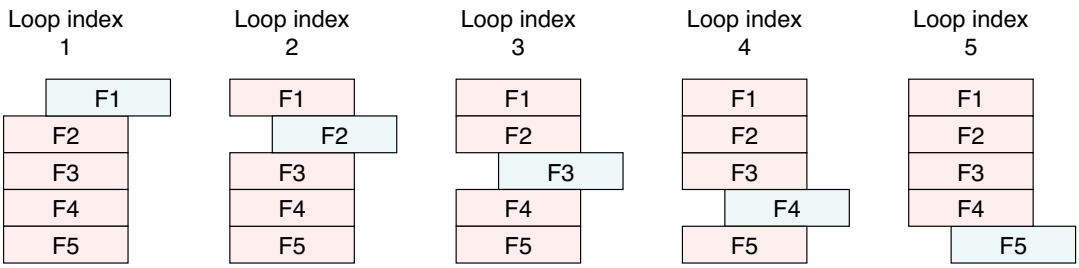


Figure 8-11: In each pass through the loop, we choose one fold for testing (in blue) and use the others (in red) for training. If we go through the loop more than five times, we repeat the pattern.

We can choose to repeat the loop as many times as we like, just repeating the cycle of fold selections (or mixing up the data so the sets always have different contents).

In an optional final step, we can train a fresh classifier with all of our data. This means we can't get an estimate for its performance. But if we watch the training carefully, and look out for overfitting (discussed in the next chapter), we can usually assume that the system that was trained on all the data is at least as good as the worst performance we obtained from cross-validation (and we hope it will be at least a little bit better).

This makes cross-validation a great option when our data is limited. We do have to repeat the train-test cycle many times, and our final performance measure is only an estimate, both of which are downsides, but we gain the ability to train with all of our data, squeezing every little bit of information out of our input set and using it to make our classifier better.

We discussed *k*-fold cross-validation with a classifier, but the algorithm is broadly applicable to almost any kind of learner.

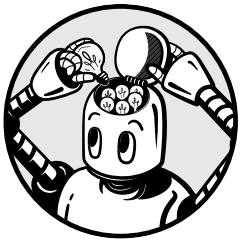
Summary

This chapter was all about training a deep learning system and deciding if it's working well enough to be deployed. We focused on a classifier, but the general thinking holds for any such system. We saw that we split the data into two pieces: a training set and a test set. We learned about the problems of overfitting and data leakage, and we also saw how we can use a validation set to get a rough idea of how well the system is learning after every epoch. Finally, we looked at cross validation, a technique typically used with small dataset, to estimate a system's performance.

In the next chapter, we'll take a closer look at overfitting and underfitting.

9

OVERFITTING AND UNDERFITTING



Whether we're a person or a computer, learning general rules about a subject from a finite set of examples is a tough challenge. If we don't pay enough attention to the details of the examples, our rules will be too general to be of much use when we're considering new data. On the other hand, if we pay too much attention to the details in the examples, our rules will be too specific, and again we'll do a bad job at evaluating new data.

These phenomena are respectively called *underfitting* and *overfitting*. The more common and troublesome problem of the two is overfitting, and if unchecked, it can leave us with a system that's all but useless. We control overfitting and rein it in with techniques known collectively as *regularization*.

In this chapter we look at the causes of overfitting and underfitting, and how to address them. Finally, we wrap up the chapter by seeing how to use Bayesian methods to fit a straight line to a bunch of data points.

Finding a Good Fit

When our system learns from the training data so well that it does poorly when presented with new data, we say that it's *overfitting*. When it doesn't learn from the training data well enough and does poorly when presented with new data, we say that it's *underfitting*. Since overfitting is usually a harder problem, we'll look at it first.

Overfitting

Let's approach our discussion of overfitting with a metaphor. Suppose we've been invited to a big open-air wedding where we know almost nobody. Over the course of the afternoon, we drift through the gathering guests, exchanging introductions and small talk. We've decided to make an effort to remember people's names, so each time we meet someone, we make up some kind of mental association between their appearance and their name (Foer 2012; Proctor 1978). One of the people we meet is a fellow named Walter who has a big walrus mustache. We make a mental picture of Walter as a walrus and try to make that picture stick in our minds. Later, we meet someone named Erin, and we notice she's wearing beautiful turquoise earrings. We make a mental picture of her earrings that have been shortened in one direction, so *earring* becomes *Erin*. We make a similar mental image for everyone we meet, and as we mingle and bump into some of the same people again, we remember their names with ease. The system is working great.

That evening at the reception we encounter lots of new people. At one point we bump into someone with a big walrus mustache. We smile and say, "Hi again, Walter!" only to get a confused expression. This is Bob, someone we haven't met before. The same thing can happen repeatedly. We might be introduced to someone with beautiful earrings, but this is Susan, not Erin. The problem is that our mental pictures have misled us. It's not that we didn't learn people's names properly, because we did. We just learned them in a way that worked only for the people in the original group and didn't generalize when we met more people.

To associate someone's appearance with their name, we need some kind of connection between the two ideas. The more robust that connection, the better we can be at recognizing that person in a new context, even if they're wearing a hat or glasses or something else that alters their appearance. In the case of our wedding, we learned people's names by connecting them to a single idiosyncratic feature. The problem is that when we met someone else with that same feature at the reception, we had no way to determine that this was someone new.

At the pre-wedding party, we thought we were doing well because when we evaluated our performance using the training data (the names of people at the wedding), we got most of the results right. If we focused on the number of successes, we say that we achieved a high *training accuracy*. If we focus instead on the number of failures, we'd say we had a low *training error* (or *training loss*). But when we then went to the reception and needed to

evaluate new data (the names of the additional people we met), our *generalization accuracy* was low, or equivalently, our *generalization error* (or *generalization loss*) was high.

We saw an example of this same problem in Chapter 8, where we erroneously identified a husky on a couch as being a Yorkshire terrier, because we used the couch as our only cue for identifying the breed of the dog on it.

Our errors with both people and dogs were due to overfitting. In other words, we learned how to classify the data in front of us, but we used specific details in that data, rather than learning general rules that would apply to new data as well.

Machine learning systems are really good at overfitting. Sometimes we say that they're good at *cheating*. If there's some quirk in the input data that helps the system get the correct result, it finds and exploits that quirk, like our story in Chapter 8 of how a system was supposed to be solving the hard problem of finding camouflaged tanks in photos of trees but probably was taking the easy way out and simply noting whether the sky was sunny or cloudy.

We can take two actions to control overfitting. First, we can catch when the rules get too specific and stop the learning process at that moment. Second, using *regularization* methods, we can delay the onset of overfitting by encouraging the system to keep learning general rules as long as possible. We'll look at each approach in a moment.

Underfitting

The opposite of overfitting is underfitting. In contrast to overfitting, which results from using rules that are too precise, underfitting describes the situation when our rules are too vague or generic. At the wedding party, we might underfit by creating a rule that says, "people wearing pants are named Walter." Although this is accurate for one particular piece of data, this rule is not going to generalize well!

Underfitting is usually much less of a problem in practice than overfitting. We can often cure underfitting just by using more training data. With more examples, the system can work out better rules for understanding each piece of data.

Detecting and Addressing Overfitting

How do we know when we've started to overfit our data? Suppose that we're using a validation set to estimate a system's generalization error after each epoch (when we're done training, as usual, we use the one-time test set to get a more reliable generalization error). The error made by the system in response to the validation data is called the *validation error*. It's an estimate of the errors the system will make when it's deployed, which is called the *generalization error*. When the validation error flattens out, or starts to become worse, while the training error is improving, we're overfitting. That's our cue to stop learning. Figure 9-1 shows the idea visually.

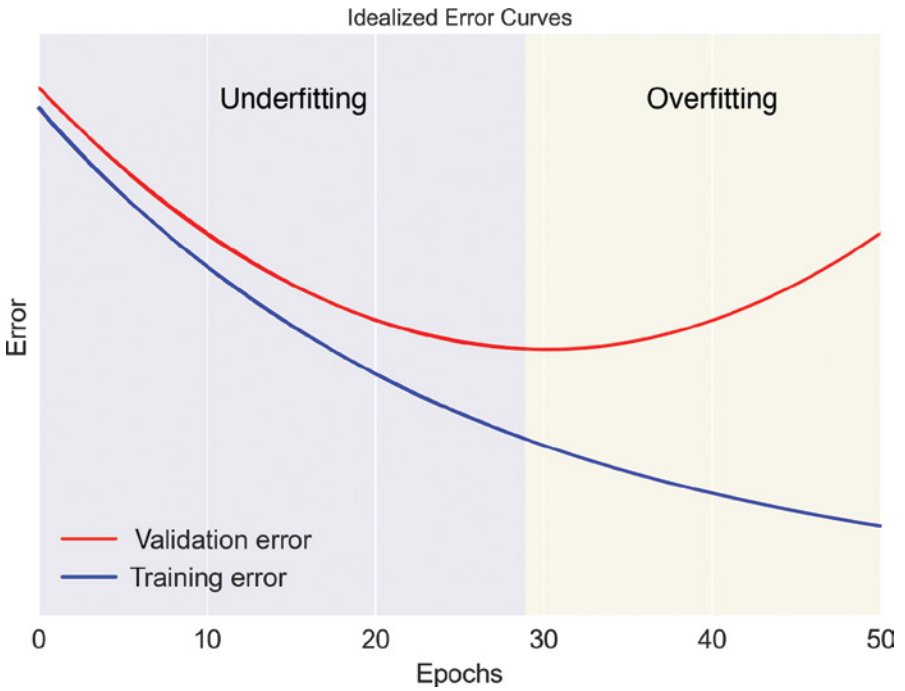


Figure 9-1: The training and validation errors both go down steadily near the start of training, but after a certain point, the validation error starts to increase while the training error continues to decrease, signaling overfitting.

In Figure 9-1, note that the training error continues to decrease as we move into the zone of overfitting, where the validation error is going up. That's because we're still learning from the training data, but now we're learning information specific to that data, rather than general rules. It's the performance on the validation set that lets us see that this is happening, because our validation error (estimating our generalization error) is getting worse. The longer we train like this, the worse our system will perform when we deploy it.

Let's see this in action. Suppose that a store's owner subscribes to a service that provides her with background music. The company provides a variety of streams with music at different tempos, and they've given her a control that lets her choose the tempo of the music at any time. Rather than setting the tempo once at the start of the day and forgetting about it, she's been finding herself adjusting it frequently through the day, and it's become a distracting chore. She's hired us to build a system that automatically adjusts the music throughout the day, the way she wants it.

The first step is to gather data. So, the next morning, we sit across from the controls and watch. Each time she adjusts the tempo, we note the time and new setting. Our collected data are shown in Figure 9-2.

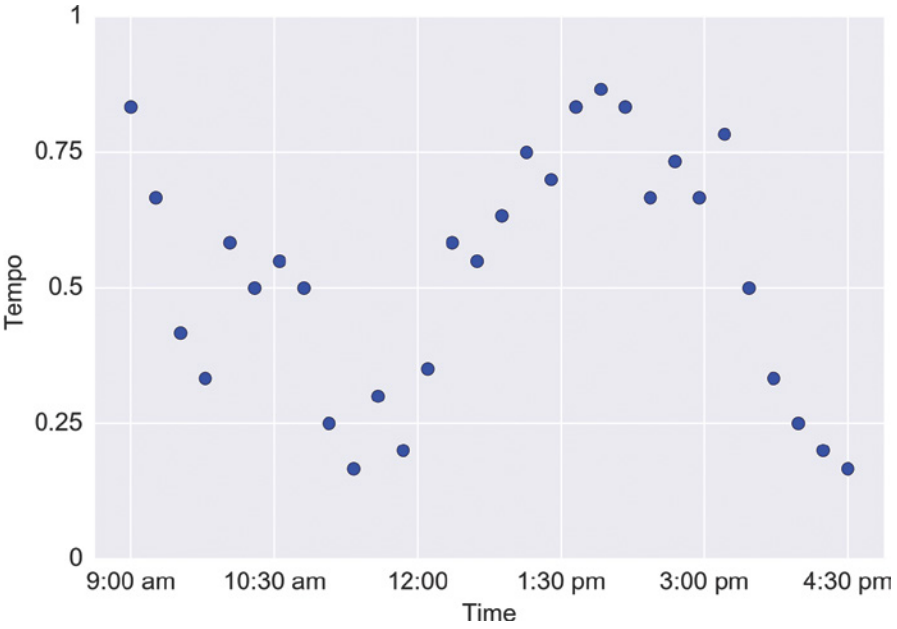


Figure 9-2: Our recorded data shows the tempo chosen by our store owner each time she adjusted it during the day.

Back in our labs that evening, we fit a curve to the data, as in Figure 9-3.

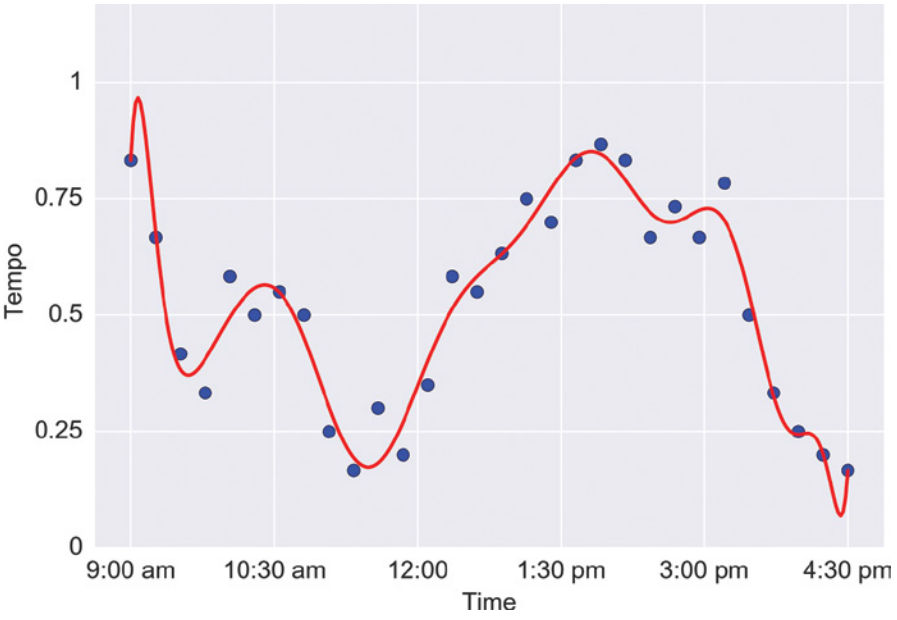


Figure 9-3: A curve to fit to the data of Figure 9-2

This curve is very wiggly, but we might reason that it's a good solution, because it does a good job of matching her recorded choices. The next morning, we program the system to follow this pattern. By the middle of the afternoon the owner is complaining because the tempo of the music is changing too often and too dramatically. It's distracting her customers.

This curve is overfitting the data as a result of matching the observed values too precisely. Her choices on the day we measured the data were based on the particular songs that were playing on that day. Because the service doesn't play the same songs at the same time every day, we don't want to reproduce the data from that one day's observations so closely. By accommodating every bump and wiggle, we're paying too much attention to the idiosyncrasies in the training data.

It would be great if we could watch her choices for several more days and use all of that data to come up with a more general plan, but she doesn't want us taking up room in her store again. The data we have is all we're going to get. We want a schedule with less variation, so the next night we reduce the accuracy of our match to the data. We aim for something that doesn't jump around as much as before and get the gentle curve of Figure 9-4.

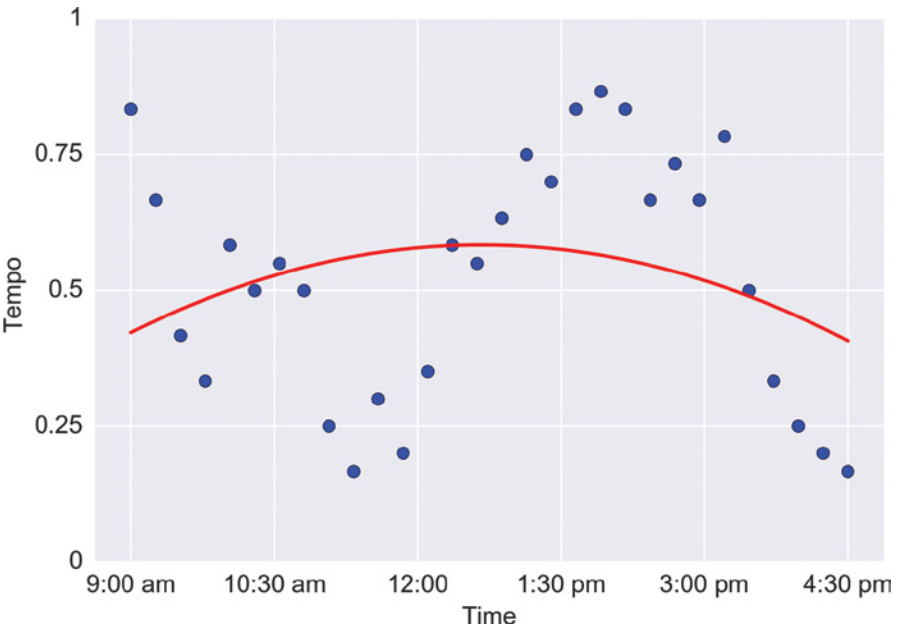


Figure 9-4: A gentle curve for matching our tempo data

We find the next day that our client still isn't satisfied because this curve is much too coarse and ignores important features like her desire to use slower tempos in the morning and more upbeat songs in the afternoon. This curve is underfitting the data.

What we want is a solution that's not trying to match all of the data exactly but is getting a good feeling for the general trends. We want something that's not too precise a match, or too loose, but "just right." The next day, we set up the system according to the curve in Figure 9-5.

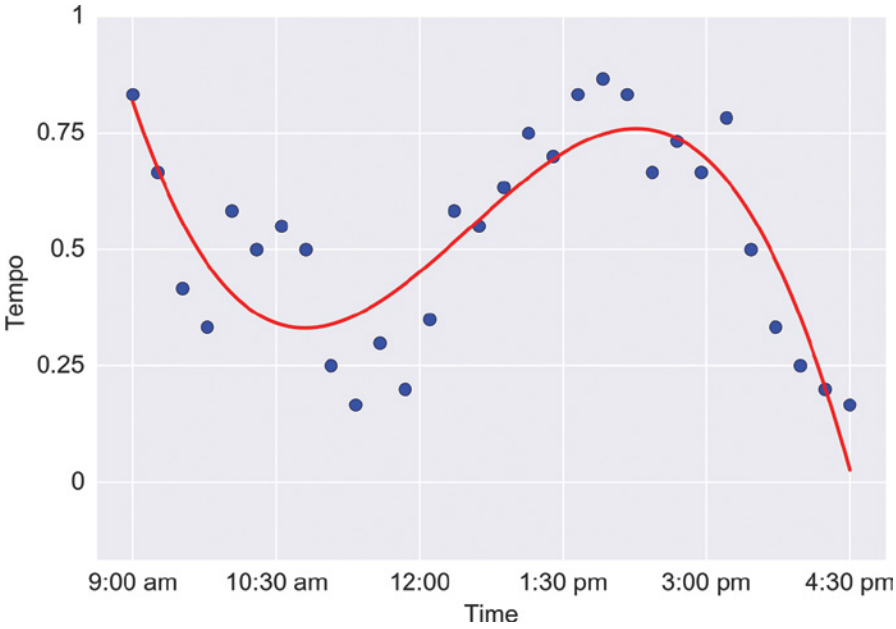


Figure 9-5: A curve that matches our tempo data well enough, but not too well

Our client is happy with this curve and the tempos of the songs it chooses over the day. We've found a good compromise between underfitting and overfitting. In this example, finding the best curve was a matter of personal taste, but later on we'll see algorithmic ways to find this sweet spot between underfitting and overfitting.

Figure 9-6 shows another example of overfitting, this time in classifying two categories of two-dimensional (2D) points.

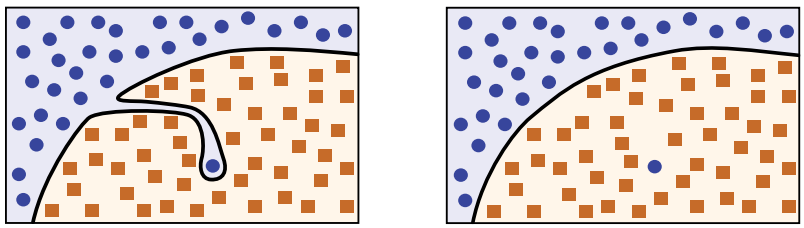


Figure 9-6: A likely case of overfitting. (Inspired by Bullinaria 2015.)

On the left side of Figure 9-6, we have one circular point deep in square territory, resulting in a complicated boundary curve. We call this kind of isolated point an *outlier*, and it's natural to treat it with suspicion. Maybe this is the result of a measuring or recording error, or maybe it's

just one very unusual piece of perfectly valid data. Getting more data would give us a better sense of which case describes this oddity, but if all we have is this one set of data to work with, we need to decide what to do. By drawing the boundary to accommodate this one data point, we risk misclassifying some future data points as blue circles, even though they were solidly inside the brown square region, because they landed on the blue side of this strange boundary curve. It might be better to prefer a simpler curve like that on the right of Figure 9-6, and accept this one point as an error.

Now that we've seen what overfitting looks like, let's look at how we can prevent it from happening.

Early Stopping

Generally speaking, when we start training our model, we are underfitting. The model won't have seen enough examples yet to figure out how to handle them properly, so its rules are general and vague.

As we train more and the model refines its boundaries, the training and validation errors both typically drop. To discuss this, let's repeat Figure 9-1 for convenience here as Figure 9-7.

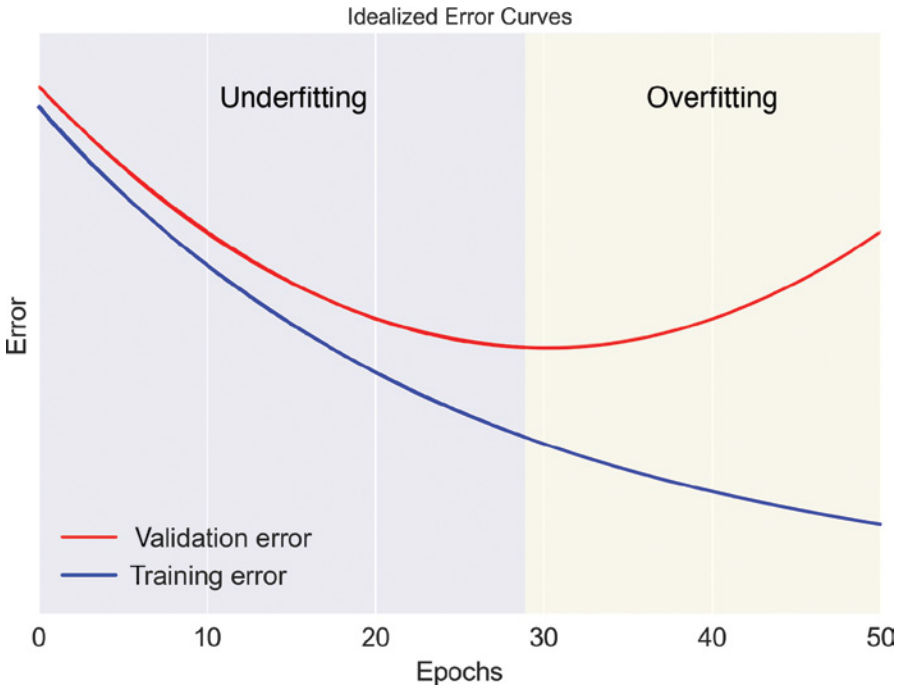


Figure 9-7: A repeat of Figure 9-1 for convenience

At some point, we'll find that although the training error is continuing to drop, the validation error is starting to rise (it may go flat for a while first). Now we're overfitting. The training error is dropping because we're

getting more and more details right. But we're now tuning our results too much to the training data, and the generalization error (or its estimate, the validation error) is going up.

From this analysis we can come up with a good guiding principle: *when we start overfitting, stop training*. That is, when we get to around 28 epochs in Figure 9-7, and we find that the validation error is going up even as training error is dropping, we should stop training. This technique of ending training just as the validation error starts to rise is called *early stopping*, since we're stopping our training process before the training error has reached zero. It may be helpful to think of this idea as *last-minute stopping*, since we're training for as long as we can, only stopping when we've found the best representation of our data without overfitting.

In practice, our error measurements are rarely as smooth as the idealized curves in Figure 9-7. They tend to be noisy and may even go the "wrong" way for short periods, so it can be hard to find the exact right place to stop. Most library functions for early stopping offer a few variables that let us tell them to implicitly smooth out these error curves so they can detect when the validation error really is rising and not just experiencing a momentary increase.

Regularization

We always want to squeeze as much information as we can out of our training data, stopping just short of overfitting. Early stopping ends learning when the validation error starts rising, but what if there was a way to delay that phenomenon, so we can train longer and continue to push down both training and validation errors?

By analogy, consider cooking a turkey in the oven. If we just put the turkey in a pan and cook it on high heat, the outside eventually starts to burn. But say we want to cook the turkey for longer, without burning it. One way to do this is to wrap it in aluminum foil. The foil delays the onset of burning, letting us cook the turkey for longer.

The techniques that delay the onset of overfitting are collectively known as *regularization methods*, or simply *regularization*. Remember that the computer doesn't know that it's overfitting. When we ask it to learn from the training data, it learns from that data as well as it can. It doesn't know when it crosses the line from "good knowledge of the input data" to "overly specific knowledge of this particular input data," so it's up to us to manage the issue.

A popular way to perform regularization, or delay the start of overfitting, is to limit the values of the parameters used by the classifier. Conceptually, the core argument for why this staves off overfitting is that by keeping all of the parameters to small numbers, we prevent any one of them from dominating (Domke 2008). This makes it harder for the classifier to become dependent on specialized, narrow idiosyncrasies.

To see this, think back to our example of remembering people's names. When we memorized the name of Walter, who wore a walrus mustache, that

one piece of information dominated everything else we remembered. The other facts we could have learned from looking at him included that he was a man, he was almost six feet tall, he had long gray hair, he had a big smile and a low voice, he wore a dark-red shirt with brown buttons, and so on. But instead, we focused on his mustache, and ignored all these other useful cues. Later, when we saw a completely different person with a walrus mustache, that one feature dominated all the others and we mistook that person for Walter.

If we force all of the features we notice to have values in roughly the same range, then "has a walrus mustache" doesn't get the chance to dominate, and the other features continue to matter when we remember the name of a new person. Regularization techniques make sure that no one parameter, or no small set of parameters, dominates all the others.

Note that we're not trying to set all the parameters to the *same* value, which would make them useless. We're just trying to make sure they're all in roughly the same range. Pushing the parameters down to small values allows us to learn longer and extract more information from our training data before overfitting occurs.

The best amount of regularization to apply varies from one learner and dataset to the next, so we usually have to try out a few values and see what works best. We specify the amount of regularization to apply with a hyperparameter that's traditionally written as a lowercase Greek λ (lambda), though sometimes other letters are used. Most commonly, larger values of λ mean more regularization.

Keeping the parameter values small also usually means that the classifier's boundary curves don't get as complex and wiggly as they otherwise could. We can use the regularization parameter λ to choose how complex we want our boundary to be. High values give us smooth boundaries, whereas low values let the boundary fit more precisely to the data it's looking at.

In later chapters we'll work with learning architectures that have multiple layers of processing. Such systems can use additional, specialized regularization techniques called *dropout*, *batchnorm*, *layer norm*, and *weight regularization* that can help control overfitting on those types of architectures. All of these methods are designed to prevent any elements of the network from dominating the results.

Bias and Variance

The statistical terms *bias* and *variance* are intimately related to underfitting and overfitting, and often they come up when those topics are discussed. We can say that bias measures the tendency of a system to consistently learn the wrong things, and variance measures its tendency to learn irrelevant details (Domingos 2015). Another way to think of these is that a large amount of bias means that a system is prejudiced toward a particular kind of result, and a large amount of variance means that the answers returned by the system are too specific to the data.

We're going to take a graphical approach to these two ideas by discussing them in terms of 2D curves. These curves might be the solutions to a regression problem, like our earlier task of setting the tempo for a store's background music over time. Or the curves could be the boundary curves between two regions of the plane, as in a classification problem. The ideas of bias and variance are not limited to any one type of algorithm, or to 2D data. But we'll stick to 2D curves because we can draw and interpret them. Let's focus on finding a good fit to an underlying noisy curve and see how the ideas of bias and variance let us describe how our algorithms behave.

Matching the Underlying Data

Let's suppose that an atmospheric researcher friend of ours has come to us for some help. She's measured the wind speed at a certain spot at the top of a mountain, at the same time, every day, for several months. Her measured data is in Figure 9-8.

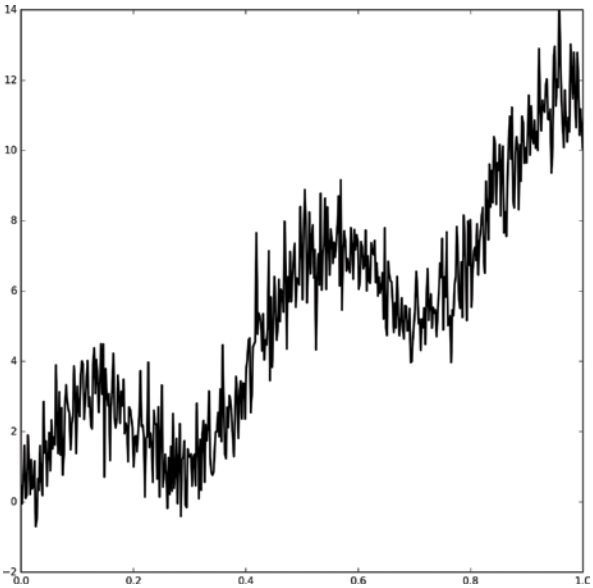


Figure 9-8: Wind speed over time as measured by an atmospheric scientist. In this data there's a clear underlying curve, but there's also plenty of noise (base curve inspired by Macskassy 2008).

She believes that the data she's measured is the sum of an *idealized curve*, which is the same from year to year, and *noise*, which accounts for unpredictable day-to-day fluctuations. The data she measured is called a *noisy curve*, since it's the sum of the idealized curve and the noise. Figure 9-9 shows the idealized curve and the noise that, when added together, make Figure 9-8.

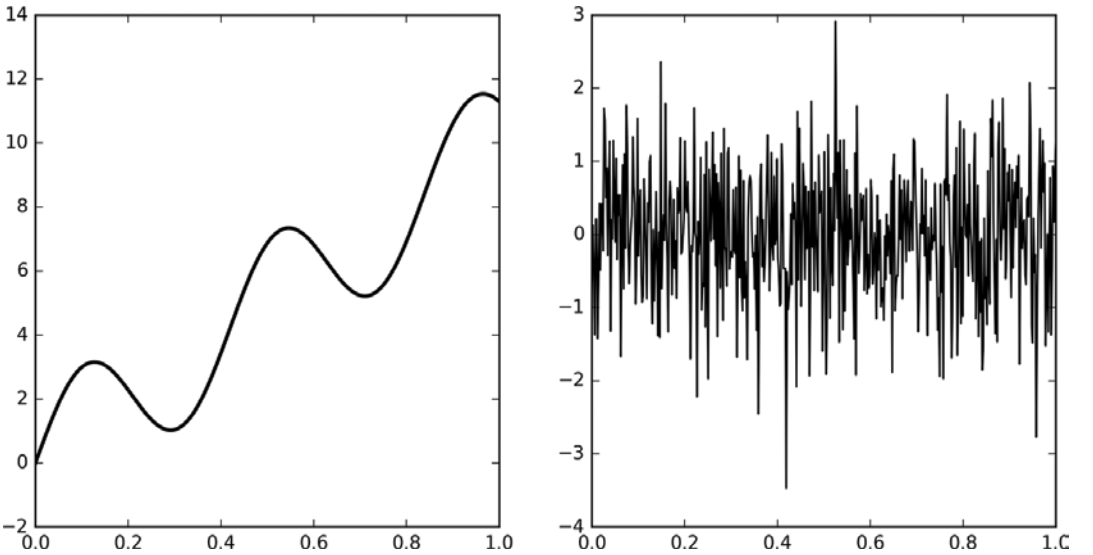


Figure 9-9: The data from Figure 9-8 split into two pieces. Left: The underlying "idealized" curve we seek. Right: The noise that nature added to the idealized curve to give us the noisy, measured data. Note that the two graphs have different vertical scales.

Our atmospheric researcher believes that she has a good model for describing the noise (maybe it follows the uniform or Gaussian distributions we saw in Chapter 2). But her description of the noise is statistical, so she can't use it to fix her day-to-day measurements. In other words, if she knew the exact values of the noise on the right of Figure 9-9, she could subtract them from the measurements in Figure 9-8 to get at her goal, the clean curve on the left of Figure 9-9. But she doesn't know those noise values. She has a statistical model that can generate lots of curves like those on the right of Figure 9-9, but she doesn't have the specific values that correspond to her data.

Here's one approach to cleaning up the noisy data. We can go back to the noisy data in Figure 9-8 and try to fit a smooth curve to it (Bishop 2006). By choosing the complexity of the curve to be wiggly enough to follow the data, but not so wiggly that it tries to match each point exactly, we hope to get a pretty fair match to the general shape of the curve, which is a good starting point for finding the underlying smooth curve.

There are many ways to fit a smooth curve to noisy data. Figure 9-10 shows one such curve. The little wiggle at the right end is typical of the sort of curve we used, which tends to jump around a bit near the edges of the dataset.

That doesn't look too far off. But can we do better?

Let's apply the ideas of bias and variance to the problem of finding the idealized curve. The idea is inspired by the method of bootstrapping that we discussed in Chapter 2, but we won't actually use the bootstrapping technique.

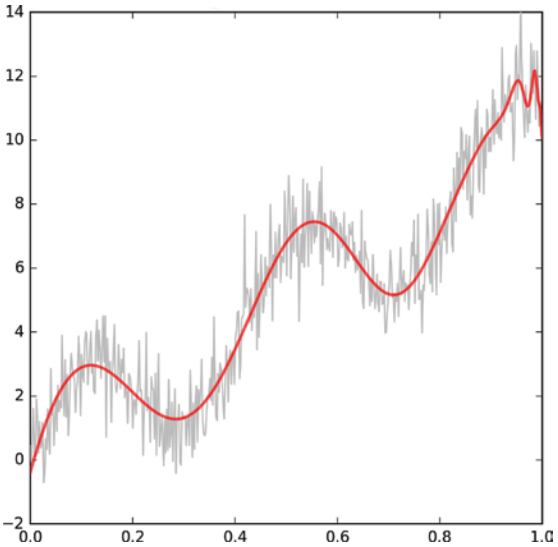


Figure 9-10: Fitting our noisy data with a curve using a curve-fitting algorithm

Let's make 50 versions of the original noisy data, but each version contains just 30 points selected randomly, without replacement. The first five of these reduced datasets are shown in Figure 9-11.

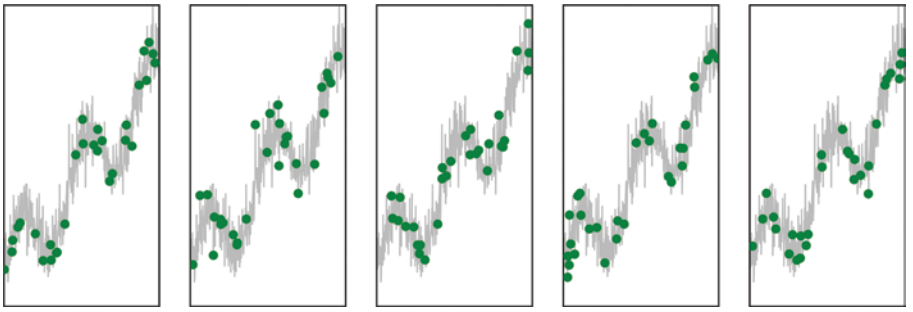


Figure 9-11: Five of the 50 smaller versions of our noisy starting data. Each version consists of 30 samples chosen from the original data without replacement. These are the first five versions, with the chosen points shown as green dots, and the original, noisy data shown in gray for reference.

Let's try matching each of these sets of points with simple curves and then complex curves, and compare the results in terms of bias and variance.

High Bias, Low Variance

We'll first fit our data with simple, smooth curves. Because we've selected these qualities ahead of time, we expect that all of our resulting curves will look about the same. The curves that fit our five sets of data in Figure 9-11 are shown in Figure 9-12.

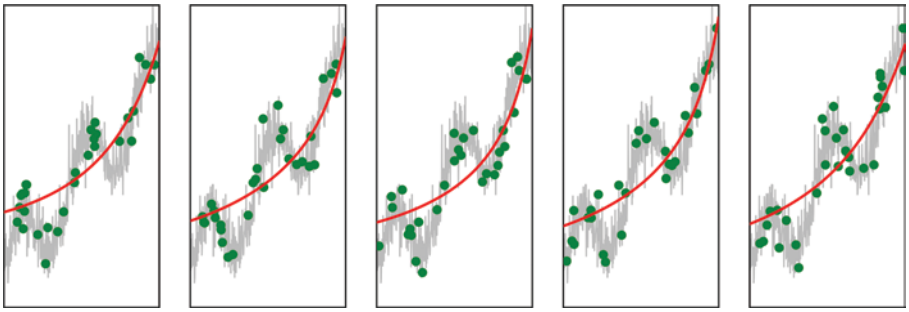


Figure 9-12: Fitting simple curves to our first five sets of points in Figure 9-11

As expected, the curves are all simple and similar. Because these curves are very similar to one another, we say that this collection of curves is showing a *high bias*. The *bias* here refers to the predetermined preference for a simple shape. Because the curves are so simple, each one lacks the flexibility to pass through more than a few of its points at the most.

The *variance* refers to how much the curves vary, or differ, from one to the next. To see the variance of these high-bias curves, we can draw all 50 curves on top of one another, as in Figure 9-13.

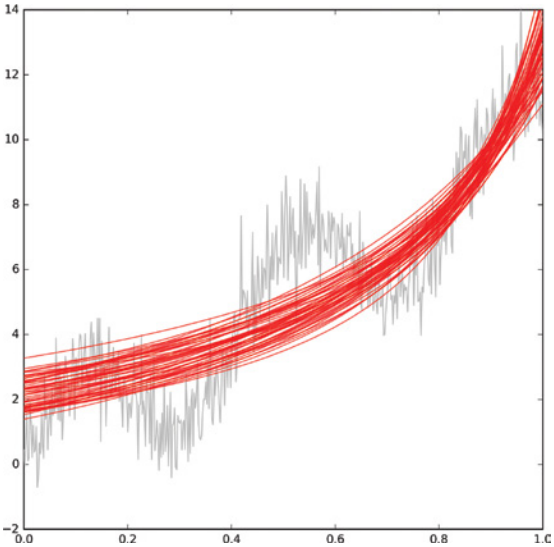


Figure 9-13: The curves for all 50 of our 30-point samples from the original data, overlaid on one another

As expected, the curves are quite similar. We say that they demonstrate low variance.

To sum up, this collection of curves has a high bias, because they all have about the same shape, and a low variance, because the individual curves aren't being influenced much by the data.

Low Bias, High Variance

Now let's try reducing our constraint that the curves need to be simple. That lets us fit complex curves to our data, so that each one comes closer to matching its green points. Figure 9-14 shows these curves applied to our first five sets of data. Compared to Figure 9-12, these curves are much wigglier, with multiple hills and valleys. Though they still don't pass directly through too many points, they come a lot closer to them.

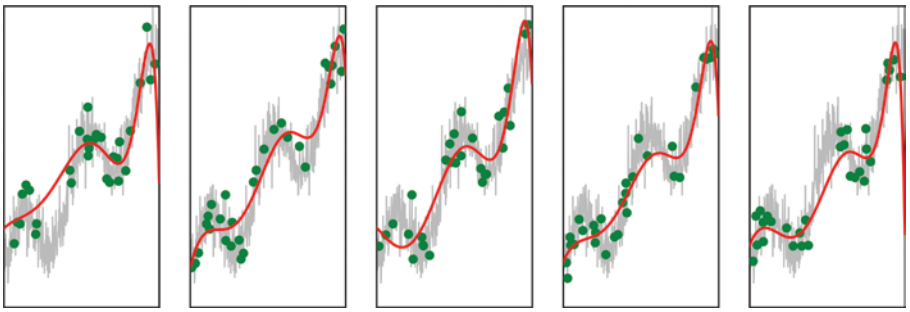


Figure 9-14: The complex curves created for the first five sets of points from our noisy data

Since the shapes of these curves are more complex and flexible, they're more influenced by the data than by any starting assumptions. Because we are placing fewer constraints on the curve shapes, we say that the collection has *low bias*. On the other hand, they're quite different from one another. We can see this by drawing all 50 curves on top of one another, as shown in Figure 9-15. Because the curves veer off wildly at the start and end, we also show an expanded vertical scale that covers those big swings.

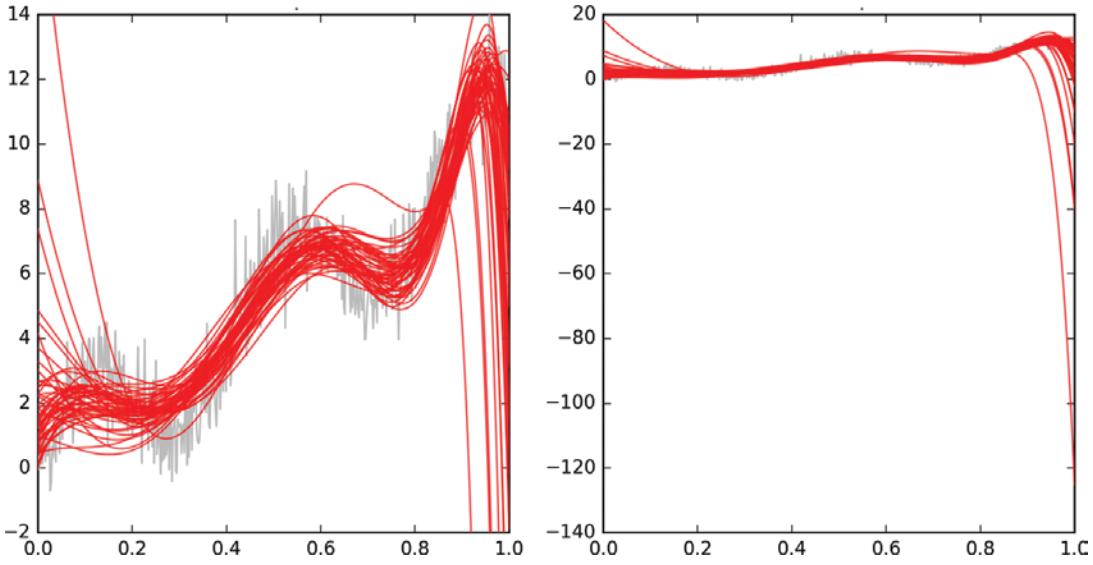


Figure 9-15: The complex curves fit to our 50 sets of data. The plot on the right shows the entire vertical scale of the curves.

These curves don't all follow the same shape, so they have low bias. Furthermore, they are quite different from one another, and each is strongly influenced by its data, so the collection has *high variance*.

Comparing Curves

Let's recap our curve fitting experiment so far.

Our atmospheric scientist asked us for a curve that matches the underlying idealized curve in her data. We created 50 small sets of points, randomly extracted from her original, noisy data. When we fit simple, smooth curves to those sets of points, the curves consistently missed most of the data points. That set of curves had a high bias, or a predisposition to a particular result (smooth and simple). The curves were not much influenced by the data they were intended to match, so that set of curves had a low variance.

On the other hand, when we fit complex and wiggly curves to these sets of points, the curves were able to fit to the data and came much closer to most of the points. Because they were influenced more by the data than by any predisposition to a particular shape, that set of curves had a low bias. But the adaptability of the curves means that they were all significantly different from one another. In other words, that set of curves had a high variance.

So, the first set had high bias and low variance, and the second set has low bias and high variance.

Ideally, we'd like curves with a low bias (so we're not imposing our preconceived ideas on their possible shapes), and low variance (so our different curves all create roughly the same match to the original, noisy data). Unfortunately, in most real situations, as either measure goes down, the other goes up. This means it's up to us to find the best *bias-variance tradeoff* for each specific situation. We'll come back to this issue in a moment.

Notice that bias and variance are properties of *families*, or collections, of curves. It doesn't make sense to discuss the bias and variance of a single curve. Bias and variance often come up in machine learning discussions as ways to describe the complexity or power of a model or algorithm.

We can now see how bias and variance help us describe underfitting and overfitting. In the beginning of training, as the system tries to find the right way to represent the training data, it's producing general rules, or underfitting. If these rules are boundaries between classes of data, they have the form of curves. If we train on multiple similar but different datasets, we'll see curves that are simple in shape and like one another. That is, they have high bias and low variance.

Later in training, the curves for each dataset are more complicated. There are fewer preconditions on their shape, so they have low bias, and they can closely match the training data, so they have high variance. When we let a system train for too long, the high-variance curves start following the input data too tightly, causing overfitting.

Figure 9-16 shows the tradeoff of bias and variance graphically.

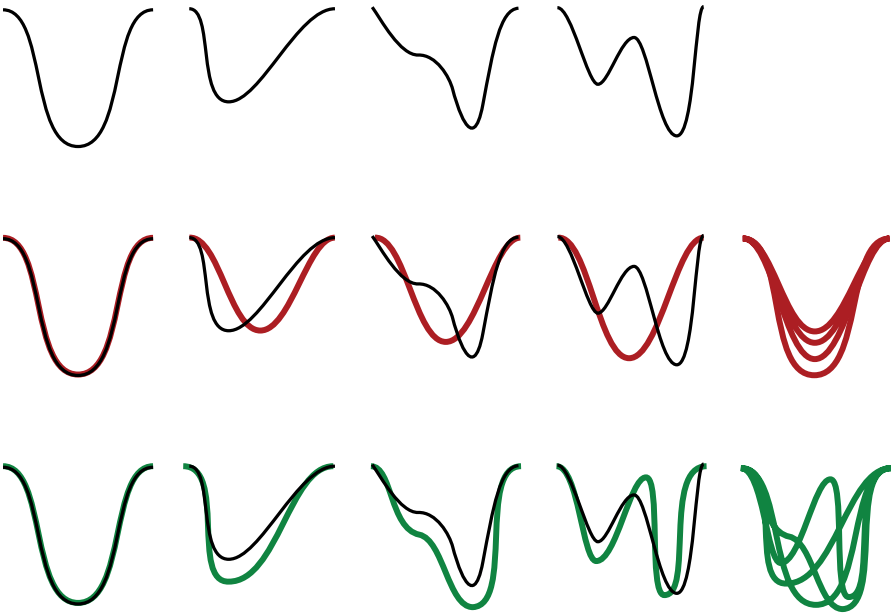


Figure 9-16: Top row: Four curves we'd like to match. Middle row: Using curves with high bias and low variance. Bottom row: Curves with low bias and high variance. The far-right image in the bottom two rows shows the four curves superimposed.

In the middle row, high bias gives us nice, simple curves (which avoid overfitting), but their low variance means they can't match the data very well. In the bottom row, low bias lets the curves better match the data, but their high variance means that the curves can match too well (which risks overfitting).

In general, neither bias nor variance is inherently better or worse than the other, so we shouldn't always be tempted to find, say, the solution with the lowest possible bias or variance. In some applications, high bias or high variance may be acceptable. For instance, if we know that our training set is absolutely representative of all future data, then we don't care about the variance and instead aim for the lowest possible bias, since matching that training set perfectly is just what we want. On the other hand, if we know that our training set is not a good representative of future data (but it's the best we have at the moment), we may not care about bias, since matching this lousy dataset isn't important, but we want the lowest variance we can get so that we have the best chance of at least doing something reasonable on future data.

In general, we need to find the right balance between these two measures in a way that works best for the goals of any particular project, given the specific algorithm and data we're working with.

Fitting a Line with Bayes' Rule

Bias and variance are a useful way to characterize how well a family of curves fits their data. Recalling our discussion of the frequentist and Bayesian philosophies from Chapter 4, we can say that bias and variance are inherently frequentist ideas. That's because the notions of bias and variance rely on drawing multiple values from a source of data. We don't rely too much on any single curve. Instead, we use averaging of all the curves to find the "true" answer that each curve approximates. Those ideas fit the frequentist approach very well.

By contrast, the Bayesian approach to fitting data asserts that the results can only be described in a probabilistic way. We list out all the ways to match our data that we think are possible and attach a probability to each one. As we gather more data, we gradually eliminate some of those descriptions and thereby make the remaining ones more probable, but we never get to a single, absolute answer.

Let's see this in practice. Our discussion is based on a visualization from *Pattern Recognition and Machine Learning* (Bishop 2006). We'll use Bayes' Rule to find a nice approximation of the noisy data atmospheric data we saw in Figure 9-8. Rather than fit a complicated curve to our data, we restrict ourselves to straight lines. That's only because doing so lets us show everything with 2D plots and diagrams. In fact, we stick to lines that are *mostly horizontal*. Again, this is just so we can draw nice diagrams that don't require higher dimensions.

The method for curve fitting with Bayes' Rule can use complex curves, or sheets in space, or even shapes with hundreds of dimensions. We will use straight lines that are mosty horizontal only because that choice keeps the pictures simple.

We'll be working with multiple lines at once, so it would be great to find a compact way to represent different groups of lines without drawing them all.

The trick will be to describe every line with two numbers. The first tells us how much the line is tilted from being perfectly horizontal, which gives us a line in any orientation. The second number tells us how much to move the line up and down.

The first number is the *slope*. A horizontal line has a slope of 0. As the line rotates clockwise, as in Figure 9-17, the slope increases. As the line rotates counterclockwise, the slope decreases.

When the line is perfectly diagonal, the slope is either 1 or –1. As it rotates to steeper orientations, the slope increases quickly until it reaches infinity for a perfectly vertical line. We can take steps to avoid this problem, but it only makes the discussion more complicated. So for the sake of simplicity, we will limit our attention to lines that have a slope between –2 and 2, which lie in the green zone in Figure 9-17.

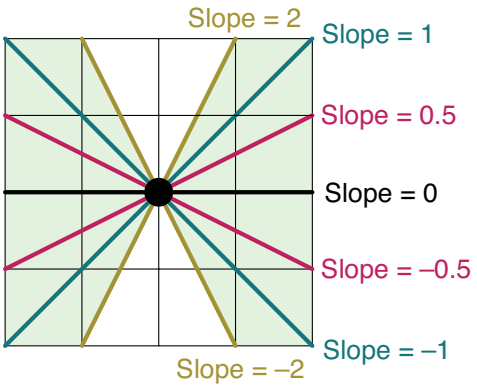


Figure 9-17: A horizontal line has a slope of 0. As the line rotates counterclockwise, the slope increases. As it rotates clockwise, the slope decreases. We will only use lines with slopes that fall in the light-green region.

The second number that describes a line is the *Y intercept*. This merely moves the whole line up or down as a whole. This number tells us the value of the line when X is zero. In other words, it's the value of the line as it crosses, or intercepts, the Y axis. Figure 9-18 illustrates the idea. Again for simplicity we'll restrict our focus to lines with a Y intercept in the range [–2, 2], tinted green in Figure 9-18.

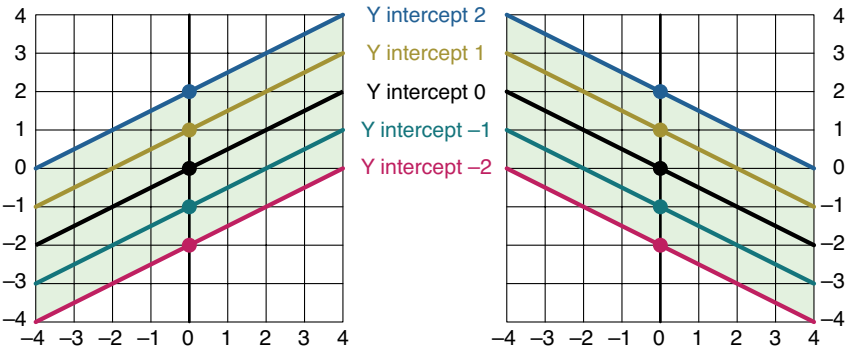


Figure 9-18: The Y intercept tells us the Y value of the line when it crosses the Y axis, regardless of its slope. We'll just use lines that have a Y intercept between –2 and 2.

Given any line, we can measure its orientation to get a value for its slope, and observe where it crosses the Y axis to get the value of the Y intercept. That's everything we need to describe the line. We can show this as a point in a new 2D grid where the axes are labeled *slope* and *Y intercept*. Let's call this an *SI* diagram, standing for slope-intercept. A normal diagram will be just an XY diagram. We can also say that the SI diagram plots lines in *SI space*, and the XY diagram shows lines in *XY space* (also called *Cartesian space*).

Figure 9-19 shows a few lines in both XY and SI diagrams.

Mathematicians call these two ways of looking at the same thing a *dual representation*, and there's a lot to be said about such things. We'll stick to just what we need for our discussion of fitting a line using Bayes' Rule.

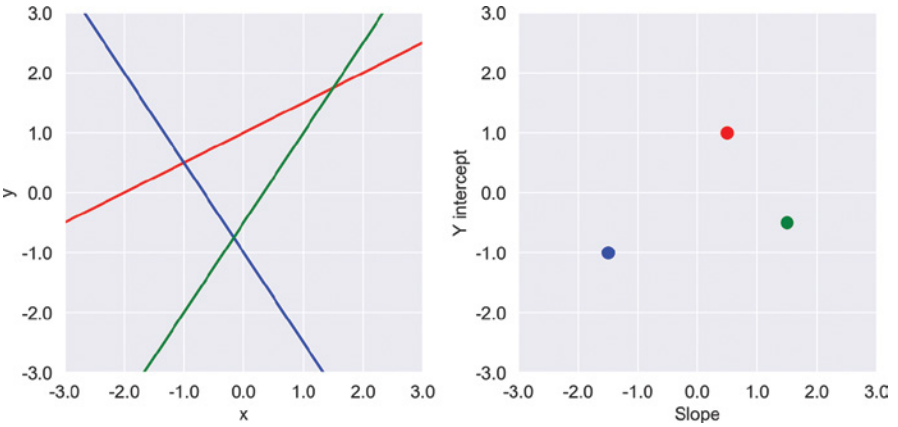


Figure 9-19: Left: Three lines in XY space. Right: Each line drawn as a dot in SI space.

Something interesting happens when we arrange a set of points in SI space along a line. When we draw their corresponding lines in XY space, they all meet at the same XY point. Figure 9-20 shows this in action. This is true any time our SI points lie on a line.

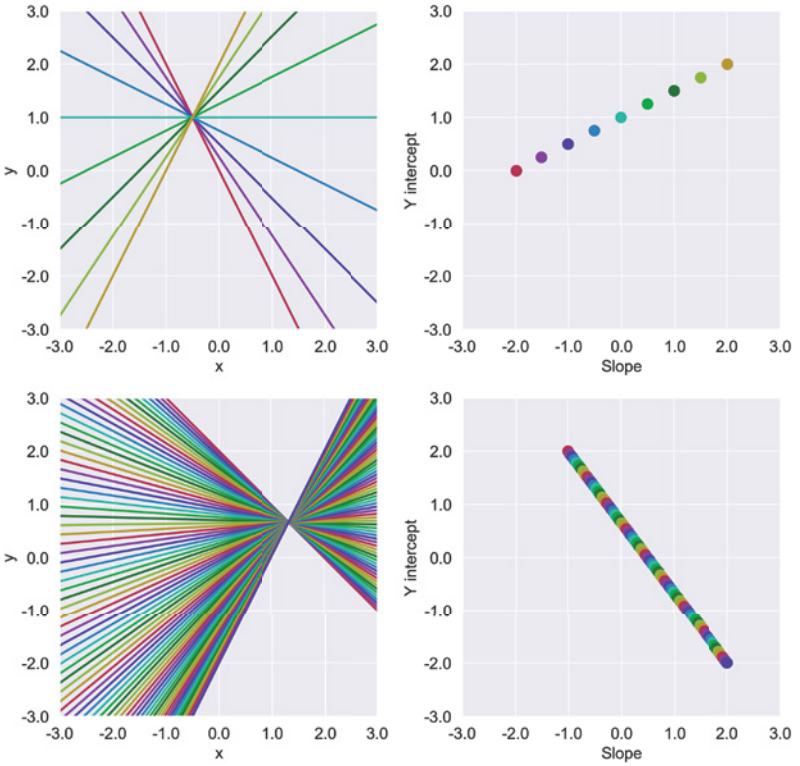


Figure 9-20: Two examples of placing dots in SI space along a line. Their corresponding lines will always meet at a single point in XY space.

As we look for the best line to fit our data, we know it probably won't be able to go through all the points. But we'd like it to come close. So instead of placing dots in SI space, let's assign every possible line a probability from 0 to 1, indicating how likely it is to be the line we're looking for. Figure 9-21 shows the idea (in Figure 9-21, and the figures to come, we scale up the probability values as needed so that they're easier to read).

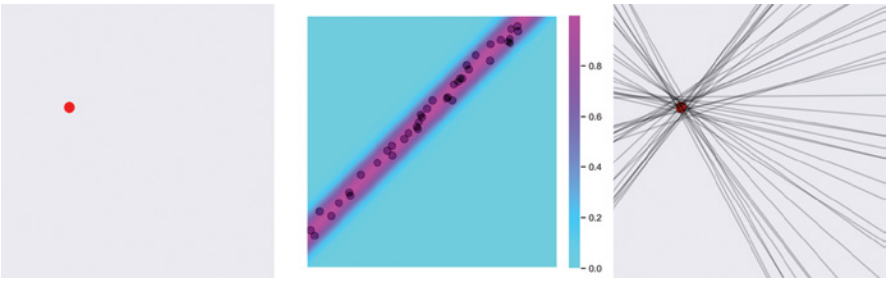


Figure 9-21: Left: A point in XY space. Middle: Every point in the SI graph is assigned a probability from 0 to 1 (blue to purple), telling us how close that line comes to the point. We show some representative lines with black dots. Right: The black dots in the middle figure drawn as their lines in XY space. Note that they all pass through our original red dot, or come close to it.

Let's now return to the problem we want to solve: finding the best straight line approximation for a noisy set of data. Let's start with an arbitrary, broad Gaussian bump in SI space as a prior, as in the top left of Figure 9-22. This says that any line might be our answer, but those in the bright purple region are the most likely. We've chosen some points in this graph according to their probabilities, and drawn them in the upper right. We're getting a lot of very different lines, confirming our very vague prior when it comes to choosing lines.

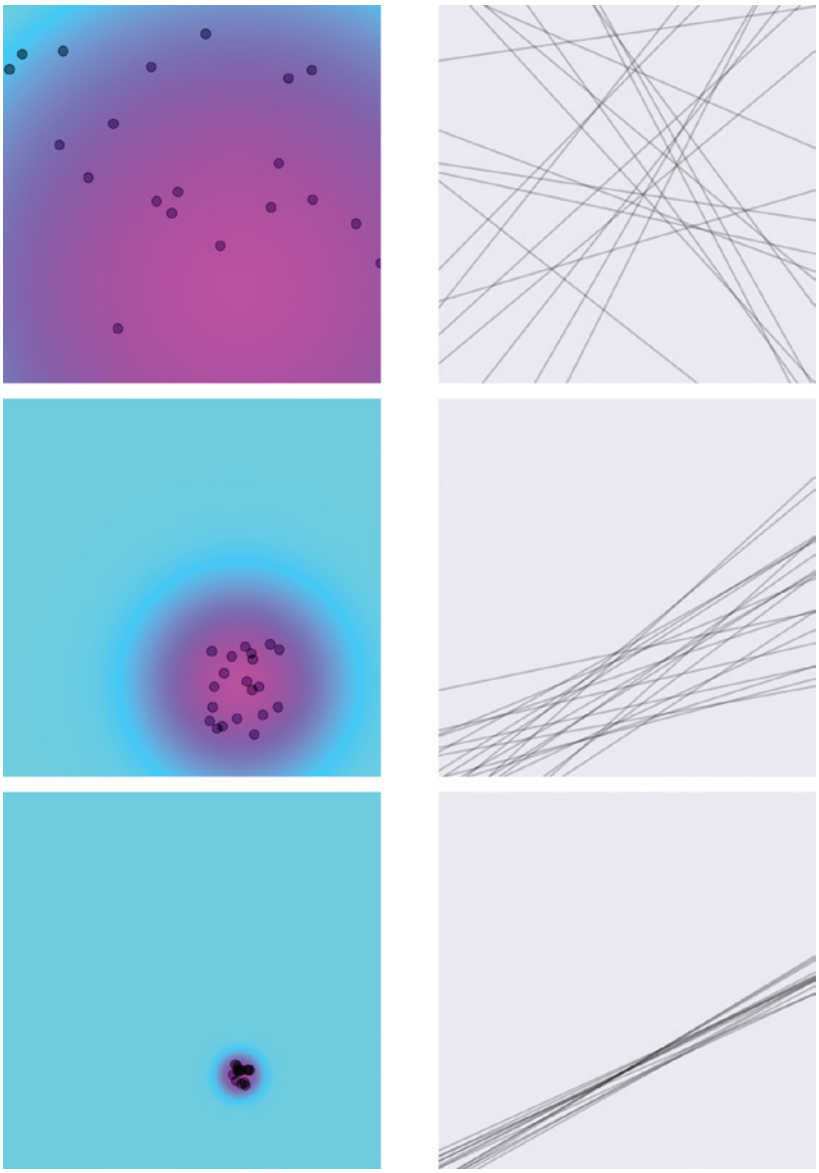


Figure 9-22: Top left: A prior in SI space, along with some points chosen from that probability distribution. Top right: Those points drawn as lines in XY space. Second and third rows: Like the top row, but with smaller priors.

We can see in Figure 9-22 that as the prior becomes smaller, we get a more refined choice of lines. So we'd hope that Bayes' Rule will follow this kind of change, and give us a small posterior (or prior) resulting in a small collection of lines that can fit our data.

Now we're ready to use Bayes' Rule to match our data! Figure 9-23 shows the process.

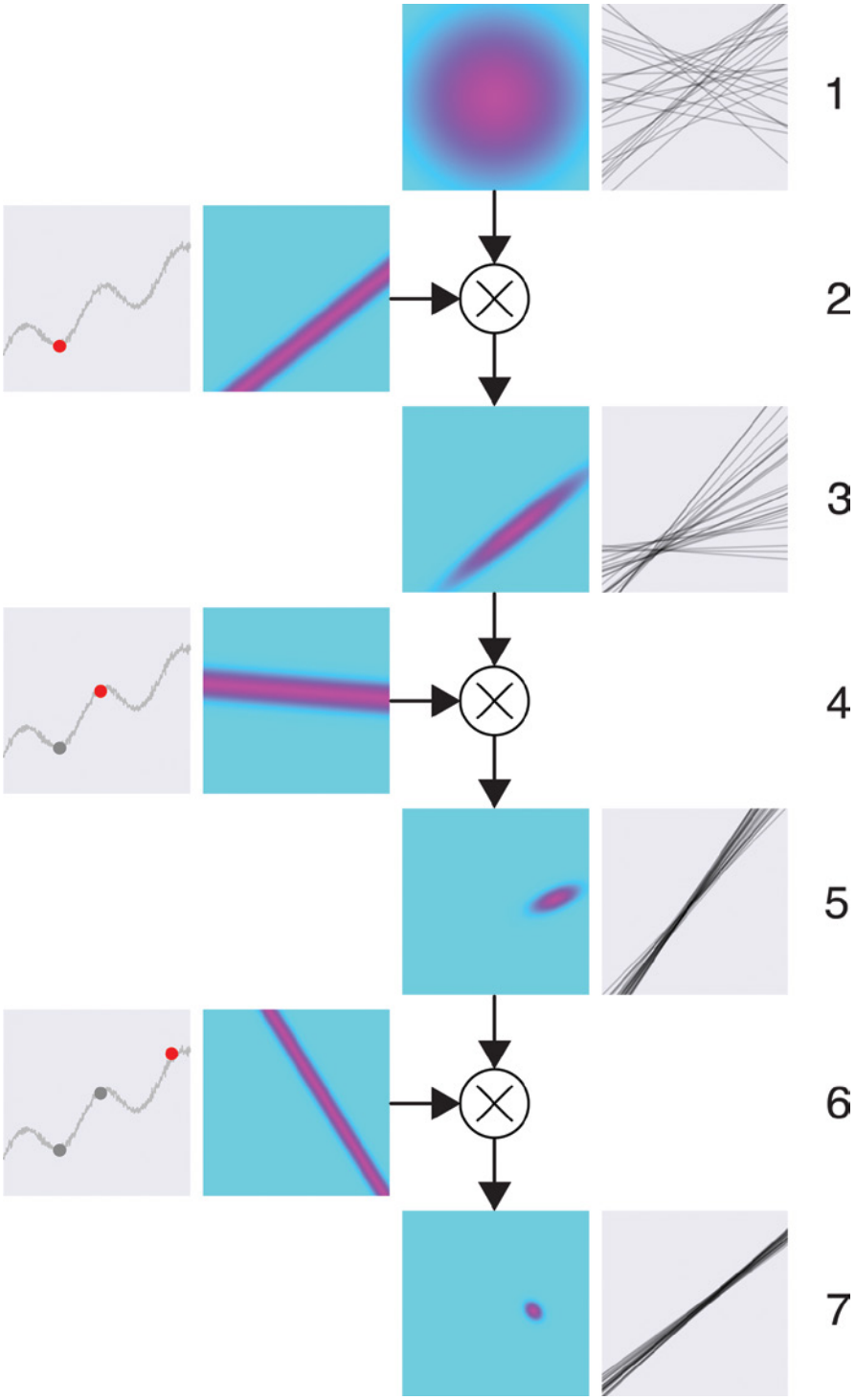


Figure 9-23: Fitting a straight line through our data with Bayes' Rule (figure inspired by Bishop 2006)

Let's walk through what's happening in Figure 9-23 row by row. In row 1, we show our prior, or our starting guess of the distribution of straight lines that will fit our data. We arbitrarily chose a Gaussian with its center in the middle. This prior means that we're guessing that our data is mostly likely fit by a straight line that is horizontal and has a Y intercept of 0. That is, it's the X axis itself. But the Gaussian goes all the way out to the edges (it doesn't quite reach 0 anywhere in the diagram), so any of our available lines are possible. We could look at the data and pick a better starting prior, but this one is simple and, because it has at least some probability for every line we'll consider as a candidate, it's an acceptable start.

The image on the right of row 1 shows 20 lines picked at random from this prior, with more probable lines being more likely to be picked.

The left image in row 2 shows our noisy dataset, and a point picked at random, shown in red. The likelihood diagram for all lines that go through (or near) that point is shown to its right. Now we apply Bayes' Rule, and multiply the prior in row 1 by the likelihood in row 2.

The result is the left figure in row 3. This is the posterior, or the result of multiplying each point in the prior (how likely we thought that line was), with the corresponding point in the new point's likelihood (how likely it is that each line fits this piece of data). We're not showing the Bayes' Rule step of dividing by the evidence because we're scaling our pictures to span the whole range of colors, so this is really a scaled version of the posterior. For simplicity, let's refer to it as the posterior anyway.

Notice that the posterior on line 3 is a new 2D distribution, represented by a new blob. To its right we see another 20 lines drawn at random from that distribution. We can see a big empty space near the top of the figure that wasn't there in row 1. The system has learned from this one step of Bayes' Rule that none of the lines that pass through that space are likely to match the data we've seen. The posterior from line 3 becomes our new prior for the next data point that comes along.

In row 4 we pick a new data point from the input, again shown in red. To its right is the likelihood for lines with respect to this point. In row 5 we apply Bayes' Rule again and multiply our prior (the posterior from line 3) with the likelihood from row 4 to get a new posterior. Notice that the posterior has shrunk in size, telling us that the collection of lines that probably fit both points is smaller than the collection that fits just the first one. To the right, we show lines drawn from this distribution. Notice how much they've grouped together in the same general direction as the two points we just learned from.

We repeat the process again with a new point and likelihood in line 6, and a new posterior and set of lines in line 7. The lines from this posterior are looking very similar, and the trend seems to be approaching a good fit to our data. By using more and more points, we get an increasingly limited range of probable lines.

We can see from this example why Bayes' Rule is so useful in training a learning system. Think of our training data as the points on the curve, and

the evolving prior as the output from our system. As we provide the system with more samples (in this case, points), the system is able to fine-tune itself to deliver the response we're looking for.

It might be tempting to look at the lines in the bottom right of Figure 9-21 and apply our ideas of bias and variance to them, but that's not thinking like a Bayesian. In the Bayesian framework, these aren't a family of lines that approximate some true answer that we can discover by using various forms of averaging. Instead, a Bayesian sees all of these lines as accurate and correct, but with different probabilities. Computing the bias and variance of lines drawn from this collection is possible, but not meaningful in a Bayesian sense.

Both the frequentist and Bayesian approaches let us fit lines (or curves) to data. They just take very different attitudes and use different mechanisms, giving us two different ways to find good answers to our problem.

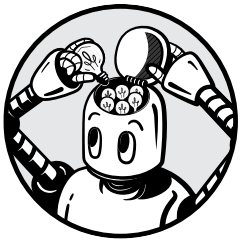
Summary

In this chapter, we looked at a few ways in which a learning system can fail to generalize. When a learning system underperforms because the curves are not good fits to the data, we are underfitting. When a learning system underperforms on new data, but excels on the training data, we are overfitting: the system has learned too many of the quirks and idiosyncrasies of the training data. We saw how we can prevent overfitting by watching training and validation performance and using regularization methods. We ended the chapter by considering bias and variance's relationship to overfitting, and seeing how we can fit a straight line to noisy data using Bayes' Rule.

In the next chapter, we'll look at data and how we can properly prepare it for our learning systems.

10

DATA PREPARATION



Machine learning algorithms can only work as well as the data they're trained on. In the real world, our data can come from noisy sensors, computer programs with bugs, or even incomplete or inaccurate transcriptions of paper records. We always need to look at our data and fix any problems before we use it.

A rich body of methods has been developed for just this job. They're referred to as techniques for *data preparation*, or *data cleaning*. The idea is to process our data before learning from it so that our learning systems can use the data most efficiently.

We also want to make sure that the data itself is well suited to machine learning, which may mean adjusting it, for example, by scaling numbers, or combining categories. This work is essential because the particular way the data is structured, and the numerical ranges it spans, can have a strong effect on the information an algorithm can extract from it.

Our goal in this chapter is to see how we can adjust the data we're given, without changing its meaning, to get the most efficient and effective learning process. We begin with techniques to confirm that our data is clean and ready for training. We then consider methods for examining the data itself, and for making sure that we've got it in the best form for machine learning. This can involve doing simple things like replacing strings with numbers or taking more interesting actions like scaling the data. Finally, we look at ways to reduce the size of our training data. This lets our algorithms run and learn more quickly.

Basic Data Cleaning

Let's start by considering some simple ways to ensure that our data is well cleaned. The idea is to make sure that we're starting out with data that has no blanks, incorrect entries, or other errors.

If our data is in textual form, then we want to make sure there are no typographical errors, misspellings, embedded unprintable characters, or other such corruptions. For example, if we have a collection of animal photos along with a text file describing them and our system is case-sensitive, then we want to make sure that every giraffe is labeled as giraffe and not girafe or Giraffe, and we want to avoid other typos or variants like beautiful giraffe or giraffe-extra tall. Every reference to a giraffe needs to use the identical string.

We should also look for other common-sense things. We want to remove any accidental duplicates in our training data, because they will skew our idea of what data we're working with. If we accidentally include a single piece of data multiple times, our learner will interpret it as multiple, different samples that just happen to have the same value, and thus that sample may have more influence than it should.

We also want to make sure we don't have any typographical errors, like missing a decimal point so we specify a value of 1,000 rather than 1.000, or putting two minus signs in front of a number rather than just one. It's not uncommon to find some hand-composed databases with blanks or question marks in them, signifying that people didn't have any data to enter. Some computer-generated databases can include a code like `NaN` (not a number), which is a placeholder indicating that the computer wanted to print a number but didn't have a valid number to show. More troubling, sometimes when people are missing data for a numerical field, they enter something like 0 or –1. We have to find and fix all such issues before we start learning from the data.

We also need to make sure that the data is in a format that will be properly interpreted by the software we're giving it to. For example, we can use a format known as *scientific notation* to write very large and very small numbers. The problem is that such notation has no official format. Different programs use slightly different forms for this type of output, and other programs that read that data (like the library functions we often use in deep learning) can misinterpret forms they're not expecting. For example, in

scientific notation, the value 0.007 is commonly printed out as `7e-3` or `7E-3`. When we provide the sequence `7e-3` as an input, a program may interpret it as $(7 \times e) - 3$, where e is Euler's constant, which has a value of about 2.7. The result is that the computer thinks that `7e-3` means that we're asking it to first multiply the values of 7 and e together, and then subtract 3, giving us about 16 rather than 0.007. We need to catch these sorts of things so that our programs properly interpret their inputs.

We also want to look for missing data. If a sample is missing data for one or more features, we might be able to patch the holes manually or algorithmically, but it might be better to simply remove the sample altogether. This is a subjective call that we usually make on a case-by-case basis.

Lastly, we want to identify any pieces of data that seem dramatically different from all the others. Some of these *outliers* might be mere typos, like a forgotten decimal point. Others might be the result of human error, like a misdirected copy and paste, or when someone forgot to delete an entry from a spreadsheet. When we don't know if an outlier is a real piece of data or an error of some kind, we have to use our judgment to decide whether to leave it in or remove it manually. This is a subjective decision that depends entirely on what our data represents, how well we understand it, and what we want to do with it.

Though these steps may seem straightforward, in practice, carrying them out can be a major effort depending on the size and complexity of our data and how messed up it is when we first get it. Many tools are available to help us clean data. Some are stand-alone, and others are built into machine-learning libraries. Commercial services will also clean data for a fee.

It's useful to keep in mind this classic computing motto: *garbage in, garbage out*. In other words, our results are only as good as our starting data, so it's vital that we start with the best data available, which means working hard to make it as clean as we possibly can.

Now that we've taken care of the essential small stuff, let's turn our attention to making the data well suited for learning.

The Importance of Consistency

Preparing numbers for learning means applying *transformations* to them, without changing the relationships among them that we care about. We cover several such transformations later in this chapter, where we might scale all the numbers to a given range or eliminate some superfluous data so that the learner has less work to do. When we do these things, we must always obey a vital principle: any time we modify our training data in some way, *we must also modify all future data the same way*.

Let's look at why this is so important. When we make any changes to our training data, we typically modify or combine the values in ways that are designed to improve the computer's learning efficiency or accuracy. Figure 10-1 shows the idea visually.

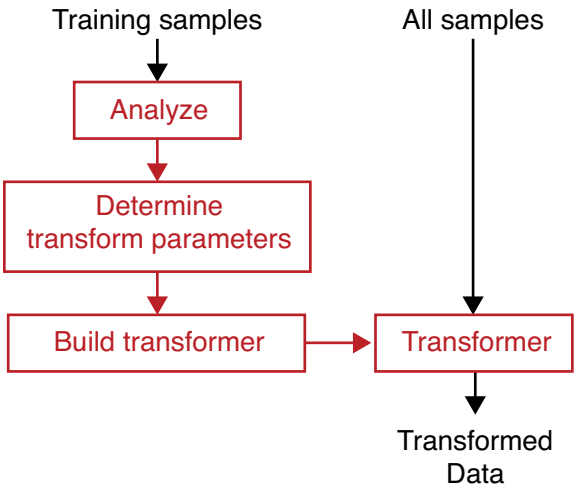


Figure 10-1: The flow of preprocessing for training and evaluating

As the figure shows, we typically determine any needed transformations by looking at the entirety of the training set. We transform that data to train our learner, and we also use the same transformation for all new data that comes after we've released our system to the world. The key point is that we must apply the *identical* modifications to all new data before we give it to our algorithm for as long as our system is in use. This step must not be skipped.

The fact that we need to reuse the same transformation on all data we evaluate pops up again and again in machine learning, often in subtle ways. Let's first look at the problem in a general way with a visual example.

Suppose we want to teach a classifier how to distinguish pictures of cows from pictures of zebras. We can collect a huge number of photos of both animals to use as training data. What most obviously distinguishes pictures of these two animals are their different black-and-white markings. To make sure our learner pays attention to these elements, we may decide to crop each photo to isolate the animal's hide, and then we train with those isolated patches of texture. These cropped photos are all that the learner sees. Figure 10-2 shows a couple of samples.

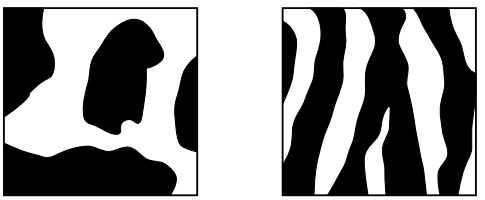


Figure 10-2: Left: A patch of texture from a cow. Right: A patch of texture from a zebra.

Suppose that we've trained our system and deployed it, but we forget to tell people about this preprocessing step of cropping each image to just the texture. Without knowing this vital information, a typical user might give our system complete pictures of cows and zebras, like those in Figure 10-3, and ask the system to identify the animal in each one.



Figure 10-3: Left: A photo of a cow. Right: A photo of a zebra. If we trained our system on the isolated patterns of Figure 10-2, it could be misled by all the extra details in the photos.

Humans can pick out the hide patterns from these photos. A computer, on the other hand, can be misled by the legs, the heads, the ground, and other details, thus reducing its ability to give us good results. The difference between the prepared data of Figure 10-2 and the unprepared data of Figure 10-3 can result in a system that performs beautifully on our training data but gives lousy results in the real world. To avoid this, all new data, like that in Figure 10-3, must be cropped to produce inputs that are just like the training data in Figure 10-2.

Forgetting to transform new data in the same way as we transformed the training data is an easy mistake to make but usually causes our algorithms to underperform, sometimes to the point of becoming useless. The rule to remember is this: we determine how to modify our training data, then we modify it, and then we *remember how we modified it*. Any time we deal with more data, we must first *modify that data in the identical way* that the training data was modified. We'll come back to this idea later and see how it's used in practice.

Types of Data

Typical databases contain different types of data: floating-point numbers, strings, integers that refer to categories, and so on. We'll treat each of these data types in its own way, so it's useful to distinguish them and give each unique type its own name. The most common naming system is based on whether a kind of data can be sorted. Though we rarely use explicit sorting when we do deep learning, this naming system is still convenient and widely used.

Recall that each sample is a list of values, each of which is called a *feature*. Each feature in a sample can be either of two general varieties: *numerical* or *categorical*.

Numerical data is simply a number, either floating-point or integer. We also call this *quantitative* data. Numerical, or quantitative, data can be sorted just by using its values.

Categorical data is just about anything else, though often it's a string that describes a label such as cow or zebra. The two types of categorical data correspond to data that can be naturally sorted and that which can't.

Ordinal data is categorical data that has a known order (hence the name), so we can sort it. Strings can be sorted alphabetically, but we can also sort them by meaning. For example, we can think of the rainbow colors as ordinal, because they have a natural ordering in which they appear in a rainbow, from red to orange on to violet. To sort the names of colors by rainbow order, we need to use a program that understands the orders of colors in a rainbow. Another example of ordinal data are strings that describe a person at different ages, such as infant, teenager, and elderly. These strings also have a natural order, so we can sort them as well, again by some kind of custom routine.

Nominal data is categorical data without a natural ordering. For example, a list of desktop items such as paper clip, stapler, and pencil sharpener has no natural or built-in ordering, nor does a collection of pictures of clothing, like socks, shirts, gloves, and bowler hats. We can turn nominal data into ordinal data just by defining an order and sticking to it. For example, we can assert that the order of clothing should be from head to toes, so our previous example would have the order bowler hats, shirts, gloves, and socks, thereby turning our pictures into ordinal data. The order we create for nominal data doesn't have to make any particular kind of sense, it just has to be defined and then used consistently.

Machine learning algorithms require numbers as input, so we convert string data (and any other nonnumerical data) into numbers before we learn from it. Taking strings as an example, we could make a list of all the strings in the training data, and assign each one a unique number starting with 0. Many libraries provide built-in routines to create and apply this transformation.

One-Hot Encoding

Sometimes it's useful to turn integers into lists. For instance, we might have a classifier with 10 classes where class 3 might be toaster and class 7 might be ball-point pen, and so on. When we manually assign a label to a photo of one of these objects, we consult this list and give it the correct number. When the system makes a prediction, it gives us back a list of 10 numbers. Each number represents the system's confidence that the input belongs to the corresponding class.

This means we are comparing our label (an integer) with the classifier's output (a list). When we build classifiers, it makes sense to compare lists to lists, so we need a way to turn our label into a list.

That's easily done. Our list form of the label is just the list we want from the output. Let's suppose that we're labeling a picture of a toaster. We want

the system's output to be a list of ten values, with a 1 in the slot for class 3, corresponding to complete certainty that this is a toaster, and a 0 in every other slot, indicating complete certainty that the image is none of those other things. So, the list form of our label is the very same thing: ten numbers, all 0, except for a 1 in slot 3.

Converting a label like 3 or 7 into this kind of list is called *one-hot encoding*, referring to only one entry in the list being "hot," or marked. The list itself is sometimes called a *dummy variable*. When we provide class labels to the system during training, we usually provide this one-hot encoded list, or dummy variable, rather than a single integer.

Let's see this in action. Figure 10-4(a) shows the eight colors in the original 1903 box of Crayola Crayons (Crayola 2016). Let's suppose these colors appear as strings in our data. The one-hot labels that we provide to the system as our labels are shown in the rightmost column.

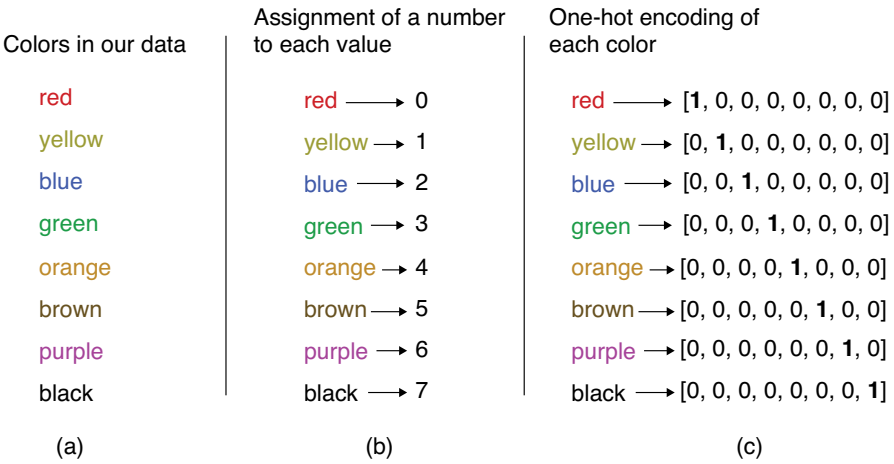


Figure 10-4: One-hot encoding for the original eight Crayola colors in 1903. (a) The original eight strings. (b) Each string is assigned a value from 0 to 7. (c) Each time the string appears in our data, we replace it with a list of eight numbers, all of which are 0 except for a 1 in the position corresponding to that string's value.

So far, we've converted data in one form to another form. Now let's look at some transformations that actually change the values in our data.

Normalizing and Standardizing

We often work with samples whose features span different numerical ranges. For instance, suppose we collected data on a herd of African bush elephants. Our data describes each elephant with four values:

1. Age in hours (0, 420,000)
2. Weight in tons (0, 7)
3. Tail length in centimeters (120, 155)
4. Age relative to the historical mean age, in hours (–210,000, 210,000)

These are significantly different ranges of numbers. Generally speaking, because of the numerical nature of the algorithms we use, larger numbers may influence a learning program more than smaller ones. The values in feature 4 are not only large, but they also can be negative.

For the best learning behavior, we want all of our data to be roughly comparable, or to fit in roughly the same range of numbers.

Normalization

A common first step in transforming our data is to *normalize* each feature. The word *normal* is used in everyday life to mean "typical," but it also has specialized technical meanings in different fields. In this context, we use the word in its statistical sense. We say that when we scale data into some specific range, the data has been *normalized*. The most popular choice of ranges for normalization are [–1,1] and [0,1], depending on the data and what it means (it doesn't make sense to speak of negative apples or ages, for instance). Every machine learning library offers a routine to do this, but we have to remember to call it.

Figure 10-5 shows a 2D dataset that we'll use for demonstration. We've chosen a guitar because the shape helps us see what happens to the points as we move them around. We also added colors strictly as a visual aid, only to help us see how the points move. The colors have no other meaning.

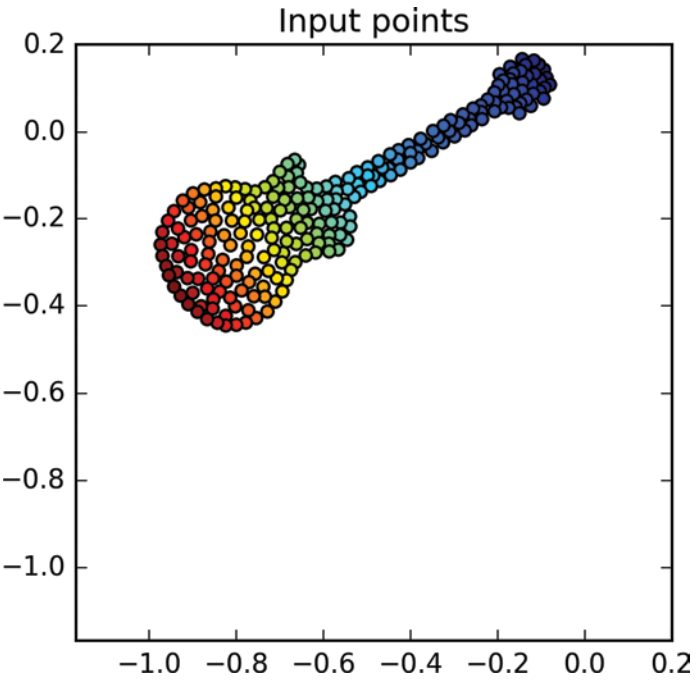


Figure 10-5: A guitar shape made of 232 points

Typically, these points are the results of measurements, say the age and weights of some people, or the tempo and volume of a song. To keep things generic, let's call the two features x and y.

Figure 10-6 shows the results of normalizing each feature in our guitar-shaped data to the range [–1,1] on each axis.

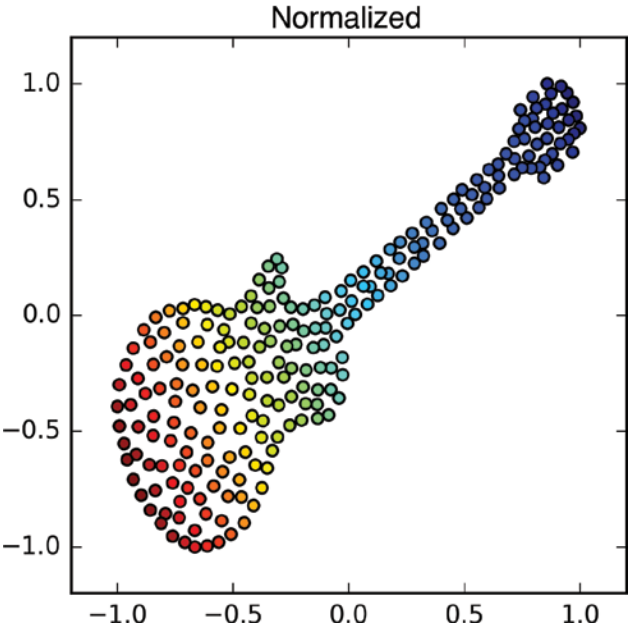


Figure 10-6: The data of Figure 10-5 after normalization to the range [–1,1] on each axis. The skewing of the shape is due to it being stretched more along the Y axis than the X.

In Figure 10-6, the x values are scaled from –1 to 1, and the y values are independently scaled from –1 to 1. The guitar shape resulting from this operation has skewed a little bit because it's been stretched more vertically than horizontally. This happens any time the different dimensions of the starting data span different ranges. In our case, the x data originally spanned the range of about [–1, 0] and the y data spanned about [–0.5, 0.2]. When we adjusted the values, we had to stretch the y values apart more than the x values, causing the skewing we see in Figure 10-6.

Standardization

Another common operation involves *standardizing* each feature. This is a two-step process. First, we add (or subtract) a fixed value to all the data for each feature so that the mean value of every feature is 0 (this step is also called *mean normalization* or *mean subtraction*). In our 2D data, this moves the entire dataset left-right and up-down so that the mean value is sitting right on (0,0). Then, instead of normalizing or scaling each feature to lie between –1 and 1, we scale it so that it has a standard deviation of 1 (this step is also called *variance normalization*). Recall from Chapter 2 that this means about 68 percent of the values in that feature lie in the range of –1 to 1.

In our 2D example, the x values are stretched or compressed horizontally until about 68 percent of the data is between –1 and 1 on the X axis, and then the y values are stretched or compressed vertically until the same thing is true on the Y axis. This necessarily means that points will land outside of the range [–1, 1] on each axis, so our results are different than what we get from normalization. Figure 10-7 shows the application of standardization to our starting data in Figure 10-5.

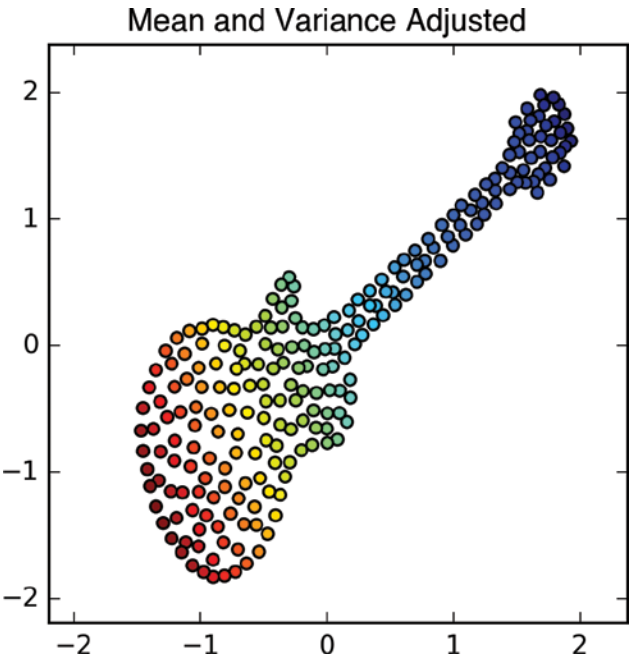


Figure 10-7: The data of Figure 10-5 after standardization

Here again we see that when the original shape doesn't fit a normal distribution, a transformation like standardization can skew or otherwise distort the shape of the original data. Most libraries offer routines to normalize or standardize any or all of our features in one call. This makes it convenient to satisfy some algorithms that require their input to be normalized or standardized.

Remembering the Transformation

Both normalization and standardization routines are controlled by parameters that tell them how to do their jobs. Most library routines analyze the data to find these parameters and then use them to apply the transformation. Because it's so important to transform future data with the same operations, these library calls always give us a way to hang onto these parameters so we can apply the same transformations again later.

In other words, when we later receive a new batch of data to evaluate, either to evaluate our system's accuracy or to make real predictions out in the field, we do *not* analyze that data to find new normalizing or

standardizing transformations. Instead, we apply the same normalizing or standardizing steps that we determined for the training data.

A consequence of this step is that the newly transformed data is almost never itself normalized or standardized. That is, it won't be in the range [–1,1] on both axes, or it won't have its average at (0,0) and contain 68 percent of its data in the range [–1,1] on each axis. That's fine. What's important is that we're using the same transformation. If the new data isn't quite normalized or standardized, so be it.

Types of Transformations

Some transformations are *univariate*, which means they work on just one feature at a time, each independent of the others (the name comes from combining *uni*, for one, with *variate*, which means the same as variable or feature). Others are *multivariate*, meaning they work on many features simultaneously.

Let's consider normalization. This is usually implemented as a univariate transformer that treats each feature as a separate set of data to be manipulated. That is, if it's scaling 2D points to the range [0,1], it would scale all the x values to that range, and then independently scale all the y values. The two sets of features don't interact in any way, so how the X axis gets scaled does not depend at all on the y values, and vice versa. Figure 10-8 shows this ideal visually for a normalizer applied to data with three features.

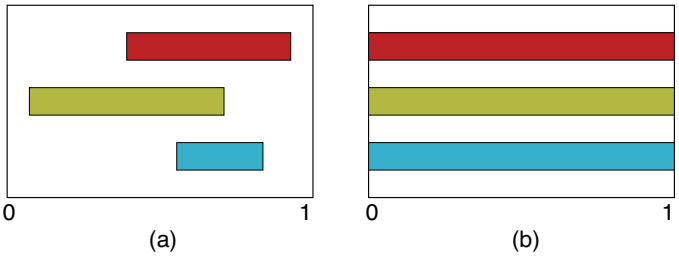


Figure 10-8: When we apply a univariate transformation, each feature is transformed independently of the others. Here we are normalizing three features to the range [0,1]. (a) The starting ranges of three features. (b) Each of the three ranges is independently shifted and stretched to the range [0,1].

By contrast, a multivariate algorithm looks at multiple features at a time and treats them as a group. The most extreme (and most common) version of this process is to handle all of the features simultaneously. If we scale our three colored bars in a multivariate way, we move and stretch them all as a group until they collectively fill the range [0,1], as illustrated in Figure 10-9.

We can apply many transformations in either a univariate or multivariate way. We choose based on our data and application. For instance, the univariate version made sense in Figure 10-6 when we scaled our x and y samples because they're essentially independent. But suppose our features are temperature measurements made at different times over the course of different days? We probably want to scale all the features together so that, as a collection, they span the range of temperatures we're working with.

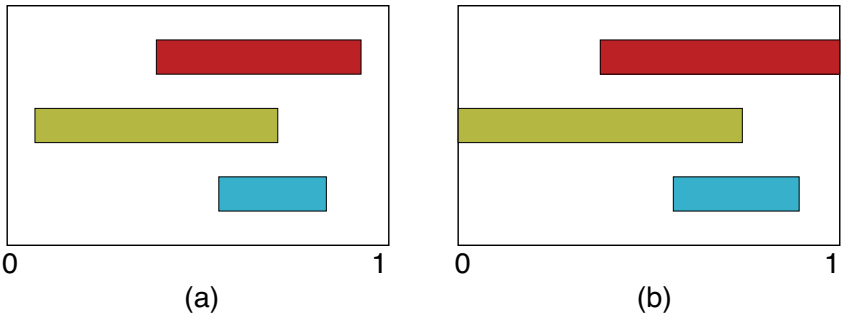


Figure 10-9: When we apply a multivariate transformation, we treat multiple features simultaneously. Here we are again normalizing to the range [0,1]. (a) The starting ranges of three features. (b) The bars are shifted and stretched as a group so that their collective minimum and maximum values span the range [0,1].

Slice Processing

Given a dataset, we need to think about how we select the data we want to transform. There are three approaches, depending on whether we think of *slicing*, or extracting, our data by sample, by feature, or by element. These approaches are respectively called *samplewise*, *featurewise*, and *elementwise* processing.

Let's look at them in that order. For this discussion, let's assume that each sample in our dataset is a list of numbers. We can arrange the whole dataset in a 2D grid, where each row holds a sample and each element in that row is a feature. Figure 10-10 shows the setup.

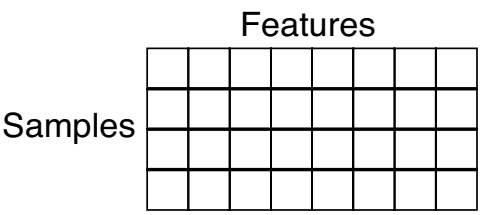


Figure 10-10: Our database for the coming discussion is a 2D grid. Each row is a sample that contains multiple features, which make up the columns.

Samplewise Processing

The samplewise approach is appropriate when all of our features are aspects of the same thing. For example, suppose our input data contains little snippets of audio, such as a person speaking into a cell phone. Then the features in each sample are the amplitude of the audio at successive moments, as in Figure 10-11.

If we want to scale this data to the range [0,1], it makes sense to scale all the features in a single sample so the loudest parts are set to 1 and the quietest parts to 0. Thus, we process each sample, one at a time, independent of the other samples.

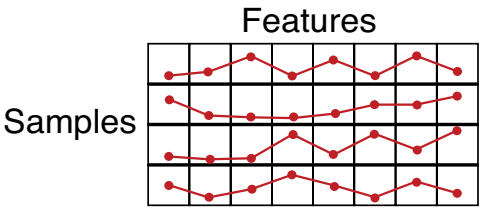


Figure 10-11: Each sample consists of a series of measurements of a short audio waveform. Each feature gives us an instantaneous measurement of the volume of the sound at that moment.

Featurewise Processing

The featurewise approach is appropriate when our samples represent essentially different things.

Suppose we've taken a variety of weather measurements each evening, recording the temperature, rainfall, wind speed, and humidity. This gives us four features per sample, as in Figure 10-12.

	Temperature	Rain fall	Wind speed	Humidity
June 3	60	0.2	4	0.1
June 6	75	0	8	0.05
June 9	70	0.1	12	0.2
	↓	↓	↓	↓
	[60, 75]	[0, 0.2]	[4, 12]	[0.05, 0.2]
	↓	↓	↓	↓
	0	1	0	0.33
	1	0	0.5	0
	0.66	0.5	1	1

Figure 10-12: When we process our data featurewise, we analyze each column independently. Top three lines: The original data. Middle line: The range. Bottom line: The scaled data.

It doesn't make sense to scale this data on a samplewise basis, because the units and measurements are incompatible. We can't compare the wind speed and the humidity on an equal footing. But we can analyze all of the humidity values together, and the same is true for all the values for temperature, rainfall, and humidity. In other words, we modify each feature in turn.

When we process data featurewise, each column of feature values is sometimes called a *fibre*.

Elementwise Processing

The elementwise approach treats each element in the grid of Figure 10-10 as an independent entity and applies the same transformation to every element in the grid independently. This is useful, for example, when all of our data represents the same kind of thing, but we want to change its units. For instance, suppose that each sample corresponds to a family with eight members and contains the heights of each of the eight people. Our measurement team reported their heights in inches, but we want the heights in millimeters.

We need only multiply every entry in the grid by 25.4 to convert inches to millimeters. It doesn't matter if we think of this as working across rows or along columns, since every element is handled the same way.

We do this frequently when we work with images. Image data often arrives with each pixel in the range [0,255]. We often apply an elementwise scaling operation to divide every pixel value in the entire input by 255, giving us data from 0 to 1.

Most libraries allow us to apply transformations using any of these interpretations.

Inverse Transformations

We've been looking at different transformations that we can apply to our data. However, sometimes we want to undo, or *invert*, those steps so we can more easily compare our results to our original data.

For example, suppose we work for the traffic department of a city that has one major highway. Our city is far north, so the temperature often drops below freezing. The city managers have noticed that the traffic density seems to vary with temperature, with more people staying home on the coldest days. In order to plan for roadwork and other construction, the managers want to know how many cars they can predict for each morning's rush-hour commute based on the temperature. Because it takes some time to measure and process the data, we decide to measure the temperature at midnight each evening and then predict how many cars will be on the road between 7 and 8 AM the next morning. We're going to start using our system in the middle of winter, so we expect temperatures both above and below freezing (0° Celsius).

For a few months, we measure the temperature at every midnight, and we count the total number of cars passing a particular marker on the road between 7 and 8 AM the next morning. The raw data is shown in Figure 10-13.

We want to give this data to a machine-learning system that will learn the connection between temperature and traffic density. After deployment, we feed

in a sample consisting of one feature, describing the temperature in degrees, and we get back a real number telling us the predicted number of cars on the road.

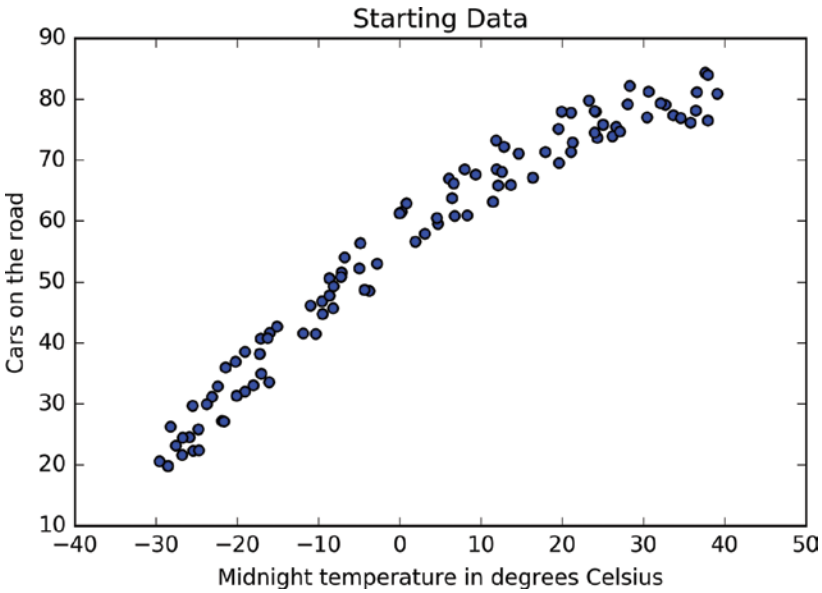


Figure 10-13: Each midnight we measure the temperature, and then the following morning, we measure the number of cars on the road between 7 and 8 AM.

Let's suppose that the regression algorithm we're using works best when its input data is scaled to the range [0,1]. We can normalize the data to [0,1] on both axes, as in Figure 10-14.

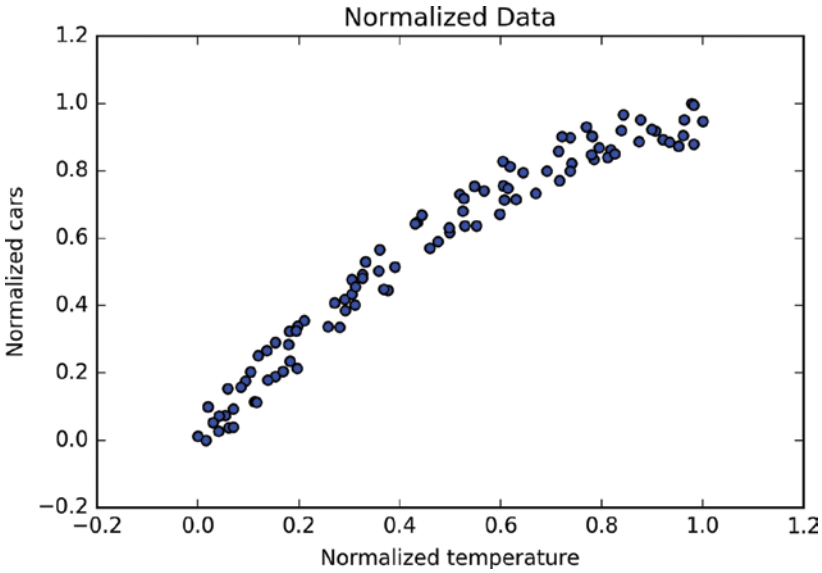


Figure 10-14: Normalizing both ranges to [0,1] makes the data more amenable for training.

This looks just like Figure 10-13, only now both our scales (and data) run from 0 to 1.

We've stressed the importance of remembering this transformation so we can apply it to future data. Let's look at those mechanics in three steps. For convenience, let's use an object-oriented philosophy, where our transformations are carried out by objects that remember their own parameters.

The first of our three steps is to create a transformer object for each axis. This object is capable of performing this transformation (also called a *mapping*).

Second, let's give that object our input data to analyze. It finds the smallest and largest values and uses them to create the transformation that shifts and scales our input data to the range [0,1]. We'll give the temperature data to the first transformer and the vehicle count data to the second transformer.

So far, we've only created the transformers, but we haven't applied them. Nothing has changed in any of our data.

Figure 10-15 shows the idea.

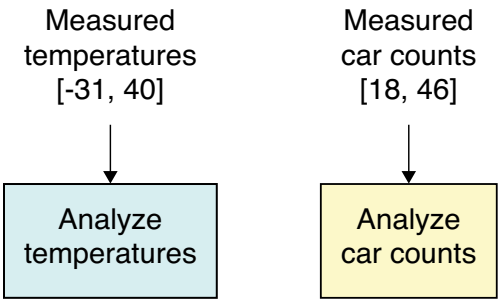


Figure 10-15: Building transformation objects. Left: The temperature data is fed to a transformation object, represented by a blue rectangle. Right: We also build a yellow transformer for the car counts.

The third step is to give our data to the transform objects again, but this time, we tell them to apply the transformation they have already computed. The result is a new set of data that has been transformed to the range [0,1]. Figure 10-16 shows the idea.

Now we're ready to learn. We give our transformed data to our learning algorithm and let it figure out the relationship between the inputs and the outputs, as shown schematically in Figure 10-17.

Let's assume that we've trained our system, and it's doing a good job of predicting car counts from temperature data.

The next day, we deploy our system on a web page for the city managers. On the first night, the manager on duty measures a midnight temperature of –10° Celsius. She opens up our application, finds the input box for the temperature, types in –10, and hits the big "Predict Traffic" button.

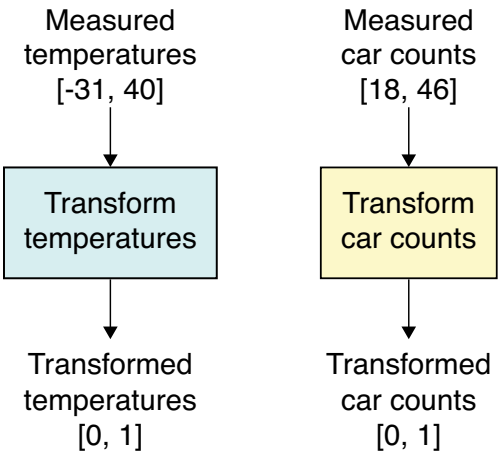


Figure 10-16: Each feature is modified by the transformation we previously computed for it. The output of the transformations goes into our learning system.

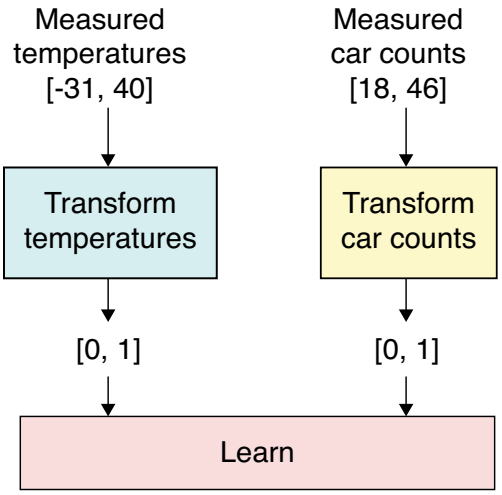


Figure 10-17: A schematic view of learning from our transformed features and targets

Uh-oh, something's gone wrong. We can't just feed –10 into our trained system, because as Figure 10-17 shows, it's expecting a number in the range of 0 to 1. We need to transform the data somehow. The only way that makes sense is to apply the same transformation that we applied to the temperatures when we trained our system. For example, if in our original dataset –10 became the value 0.29, then if the temperature is –10 tonight, we should enter 0.29, not –10.

Here's where we see the value of saving our transformation as an object. We can simply tell that object to take the same transformation that

it applied to our training data and now apply it to this new piece of data. If –10 turned into 0.29 during training, any new input of –10 turns into 0.29 during deployment as well.

Let's suppose that we correctly give the temperature 0.29 to the system, and it produces a traffic density of 0.32. This corresponds to the value of some number of cars transformed by our car transformation. But that value is between 0 and 1, because that was the range of the data we trained on representing car counts. How do we undo that transformation and turn it into a number of cars?

In any machine learning library, every transformation object comes with a routine to *invert*, or undo, its transformation, providing us with an *inverse transformation*. In this case, it inverts the normalizing transformation it's been applying so far. If the object transformed 39 cars into the normalized value 0.32, then the inverse transformation turns the normalized value 0.32 back into 39 cars. This is the value we print out to the city manager. Figure 10-18 shows these steps.

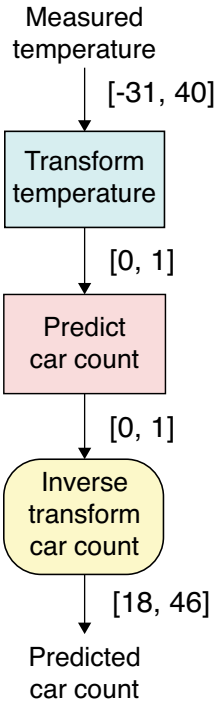


Figure 10-18: When we feed a new temperature to our system, we transform it using the transformation we computed for our temperature data, turning it into a number from 0 to 1. The value that comes out is then run through the inverse of the transformation we computed for the car data, turning it from a scaled number into a number of cars.

One thing that can seemingly go wrong here is if we get new samples outside of the original input range. Suppose we get a surprisingly cold temperature reading one night of –50° Celsius, which is far below the

minimum value in our original data. The result is that the transformed value is a negative number, outside of our [0,1] range. The same thing can happen if we get a very hot night, giving us a positive temperature that transforms to a value greater than 1, which is again outside of [0,1].

Both situations are fine. Our desire for scaling inputs to [0,1] is to make training go as efficiently as possible, and also to keep numerical issues in check. Once the system is trained, we can give it any values we want as input, and it calculates a corresponding output. Of course, we still have to pay attention to our data. If the system predicts a negative number of cars for tomorrow, we don't want to make plans based on that number.

Information Leakage in Cross-Validation

We've seen how to build the transformation from the training set and then retain that transformation and apply it, unchanged, to all additional data. If we don't follow this policy carefully, we can get *information leakage*, where information that doesn't belong in our transformation accidentally sneaks (or leaks) into it, affecting the transformation. This means that we don't transform the data the way we intend. Worse, this leakage can lead to the system getting an unfair advantage when it evaluates our test data, giving us an overinflated measure of accuracy. We may conclude that our system is performing well enough to be deployed, only to be disappointed when it has much worse performance when used for real.

Information leakage is a challenging problem because many of its causes can be subtle. As an example, let's see how information leakage can affect the process of cross-validation, which we discussed in Chapter 8. Modern libraries give us convenient routines that provide fast and correct implementations of cross-validation, so we don't have to write our own code to do it. But let's look under the hood anyway. We'll see how a seemingly reasonable approach leads to information leakage, and then how we can fix it. Seeing this in action will help us get better at preventing, spotting, and fixing information leakage in our own systems and algorithms.

Recall that in cross-validation, we set aside one fold (or section) of the starting training set to serve as a temporary validation set. Then we build a new learner and train it with the remaining data. When we're done training, we evaluate the learner using the saved fold as the validation set. This means that each time through the loop, we have a new training set (the starting data without the samples in the selected fold). If we're going to apply a transformation to our data, we need to build it from just the data that's being used as the training set for that learner. We then apply that transformation to the current training set, and we apply *the same transformation* to the current validation set. The key thing to remember is that because in cross-validation we create a new training set and validation set each time through the loop, we need to build a new transformation each time through the loop as well.

Let's see what happens if we do it incorrectly. Figure 10-19 shows our starting set of samples at the left. They're analyzed to produce a

transformation (shown by the red circle), which is then applied by a transformation routine (marked T). To show the change, we colored the transformed samples red, like the transformation.

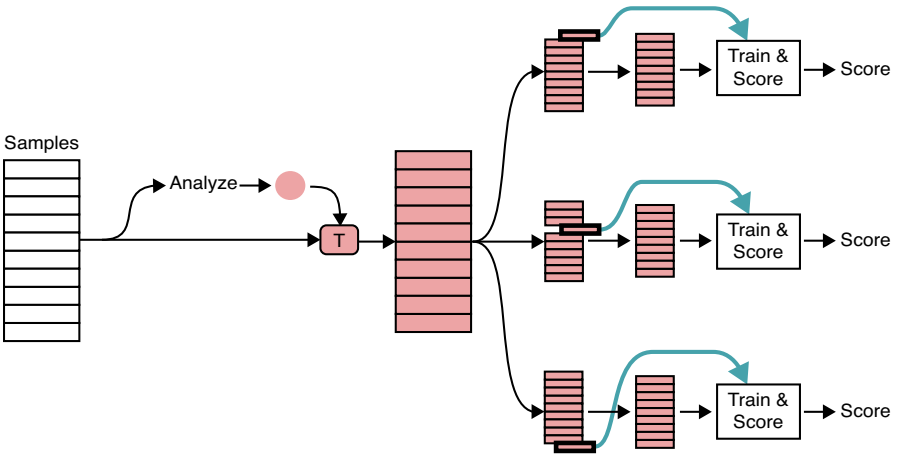


Figure 10-19: A wrong way to do cross-validation: building one transform based on all the original training data

Then we enter the cross-validation process. Here the loop is "unrolled," so we're showing several instances of the training sessions, each associated with a different fold. Each time through the loop, we remove a fold, train on the remaining samples, and then test with the validation fold and create a score.

The problem here is that when we analyzed the input data to build the transformation, we included the data in every fold in our analysis. To see why this is a problem, let's look more closely at what's going on with a simpler and more specific scenario.

Suppose that the transformation we want to apply is scaling all the features in the training set, as a group, to the range 0 to 1. That is, we'll do samplewise multivariate normalization. Let's say that in the very first fold, the smallest and largest feature values are 0.01 and 0.99. In the other folds, the largest and smallest values occupy smaller ranges. Figure 10-20 shows the range of data contained in each of the five folds. We're going to analyze data in all the folds and build our transformation from that.

In Figure 10-20, our dataset is shown at the left, split into five folds. Inside each box, we show the range of values in that fold, with 0 at the left and 1 at the right. The top fold has features running from 0.01 to 0.99. The other folds have values that are well within this range. When we analyze all the folds as a group, the range of the first fold dominates, so we only stretch the whole dataset by a little bit.

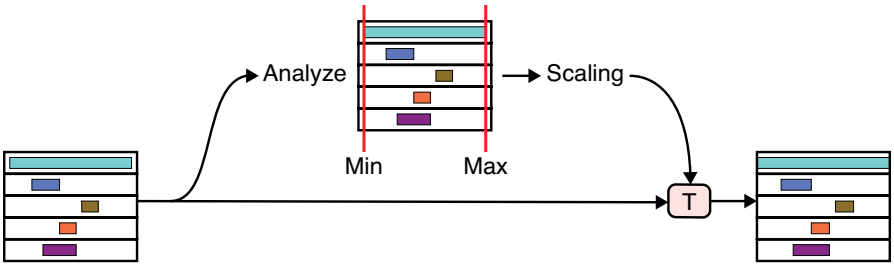


Figure 10-20: A wrong way to transform our data for cross-validation is to transform everything at once before the loop.

Now let's proceed with the cross-validation loop. Our input data is the stack of five transformed folds at the far right of Figure 10-20. Let's start by extracting the first fold and setting it aside; then we can train with the rest of the data, and validate. But we've done something bad here, because *our training data's transformation was influenced by the validation data*. This is a violation of our basic principle that we create the transform using only the values in the training data. But here we used what is now the validation data when we computed our transform. We say that information has *leaked* from this step's validation data into the transformation parameters, where it doesn't belong.

The right way to build the transformation for the training data is to remove the validation data from the samples, then build the transformation from the remaining data, and then apply that transformation to both the training and validation data. Figure 10-21 shows this visually.

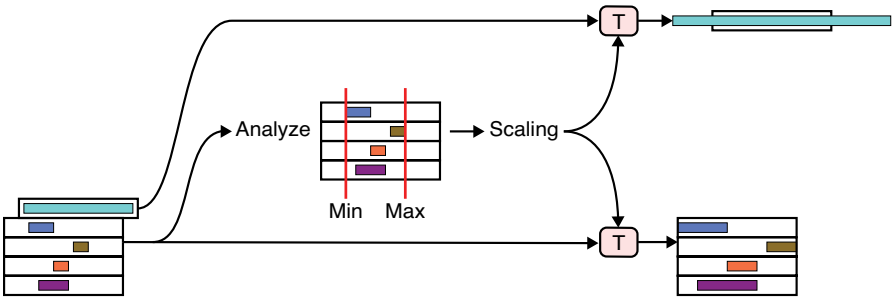


Figure 10-21: The proper way to transform our data for cross-validation is to first remove the fold samples and then compute the transformation from the data that remains.

Now we can apply that transformation to both the training set and the validation data. Note that here the validation data ends up way outside the range [0,1], which is no problem, because that data really is more extreme than the training set.

To fix our cross-validation process, we need to use this scheme in the loop and compute a new transformation for every training set. Figure 10-22 shows the right way to proceed.

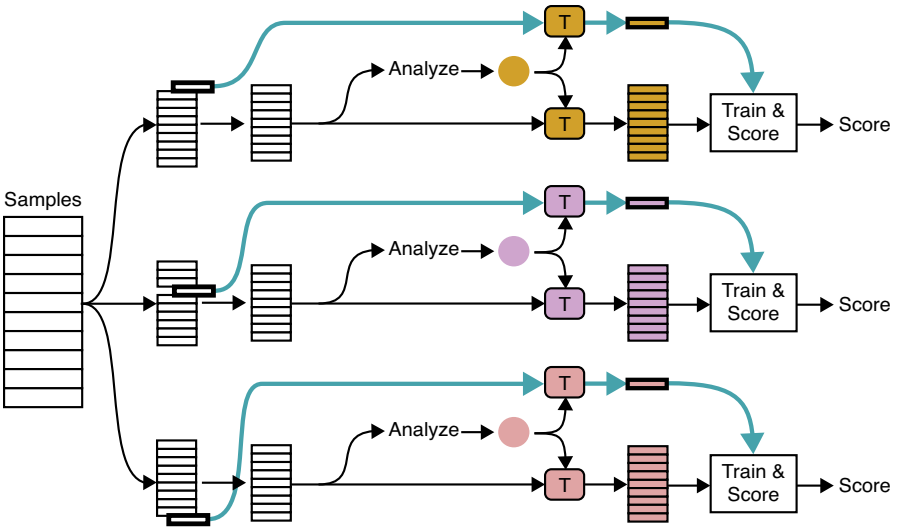


Figure 10-22: The proper way to do cross-validation

For each fold we want to use as a validation set, we analyze the starting samples with that fold removed and then apply the resulting transform to both the training set and the validation set in the fold. The different colors show that each time through the loop, we build and apply a different transformation.

We've discussed information leakage in the context of cross-validation because it's a great example of this tricky topic. Happily, the cross-validation routines in modern libraries all do the right thing, so we don't have to worry about this problem ourselves when we use library routines. But this doesn't take the responsibility off of us when we write our own code. Information leakage is often subtle, and it can creep into our programs in unexpected ways. It's important that we always think carefully about possible sources of information leakage when we build and apply transformations.

Shrinking the Dataset

We've been looking at ways to adjust the numbers in our data, and how to select the numbers that go into each transformation. Now let's look at a different kind of transformation, designed not just to manipulate the data, but to actually compress it. We will literally create a new dataset that's smaller than the original training set, typically by removing or combining features in each sample.

This has two advantages: improved speed and accuracy when training. It stands to reason that the less data we need to process during training, the faster our training goes. By going faster, we mean we can pack

in more learning in a given amount of time, resulting in a more accurate system.

Let's look at a few ways to shrink our dataset.

Feature Selection

If we've collected features in our data that are redundant, irrelevant, or otherwise not helpful, then we should eliminate them so that we don't waste time on them. This process is called *feature selection*, or sometimes, *feature filtering*.

Let's consider an example where some data is actually superfluous. Suppose we're hand-labeling images of elephants by entering their size, species, and other characteristics into a database. For some reason nobody can quite remember, we also have a field for the number of heads. Elephants have only one head, so that field's going to be nothing but 1's. So that data is not just useless, it also slows us down. We ought to remove that field from our data.

We can generalize this idea to removing features that are *almost* useless, that contribute very little, or that simply make the least contribution to getting the right answer. Let's continue with our collection of elephant images. We have values for each animal's height, weight, last known latitude and longitude, trunk length, ear size, and so on. But for this (imaginary) species, the trunk length and ear size may be closely correlated. If so, we can remove (or *filter out*) either one and still get the benefit of the information they each represent.

Many libraries offer tools that can estimate the impact of removing each field from a database. We can then use this information to guide us in simplifying our database and speeding our learning without sacrificing more accuracy than we're willing to give up. Because removing a feature is a transformation, any features we remove from our training set must be removed from all future data as well.

Dimensionality Reduction

Another approach to reducing the size of our dataset is combining features, so one feature can do the work of two or more. This is called *dimensionality reduction*, where *dimensionality* refers to the number of features.

The intuition here is that some of the features in our data might be closely related without being entirely redundant. If the relationship is strong, we might be able to combine those two features into just one new one. An everyday example of this is the *body mass index (BMI)*. This is a single number that combines a person's height and weight. Some measurements of a person's health can be computed with just the BMI. For example, charts that help people decide if they need to lose weight can be conveniently indexed by age and BMI (CDC 2017).

Let's look at a tool that automatically determines how to select and combine features to make the smallest impact on our results.

Principal Component Analysis

Principal component analysis (PCA), is a mathematical technique for reducing the dimensionality of data. Let's get a visual feel for PCA by looking at what it does to our guitar data.

Figure 10-23 shows our starting guitar data again. As before, the colors of the dots are just to make it easier to track them in the following figures as the data is manipulated, and don't have any other meaning.

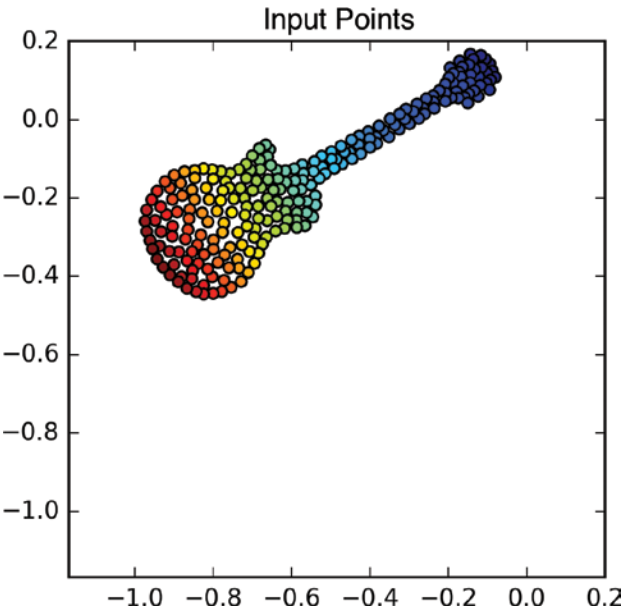


Figure 10-23: The starting data for our discussion of PCA

Our goal is to crunch this two-dimensional data down to one-dimensional data. That is, we combine each set of paired x and y values to create a single new number based on both of them, just as BMI is a single number that combines a person's height and weight.

Let's start by standardizing the data. Figure 10-24 shows this combination of setting each dimension to have a mean of 0 and a standard deviation of 1, as we saw before.

We already know that we're going to try to reduce this 2D data to just 1D. To get a feel for the idea before we actually apply it, let's go through the process with one key step missing, and then we'll put that step back in.

To get started, let's draw a horizontal line on the X axis. We'll call this the *projection line*. Then we'll *project*, or move, each data point to its closest spot on the projection line. Because our line is horizontal, we only need to move our points up or down to find their closest point on the projection line.

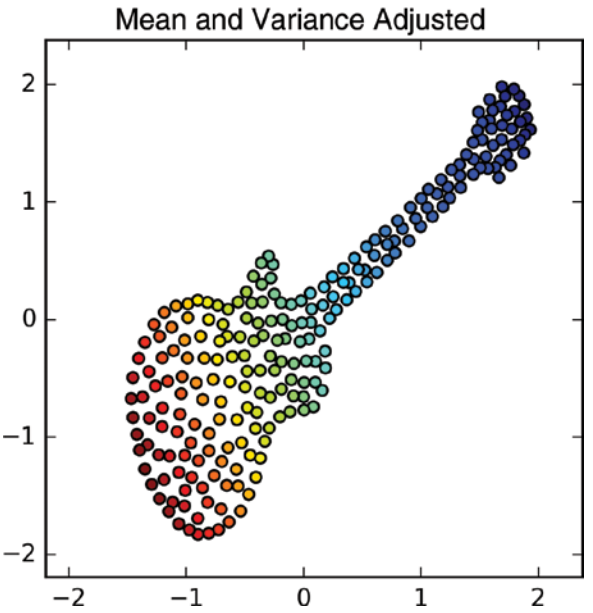


Figure 10-24: Our input data after standardizing

The results of projecting the data in Figure 10-24 onto a horizontal projection line are shown in Figure 10-25.

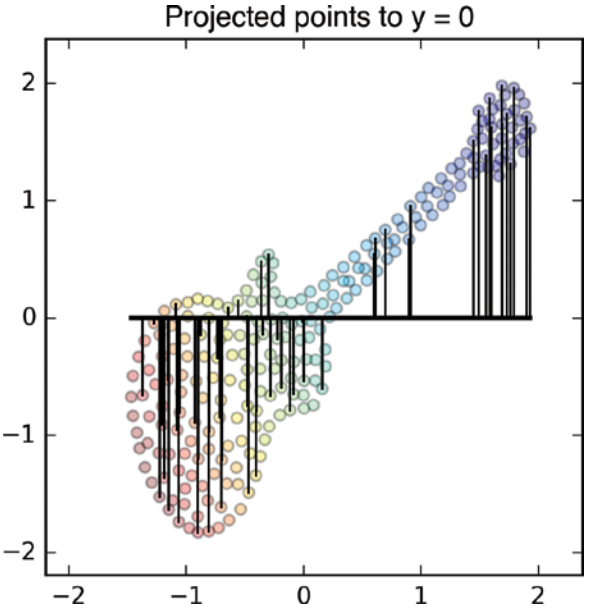


Figure 10-25: We project each data point of the guitar by moving it to its nearest point on the projection line. For clarity, we're showing the path taken by only about 25 percent of the points.

The results after all the points are processed is shown in Figure 10-26.

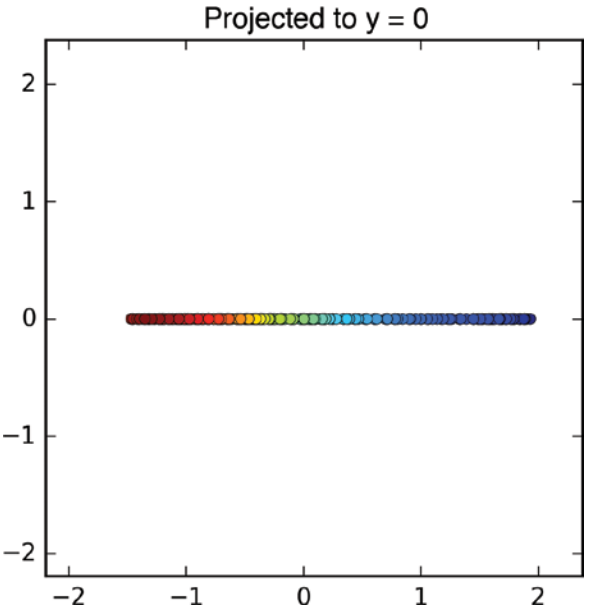


Figure 10-26: The result of Figure 10-25 after all the points have been moved to the projection line. Each point now is described only by its x coordinate, resulting in a one-dimensional dataset.

This is the one-dimensional dataset we were after, because the points only differ by their x value (the y value is always 0, so it's irrelevant). But this would be a lousy way to combine our features, because all we did was throw away the y values. It's like computing BMI by simply using the weight and ignoring the height.

To improve the situation, let's include the step we skipped. Instead of using a horizontal projection line, we rotate the line around until it's passing through the direction of maximum variance. Think of this as the line that, after projection, has the largest range of points. Any library routine that implements PCA finds this line for us automatically. Figure 10-27 shows this line for our guitar data.

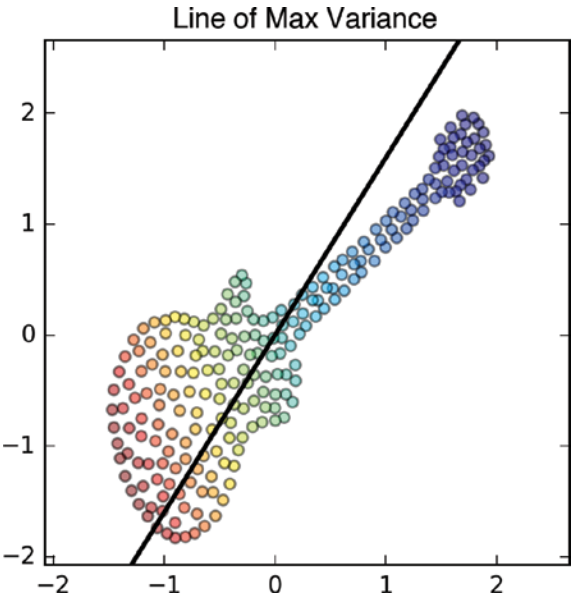


Figure 10-27: The thick black line is the line of maximum variance through our original data. This is our projection line.

Now we continue just like before. We project each point onto this projection line by moving it to its closest point on the line. As before, we do this by moving perpendicular to the line until we intersect it. Figure 10-28 shows this process.

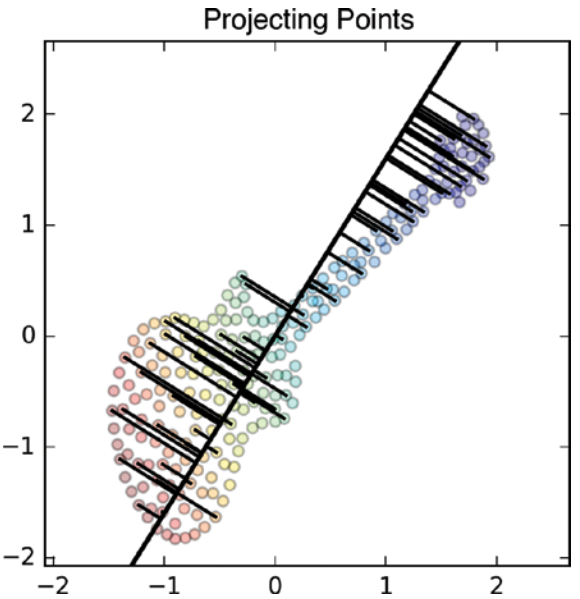


Figure 10-28: We project the guitar data onto the projection line by moving each point to its closest point on the line. For clarity, we're showing the path taken by only about 25 percent of the points.

The projected points are shown in Figure 10-29. Note that they all lie on the line of maximum variance that we found in Figure 10-27.

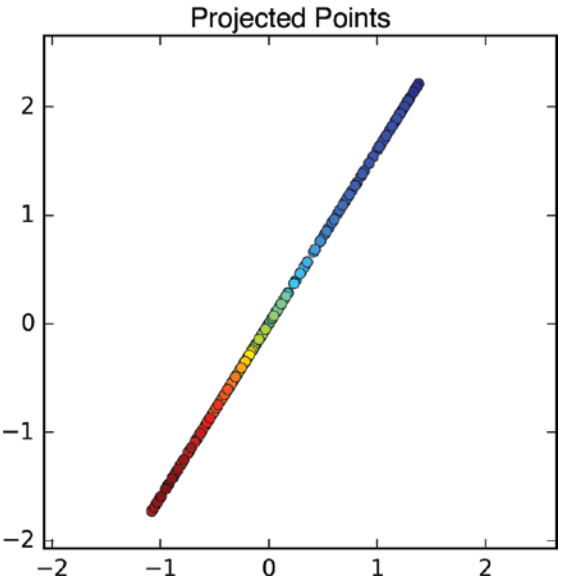


Figure 10-29: The points of the guitar dataset projected onto the line of maximum variance

For convenience, we can rotate this line of points to lie on the X axis, as shown in Figure 10-30. Now the y coordinate is irrelevant again, and we have 1D data that includes information about each point from both its original x and y values.

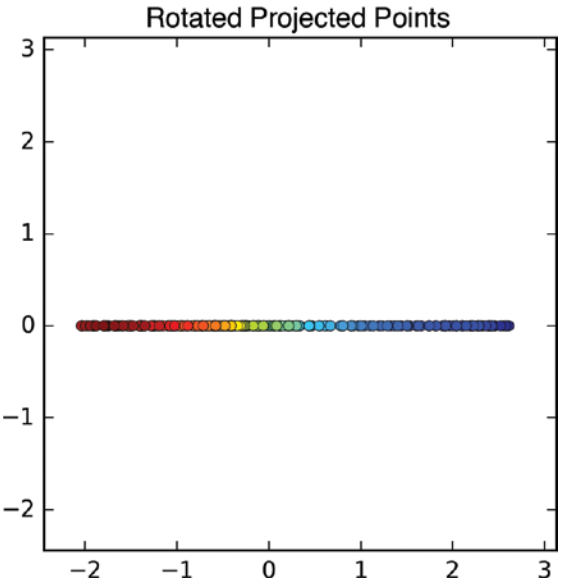


Figure 10-30: Rotating the points of Figure 10-29 into a horizontal position

Although the straight line of points in Figure 10-30 looks roughly like the line of points in Figure 10-26, they're different, because the points are distributed differently along the X axis. In other words, they have different values, because they were computed by projecting onto a tilted line, rather than a horizontal one. Figure 10-31 shows the two projections together. The PCA result is not just longer, but the points are also distributed differently.

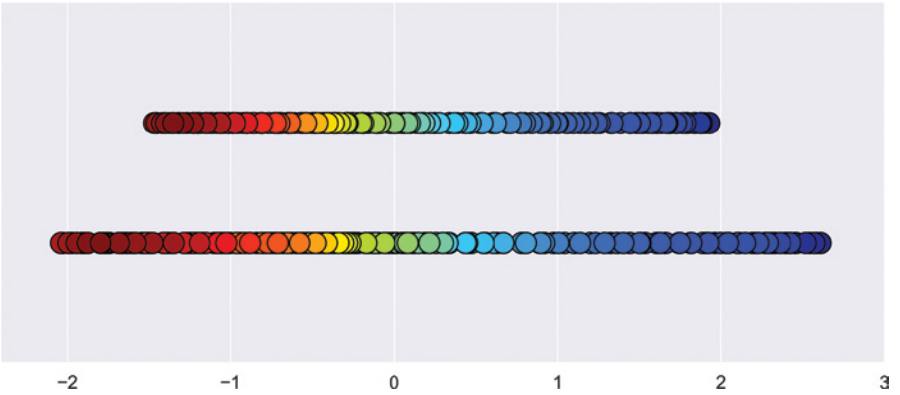


Figure 10-31: Comparing the points created by projection to y = 0 in Figure 10-26 (top) and the PCA algorithm in Figure 10-30 (bottom)

All of the steps we just discussed are carried out automatically by a machine learning library when we call its PCA routine.

The beauty of the 1D data created by this projection step is that every point's single value (its x value) is a combination of the 2D data it started with. We reduced the dimensionality of our dataset by one dimension, but we did so while retaining as much information as we could. Our learning algorithms now only have to process one feature rather than two, so they'll run faster. Of course, we've thrown some information away, so the accuracy may suffer. The trick to using PCA effectively is to choose dimensions that can be combined while still staying within the performance goals of our project.

If we have 3D data, we can imagine placing a plane in the middle of the cloud of samples and projecting our data down onto the plane. The library's job is to find the best orientation of that plane. This takes our data from 3D to 2D. If we want to go all the way down to 1D, we can imagine projecting the data onto a line through the volume of points. In practice, we can use this technique in problems with any number of dimensions, where we may reduce the dimensionality of the data by tens or more.

The critical questions for this kind of algorithm include: How many dimensions should we try to compress? Which dimensions should be combined? How they should get combined? We usually use the letter *k* to stand for the number of dimensions remaining in our data after PCA has done its work. So, in our guitar example, *k* was 1. We can call *k* a parameter of the algorithm, though usually we call it a hyperparameter of the entire learning

system. As we've seen, the letter *k* is used for lots of different algorithms in machine learning, which is unfortunate; it's important to pay attention to the context when we see references to *k*.

Compressing too little means our training and evaluation steps are going to be inefficient, but compressing too much means we risk eliminating important information that we should have kept. To pick the best value for the hyperparameter *k*, we usually try out a few different values to see how they do and then pick the one that seems to work best. We can automate this search using the techniques of *hyperparameter searching* provided by many libraries.

As always, whatever PCA transformations we use to compress our training data must also be used in the same way for all future data.

PCA for Simple Images

Images are an important and special kind of data. Let's apply PCA to a simple set of images.

Figure 10-32 shows a set of six images, perhaps drawn from a huge dataset of tens of thousands of such pictures. If these grayscale images are 1,000 pixels on a side, each contains 1,000 × 1,000, or 1 million, pixels. Is there any better way to represent them than with a million numbers each?

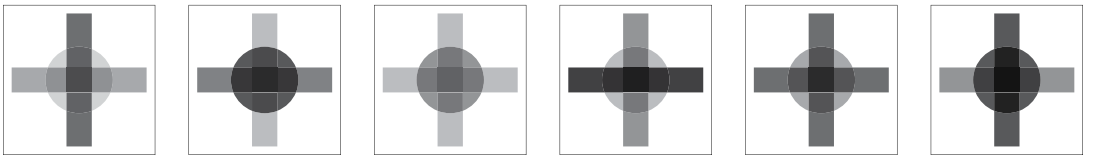


Figure 10-32: Six images we'd like to represent

Let's start by observing that each image in Figure 10-32 can be recreated from the three images in Figure 10-33, each scaled by a different amount and then added together.

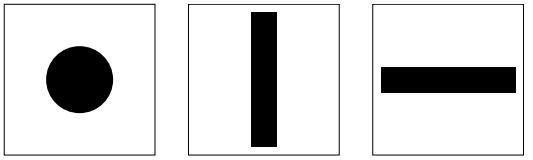


Figure 10-33: We can create all six images in Figure 10-32 by scaling these three by different amounts and adding the results.

For example, we can reconstruct the first image in Figure 10-32 by adding 20 percent of the circle, 70 percent of the vertical box, and 40 percent of the horizontal box. We often call these scaling factors the *weights*. The weights for each of the six starting images are shown in Figure 10-34.

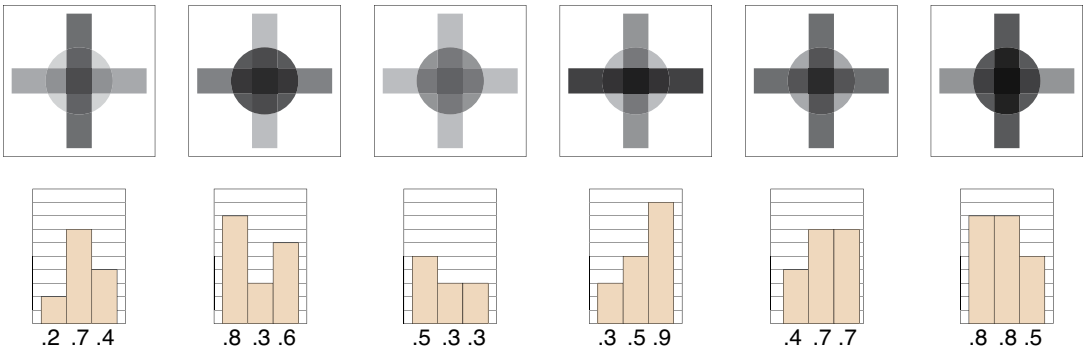


Figure 10-34: The weights to use when scaling the three images in Figure 10-33 to recover the images in Figure 10-32

Figure 10-35 shows the process of scaling and combining the components to recover the first original image.

In general, we can represent any image of this type with the three weights of the component images that make the best match. To reconstruct any of the input images, we need the three simpler pictures (1 million values each) plus the three numbers for that specific image. If we had 1,000 images, storing each one would take a total of 1,000 megabytes. But using this compressed form, we need a total of only 3.001 megabytes.

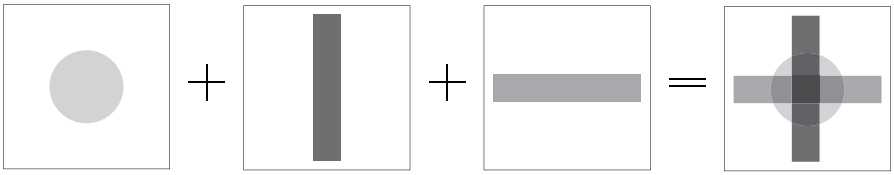


Figure 10-35: Recovering the first image in Figure 10-32 by scaling the images in Figure 10-33

For these simple images, it's easy to find the components as three geometric shapes. But when we have more realistic pictures, this won't generally be possible.

The good news is that we can use the projection technique we discussed earlier. Instead of projecting a collection of points to create a new set of points on a line, we can project a collection of images to create a new image. This is a more abstract process than the one we followed with the guitar points, but the concept is the same. Let's get a feeling for how PCA handles images by skipping the mechanics and focusing on the results.

Consider again the six starting images of Figure 10-32. Remember that these are grayscale images, not vector drawings. Let's ask PCA to find a grayscale image that comes closest to representing all the images, in the same way that our diagonal line came closest to representing all the points in our guitar. Then we can represent each starting image as a sum of this image, scaled by an appropriate amount, plus whatever is left over. Figure 10-36 shows this process.

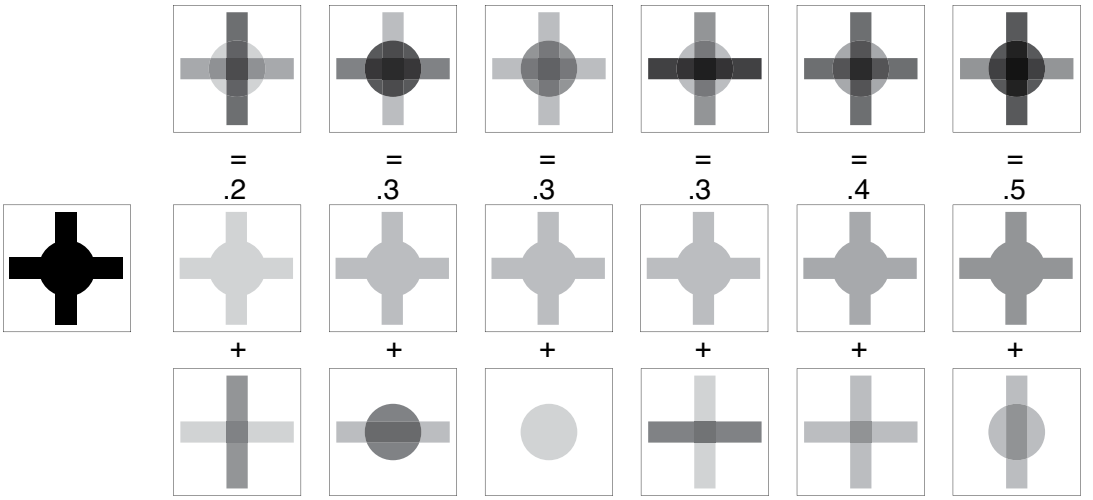


Figure 10-36: Running PCA on the images from Figure 10-32

The starting images from 10-32 are shown at the top of Figure 10-36.

Now we ask PCA to find an image that corresponds to the line in Figure 10-28. That is, an image that, in some sense, captures something from all of the inputs. Let's say it found the picture at the left in Figure 10-36, showing the two bars and the circle all in black and overlaid. We'll call this the *shared image*, which isn't a formal term but is useful here.

Let's now represent each image at the top of Figure 10-36 as a combination of a scaled version of the shared image, and some other image. To do this, we find the lightest pixel in each input image, and scale the shared image to that intensity. That scaling factor is shown at the top of the copies of the common image in the middle row, where the copies have been scaled in intensity by that amount. If we subtract each scaled common image from the source image above it in the figure, we get their difference. We could write this as "source – common = difference," or equivalently, "source = common + difference," which is shown in the figure.

We can then run PCA again, this time on the bottom row of Figure 10-36. Again, it projects these six images to create a new image that is the best match for all of them. As before, we can represent each picture as the sum of a scaled version of what's common, plus whatever is left. Figure 10-37 shows the idea. In this demonstration, we're supposing that PCA created an image of the two overlapping boxes as the best matching image.

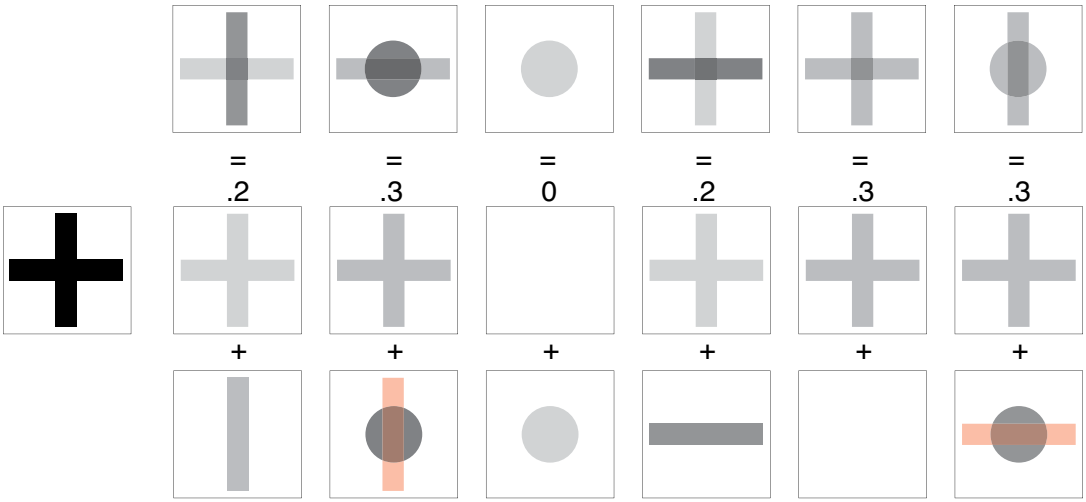


Figure 10-37: Running PCA on the bottom row of Figure 10-36

Something interesting happened in two images on the bottom row. Let's look at the second column from the left. The image we want to match on the top row is a circle and horizontal box, but we're trying to match it with a scaled pair of crossed boxes. To match the top image, we need to add in some of the circle, but subtract the vertical box that we just introduced. That just means setting the corresponding data in the bottom image to negative values. These are perfectly valid numbers to place into our data, though we have to be careful if we try to display this image directly. If we reconstruct the top image by adding up the two images beneath it, the negative values in the red region in the bottom row cancel out the positive values in the vertical box in the middle row, so the sum of these images matches the circle and horizontal box at the top. The same reasoning applies to the horizontal box in the rightmost column.

Figure 10-38 summarizes the two steps we've seen so far.

We took only two steps here, but we can repeat this process dozens or hundreds of times.

To represent each starting image, we only need the collection of common images and the weight we assigned to each. Since the common images are shared by all the images, we can consider them a shared resource. Then each image can be completely described by a reference to this shared resource, and the list of weights to be applied to the common images.

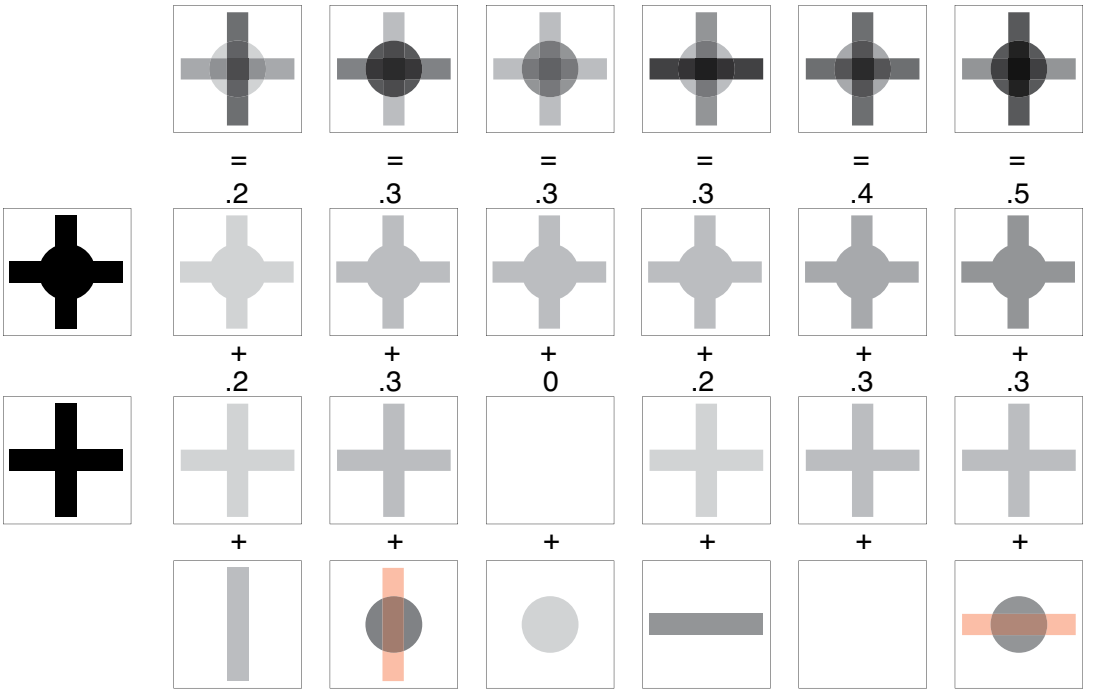


Figure 10-38: Representing each starting image (top row) as the sum of two scaled component images (second and third rows), plus whatever's left (bottom row)

Each of the common images is called a *component*, and the one we create at every step is the *principal* component (since it's the best of all possible components, in the same way that the line in Figure 10-28 was the best line). We find these principal components by analyzing the input images. Hence, the name *principal component analysis*.

The more components we include, the more accurately each reconstructed image will match its original. We usually aim to produce enough components so that each reconstructed image has all the qualities we care about in its original.

In this discussion, most of the weights were positive, but we also saw some component images that should be subtracted, not added, and thus they produce weights with negative values. This is so that the final pixels, when all the components are summed together, have the desired values.

How well did we do with just two component images? Figure 10-39 shows the original six images and our reconstructed images using the sum of the middle two rows of Figure 10-38.

The matches aren't perfect, but they're a good start, particularly for just two components. So, we have a common pool of two images (requiring 1 million numbers each), and then each image itself can be described with just two numbers. The beauty of this scheme is that our algorithms never need to see the common images. We just need those for finding the weights that describe each input image (and to reconstruct the images, if we want). As far as the learning algorithm is concerned, each image is described by

only two numbers. This means our algorithms consume less memory and run more quickly.

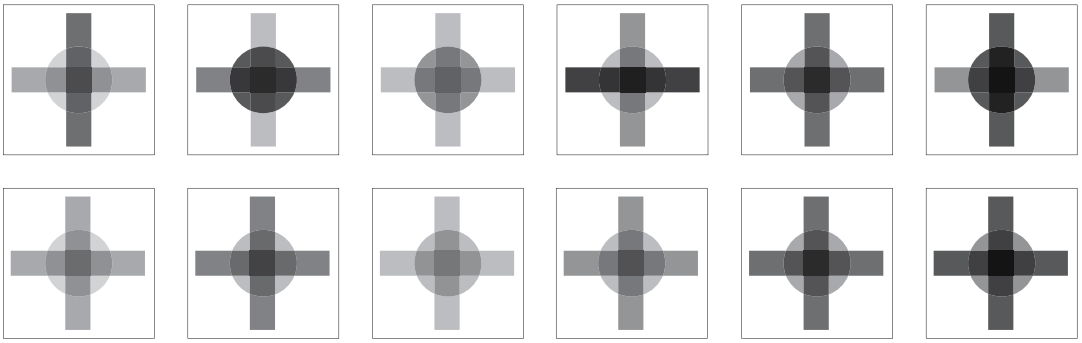


Figure 10-39: Our starting six images (top), and the reconstructed images from Figure 10-35c (bottom)

We skipped a step in this example: normally when we use PCA, we standardize all the images first. That's included in our next example.

PCA for Real Images

The images in the last section were contrived for simplicity. Now we'll apply PCA to real pictures.

Let's begin with the six pictures of huskies shown in Figure 10-40. To make our processing easier to see, these images are only 45 × 64 pixels. These were aligned by hand so that the eyes and nose are in about the same location in each image. This way each pixel in each image has a good chance of representing the same part of a dog as the corresponding pixel in the other images. For instance, a pixel just below the center is likely to part of a nose, one near the upper corners is likely to be an ear, and so on.



Figure 10-40: Our starting set of huskies

A database of six dogs isn't a lot of training data, so let's enlarge our database using the idea of *data augmentation*, a common strategy for *amplifying* or *enlarging* a dataset. In this case, let's run through our set of six images in random order over and over. Each time through, we'll pick an image, make a copy, randomly shift it horizontally and vertically up to 10 percent on each axis, rotate it up to five degrees clockwise or counterclockwise, and maybe flip it left to right. Then we append that transformed image to our training set. Figure 10-41 shows the results of the first two passes through

our six dogs. We used this technique of creating variations to build a training set of 4,000 dog images.

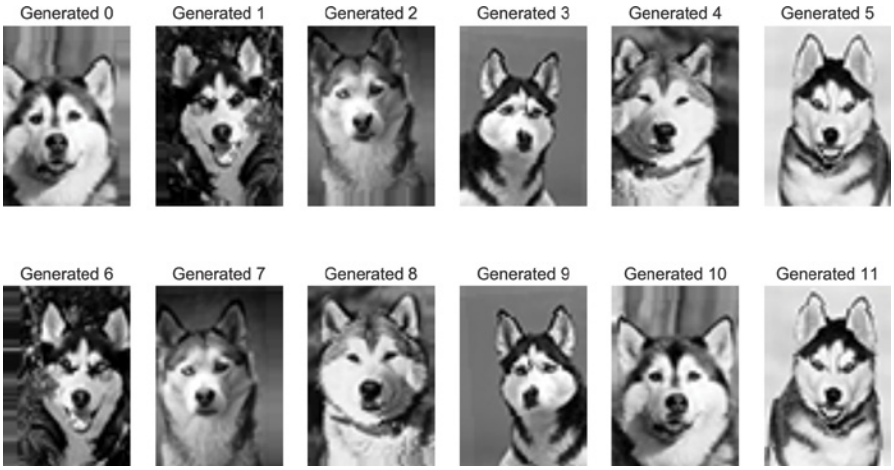


Figure 10-41: Each row shows a set of new images created by shifting, rotating, and perhaps horizontally flipping each of our input images.

Since we'd like to run PCA on these images, our first step is to standardize them. This means we analyze the same pixel in each of our 4,000 images and adjust the collection to have zero mean and unit variance (Turk and Pentland 1991). The standardized versions of our first six generated dogs are shown in Figure 10-42.



Figure 10-42: Our first six huskies after standardization

Since 12 images fit nicely into a figure, let's begin by arbitrarily asking PCA to find the 12 images that, when added together with appropriate weights, best reconstruct the input images. Each of the projections (or components) found by PCA is technically known as an *eigenvector*, from the German *eigen* meaning "own" (or roughly "self"), and *vector* from the mathematical name for this kind of object. When we create eigenvectors of particular types of things, it's common to create a playful name by combining the prefix eigen with the object we're processing. Hence, Figure 10-43 shows our 12 *eigendogs*.

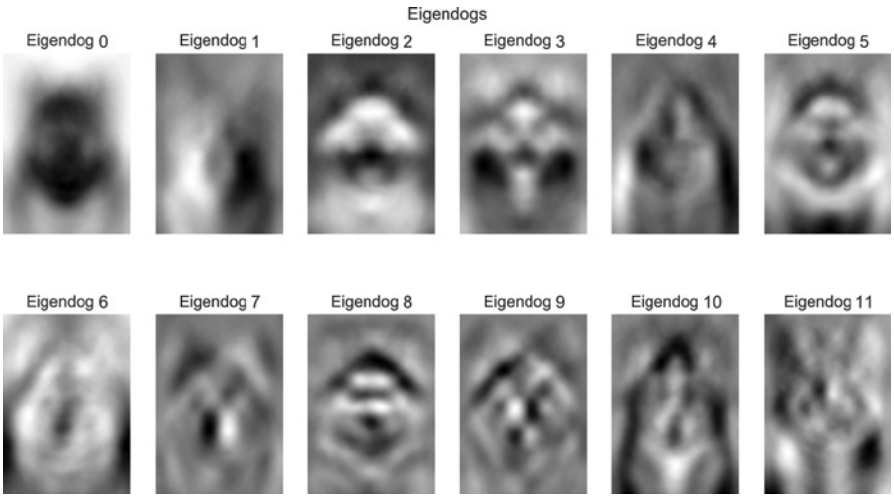


Figure 10-43: The 12 eigendogs produced by PCA

Looking at the eigendogs tells us a lot about how PCA is analyzing our images. The first eigendog is a big smudge that is darker roughly where most of the dogs appear in the image. This is the single image that comes closest to approximating every input image. The second eigendog gives us refinements to the first, capturing some of the left-right shading differences.

The next eigendog provides additional detail, and so it goes through all 12 eigendogs. So, the first eigendog captures the broadest, most common features, and each additional eigendog lets us recover a little more detail.

PCA is able not only to create the eigendogs of Figure 10-43, but also to take as input any picture, and tell us the weight to apply to each eigendog image so that, when the weighted images are added together, we get the best approximation to the input image.

Let's see how well we can recover our original images by combining these 12 eigendogs with their corresponding weights. Figure 10-44 shows the weights that PCA found for each input image. We create the reconstructed dogs by scaling each eigendog image from Figure 10-43 with its corresponding weight, and then adding the results together.

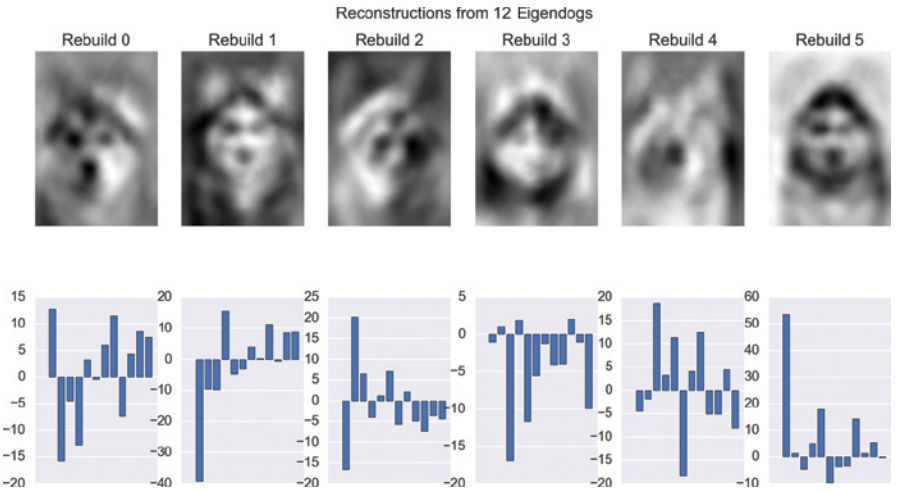


Figure 10-44: Reconstructing our original inputs from a set of 12 eigendogs. Top row: The reconstructed dogs. Bottom row: The weights applied to the eigendogs of Figure 10-43 to build the image directly above. Notice that the vertical scales on the bottom row are not all the same.

The recovered dogs in Figure 10-44 are not great. We've asked PCA to represent all 4,000 images in our training set with just 12 pictures. It did its best, but these results are pretty blurry. They do seem to be on the right track, though.

Let's try using 100 eigendogs. The first 12 eigendog images look just like those in Figure 10-43, but then they get more complicated and detailed. The results of reconstructing our first set of 6 dogs are shown in Figure 10-45.

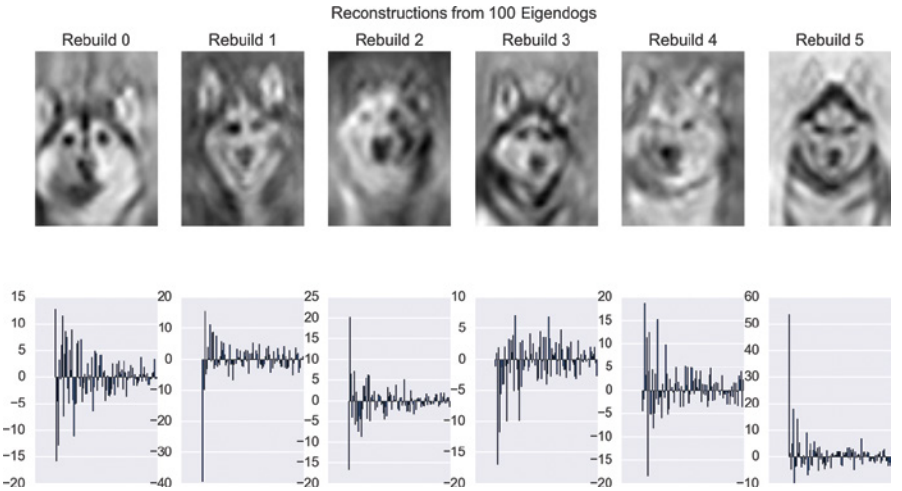


Figure 10-45: Reconstructing our original inputs from a set of 100 eigendogs

That's better! They're starting to look like dogs. But it seems that 100 eigendogs is still not enough.

Let's crank up our number of eigendogs to 500 and try again. Figure 10-46 shows the results.

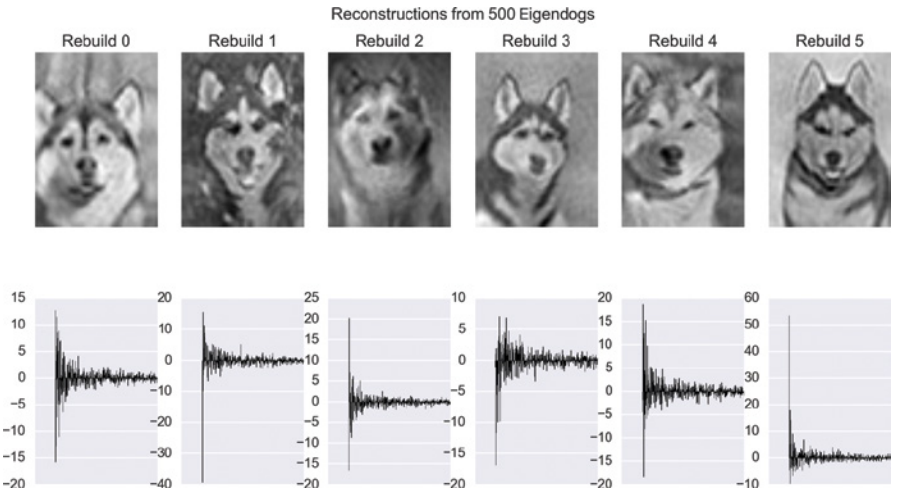


Figure 10-46: Reconstructing our original inputs from a set of 500 eigendogs

These are looking pretty great. They are all easily recognized as the 6 standardized dogs in Figure 10-42. They're not perfect, but considering that we're adding together different amounts of 500 shared images, we've done a fine job of matching the original images. There's nothing special about these first 6 images. If we look at any of the 4,000 images in our database, they all look this good. We could keep increasing the number of eigendogs, and the results would continue to improve, with the images getting increasingly sharper and less noisy.

In each plot of the weights, the eigendog images that get the most weight are the ones at the start, which capture the big structures. As we work our way down the list, each new eigendog is generally weighted a little less than the one before, so it contributes less to the overall result.

The value of PCA here is not that we can make images that look just like the starting set, but rather, that we can use the eigendogs' representation to reduce the amount of data our deep learning system has to process. This is illustrated in Figure 10-47. Our set of input dogs goes into PCA, which generates a set of eigendogs. Then each dog we'd like to classify goes into PCA again, which gives us the weights for that image. Those are the values that go into the classifier.

As we mentioned earlier, rather than training our categorizer on all of the pixels from each image, we can train it on just that image's 100 or 500 weights. The categorizer never sees a full image of a million pixels. It never even sees the eigendogs. It just gets a list of the weights for each image, and that's the data it uses for analysis and prediction during training. When we

want to classify a new image, we provide just its weights, and the computer gives us back a class. This can save a lot of computation, which translates to a savings in time, and perhaps increased quality of final results.

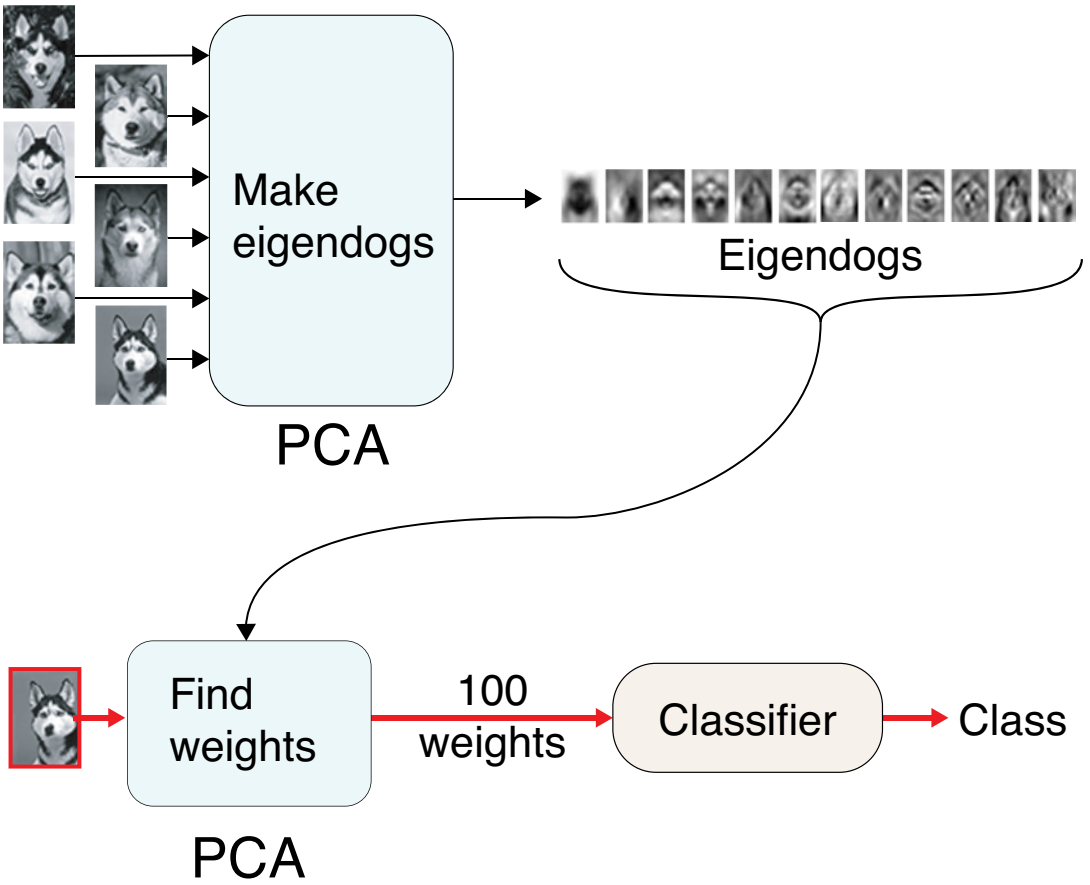


Figure 10-47: In the top row we first we use PCA to build a set of eigendogs, and then in the bottom row we find the weights for each input to our classifier, which only uses those weights to find the input's class.

To summarize, the data we hand the classifier is not each input image, but its weights. The classifier then proceeds to work out which breed of dog it's looking at based just on those weights. Often we need only a few hundred weights to represent input samples with many thousands, or even millions, of features.

Summary

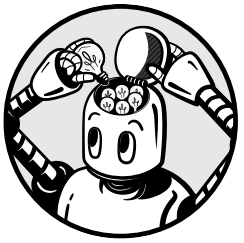
In this chapter, we looked at ways to prepare data. We saw that it's important to inspect our data before we do anything with it and make sure that it's clean. Once our data is clean, we can transform it to better fit our learning algorithms in a number of ways. These transformations are built from the training data only. It's important to remember that any transforms we

apply to the training data must then be applied to every additional sample we give to our algorithm, from validation and test data to deployment data provided by real-world users.

In the next chapter, we'll dig into classifiers and survey some of the most important algorithms for the job.

11

CLASSIFIERS



In this chapter, we introduce four important classification algorithms, building on the basics of classification that we covered in Chapter 7. We often use these algorithms to help us study and understand our data. In some cases, we can even build a final classification system from them. In other cases, we can use the understanding we gain from these methods to design a deep learning classifier, as discussed in later chapters.

We will usually illustrate our classifiers using 2D data and usually only two classes because that's easy to draw and understand, but modern classifiers are capable of handling data with any number of dimensions (or features) and huge numbers of classes. Modern libraries let us apply most of these algorithms to our own data with just a handful of lines of code.

Types of Classifiers

Before we get into specific algorithms, let's break down the world of classifiers into two main approaches: *parametric* and *nonparametric*.

In the parametric approach, we usually think of the algorithm as starting with a preconceived description of the data it's working with, and it then searches for the best parameters of that description to make it fit. For instance, if we think that our data follows a normal distribution, we can look for the mean and standard deviation that best fit it.

In the nonparametric approach, we let the data lead the way, and we try to come up with some way to represent it only after we've analyzed it. For example, we may look at all of the data and attempt to find a boundary that splits it into two or more classes.

In reality, these two approaches are more conceptual than strict. For example, we can argue that simply choosing a particular kind of learning algorithm means that we're making assumptions about our data. And we can argue that we're always learning about the data itself by processing it. But these are useful generalizations, and we'll use them to organize our discussion.

Let's start by looking at two nonparametric classifiers.

k-Nearest Neighbors

We begin with a nonparametric algorithm called *k-nearest neighbors*, or *kNN*. As usual, the letter *k* at the start refers not to a word but to a number. We can pick any integer that's 1 or larger. Because we set this value before the algorithm runs, it's a hyperparameter.

In Chapter 7, we saw an algorithm called *k-means clustering*. Despite the similarity in names, that algorithm and *k*-nearest neighbor are different techniques. One key difference is that *k*-means clustering learns from unlabeled data, whereas kNN works with labeled data. In other words, *k*-means clustering and kNN fall into the classes of unsupervised and supervised learning, respectively.

kNN is fast to train, because all it does is save a copy of every incoming sample into a database. The interesting part comes when training is complete, and a new sample arrives to be classified. The central idea of how kNN classifies a new sample is appealingly geometric, as shown in Figure 11-1.

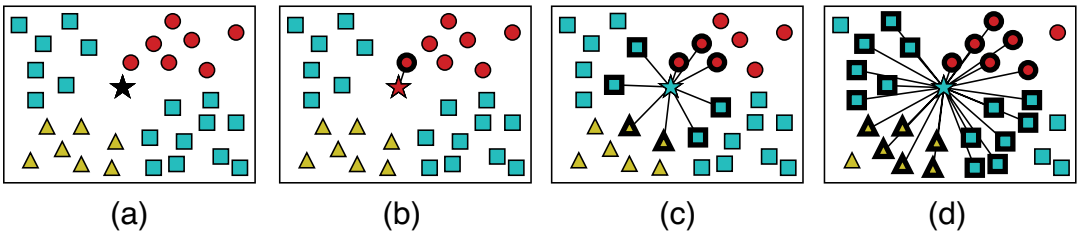


Figure 11-1: To find the class for a new sample, shown as a star, we find the most popular of its k *neighbors.*

In Figure 11-1(a) we have a sample point (a star) amid a bunch of other samples that represent three classes (circle, square, and triangle). To determine a class for our new sample, we look at the k nearest samples (or *neighbors*), and we count up their classes. Whichever class is most populous becomes the new sample's class. We show which samples are considered for different values of k by a line to each of the k nearest samples. In Figure 11-1(b) we've set k to 1, meaning we want to use the class of the nearest sample. In this case it's a red circle, so this new sample is classified as a circle. In Figure 11-1(c) we've set k to 9, so we look at the nine nearest points. Here we find 3 circles, 4 squares, and 2 triangles. Because there are more squares than any other class, the star is classified as a square. In Figure 11-1(d) we've set k to 25. Now we have 6 circles, 13 squares, and 6 triangles, so again the star is classified as a square.

To sum this up, kNN accepts a new sample to evaluate, along with a value of k. It then finds the closest k samples to the new sample, and we assign the new sample to the class with the largest number of representatives among the k samples we found. There are various ways to break ties and handle exceptional cases, but that's the basic idea.

Note that kNN does not create explicit boundaries between groups of points. There's no notion here of regions or areas that samples belong to. We say that kNN is an *on-demand*, or *lazy*, algorithm because it does no processing of the samples during the learning stage. When learning, kNN just stashes the samples in its internal memory and it's done.

kNN is attractive because it's simple, and training it is usually exceptionally fast. On the other hand, kNN can require a lot of memory, because (in its basic form) it's saving all the input samples. Using large amounts of memory can slow down the algorithm. Another problem is that the classification of new points is often slow (compared to other algorithms we'll see) because of the cost of searching for neighbors. Every time we want to classify a new piece of data, we have to find its k nearest neighbors, which requires work. Of course, there are many ways to enhance the algorithm to speed this up, but it still remains a relatively slow way to classify. For applications where classification speed is important, like real-time systems and websites, the time required by kNN to produce each answer can take it out of the running.

Another problem with this technique is that it depends on having lots of neighbors nearby (after all, if the nearest neighbors are all very far away, then they don't offer a good proxy for other examples that are like the one we're trying to classify). This means we need a lot of training data. If we have lots of features (that is, our data has many dimensions) then kNN quickly succumbs to the curse of dimensionality, which we discussed in Chapter 7. As the dimensionality of the space goes up, if we don't also significantly increase the number of training samples, then the number of samples in any local neighborhood drops, making it harder for kNN to get a good collection of nearby points.

Let's put kNN to the test. In Figure 11-2 we show a "smile" dataset of 2D data that falls into two classes. In this figure, and the similar ones that follow, the data is made of points. Since points are hard to see, we're drawing a filled circle around each point as a visual aid.

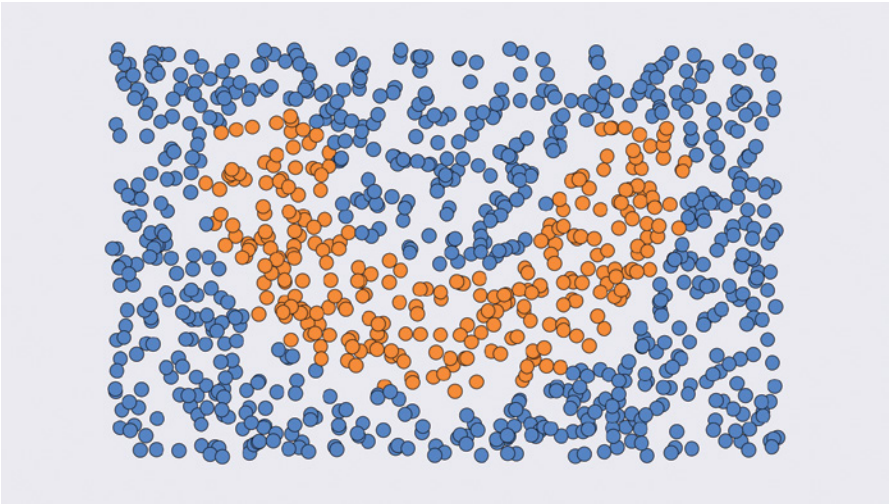


Figure 11-2: A smile dataset of 2D points. There are two classes, blue and orange.

Using kNN with different values of *k* gives us the results in Figure 11-3.

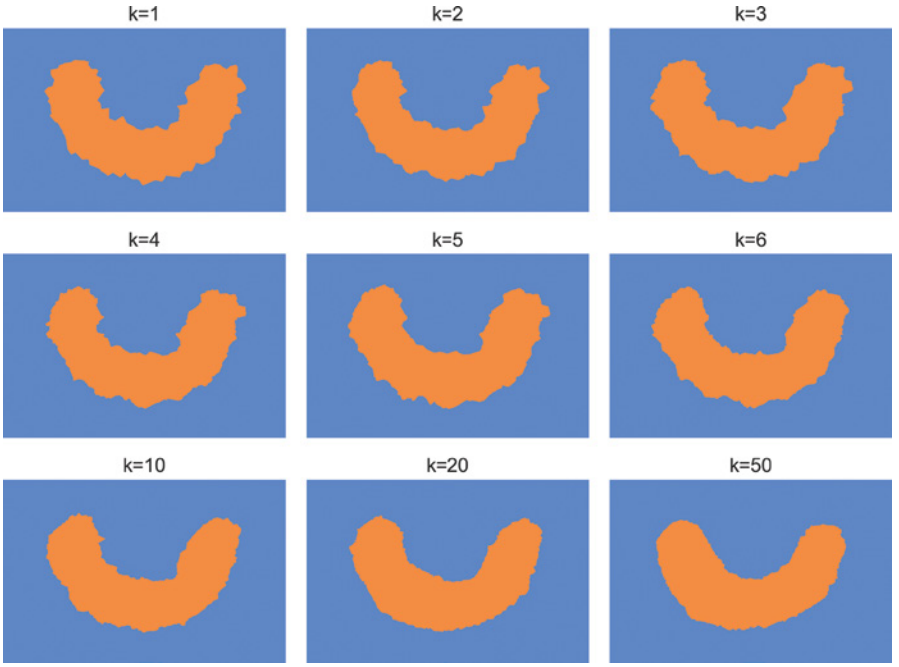


Figure 11-3: Classifying with the points in Figure 11-2 using kNN for different values of k

Although kNN doesn't produce explicit classification boundaries, we can see that when *k* is small and we compare our input point with only a few neighbors, the space is broken up into sections that share a rather rough border. As *k* gets larger and we use more neighbors, that border smooths out because we're getting a better overall picture of the environment around the new sample.

To make things more interesting, let's add some noise to our data so that the edges aren't so easy to find. Figure 11-4 shows a noisy version of Figure 11-2.

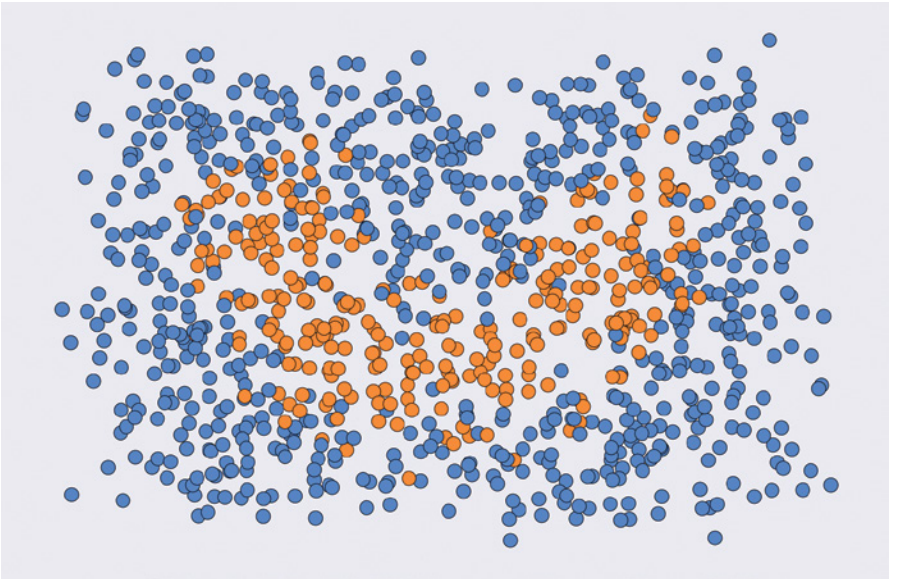


Figure 11-4: A noisy version of the smile dataset from Figure 11-2

The results for different values of k are shown in Figure 11-5.

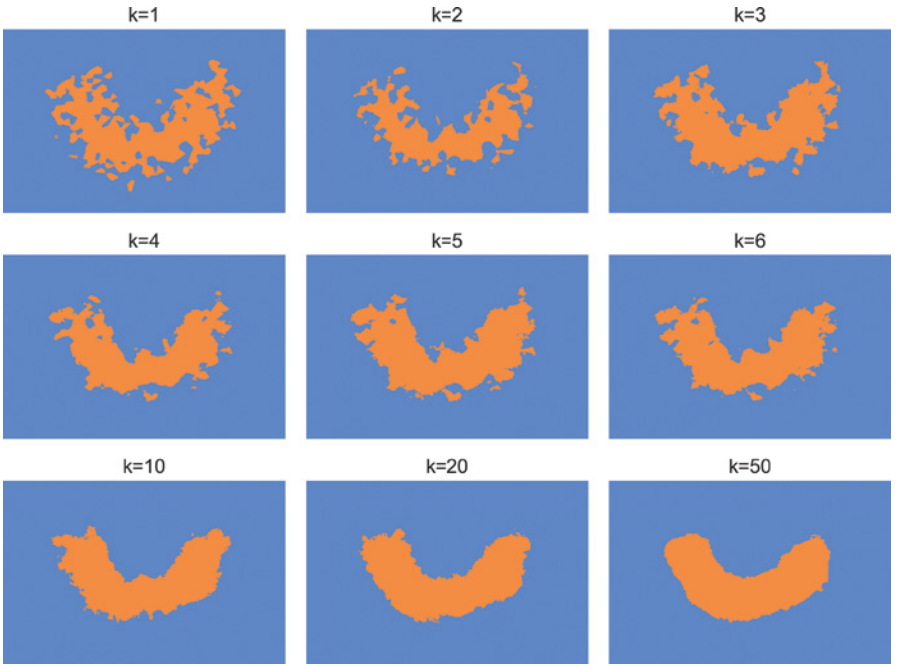


Figure 11-5: Using kNN and Figure 11-4 to assign a class to points in the plane

We can see that in the presence of noise, small values of k lead to ragged borders. In this example, we have to get k up to 50 before we see fairly smooth boundaries. Also notice that as k increases, the smile shape contracts, since the perimeter gets eroded by the larger number of background points.

Because kNN doesn't explicitly represent the boundaries between classes, it can handle any kind of boundaries, or any distribution of classes. To see this, let's add some eyes to our smile, creating three disconnected sets of the same class. The resulting noisy data is in Figure 11-6.

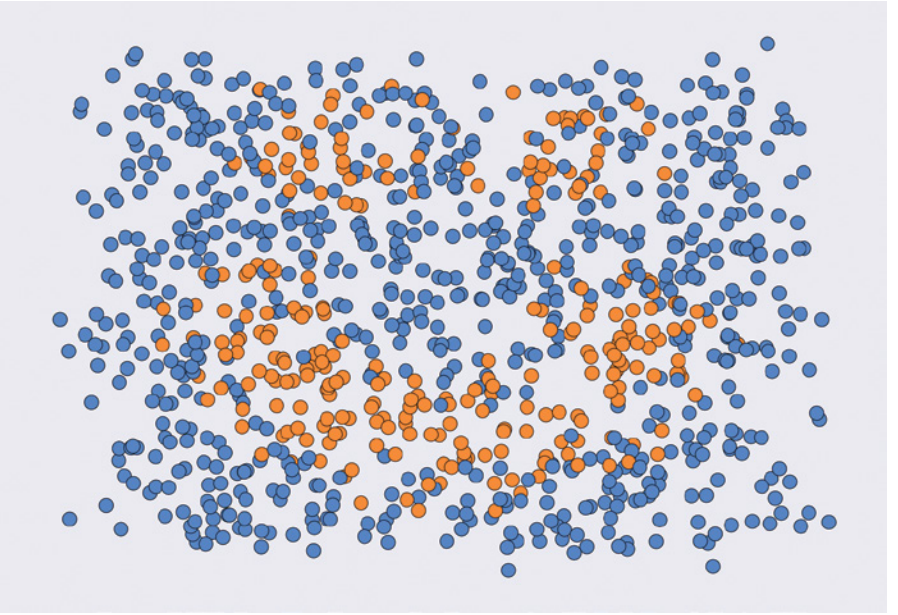


Figure 11-6: A noisy dataset of a smile and two eyes

The resulting classifications for different values of k are shown in Figure 11-7.

In this example, a value of about 20 for k looks best. Too small a value of k can give us ragged edges and noisy results, but too large a value of k can start to erode the features. As is so often the case, finding the best hyperparameter for this algorithm for any given dataset is a matter of repeated experimentation. We can use cross-validation to automatically score the quality of each result, which is particularly useful when there are many dimensions.

kNN is a great nonparametric algorithm: it's easy to understand and program, and when the dataset isn't too big, training is extremely fast and classification of new data isn't too slow. But when the dataset gets large, kNN becomes less appealing: memory requirements go up because every sample is kept, and classification gets slower because the search gets slower. These problems are shared by most nonparametric algorithms.

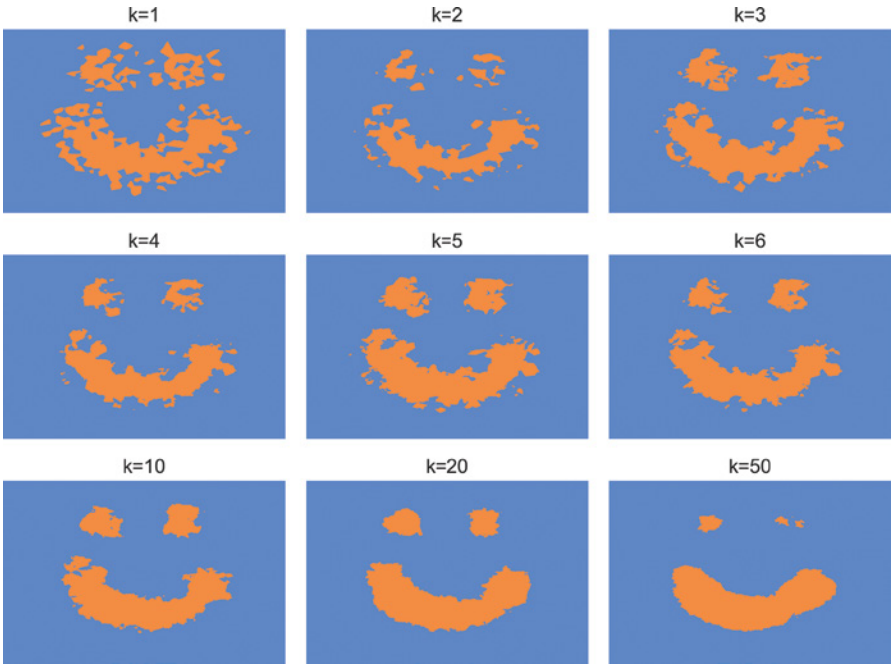


Figure 11-7: kNN doesn't create boundaries between clusters of samples, so it works even when a class is broken up into pieces.

Decision Trees

Let's consider another nonparametric classification method, called *decision trees*. This algorithm builds a data structure from the points in the sample set, which is then used to classify new points. Let's begin by taking a look at this structure, and then we'll see how to build it.

Introduction to Trees

We can illustrate the basic idea behind decision trees with the familiar parlor game called 20 Questions. In this game, one player (the chooser) thinks of a specific target object, which is often a person, place, or thing. The other player (the guesser) then asks a series of yes/no questions. If the guesser can correctly identify the target in 20 or fewer questions, they win. One reason the game endures is that it's fun to narrow down the enormous number of possible people, places, and things to one specific instance with such a small number of simple questions (perhaps surprisingly, with 20 yes/no questions we can only distinguish just over a million distinct targets).

We can draw a typical game of 20 questions in graphical form, as in Figure 11-8.

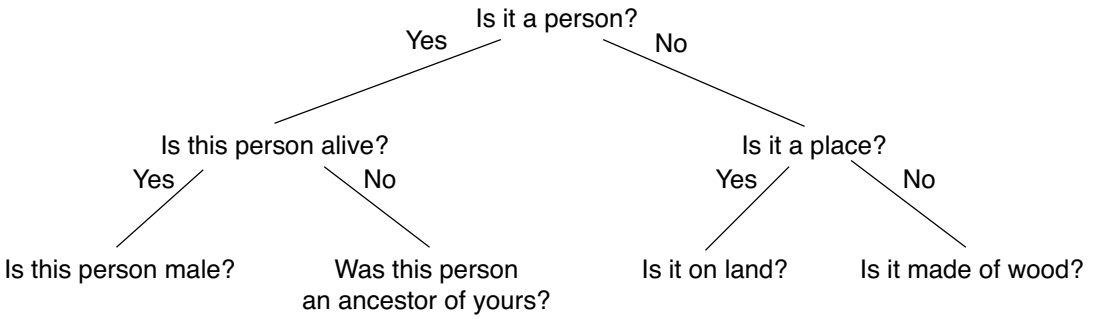


Figure 11-8: A tree for playing 20 questions. Note that after each decision there are exactly two choices, one each for "yes" and "no."

We call a structure like Figure 11-8 a *tree* because it looks something like an upside-down tree. Such trees have a bunch of associated terms that are worth knowing. We say that each splitting point in the tree is a *node*, and each line connecting nodes is a *link*, *edge*, or *branch*. Following the tree analogy, the node at the top is the *root*, and the nodes at the bottom are *leaves*, or *terminal nodes*. Nodes between the root and the leaves are called *internal nodes* or *decision nodes*.

If a tree has a perfectly symmetrical shape, we say the tree is *balanced*, otherwise it's *unbalanced*. In practice, almost all trees are unbalanced when they're made, but we can run algorithms to make them closer to being balanced if a particular application prefers that. We also say that every node has a *depth*, which is a number that gives the smallest number of nodes we must go through to reach the root. The root has a depth of 0, the nodes immediately below it have a depth of 1, and so on.

Figure 11-9 shows a tree with these labels.

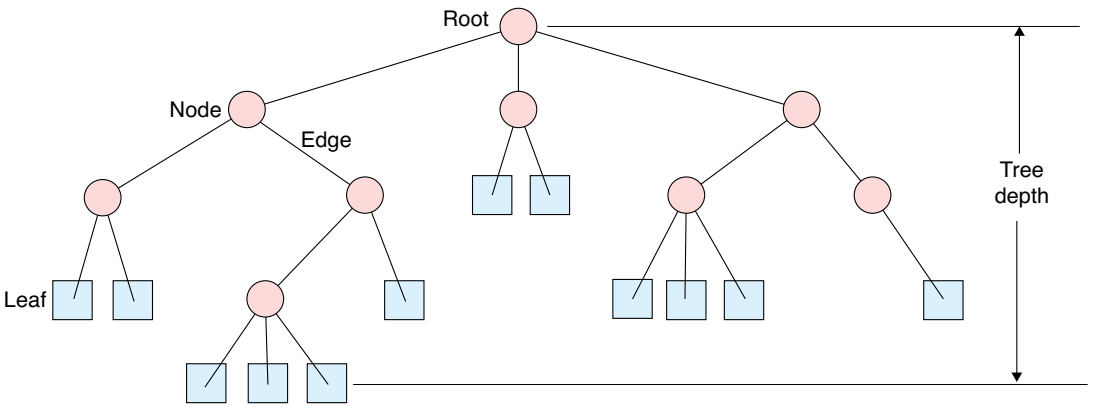


Figure 11-9: Some terminology for a tree

It's also common to use the terms associated with family trees, though these abstract trees don't require the union of two nodes to produce children. Every node (except the root) has a node above it. We call this the *parent* of that node. The nodes immediately below a parent node are its *children*. We sometimes distinguish between *immediate children* that are directly connected to a parent, and *distant children* that are at the same depth as immediate children, but connected to the parent through a sequence of other nodes. If we focus our attention on a specific node, then that node and all of its children taken together are called a *subtree*. Nodes that share the same immediate parent are called *siblings*.

Figure 11-10 shows some of these ideas graphically.

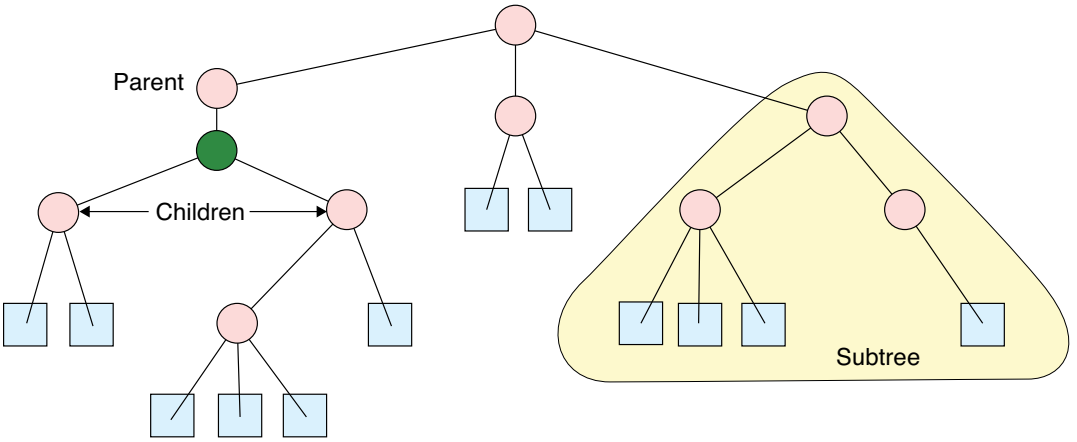


Figure 11-10: Using familiar terms with trees. The green node's parent is immediately above it, and its children are immediately below.

With this vocabulary in mind, let's return to our game of 20 Questions.

Using Decision Trees

An interesting quality of the 20 Questions tree shown in Figure 11-8 is that it's *binary*; every parent node has exactly two children: one for yes, one for no. Binary trees are a particularly easy kind of tree for some algorithms to work with. If some nodes have more than two children, we say that the tree overall is *bushy*. We can always convert a bushy tree to a binary tree if we want. An example of a bushy tree that tries to guess the month of someone's birthday is shown on the left of Figure 11-11, and the corresponding binary tree is shown on the right. Because we can easily go back and forth, we usually draw trees in whatever form is most clear and succinct for the discussion at hand.

We can use trees to classify data. When used this way, the trees are called *decision trees*. The full name of the approach is *categorical variable*

decision trees. This is to distinguish it from those times we use decision trees to work with continuous variables, as in regression problems. Those are called *continuous variable decision trees*.

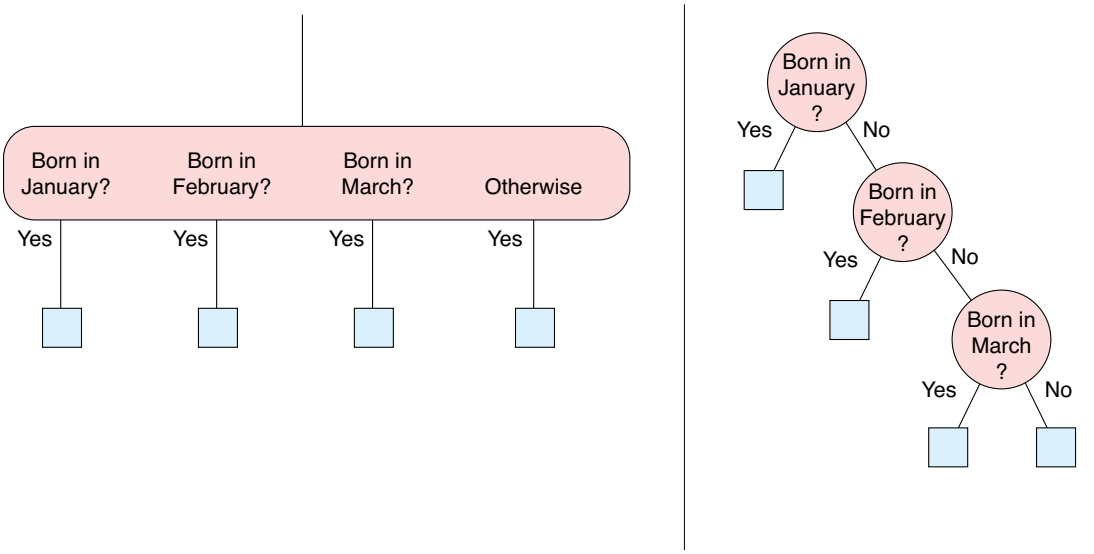


Figure 11-11: Turning a bushy tree (left) into a binary tree (right)

Let's stick with the categorical versions here. For simplicity, we'll just refer to them as decision trees, or simply trees, from now on. An example of such a tree for sorting inputs into different classes is shown in Figure 11-12.

In Figure 11-12, we start with a root node containing samples of different classes, distinguished by their shapes (and colors). To construct the tree, we *split* the samples at each node into two groups using some kind of test, resulting in a decision. For instance, the test applied at the root node might be, "Is this shape rectangular?" The test for the left child might be, "Is this shape taller than it is wide?" We'll soon see how to come up with such tests. The goal in this example is to keep splitting each node until we're left with samples of only one class. At that point, we declare that node to be a leaf, and stop splitting. In Figure 11-12, we've split our starting data into five classes. It's essential to remember the test we applied at each node. Now, when a new sample arrives, we can start at the root and apply the root's test, then the test of the appropriate child, and so on. When we finally reach a leaf, we have determined the class for that sample.

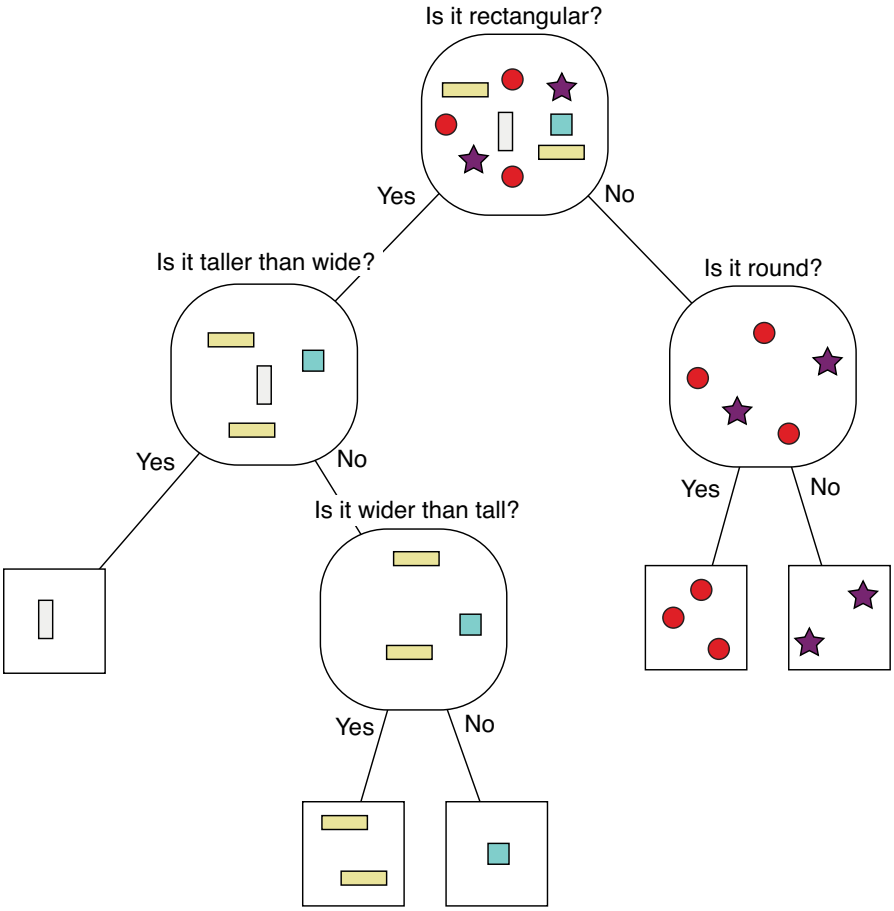


Figure 11-12: A categorical decision tree. Each class is a different shape and color.

Decision trees don't start out fully built. Instead, we build the tree based on the samples in the training set. When we reach a leaf node during training, we test to see if the new training sample has the same class as all the other samples in that leaf. If it does, we add the sample to the leaf and we're done. Otherwise, we come up with decision criteria based on some of the features that let us distinguish between this sample and the previous samples in the node. We then use this test to split the node. The test we come up with gets saved with the node, we create at least two children, and assign each sample to the appropriate child, as in Figure 11-13.

When we're done with training, evaluating new samples is easy. We just start at the root and work our way down the tree, following the appropriate branch at each node based on that node's test with this sample's features. When we land in a leaf, we report that the sample belongs to the class of the objects in that leaf.

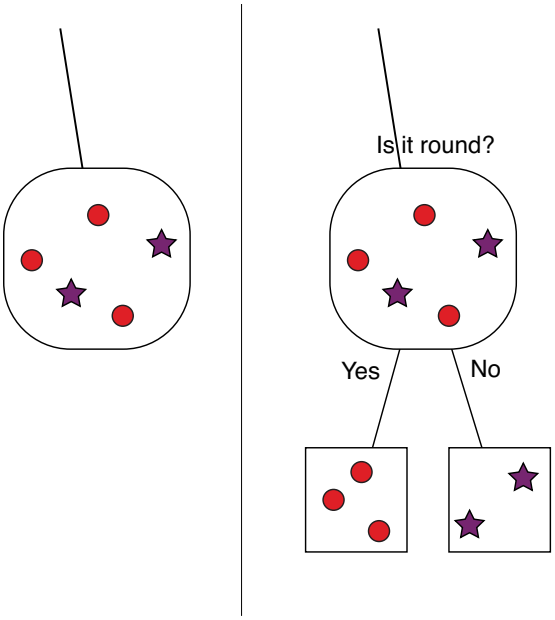


Figure 11-13: Splitting a node by applying a test to its contents

This is an idealized process. In practice, our tests can be imperfect and our leaves might contain a mixture of objects of different classes if, for efficiency or memory reasons, we chose not to split some leaves any more. For example, if we land in a node that contains samples that are 80 percent from class A and 20 percent from class B, we might report that the new sample has an 80 percent chance of being in A and a 20 percent chance of being in B. If we have to report just one class, we might report that it's A 80 percent of the time, and B the other 20 percent.

This process of making a decision tree works with just one sample at a time. In the simplest version of this technique, we don't consider the entire training set at once and try to find, say, the smallest or most balanced tree that classifies the samples. Instead, we consider one sample at a time and split the nodes in the tree as required to handle that sample. Then we do the same for the next sample, and the next, and so on, making decisions in the moment without caring about any other data yet to come. This makes for efficient training.

This algorithm makes decisions based only on the data it has seen before and the data currently under consideration. It doesn't try to plan or strategize for the future based on what it has seen so far. We call this a *greedy* algorithm, since it's focused on maximizing its immediate, short-term gains.

Decisions trees are sometimes preferred in practice over other classifiers because their results are *explainable*. When the algorithm assigns a class to a sample, we don't have to unravel some complex mathematical or algorithmic process. Instead, we can fully explain the final result just by identifying each decision along the way. This can be important in real life, too.

For example, suppose we apply for a loan at a bank, but we're turned down. When we ask for the reason, the bank can show us each test made along the way. We say that the workings of the algorithm are *transparent*. Note that this doesn't mean that they're fair or reasonable. The bank may have come up with tests that are biased against one or more social groups or that depend on what seem to be irrelevant criteria. Just because they can explain why they made their choice doesn't make the process or the results satisfactory. Legislators in particular seem to prefer laws that enforce transparency, which is easy to demonstrate, over fairness, which is much harder. Transparency is nice to have, but it doesn't mean a system is behaving the way we'd like it to.

Decision trees can make bad decisions because they are particularly prone to overfitting. Let's see why.

Overfitting Trees

Let's begin our discussion of overfitting in the tree-building process by considering a couple of examples.

The data in Figure 11-14 shows a cleanly separated set of data representing two classes. A dataset that roughly follows this kind of geometry is often called a *two-moons dataset*, presumably because the semicircles reminded someone of crescent moons.

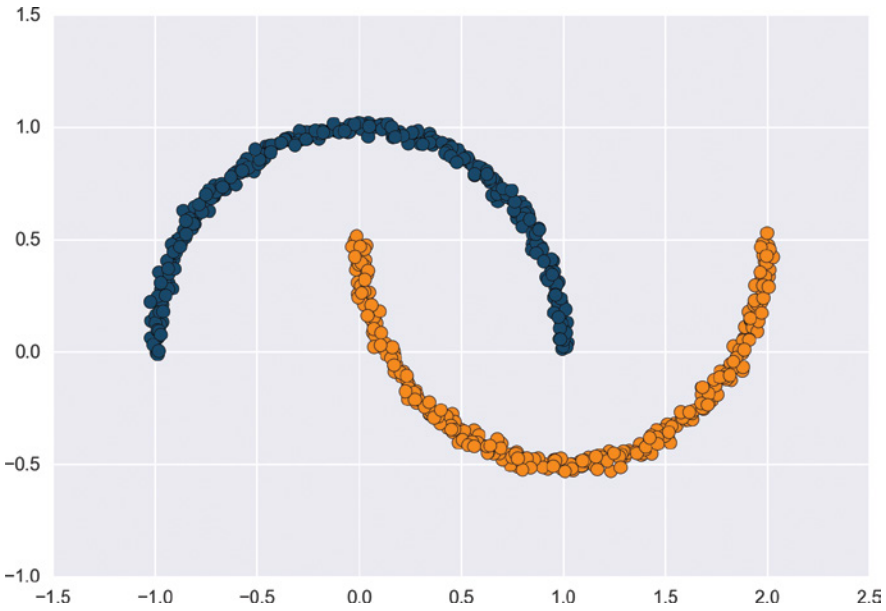


Figure 11-14: Our 600 starting data points for building a decision tree, arranged in a two-moons structure. These 2D points represent two classes, blue and orange.

Each step in building a tree potentially involves splitting a leaf, and thus replacing it with a node and two leaves, for a net increase to the tree of one internal node and one leaf. We often think about the size of our tree in terms of the number of leaves it contains.

Figure 11-15 shows the process of building a decision tree for this data.

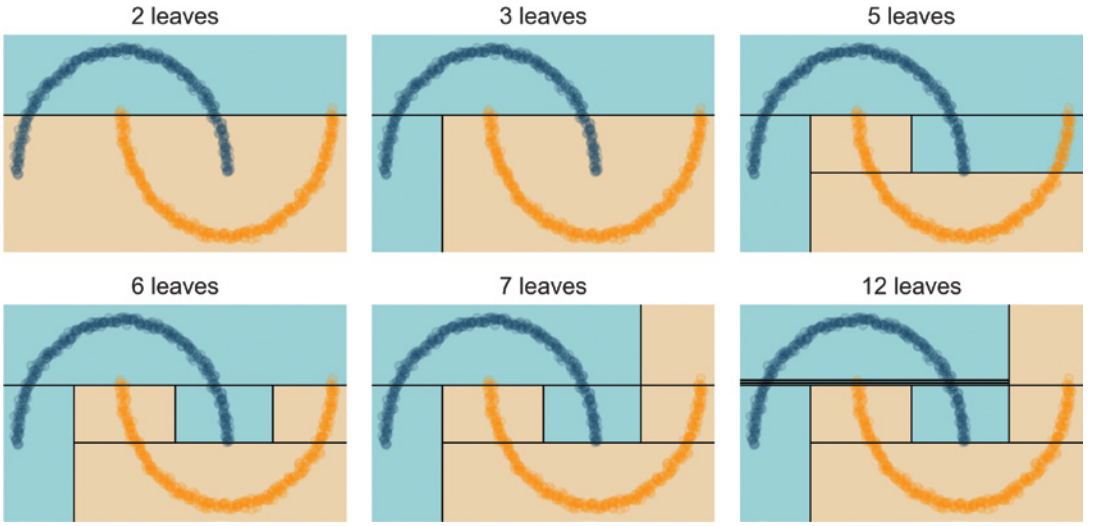


Figure 11-15: Building a decision tree for the data in Figure 11-14. Notice how the tree starts with big chunks and refines them into smaller and more precise regions.

In this figure, we drew the dataset over each image just for reference. In this example, each node corresponds to a box. We don't show it here, but the tree begins with just a single root corresponding to a blue box that covers the entire region. Then the training process receives one of the orange points near the top of the orange curve. This isn't in the blue class, so we split the root with a horizontal cut into two boxes, shown in the upper left. The next point that comes in is a blue point near the left part of the blue curve. This falls in the orange box, so we split that with a vertical cut into two boxes as shown, giving us a total of three leaves. The rest of the figure shows the evolving tree as more samples arrive.

Notice how the regions gradually refine as the tree grows in response to more training data.

This tree needs only 12 leaves to correctly classify every training sample. The final tree and the original data are shown together in Figure 11-16. This tree fits the data perfectly.

Note the two horizontal, thin rectangles. They enclose two orange samples at the top of the left side of the arc and manage to slip between the blue points (recall that the samples are points in the center of each circle). This is overfitting, because any future points that fall into those rectangles, despite the fact that they're both almost completely in a blue region, will be classified as orange.

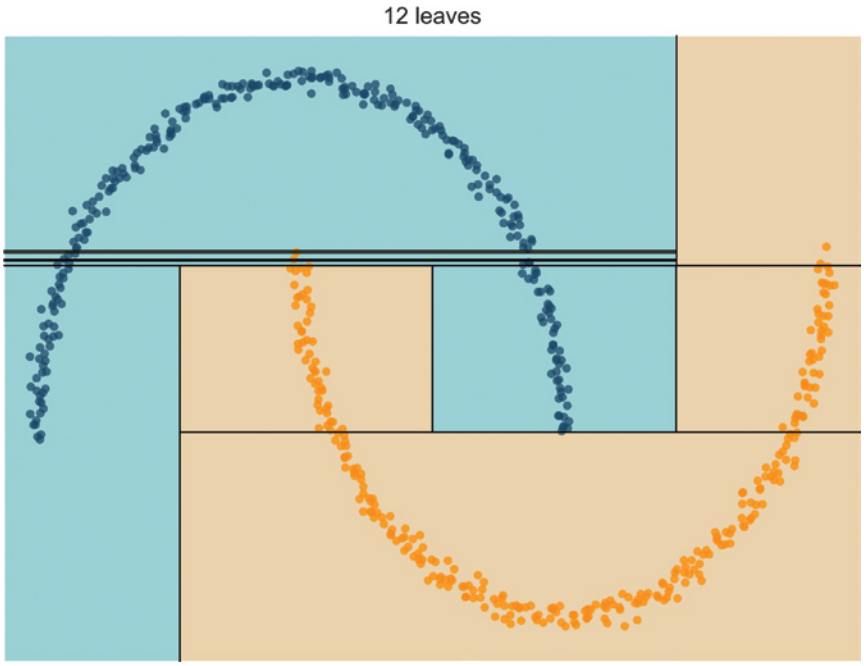


Figure 11-16: Our final tree with 12 leaves

Because decision trees are so sensitive to each input sample, they have a profound tendency to overfit. In fact, decision trees almost always overfit, because every training sample can influence the tree's shape. To see this, take a look at Figure 11-17. Here we ran the same algorithm as for Figure 11-16 two times, but in each case, we used a different, randomly chosen 70 percent of the input data.

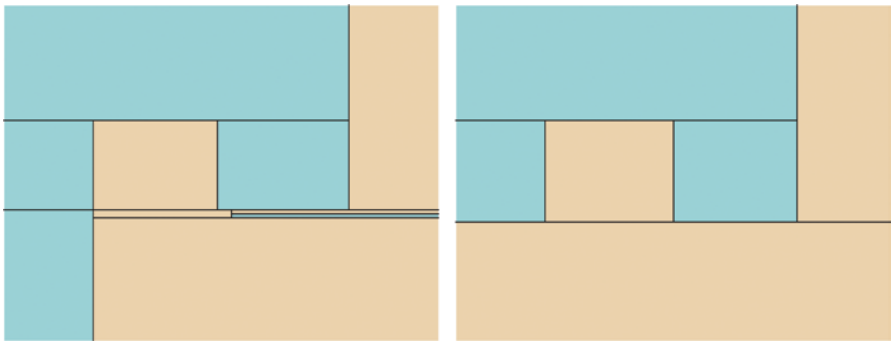


Figure 11-17: Decision trees are very sensitive to their inputs. (Left) We randomly chose 70 percent of the samples from Figure 11-14 and fit a tree. (Right) The same process, but for a different randomly selected 70 percent of the original samples.

These two decision trees are similar, but definitely not identical. The tendency of decision trees to overfit is much more pronounced when the data isn't so easily separated. Let's look at an example of that now.

Figure 11-18 shows another pair of crescent moons, but this time we added lots of noise to the samples after they had their classes assigned. The two classes no longer have a clean boundary.

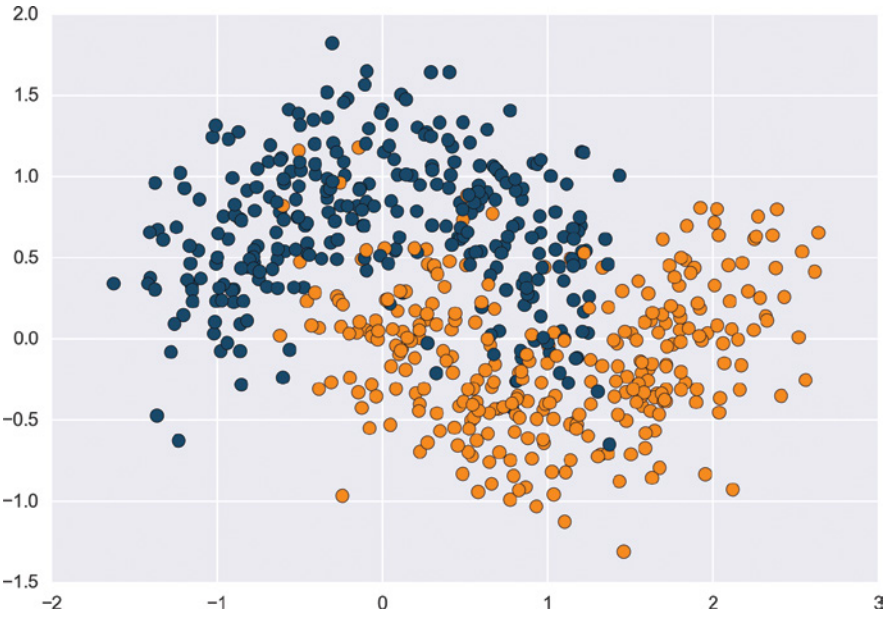


Figure 11-18: A noisy set of 600 samples for building decision trees

Fitting a tree to this data starts out with big regions, but it rapidly turns into a complicated set of tiny boxes as the algorithm splits up nodes this way and that to match the noisy data. Figure 11-19 shows the result. In this case, it required 100 leaves for the tree to correctly classify the points.

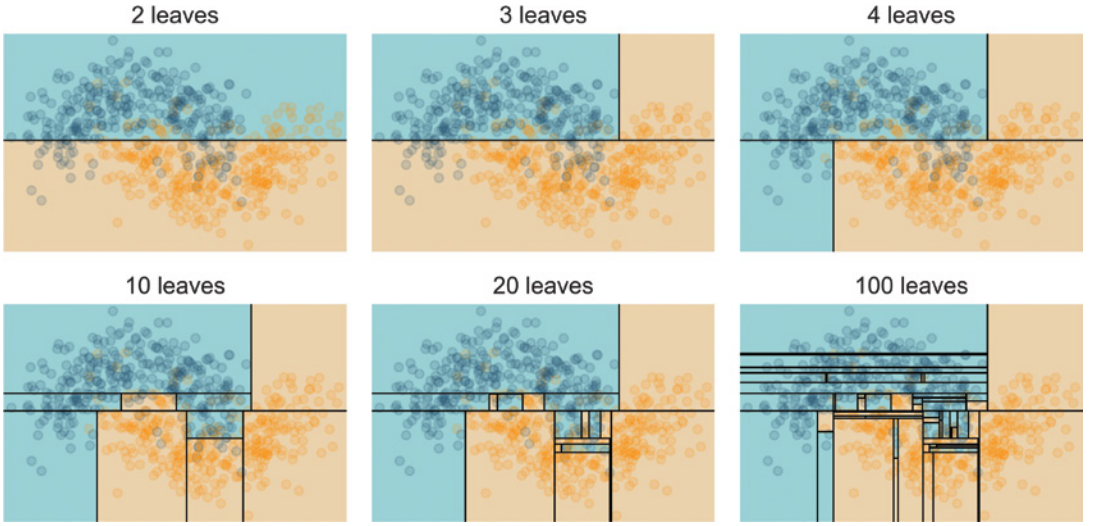


Figure 11-19: The tree-building process. Note that the second row uses large numbers of leaves.

Figure 11-20 shows a close-up of the final tree and the original data.

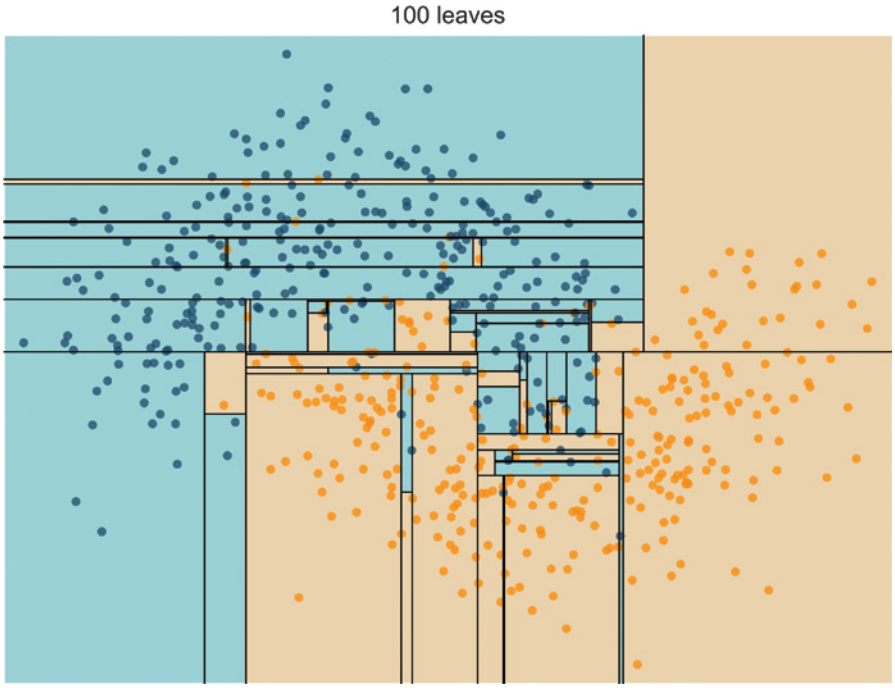


Figure 11-20: Our noisy data fit with a tree with 100 leaves. Notice how many little boxes have been used to catch just an odd sample here and there.

There's a lot of overfitting here. Though we expect most of the samples in the lower right to be orange and most of those in the upper left to be blue, this tree has carved out a lot of exceptions based on this particular dataset. Future samples that fall into those little boxes are likely to be misclassified.

Let's repeat our process of building trees using different, random 70 percent selections of the data in Figure 11-18. Figure 11-21 shows the results.

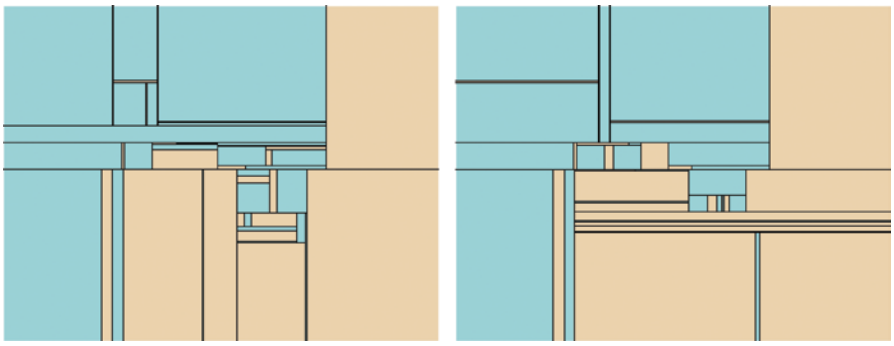


Figure 11-21: A couple of trees built with different sets of 70 percent of the samples in Figure 11-18

There are similarities, but these trees are significantly different, and lots of little bits exist only to classify a few samples. This is overfitting in action.

Although this may look pretty bad for the decision tree method, in Chapter 12 we'll see that by combining many simple decision trees into a group, or an *ensemble*, we can create robust, efficient classifiers that don't suffer as much from overfitting.

There are a few other ways we can control overfitting.

As we saw in Figure 11-15 and Figure 11-19, the first few steps of the tree's growth tend to generate big, general shapes. It's only when the tree gets very deep that we get the tiny boxes that are symptomatic of overfitting. One popular strategy to reducing overfitting is *depth limiting:* we simply limit the tree's depth while it's building. If a node is more than a given number of steps from the root, we just declare it a leaf and don't split it any more. A different strategy is setting a minimum sample requirement so that we never split a node that has less than a certain number of samples, no matter how mixed they are.

Yet another approach to reducing overfitting is to reduce the size of the tree after it's made in a process called *pruning*. This works by removing, or *trimming*, leaf nodes. We look at each leaf and characterize what would happen to the total error of the tree's results if we removed that leaf. If the error is acceptable, we simply remove the leaf from the tree. If we remove all the children of a node, then it becomes a leaf itself, and a candidate for further pruning. Pruning a tree can make it shallower, which offers the additional benefit of also making it faster when we classify new data.

Depth limiting, setting minimum sample requirements per node, and pruning all simplify the tree, but because they do so in different ways, they usually give us different results.

Splitting Nodes

Before we leave decision trees, let's return briefly to the node-splitting process, since many machine learning libraries offer us a choice of splitting algorithms to choose from. Here are two questions to ask when we consider a node: First, does it need to be split? Second, how should we split it? Let's take these in order.

When we ask if a node needs to be split, we usually consider to what extent all the samples in a given node are of the same class. We describe the uniformity of a node's contents with a number called that node's *purity*. If all the samples are in the same class, the node is completely pure. The more samples we have of other classes, the smaller the value of purity becomes. To test if a node needs splitting, we can check the purity against a threshold. If the node is too *impure*, meaning that the purity is below the threshold, we split it.

Now we can look at how to split the node. If our samples have many features, we can invent lots of different possible splitting tests. We can test the value of just one feature and ignore the others. We can look at groups of features and test on some aggregated values from them. We're free to

choose a completely different test at every node based on different features. This gives us a huge variety of possible tests to consider.

Figure 11-22 shows a node containing a mix of circles of different sizes and colors. Let's try to get all the reddish objects in one child and all the bluish ones in another. When we just look at the data (usually the best first step with any new database), it seems like the reddish circles are the biggest. Let's try using a test based on the radius of each circle. The figure shows the result of splitting on the radius using three different values.

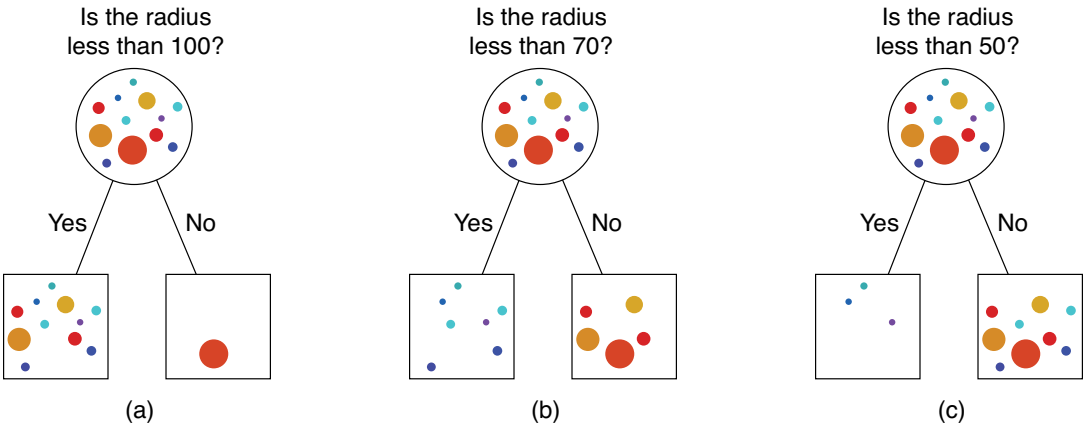


Figure 11-22: Splitting a node according to different values of the radii of the circles within it

In this example, the radius value of 70 produces the purest results, with all the blue objects in one child and all the red ones in the other. If we use this test for this node, we'll remember which feature we're splitting on (the radius) and what value to test for (70).

Since we could potentially split the node using a test based on any characteristic of the samples, we need some way to evaluate the results so we can pick the best test. Let's look at two popular ways to test these results.

Recall from Chapter 6 that *entropy* is a measure of complexity, or how many bits it takes to communicate some piece of information. The *Information Gain (IG)* measure uses this idea by comparing the entropy of a node to that of the children produced by each candidate test.

To evaluate a test, IG adds together the entropies of all the new children produced by that test and compares that result to the entropy in the parent cell. The more pure a cell is, the lower its entropy, so if a test makes pure cells, the sum of their entropies is less than the entropy of their parent. After trying different ways to split a node, we choose the split that gives us the biggest reduction in entropy (or the biggest gain in information).

Another popular way to evaluate splitting tests is called the *Gini impurity*. The math used by this technique is designed to minimize the probability of misclassifying a sample. For example, suppose a leaf has 10 samples of class A and 90 samples of class B. If a new sample ends up at that leaf, and we report that it belongs to class B, there is a 10 percent chance that

we are wrong. Gini impurity measures those errors at each leaf for multiple candidate split values. It then chooses the split that has the least chance of an erroneous classification.

Some libraries offer other measures for grading the quality of a potential split. As with so many other choices, we usually try out a few options and pick the one that works the best for the specific data we're working with.

Support Vector Machines

Let's consider our first parametric algorithm: the *support vector machine* (or *SVM*). We will use 2D data and just two classes for our illustrations (VanderPlas 2016), but like most machine learning algorithms, the ideas are easily applied to data with any number of dimensions and classes.

The Basic Algorithm

Let's begin with two blobs of points, one for each of two classes, shown in Figure 11-23.



Figure 11-23: Our starting dataset consists of two blobs of 2D samples.

We want to find a boundary between these clusters. To keep things simple, let's use a straight line. But which one? A lot of lines split these two groups. Three candidates are shown in Figure 11-24.

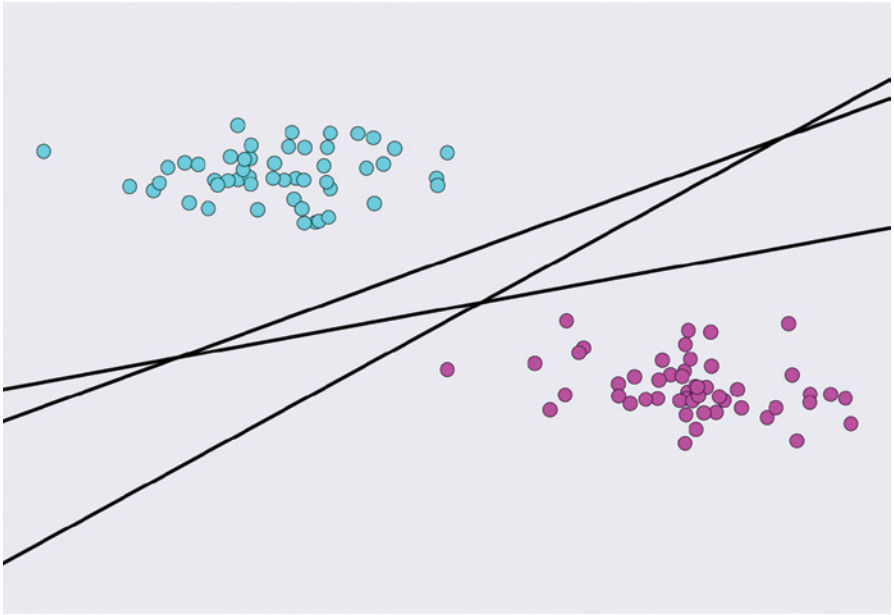


Figure 11-24: Three of the infinite number of lines that can separate our two clusters of samples

Which of these lines should we pick? One way to think about this is to imagine new data that might come in. Generally speaking, we want to classify any new sample as belonging to the class of the sample that it is nearest.

To evaluate how well any given boundary line achieves this goal, let's find its distance to the nearest sample of either class. We can use this distance to draw a symmetrical boundary around the line. Figure 11-25 shows the idea for a few different lines.

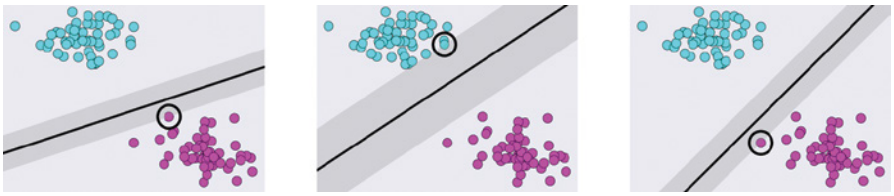


Figure 11-25: We can assign a quality to each line by finding the distance from that line to the nearest data point.

In Figure 11-25, we've drawn a circle around the sample that's closest to the line. In the leftmost figure, many new points that are closer to the lower-right cluster than the upper-left one would be incorrectly classified as part of the upper-left cluster. The same situation holds in the rightmost

figure. In the center, the line is much better, but it's now preferring the lower-right cluster a little. Since we want each new point to be assigned to the class of the sample it's closest to, we'd like our line to go right through the middle of the two clusters.

The algorithm that finds this line is called a *support vector machine*, or *SVM* (Steinwart 2008). An SVM finds the line that is farthest from all the points in both clusters. In this context, the word *support* can be thought of as meaning "nearest," *vector* is a synonym for "sample," and *machine* is a synonym for "algorithm." Thus, we can describe SVM as the "nearest sample algorithm."

The best line for our two clusters in Figure 11-23, as calculated by SVM, is shown in Figure 11-26.

Let's see how SVM finds this line.

In Figure 11-26 the circled samples are the nearest samples, or the *support vectors*. The algorithm's first job is to locate these circled points. Once it finds them, the algorithm then finds the solid line near the middle of the figure. Of all the lines that separate the two sets of points, this is the line that's farthest from every sample in each set, because it has the greatest distance to its support vectors. The dashed lines in Figure 11-26, like the circles around the support vectors, are just visual aids to help us see that the solid line in the center, found by the SVM, is the one that is as far as possible from all the samples. The distance from the solid line to the dashed lines that pass through the support vectors is called the *margin*. We can rephrase the idea by saying that the SVM algorithm finds the line with the largest margin.

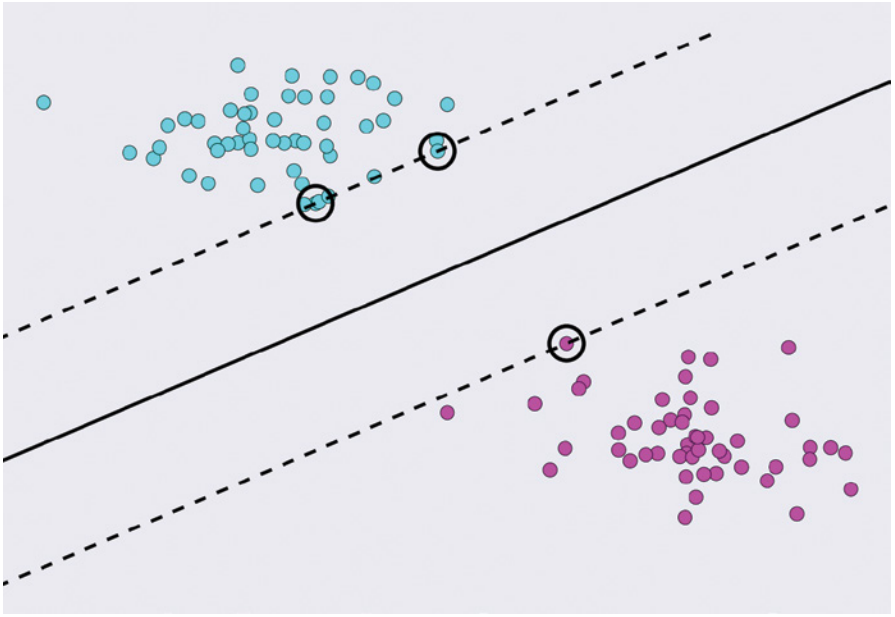


Figure 11-26: The SVM algorithm finds the line that has the greatest distance from all the samples.

What if the data is noisy, and the blobs overlap, as in Figure 11-27? Now we can't create a line surrounded by an empty zone. What's the best line to draw through these overlapping sets of samples?



Figure 11-27: A new set of data where the blobs overlap

The SVM algorithm gives us control over a parameter that's conventionally called *C*. This parameter controls how strict the algorithm is about letting points into the region between the margins. The larger the value of *C*, the more the algorithm demands an empty zone around the line. The smaller the value of *C*, the more points can appear in a zone around the line. We frequently need to search for the best value for *C* using trial and error. In practice, that usually means trying out lots of values and evaluating them with cross-validation.

Figure 11-28 shows our overlapping data with a *C* value of 100,000.

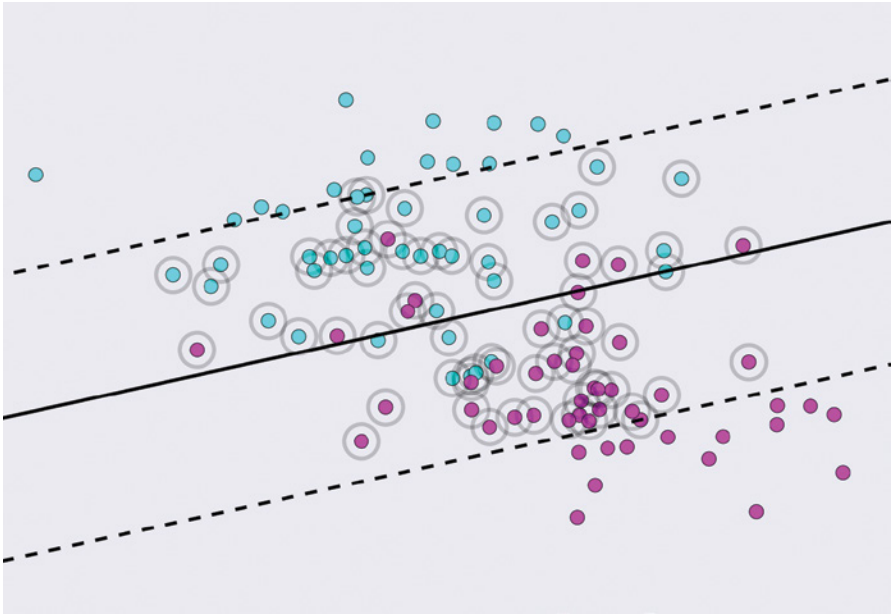


Figure 11-28: The value of C tells SVM how sensitive to be to points that can intrude into the zone around the line that's fit to the data. Here, C is 100,000.

Let's drop *C* way down to 0.01. Figure 11-29 shows that this lets fewer points into the region around the line.

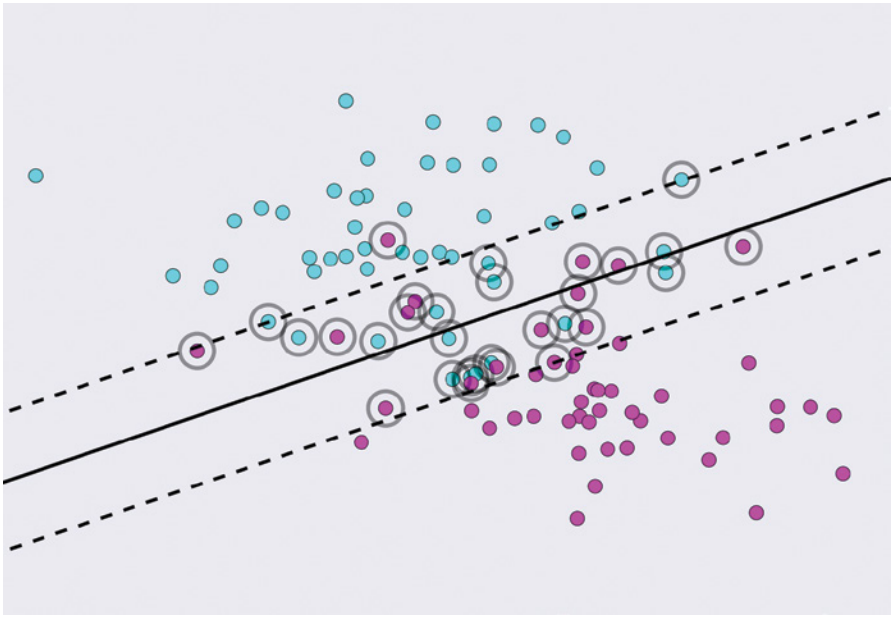


Figure 11-29: Lowering C to 0.01 lets in fewer points compared to Figure 11-28.

The lines in Figure 11-28 and Figure 11-29 are different. Which one we prefer depends on what we want from our classifier. If we think the best boundary comes from the details near the zone where the points overlap, we want a small value of C so that we only look at the points near that boundary. If we think the overall shapes of the two collections of points is a better descriptor of that boundary, we want a larger value of C to include more of those farther-away points.

The SVM Kernel Trick

A parametric algorithm is limited by the shapes it's able to find. SVM, for instance, can only find *linear* shapes, like lines and planes. If we have data that can't obviously be separated by such a shape, it may seem that SVM won't be of much use. But there's a clever trick that can sometimes let us use a linear boundary where it initially appears as if only a curved one could do the job.

Suppose that we have the data of Figure 11-30, where there's a blob of samples of one class surrounded by a ring of samples of another. There's no way we can draw a straight line to separate these two sets.

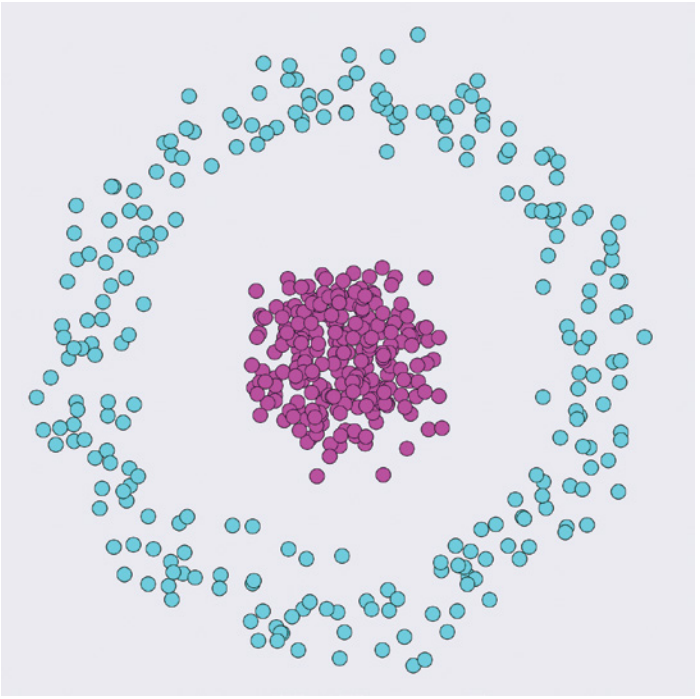


Figure 11-30: A dataset of two classes

Here comes the clever part. Suppose we temporarily add a third dimension to each point by elevating it by an amount based on that point's distance from the center of the square. Figure 11-31 shows the idea.

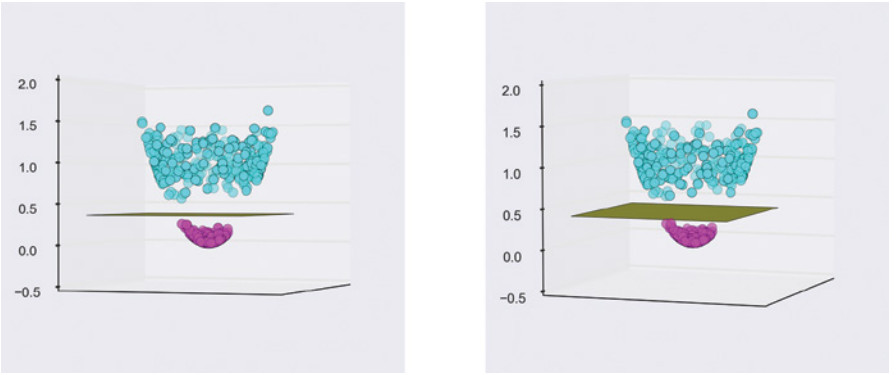


Figure 11-31: If we push each point in Figure 11-30 upward by an amount based on its distance from the center of the pink blob, we get two distinct clouds of points, which we can separate with a plane.

As we can see in Figure 11-31, we can now draw a plane (the 2D version of a straight line) between the two sets. In fact, we can use the very same idea of support vectors and margins as we did before to find the plane. Figure 11-32 highlights the support vectors for the plane between the two clusters of points.

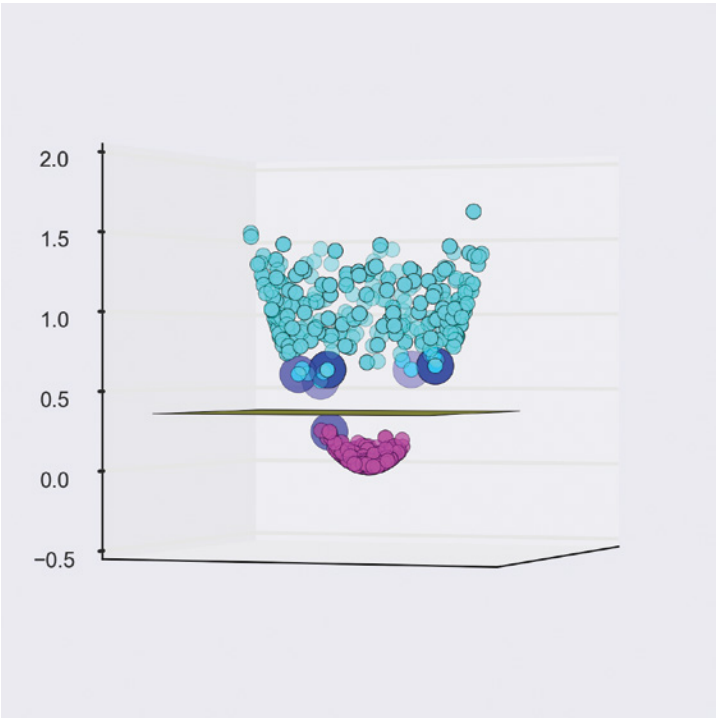


Figure 11-32: The support vectors for the plane

Now all points above the plane can be placed into one class and all those below into the other.

If we highlight the support vectors we found from Figure 11-32 in our original 2D plot, we get Figure 11-33.

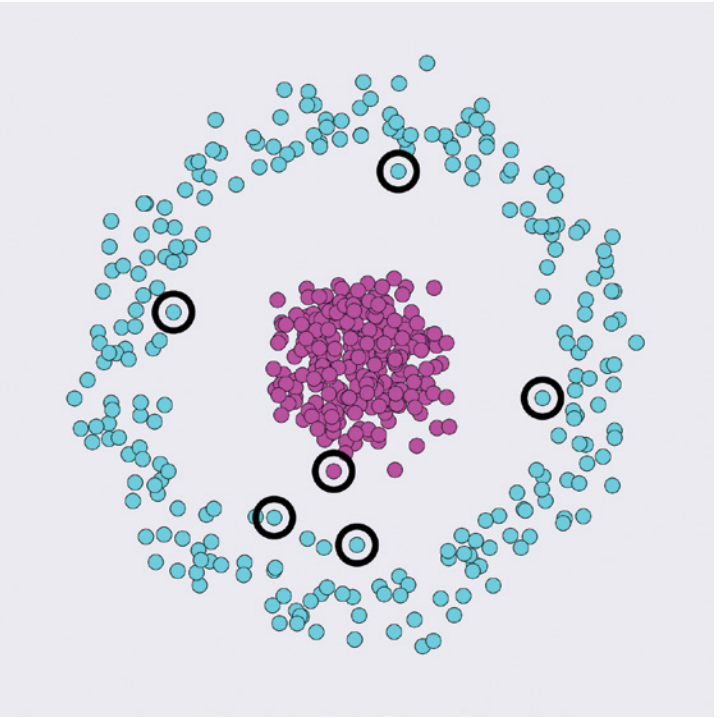


Figure 11-33: Looking down on Figure 11-32

If we include the boundary created by the plane, we get Figure 11-34.

In this case we found the right way to modify our data by looking at it and then coming up with a good 3D transformation of the data that let us split it apart. But when the data has many dimensions, we might not be able to visualize it well enough to even guess at a good transformation. Happily, we don't have to find these transformations manually.

One way to find a good transformation is to try lots and lots of them, and then select the one that works the best. The calculations required make this approach too slow to be practical, but fortunately, we can speed this up in a clever way. This idea focuses on a piece of math called the *kernel*, which lies at the heart of the SVM algorithm. Mathematicians sometimes honor a particularly clever or neat idea with the complementary term *trick*. In this case, rewriting the SVM math is called the *kernel trick* (Bishop 2006). The kernel trick lets the algorithm find the distances between transformed points without actually transforming them, which is a major efficiency boost. The kernel trick is used automatically by all major libraries, so we don't even have to ask for it (Raschka 2015).

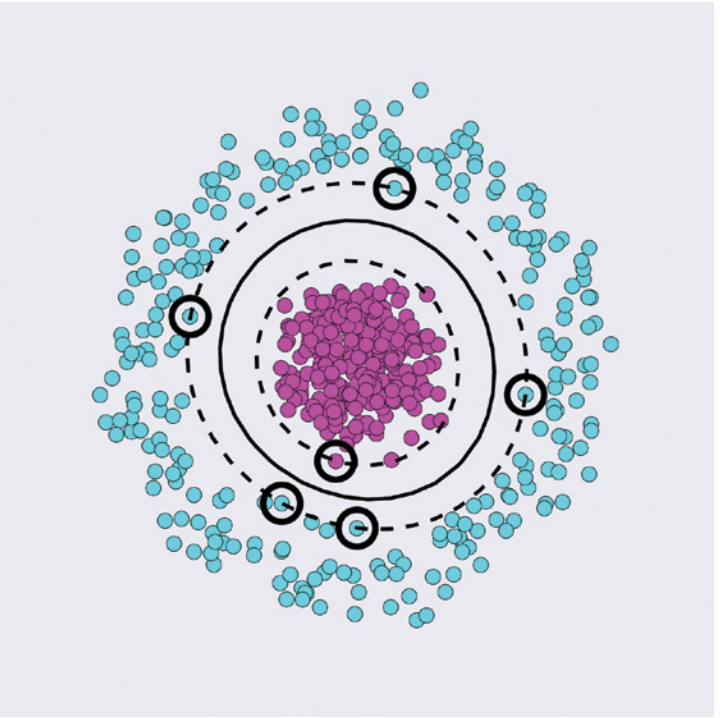


Figure 11-34: The data from Figure 11-30 with the support vectors, the dashed lines showing the margins, and the boundary created by the plane

Naive Bayes

Let's look at a parametric classifier that's often used when we need quick results, even if they're not the most accurate.

This classifier works quickly because it begins by making assumptions about the data. It is based on Bayes' Rule, which we looked at in Chapter 4.

Recall that Bayes' Rule begins with a *prior*, or a predetermined idea of what the result is likely to be. Normally when we use Bayes' Rule, we refine the prior by evaluating new pieces of evidence, creating a *posterior* that then becomes our new prior. But what if we just commit to the prior ahead of time and then see where it leads us?

The *naive Bayes* classifier takes this approach. It's called *naive* because the assumptions we make in our prior are not based on the contents of our data. That is, we make an uninformed, or naive, characterization of our data. We just assume that the data has a certain structure. If we're right, great, we get good results. The less well the data matches this assumption, the worse the results are. Naive Bayes is popular because this assumption turns out to

be correct, or nearly correct, often enough that it's worth taking a look. The interesting thing is that we never check to see if our assumption is justified. We just plow ahead as if we are certain.

In one of the more common forms of naive Bayes, we assume that every feature of our samples follows a Gaussian distribution. Recall from Chapter 2 that this is the famous bell curve: a smooth, symmetrical shape with a central peak. That's our prior. When we look at a specific feature across all of our samples, we simply try to match that as well as we can with a Gaussian curve.

If our features really do follow Gaussian distributions, then this assumption produces a good fit. The great thing about naive Bayes is that this assumption seems to work well far more frequently than we might expect.

Let's see it in action, starting with data that does satisfy the prior. Figure 11-35 shows a dataset that was created by drawing samples from two Gaussian distributions.

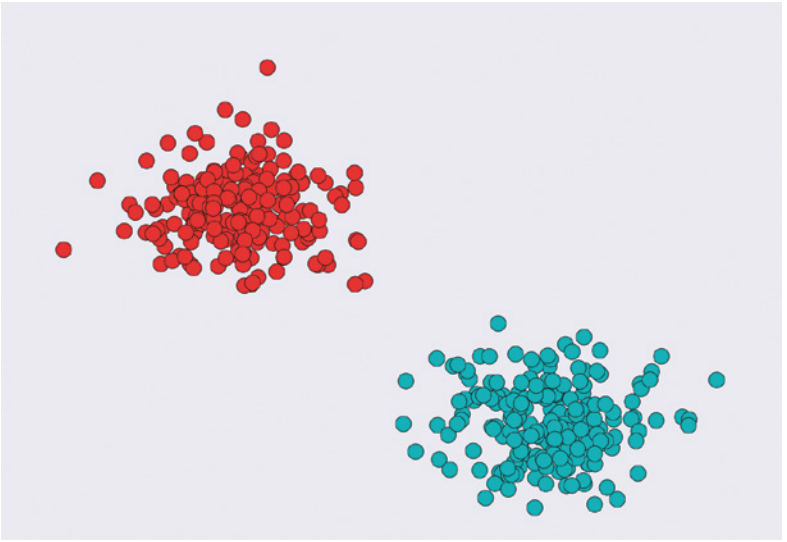


Figure 11-35: A set of 2D data for training with naive Bayes. There are two classes, red and blue.

When we give this data to a naive Bayes classifier, it assumes that each set of features comes from a Gaussian. That is, it assumes that the x coordinates of the red points follow a Gaussian, and the y coordinates of the red points also follow a Gaussian. It assumes the same thing about the x and y features of the blue points. Then it tries to fit the best four Gaussians it can to that data, creating two 2D hills. The result is shown in Figure 11-36.

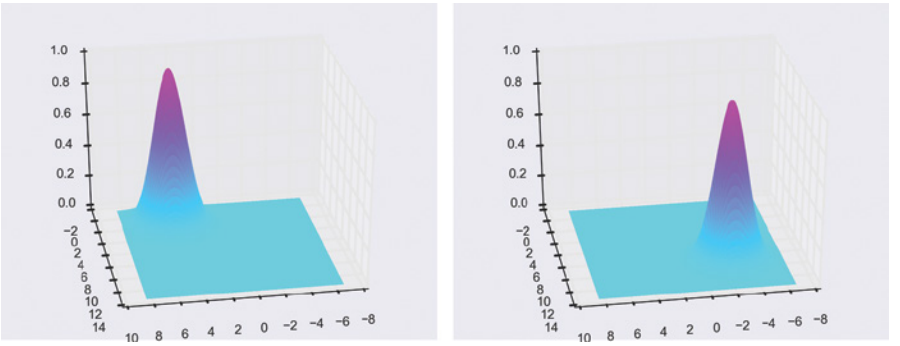


Figure 11-36: Naive Bayes fits a Gaussian to each of the x and y features of each class. Left: The Gaussian for the red class. Right: The Gaussian for the blue class.

If we overlay the Gaussian blobs and the points and look directly down, as in Figure 11-37, we can see that they form a very close match. That's no surprise, because we generated the data in a way that had exactly the distribution the naive Bayes classifier was expecting.

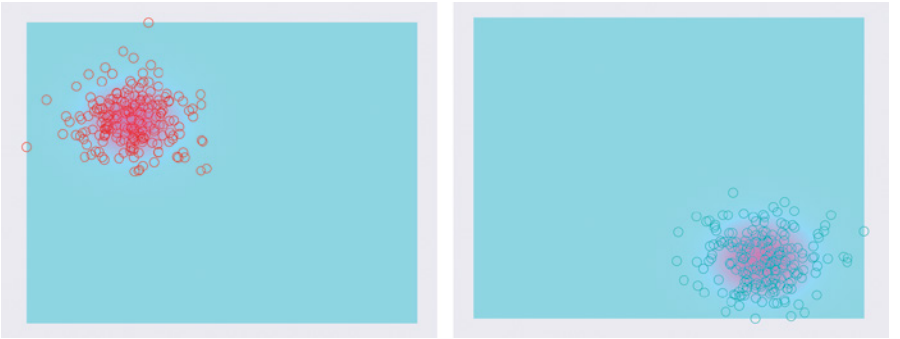


Figure 11-37: Our entire training set overlaid on the Gaussians of Figure 11-36

To see how well the classifier works in practice, let's split the training data, putting a randomly selected 70 percent of the points into a training set and the rest into a test set. Let's train with this new training set and then draw the test set on top of the Gaussians, giving us Figure 11-38.

In Figure 11-38, we drew all the points that were classified as belonging to the first class on the left, and all those in the second class on the right, maintaining their original colors. We can see that all of the test samples were correctly classified.



Figure 11-38: The test data after training with 70 percent of our starting data

Now let's try an example where we don't satisfy the prior that all features of all samples follow Gaussian distributions. Figure 11-39 shows our new starting data of two noisy crescent moons.

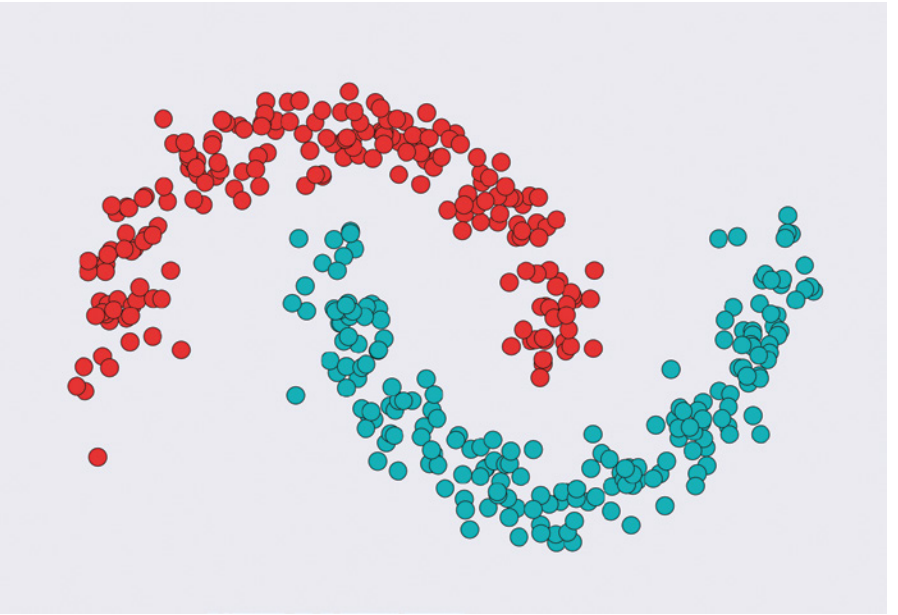


Figure 11-39: Some noisy crescent moon data

When we give these samples to the naive Bayes classifier, it assumes (as always) that the red x values, red y values, blue x values, and blue y values all come from Gaussian distributions. It finds the best Gaussians it can, shown in Figure 11-40.

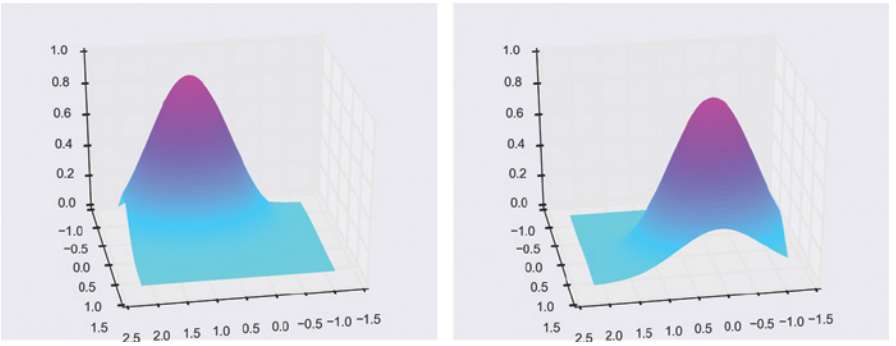


Figure 11-40: Fitting Gaussians to the crescent-moon data from Figure 11-39

Of course, these are not a good match to our data, because they don't satisfy the assumptions. Overlaying the data on the Gaussians in Figure 11-41 shows that the matches aren't abysmal, but they're pretty far off.

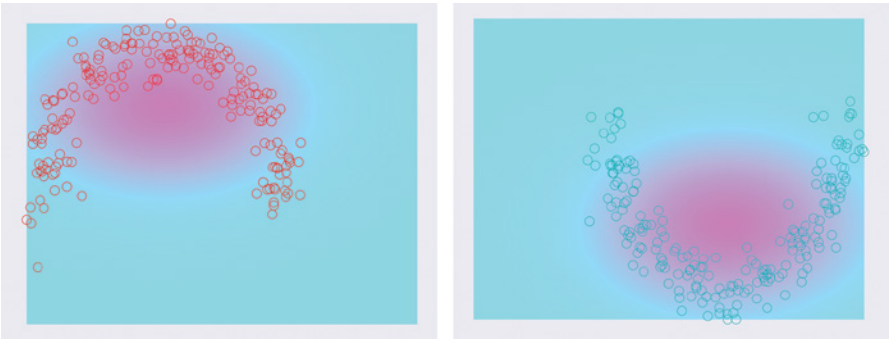


Figure 11-41: Our training data from Figure 11-39 overlaid on the Gaussians of Figure 11-40

Like before, let's now split the crescent moons into training and test sets, train on the 70 percent, and look at the predictions. In the left image of Figure 11-42 we can see all the points that we assigned to the red class. As we would hope, this has most of the red points, but some of the points from the upper-left red moon are not classified as red, and some points from the lower-right blue moon are classified as red, because their value from this Gaussian is higher than their value from the other.

On the right of Figure 11-42, we can see that the opposite situation holds for the other Gaussian. In other words, we correctly classified lots of the points, but we also misclassified points in each class.

We shouldn't be too surprised at the misclassifications because our data did not follow the assumptions made by the naive Bayes prior. What is amazing is how well the classifier did. In general, naive Bayes often does a good job on all kinds of data. This is probably because lots of real-world data comes from processes that are well described by Gaussians.

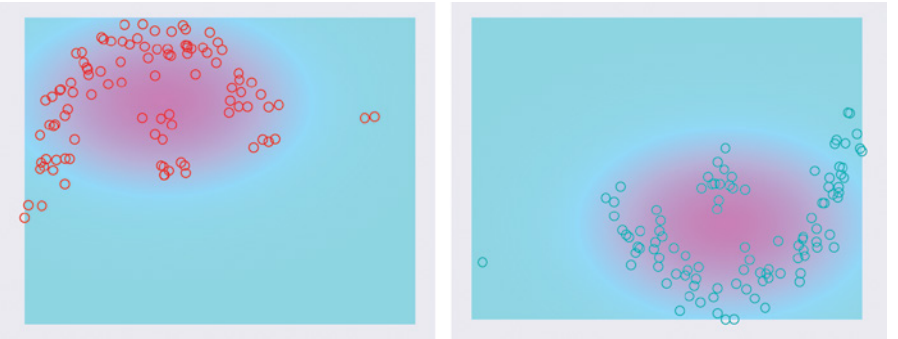


Figure 11-42: Predictions of test data from our naive Bayes classifier trained on the data in Figure 11-39

Because naive Bayes is so fast, it's common to apply it when we're trying to get a feeling for our data. If it does a great job, we might not have to look at any more complex algorithms.

Comparing Classifiers

We looked at four popular classification algorithms in this chapter. Most machine learning libraries offer all of these, along with many others. Very briefly, let's look at the pros and cons of these four classifiers, starting with the nonparametric algorithms.

The kNN method is flexible. It doesn't explicitly represent boundaries, so it can handle any kind of complicated structure formed by the class samples in the training data. It's fast to train, since it typically just saves each training sample. On the other hand, prediction is slow, because the algorithm has to search for the nearest neighbors for every sample we want to classify (there are many efficiency methods that speed up this search, but it still takes time). And because it's saving every training sample, the algorithm can consume huge gulps of memory. If the training set is larger than the available memory, the operating system typically needs to start saving data on the hard drive (or other external storage), which can slow the algorithm down significantly.

Decision trees are fast to train, and they're also fast when making predictions. They can handle weird boundaries between classes, though this can require a deeper tree. They have a huge downside due to their appetite for overfitting (though as we mentioned, we will address this issue later by using collections of small trees, so all is not lost). Decision trees have a huge appeal in practice because they are easy to interpret. Sometimes people use decision trees, even when the results are inferior to other classifiers, because their decisions are transparent, or easy to understand. Note that this doesn't mean that the choices are fair or even correct, just that they're comprehensible to humans.

Support vector machines are parametric algorithms that can make fast predictions. Once trained, they don't need much memory, since they

only store the boundaries between sample regions. And they can use the kernel trick to find classification boundaries that appear much more complicated than the straight lines (and flat planes, and higher-dimensional flat surfaces) that SVM produces. On the other hand, the training time grows with the size of the training set. The quality of the results is sensitive to the parameter C that specifies how many samples are allowed near the boundary. We can use cross-validation to try different values of C and pick out the best one.

Naive Bayes trains quickly and predicts quickly, and it's not too hard to explain its results (though they're a bit more abstract than decision trees or kNN results). The method has no parameters that we need to tune. If the classes we're working with are well separated, then the naive Bayes prior often produces good results. The algorithm works particularly well when our data is Gaussian in nature. It also works well when the data has many features because the classes of such data are often separated in ways that play to the strengths of naive Bayes (VanderPlas 2016). In practice, we often try naive Bayes early in the process of getting to know a dataset, because it's fast to train and predict and can give us a feeling for the structure of our data. If the prediction quality is poor, we can then turn to a more expensive classifier (that is, one requiring more time or memory).

The algorithms we've seen here are frequently used in practice, particularly when we're first getting to know our data, because they're usually straightforward to apply and visualize.

Summary

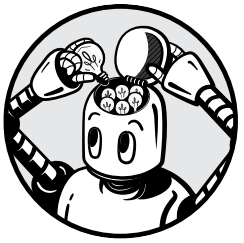
In this chapter, we covered two types of classifier. When a classifier has no preconceptions on the structure of the data it's going to look at, we say it is nonparametric. The k-nearest neighbors algorithm is of this variety, assigning a class to a sample based on its most popular neighbor. Decision trees are also nonparametric, assigning classes based on a series of decisions that are learned from the training data.

On the other hand, parametric classifiers have a preconceived notion of the structure of the data. A basic support vector machine looks for linear shapes, like lines or planes, that separate the training data by class. A naive Bayes classifier presumes that the data has a fixed distribution, usually Gaussian, and then does its best to fit that distribution to each feature in the data.

In the next chapter, we'll see how to bundle together multiple classifiers to produce ensemble classifiers that outperform their individual components.

12

ENSEMBLES



Anyone can make mistakes, even algorithms. Sometimes we might be pretty sure that our algorithms are giving us good answers, but for any number of reasons, we might harbor a bit of doubt. How can we increase our confidence in what the computer tells us?

This isn't a new problem. The Apollo spacecraft of the 1960s and '70s relied on one type of computer in the command module, which orbited the moon, and a different type of computer in the lunar module, which landed there. These computers were a critical part of almost every maneuver, so it was essential that the astronauts could trust their outputs. The computers were built with integrated circuits, which were relatively new at the time. The astronauts trusted their lives to their software and hardware, but there was always room for doubt. How could they guard against errors or malfunctions that could end the mission or even prove fatal?

The designers of these computers addressed that problem with redundancy: every circuit board was duplicated, not once, but twice, producing three copies in all. All three systems always ran in synchrony, a technique

called *triple modular redundancy*. The computers took the same inputs and computed their own independent results. The output of the group was decided by majority vote (Ceruzzi 2015). That way, if any one of the three systems got damaged, the right answer would still emerge.

We can adopt and expand on this idea in machine learning. Like the Apollo engineers, we can make multiple learners and use them all simultaneously. In machine learning, groups of similar learners are called *ensembles*. And like the Apollo computers, the output of the ensemble is the most popular result from its members. But unlike Apollo's identical software and hardware, we make each learner unique, usually by training it on slightly different data. This makes it unlikely that a mistake made by one learner will be made in exactly the same way by the others. In this way, the majority vote helps us weed out bad decisions.

In Chapter 11, we saw that decision trees easily overfit their training data, which can lead to errors when the system is deployed. In this chapter, we'll see how to combine many such trees into an ensemble. The result is an algorithm that enjoys the simplicity and transparency of decision trees but greatly reduces their problems. Let's begin with a brief discussion of how ensembles determine their final results.

Voting

Making decisions is hard for computers and humans alike. In some human societies, we deal with individual imperfections in decision-making by aggregating the opinions of many people. Laws are passed by senates, financial decisions are made by boards, and individual leaders are elected by popular vote. The thinking in all of these cases is that we can avoid errors in judgment that are unique to a single individual if we instead use the consensus of multiple, independent voters. Although this doesn't guarantee good decisions, it can sometimes help avoid problems caused by any one person's idiosyncrasies, biases, or bad judgment.

Machines have biases, too. When we use learning algorithms to make decisions, their predictions are based on the data that they trained on. If that data contained biases, omissions, underrepresentations, overrepresentations, or any other kind of systemic error, those errors are baked into the learner as well. This can have profound implications in the real world. For instance, when we use machine learning to evaluate loans for homes or businesses, to determine admissions to colleges, or to prescreen job applicants, any unfairness or bias in our training data causes similar unfairness and bias in the decisions the system makes, and bad decisions from the past are repeated in the present and propagated into the future.

One way to reduce the effects of these problems is to create multiple learners trained with different datasets. For example, we might train each system with a different training set from a different source. Since such data is often hard to come by in practice, often we train with different subsets drawn from a common pool of training data instead.

When we have trained a bunch of learners on these different datasets, we usually ask each one to evaluate each new input. Then we let the learners vote to determine a final result.

The typical way to do this is to use *plurality voting* (RangeVoting.org 2020). Put simply, each learner casts a single vote for its prediction, and whatever prediction receives the most votes is the winner (if there's a tie, the computer can either randomly select one of the tied entries, or try another round of voting). Though plurality voting is not perfect, and there exist useful alternatives, it is simple, fast, and usually produces acceptable results in machine learning (NCSL 2020).

A popular variation of plurality voting is *weighted plurality voting*. Here, every vote gets a certain *weight*, which is just a number that tells us how much influence that vote has on the result. Another variation is to ask each voter to identify their *confidence* in their decision, so more confident voters can have more impact than those that are less sure.

With those terms in place, let's now dig into making an ensemble of decision trees.

Ensembles of Decision Trees

A great way to build on the strengths of decision trees, while reducing their drawbacks, is to combine them into ensembles. To keep the following discussion specific, let's focus on using decision trees for classification.

Let's look at three popular techniques for building decision tree ensembles that can significantly outperform their individual components.

Bagging

The ensemble technique called *bagging* is a portmanteau of *bootstrap aggregating*. As the name suggests, this technique is based on the bootstrap idea we saw in Chapter 2. There, we saw how to use bootstrapping to estimate the quality of some statistical measure by evaluating lots of small subsets drawn from the starting data. In this case, let's again create many small sets built from a training set, but now let's use them to train a collection of decision trees.

Starting with our original training set of samples, we can build multiple new sets, or *bootstraps*, by picking items from the original, using sampling with replacement. This means that it's possible to pick the same sample more than once. Figure 12-1 shows the idea. Remember that each sample comes with its assigned class (shown by color) so we can train with it.

In the center of Figure 12-1, we have a starting set of eight samples, belonging to five classes. By selecting samples from this set, we can make many new sets, in this case of four samples each. This is the first step of bagging. Since we're sampling with replacement, it's possible that any given sample might appear multiple times.

Now let's create a decision tree for every bootstrap and train it on that data. We call the collection of those trees an *ensemble*.

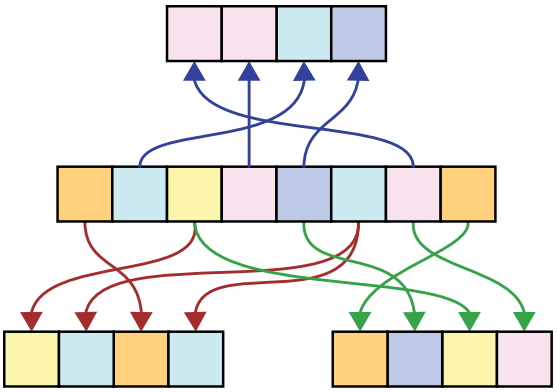


Figure 12-1: Creating three bootstraps (top and bottom) from a set of samples (center)

When training is done and we're evaluating a new sample, we give it to all the decision trees in the ensemble. Each tree produces one class prediction. We treat the predicted classes as votes in a plurality election, producing either a winner or a tie. Suppose that we have a small ensemble with just five trees. Figure 12-2 shows the process of evaluating a new sample after deployment, assigning it to one of four lettered classes.

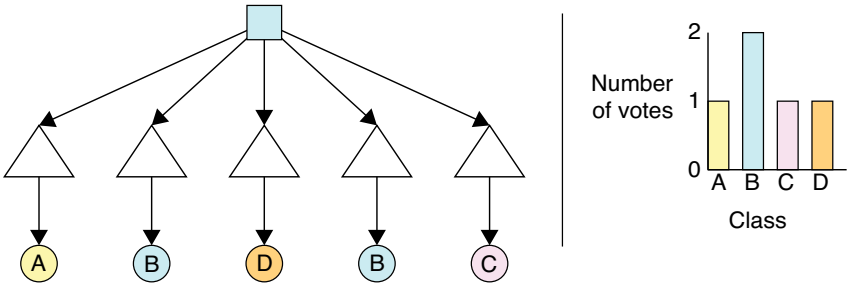


Figure 12-2: Using an ensemble to predict the class of a sample

Consider the left side of the figure. At the top, a new sample of unknown class arrives at our ensemble. The sample is given to every one of our decision trees (shown as triangles), and each one produces a predicted class, labeled A through D. Because each tree was trained on a different bootstrap, it's a little different from all the others. On the right side of the figure, we run a plurality voting election with those predicted classes. In this example, the most popular class is B. That class wins, and is therefore the output of the ensemble.

We only need to specify two parameters to create this ensemble: how many samples we should use in each bootstrap, and how many trees we want to build. Analysis shows that adding more classifiers makes the ensemble's predictions better, but at some point, adding more classifiers just makes it slower and the results stop improving. This is called *the law of diminishing returns in ensemble construction*. A good rule of thumb is to use about the same number of classifiers as classes of data (Bonab 2016),

though we can use cross-validation to search for the best number of trees for any given dataset.

Before we leave bagging, let's consider a couple of techniques that build on the basic idea. The central idea of each is to add extra randomization to our trees during training.

Random Forests

As we saw in Chapter 11, when it's time to split a decision tree's node in two, we can choose any feature (or set of features) to create the test that directs elements into one child or the other. If we choose to split based on just one feature, then we need to choose which feature we want to use and what value of that feature to test for. To compare different tests, we can use the measurements we saw in Chapter 11, such as information gain or the Gini impurity.

When building a decision tree, we often look for the best test by considering every feature. But we can also use a technique called *feature bagging*. Before looking for the best test at a node, we first choose a random subset of the features of the samples at that node, using selection without replacement. Now we're ready to look for the best test, based only on those features. We don't even consider splits based on the features we're ignoring.

Later, when we decide to split another node, we again choose a brand-new subset of features and again determine our new split using only those. The idea is shown in Figure 12-3.

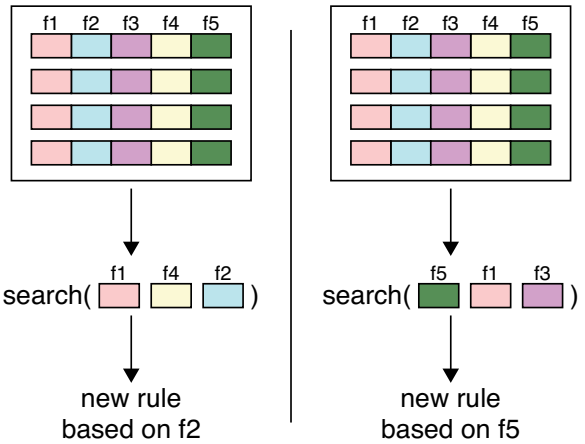


Figure 12-3: Determining which feature (f1 to f5) to use when splitting a node by feature bagging, shown for two nodes with five samples each

On the left of Figure 12-3, we randomly select a set of three features from the five available, and search those for the best feature to split on. On the right we do it again, only we pick a different random set of three features, giving us a different test. The thinking here is that by randomly choosing only a few of the features, we can avoid making the same choice for this node in every tree we train, and thus we can increase the diversity of our decisions.

When we build ensembles this way, we call the result a *random forest*. The *random* part of the name refers to our random choice of features at each node, and the word *forest* refers to the resulting collection of decision trees.

To create a random forest, we need to provide the same two parameters that we used for bagging: the size of each bootstrap and the number of trees to build. We also have to specify what fraction of the features to consider at each node. We can express this as a percentage of the number of features in the node. Alternatively, many libraries offer a variety of algorithms that pick that percentage for us.

Extra Trees

Let's look at a second way to randomize our construction of trees when building an ensemble. Normally when we split a node, we consider each feature it contains (or a random subset of them if we're building a random forest), and we find the value of that feature that best splits the samples at that node into two children. As we mentioned, we compare different possible tests using a measure like information gain.

Instead of finding the best splitting point for each feature, let's choose the splitting point randomly, based on the values that are in the node. The result of this change is an ensemble called *Extremely Randomized Trees*, or *Extra Trees*.

Although it may seem that this is destined to give us worse results for that tree, remember that decision trees are prone to overfitting. This random choice of the splitting point lets us trade off a little accuracy for reduced overfitting.

Boosting

The techniques we just finished looking at are all specific to decision trees. Now let's look at another method for building ensembles that is applicable to any kind of learner. This method is called *boosting* (Schapire 2012).

Boosting is a popular algorithm because it lets us combine a large number of small, fast, and inaccurate learners into a single accurate learner. To keep things concrete, let's continue using decision trees as our example learners. We can make the discussion even simpler by focusing on binary classifiers, which assign every sample to one of only two classes. We're going to build our ensemble out of simple classifiers that are just barely useful. To get going, let's start with a thought experiment involving a completely useless classifier and then improve it just a little.

Imagine a dataset where the samples come from two classes. Also imagine a completely random binary classifier. Regardless of a sample's features, the classifier assigns the sample to one of these two classes arbitrarily. If the samples are evenly split in the training set, we have a 50:50 chance of any sample being correctly labeled. We call this *random labeling*, because the odds of getting the right answer are up to chance.

Now suppose that we can tweak our binary classifier so that it does just barely better than chance. For example, Figure 12-4 shows a set of data of

two classes, a binary classifier that is no better than chance, and a binary classifier that is just a tiny bit better than chance.

The learner in Figure 12-4(b) is no better than chance, with half of each class getting incorrectly classified. This is a useless classifier. The learner in part (c) is just slightly less useless than the classifier in part (b) because the slight tilt in the boundary line means it does just a little better than the useless classifier.

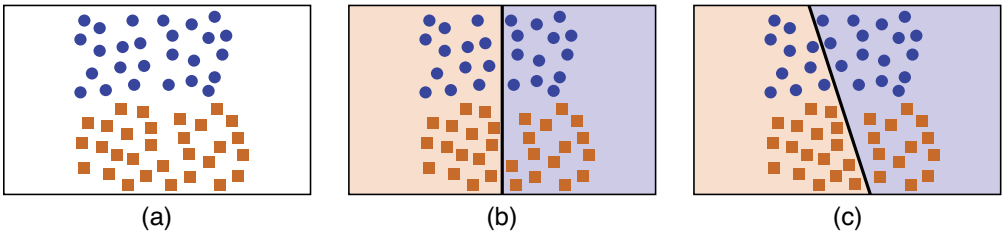


Figure 12-4: (a) Our training data. (b) A random classifier. (c) A terrible, but not quite random, classifier.

We call the classifier in Figure 12-4(c) a *weak learner*. In this situation, a weak learner is any classifier that is even the slightest bit accurate. That is, it assigns the correct class more than 50 percent of the time, but perhaps only barely. The beauty of boosting is that we can use this weak learner as part of an ensemble that produces great results.

In fact, a weak binary learner is just as useful to us even if it does *worse* than chance. That's because we have only two classes. If a classifier is below chance (that is, it assigns the wrong class more frequently than the right one), then we can just swap the output classes, and then it's doing better than chance, rather than worse. The conclusion is that as long as a binary learner isn't completely random, we are able to use it.

Weak classifiers are easy to make. Perhaps the most commonly used weak classifier is a decision tree that's only one test deep. That is, the whole tree is made up of just a root node and its two children. This ridiculously small decision tree is often called a *decision stump*. Because it almost always does a better job than randomly assigning a class to each sample, it's a fine example of a weak classifier. It's small, fast, and a little better than random.

In contrast to a weak learner, a *strong learner* is a classifier that gets the correct label most of the time. The stronger the learner, the better its percentage of being right.

The idea behind boosting is to combine multiple weak classifiers into an ensemble that acts like a strong classifier. Note that our weakness condition is just a minimum threshold. We can combine lots of strong classifiers if we want to, though using weak ones is more common because they're usually faster.

Let's see how boosting works with an example. Figure 12-5 shows a training set of samples that belong to two different classes.

What might be some good classifiers for this data? A fast and easy classifier just draws a straight line through the 2D dataset. We can see that no straight line is going to split up this data because the circular samples surround the square samples on three sides.

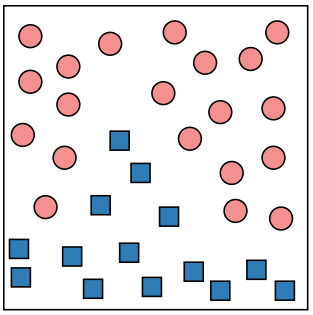


Figure 12-5: A collection of samples we're going to classify using boosting

Even though no single straight line can separate this data, we will see that multiple straight lines can, so let's use straight lines as our weak classifiers. Figure 12-6 shows the boundary line for one such classifier. We'll use A for the name of both the classifier and the boundary line it defines.

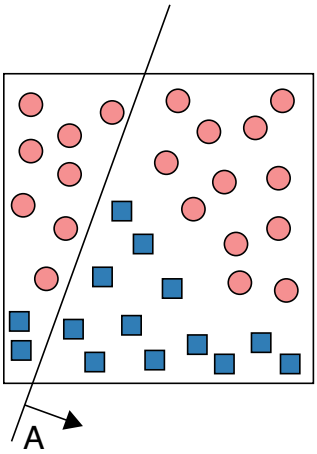


Figure 12-6: Placing one line called A into our samples cuts the two big clusters apart.

In this classifier, everything on the side of A pointed to by the arrow is classified as square, and everything on the other side is classified as a circle. Using our measure of accuracy from Chapter 3, we find that the accuracy of this learner is given by (TP + TN) / (TP + TN + FP + FN) = (12 + 8) / (12 + 8 + 12 + 2) = 20 / 34, or about 59 percent. That's a nice example of a weak learner: it's better than chance (50 percent), but not a lot better.

To use boosting, we'll want to add more lines (that is, additional weak learners) so that ultimately every region formed by the lines contains samples of a single class. After adding two more of these straight-line classifiers, we end up with Figure 12-7.

Line B has an accuracy of about 73 percent. C is terrible, with an accuracy of only about 12 percent. But as we noted before, that's fine, because if we just swap the labels that C assigns (that is, just point the arrow in the other direction) it has an accuracy of about 88 percent!

The three lines, or boundaries, in Figure 12-7 together create seven nonoverlapping regions. The figure also shows that each region contains only one class of samples. By looking at these regions, we can see a way to determine the class of a sample using just the outputs of the three classifiers.

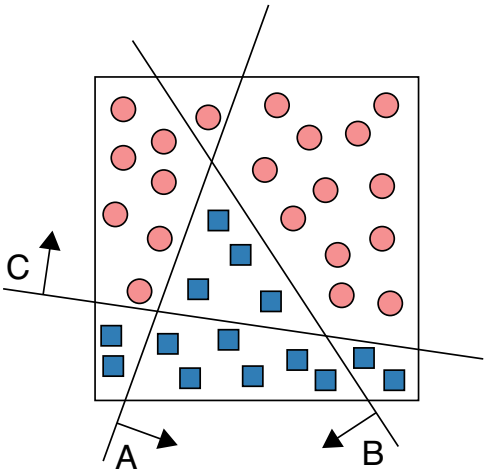


Figure 12-7: Two more lines added to Figure 12-6

Let's draw our three boundaries together. We will label each region with the learners that point toward it. The result is shown in Figure 12-8.

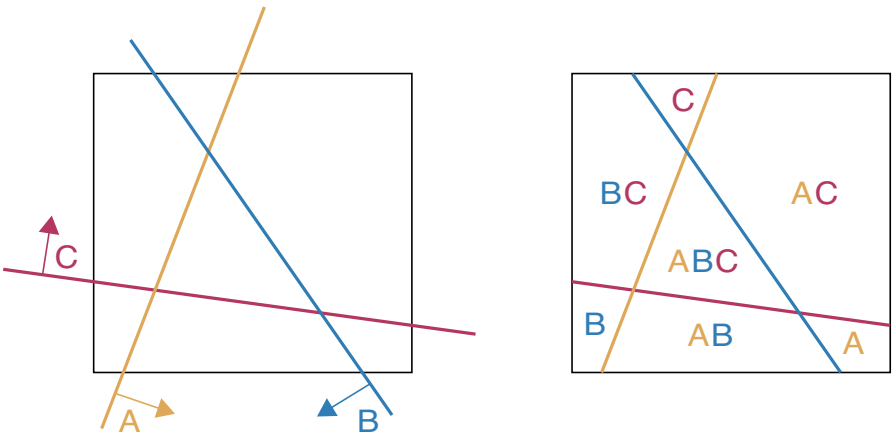


Figure 12-8: On the left, we show three lines named A, B, and C. On the right, each region is marked with the names of the learners that put that region on the positive side of their respective lines.

When our classifiers get a new sample, normally they each return a class. Instead, let's set them up to return a 1 if the sample is on the positive side of that classifier's boundary (that is, the side pointed to by the arrow in Figure 12-8), and 0 otherwise.

Now we can add up all the contributions from all three learners in each cell.

For example, consider the region at the top center, marked C in Figure 12-8. It's on the positive side of learner C, earning it a score of 1. It's on the negative side of both A and B, each of which thus contribute 0, so the sum of the three outputs is 1. The region at the bottom, marked AB, gets 1 from learners A and B, and 0 from C, giving it a total of 2. These scores are shown along with the other regions in Figure 12-9.

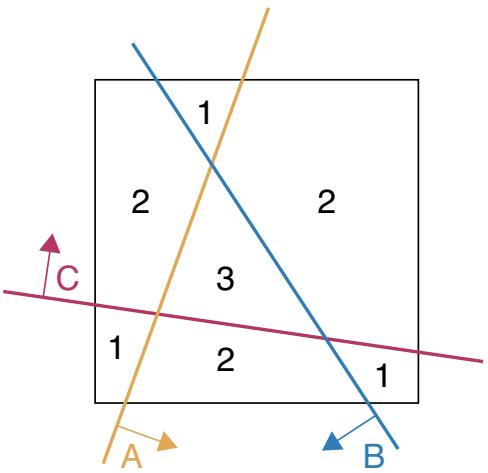


Figure 12-9: The composite score for each of the seven regions. Each letter in Figure 12-8 earns that region a 1.

We've almost created our new classifier. There are two steps to go. First, we replace the 1 we arbitrarily assigned to each classifier's output with a more useful value. Second, we find a threshold that turns the summed value in each region into a class.

Recall our discussion of weighted plurality voting from earlier in the chapter. If a region is on the positive side of a line, then that line is voting for that region. Rather than simply adding 1 from every classifier, we can assign each classifier its own voting weight. For example, if classifiers A, B, and C have voting weights 2, 3, and –4, and a point is on the positive side of A and C but not B, then A contributes 2, B contributes 0 (since the point is on the negative side of line B), and C contributes –4, for a total of 2 + 0 + –4 = –2.

The voting weight for each classifier is found for us by the boosting algorithm. Rather than going into those mechanics, let's visualize the results of a specific set of weights for this dataset so we can see their effect.

In Figure 12-10 we show the regions that are affected by the score for each learner. A dark region gets that learner's value, while a light region

does not (so the learner's value in light regions is 0). Here we use the weights 1.0, 1.5, and –2 for A, B, and C, respectively. Recall that line C was pointing in the "wrong" direction. Giving a negative value to C's weight has the effect of reversing the decisions from classifier C.

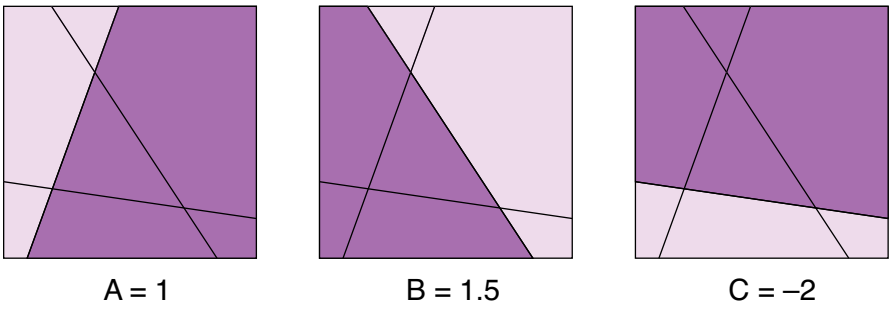


Figure 12-10: We assign a numerical value to each region that is classified as positive by each learner. The dark regions for each line get the weight associated with that line.

The sums of all of these scores are shown in Figure 12-11. Blue regions have positive sums, and red regions have negative sums. These exactly correspond to where the circles and squares fell in our dataset. Any sample in a positive region is a square, and any sample in a negative region is a circle.

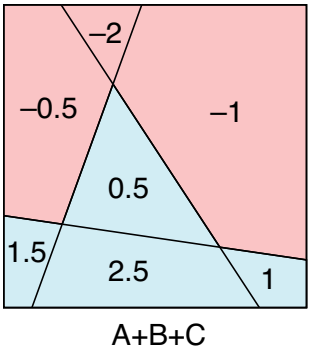


Figure 12-11: Adding up the scores from Figure 12-10

When a sample arrives, we send it to each classifier (that is, we test it against its corresponding line). For each classifier that finds the sample to be on the positive side of its line, we contribute its voting weight to a running sum. After adding up all the classifier outputs, we determine if the sample is positive or negative, which tells us which class the sample belongs to. We've correctly classified our data!

Let's look at another example. Figure 12-12 shows a new set of data.

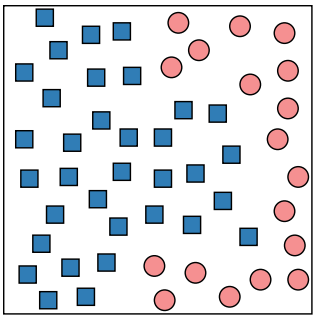


Figure 12-12: A set of data we'd like to classify using boosting

For this data, let's try using four learners. Figure 12-13 shows the four weak learners that a boosting algorithm might find in order to partition this data.

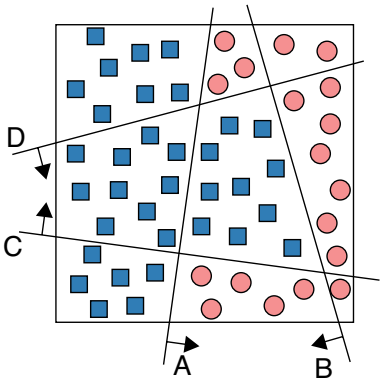


Figure 12-13: Four lines that let us classify the data of Figure 12-12

As before, the algorithm also assigns weights to these learners. Let's illustrate the results using weights –8, 2, 3, and 4 for learners A, B, C, and D respectively. Figure 12-14 shows which regions have those weights added to their overall score. Light-colored regions implicitly receive a value of 0.

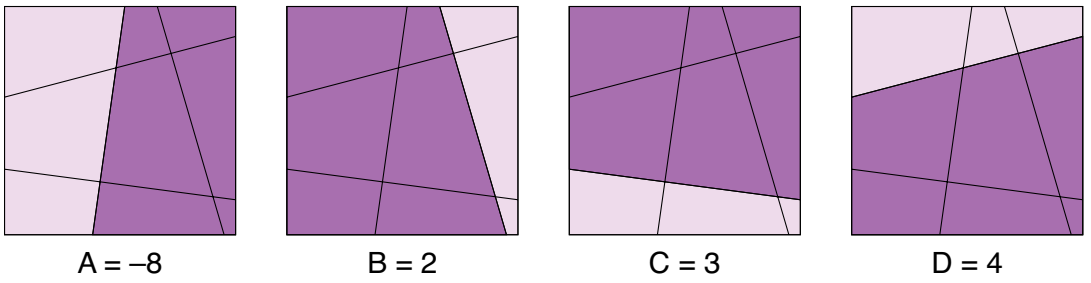


Figure 12-14: The regions corresponding to each learner

Figure 12-15 shows the sums of the contributions for each region. Again, a positive or negative sum distinguishes the two types of regions. We've found a way to combine four weak learners to correctly classify the points in Figure 12-13.

The beauty of boosting is that it takes classifiers that are simple and fast, but lousy, and by finding weights for them, it turns the ensemble into a single great classifier.

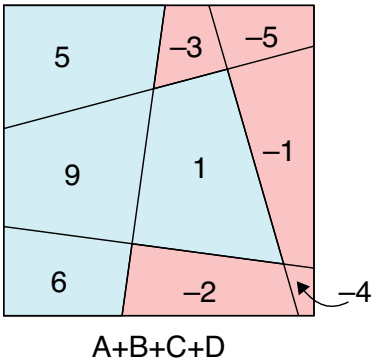


Figure 12-15: The sums of the scores of each region from Figure 12-14. Positive regions are shown in blue, and they correctly classify the points in Figure 12-13.

The only hyperparameter we need to provide is how many classifiers we want. In boosting, as in bagging, a good rule of thumb is to start with about as many classifiers as there are classes (Bonab 2016). That means that our earlier examples started on the high side, since we used three or four classifiers for only two classes. But as in so many things in machine learning, the best value is found by trial and error.

Boosting made its first appearance as part of an algorithm called *Adaboost* (Freund 1997; Schapire 2013). Although it can work with any learning algorithm, boosting has been particularly popular with decision trees. In fact, it works very well with the decision stumps we mentioned previously (these are trees that have only a root node and its two immediate children). We can think of the lines we used in Figures 12-7 and 12-13 as decision stumps since they have just one test: Is a sample on the side of the line pointed to by the arrow, or is it not?

It's worth noting that boosting is not a sure-fire way to improve all classification algorithms. The theory of boosting only covers binary classification, as in our earlier examples (Fumera 2008; Kak 2016). This is partly why boosting has been so popular and successful with decision tree classifiers.

Summary

Ensembles are collections of diversified learners. The general idea is that we gather up multiple learners of similar type, but trained on different data,

and let them all evaluate an input. We then let them vote for the class each one determines, and the winner of the most votes is reported as the class for that input. The thinking is that any errors in the individual learners are essentially voted away by the class agreed upon by the others.

Boosting is a way of using many weak learners as an ensemble to perform like a strong learner.

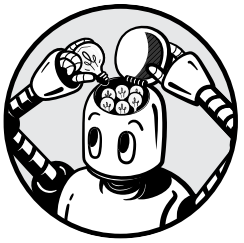
This discussion wraps up our discussion of machine learning techniques. Starting in Chapter 13, we look at the neural networks that power deep learning algorithms. We'll see that the methods we introduced here are helpful in deep learning, because they help us understand our data and make the best choices of algorithms and networks to work with that data, and produce results that are useful to us.

PART III

DEEP LEARNING BASICS

13

NEURAL NETWORKS



Deep learning algorithms are based on building a network of connected computational elements. The fundamental unit of such networks is a small bundle of computation called an *artificial neuron*, though it's often referred to simply as a *neuron*. The artificial neuron was inspired by human neurons, which are the nerve cells that make up our brain and central nervous system and are largely responsible for our cognitive abilities.

In this chapter, we see what artificial neurons look like and how to arrange them into networks. We then group them into layers, which create deep learning networks. We also look at various ways to configure the outputs of these artificial neurons so that they produce the most useful results.

Real Neurons

In biology, the term *neuron* is applied to a wide variety of complex cells distributed throughout every human body. These cells all have similar structure and behavior, but they're specialized for many different tasks. Neurons are sophisticated pieces of biology that use a mix of chemistry, physics, electricity, timing, proximity, and other means to perform their behaviors and communicate with one another (Julien 2011; Khanna 2018; Lodish et al. 2000; Purves et al. 2001). A highly simplified sketch of a neuron is shown in Figure 13-1.

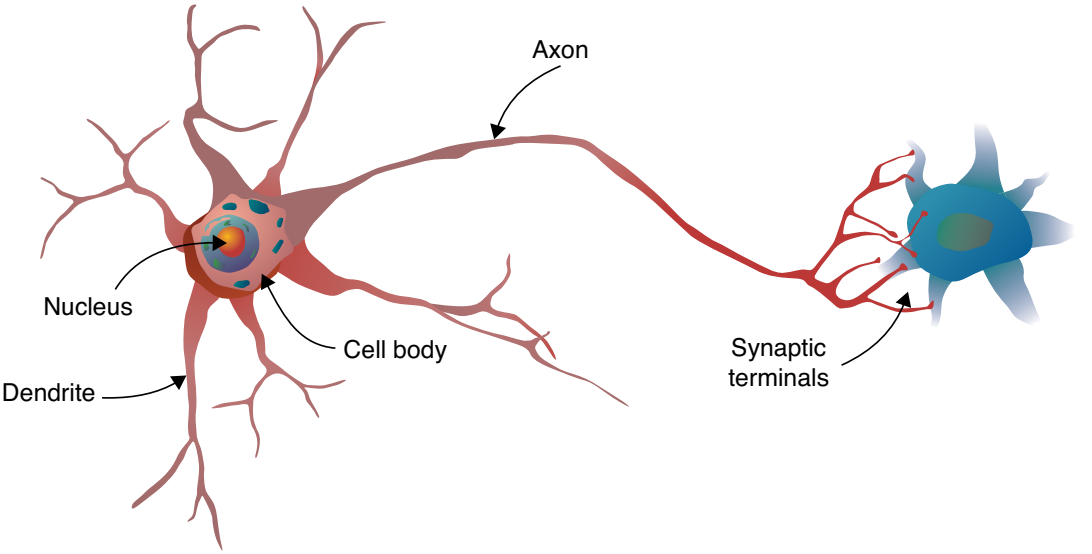


Figure 13-1: A sketch of a highly simplified biological neuron (in red) with a few major structures identified. This neuron's outputs are communicated to another neuron (in blue), only partially shown (adapted from Wikipedia 2020b).

Neurons are information processing machines. One type of information arrives in the form of chemicals called *neurotransmitters* that temporarily *bind*, or attach, onto *receptor sites* located on the neuron (Goldberg 2015). Let's sketch out what happens next in the broadest possible terms.

The chemicals that bind to the receptor sites cause electrical signals to travel into the body of the neuron. Each of these signals can be either positive or negative. All of the electrical signals arriving at the neuron's body over a short interval of time are added together and then compared to a *threshold*. If the total exceeds that threshold, a new signal is sent along the axon to another part of the neuron, causing specific amounts of neurotransmitters to be released into the environment. These molecules then bind with other neurons, and the process repeats.

In this way, information is propagated and modified as it flows through the densely connected network of neurons in the brain and central nervous system. If two neurons are physically close enough to each other that one

can receive the neurotransmitters released by the other, we say that the neurons are *connected*, even though they may not be actually touching. There is some evidence that the particular pattern of connections between neurons is as essential to cognition and identity as the neurons themselves (Sporns, Tononi, and Kötter 2005; Seung 2013). A map of an individual's neuronal connections is called their *connectome*. Connectomes are as unique as fingerprints or iris patterns.

Although real neurons and their surrounding environment are tremendously complex and subtle, the basic mechanism described here has an appealing elegance. Responding to this, some scientists have attempted to emulate or duplicate the brain by creating enormous numbers of simplified neurons and their environment, in hardware or software, hoping that interesting behavior will emerge (Furber 2012; Timmer 2014). So far, this has not delivered results that most people would call intelligence.

But we can connect up simplified neurons in specific ways to produce great results on a wide range of problems. Those are the types of structures that will be our focus in this chapter, and the rest of this book.

Artificial Neurons

The "neurons" we use in machine learning are inspired by real neurons in the same way that a stick figure drawing is inspired by a human body. There's a resemblance, but only in the most general sense. Almost all of the details are lost along the way, and we're left with something that's more of a reminder of the original, rather than even a simplified copy.

This has led to some confusion, particularly in the popular press, where "neural network" is sometimes used as a synonym for "electronic brain," and from there, it's only a short step to general intelligence, consciousness, emotions, and perhaps world domination and the elimination of human life. In reality, the neurons we use are so abstracted and simplified from real neurons that many people prefer instead to call them by the more generic name of *units*. But for better or worse, the word *neuron*, the phrase *neural net*, and all the related language are apparently here to stay, so we use them in this book as well.

The Perceptron

The history of artificial neurons may be said to begin in 1943, with the publication of a paper that presented a massively simplified abstraction of a neuron's basic functions in mathematical form, and described how multiple instances of this object could be connected into a *network*, or *net*. The big contribution of this paper was that it proved mathematically that such a network could implement any idea expressed in the language of mathematical logic (McCulloch and Pitts 1943). Since mathematical logic is the basis of machine calculation, that means neurons could perform mathematics. This was a big deal, because it provided a bridge between the fields of math, logic, computing, and neurobiology.

Building on that insight, in 1957 the *perceptron* was proposed as a simplified mathematical model of a neuron (Rosenblatt 1962). Figure 13-2 is a block diagram of a single perceptron with four inputs.

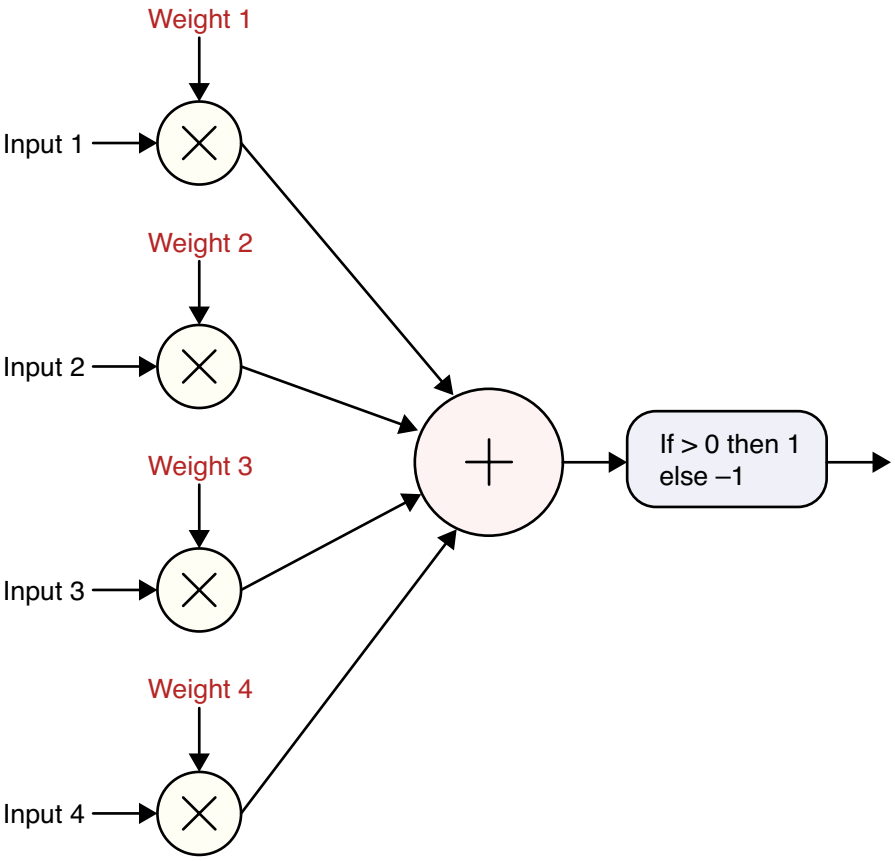


Figure 13-2: A four-input perceptron

Every input to a perceptron is represented by a single floating-point number. Each input is multiplied by a corresponding floating-point number called a *weight*. The results of these multiplications are all added together. Finally, we compare the result to a threshold value. If the result of the summation is greater than 0, the perceptron produces an output of +1, otherwise it's –1 (in some versions, the outputs are 1 and 0, rather than +1 and –1).

Though the perceptron is a vastly simplified version of a real neuron, it's proven to be a terrific building block for deep learning systems.

The history of the perceptron is an interesting part of the culture of machine learning, so let's look at just a couple of its key events; more complete versions may be found online (Estebon 1997; Wikipedia 2020a).

After the principles of the perceptron had been verified in software, a perceptron-based computer was built at Cornell University in 1958. It was a rack of wire-wrapped boards the size of a refrigerator, called the Mark

I Perceptron (Wikipedia 2020c). The device was built to process images, using a grid of 400 photocells that could digitize an image at a resolution of 20 by 20 pixels (the word *pixel* hadn't yet been coined). The weight applied to each input of the perceptron was set by turning a knob that controlled an electrical component called a potentiometer. To automate the learning process, electric motors were attached to the potentiometers so the device could literally turn its own knobs to adjust its weights and thereby change its calculations, and thus its output. The theory guaranteed that, with the right data, the system could learn to separate two different classes of inputs that could be split with a straight line.

Unfortunately, not many interesting problems involve sets of data that are separated by a straight line, and it proved hard to generalize the technique to more complicated arrangements of data. After a few years of stalled progress, a book proved that the original perceptron technique was fundamentally limited (Minsky and Papert 1969). It showed that the lack of progress wasn't due to a lack of imagination, but the result of theoretical limits built into the structure of a perceptron. Most interesting problems, and even some very simple ones, were provably beyond the ability of a perceptron to solve.

This result seemed to signal the end of perceptrons for many people, and a popular consensus formed that the perceptron approach was a dead end. Enthusiasm, interest, and funding all dried up, and most people directed their research to other problems. This period, which lasted roughly between the 1970s and 1990s, was called the *AI winter*.

But despite a widespread interpretation that the perceptron book had closed the door on perceptrons in general, in fact it had only shown the limitations of how they'd been used up to that time. Some people thought that writing off the whole idea was an overreaction and that perhaps the perceptron could still be a useful tool if applied in a different way. It took roughly a decade and a half, but this point of view eventually bore fruit when researchers combined perceptrons into larger structures and showed how to train them (Rumelhart, Hinton, and Williams 1986). These combinations easily surpassed the limitations of any single unit. A series of papers then showed that careful arrangements of multiple perceptrons, beefed up with a few minor changes, could solve complex and interesting problems.

This discovery rekindled interest in the field, and soon research with perceptrons became a hot topic once again, producing a steady stream of interesting results that have led to the deep learning systems we use today. Perceptrons remain a core component of many modern deep learning systems.

Modern Artificial Neurons

The neurons we use in modern neural networks are only slightly generalized from the original perceptrons. There are two changes: one at the input, and one at the output. These modified structures are still sometimes called perceptrons, but there's rarely any confusion because the new versions are used almost exclusively. More commonly, they're just called *neurons*. Let's look at these two changes.

The first change to the perceptron of Figure 13-2 is to provide each neuron with one more input, which we call the *bias*. This is a number that doesn't come from the output of a previous neuron. Instead, it's a number that's directly added into the sum of all the weighted inputs. Every neuron has its own bias. Figure 13-3 shows our original perceptron, but with the bias term included.

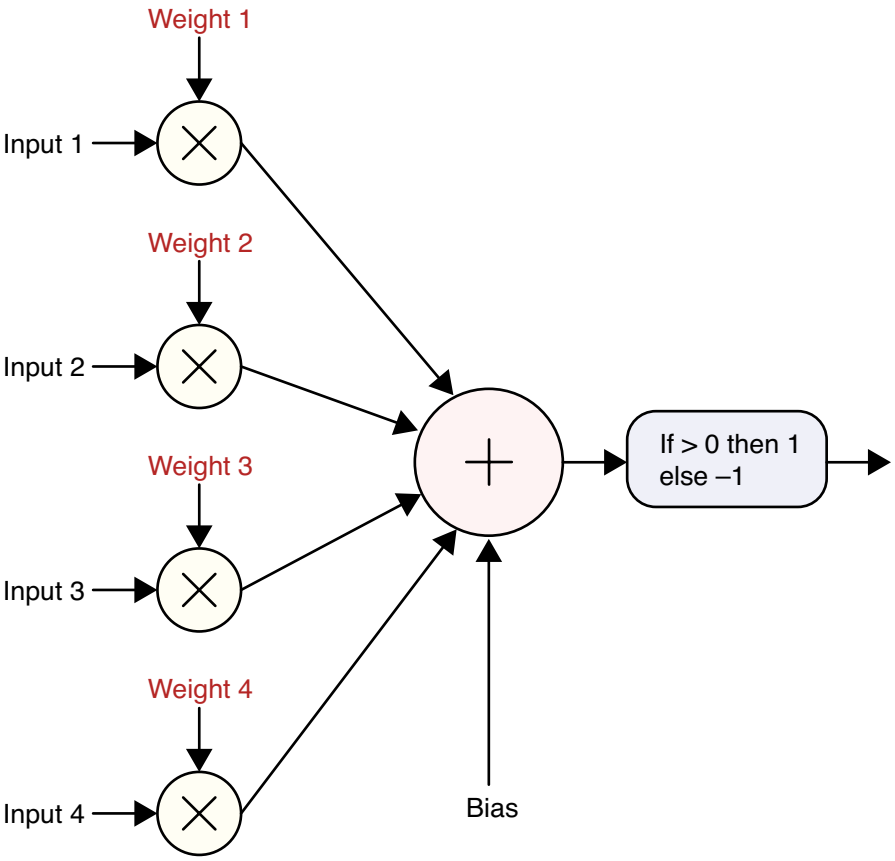


Figure 13-3: The perceptron of Figure 13-2, but now with a bias term

Our second change to the perceptron of Figure 13-2 is at the output. The perceptron in that figure tests the sum against a threshold of 0, and then produces either a –1 or 1 (or 0 or 1). We generalize this by replacing the testing step with a mathematical function that takes the sum (including the bias) as input and returns a new floating-point value as output. Because the output of a real neuron is called its *activation*, we call this function that calculates the artificial neuron's output the *activation function*. The little test shown in Figure 13-2 is an activation function, but one that's rarely used anymore. Later in this chapter we'll survey a variety of activation functions that have proved to be popular and useful in practice.

Drawing the Neurons

Let's identify a convention that's used by most drawings of artificial neurons. In Figure 13-3 we showed the weights explicitly, and we also included the multiplication steps to show how the weights multiply the inputs. This takes a lot of room on the page. When we draw diagrams with a lot of neurons, all of these details can make for a cluttered and dense figure. So instead, in virtually all neural network diagrams, the weights and their multiplications are implied.

This is important, and bears repeating: in neural network diagrams, the weights, and the steps where they multiply the inputs, are not drawn. Instead, we're supposed to know that they are there and mentally include them in the diagram. If we show the weights at all, we typically label the lines from the inputs with the name of the weight. Figure 13-4 shows Figure 13-3 drawn in this style.

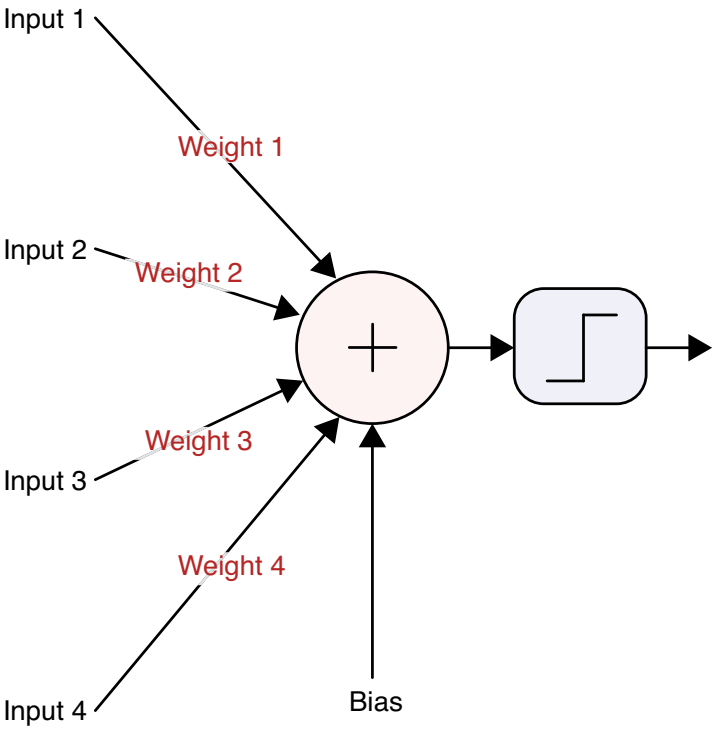


Figure 13-4: A neuron is often drawn with the weights on the arrows.

In Figure 13-4, we also changed the threshold test at the end to a little picture. This is a drawing of a function called a *step*, and it's meant to give us a visual reminder that any activation function can go into that spot. Basically, a number goes into that step, and a new number comes out, determined by whichever function we choose for the job.

We usually simplify things again. This time we omit the bias by pretending it's one of the inputs. This not only makes the diagram simpler, but it makes the math simpler as well, which, in this case, also leads to more efficient algorithms. This simplification is called the *bias trick* (the word *trick* comes from mathematics, where it's a complimentary term sometimes used for a clever simplification of a problem). Rather than change the value of the bias, we set the bias to always be 1, and change the weight applied to it before it gets summed up with the other inputs. Figure 13-5 shows this change in labeling. Though the bias term is always 1 and only its weight can change, we usually ignore the distinction and just talk about the value of the bias.

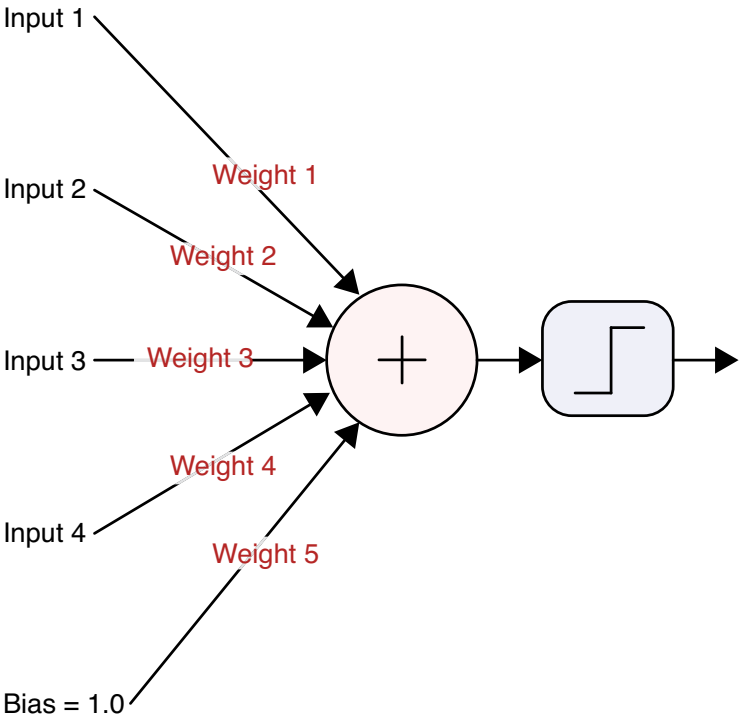


Figure 13-5: The bias trick in action. Rather than show the bias term explicitly, as in Figure 13-4, we pretend it's another input with its own weight.

We want our artificial neuron diagrams to be as simple as possible because when we start building up networks we'll be showing lots of neurons at once, so most of these diagrams take two additional steps of simplification. First, they don't show the bias at all. We're supposed to remember that the bias is included (along with its weight), but it's not shown. Second, the weights are often omitted as well, as in Figure 13-6. This is unfortunate, because the weights are the most important part of the neuron for us. The reason for this is that they are the only things we can change during training. Despite being left out of most drawings, they're so essential that we repeat the key idea yet again: *even though we don't show the weights explicitly, the weights are always there.*

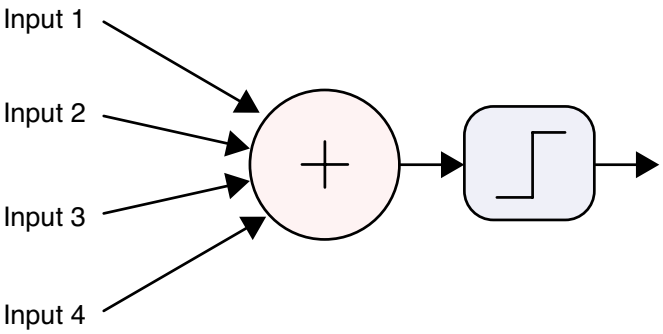


Figure 13-6: A typical drawing of an artificial neuron. The bias term and the weights are not shown, but they are definitely present.

Like real neurons, artificial neurons can be wired up in networks, where each input comes from the output of another neuron. When we connect neurons together into networks, we draw "wires" to connect one neuron's output to one or more other neurons' inputs. Figure 13-7 shows this idea visually.

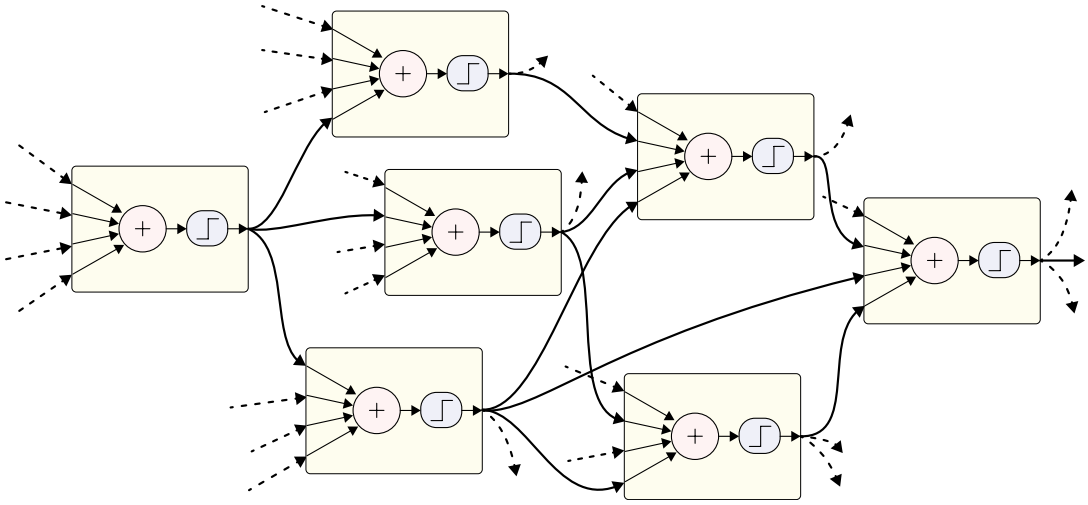


Figure 13-7: A piece of a larger network of artificial neurons. Each neuron receives its inputs from other neurons. The dashed lines show connections to and from outside this little cluster.

This is a *neural network*. Usually the goal of a network like Figure 13-7 is to produce one or more values as outputs. We'll see later how we can interpret the numbers at the outputs in meaningful ways.

Even though we've said that we usually don't draw the weights, in discussions, sometimes it's useful to refer to individual weights. Let's look at a common convention for weight names. Figure 13-8 shows six neurons. For convenience, we've labeled each neuron with a letter. Each weight corresponds to how the output of one specific neuron is changed on its way to another specific neuron. Each of these connections is shown as a line in the

figure. To name a weight, we combine the name of the output neuron with the input neuron. For example, the weight that multiplies the output of A before it's used by D is called AD.

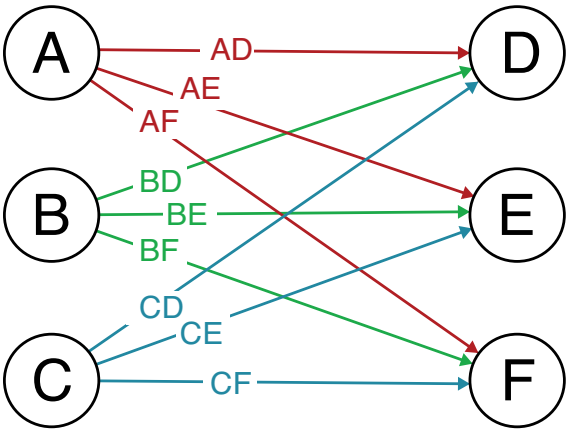


Figure 13-8: The weights are named by combining the names of the output and input neurons.

From a structural point of view, it makes no difference whether we draw the weights inside each neuron, or on the wires that carry values to it. Various authors assume one or the other if it makes their discussion easier to follow, but we can always take the other viewpoint if we like.

In Figure 13-8, we named the weight from neuron A to neuron D as AD. Some authors flip this around and write DA, because it's a more direct match to how we often write the equations. It's always worth a moment to check which order is being used in diagrams like this.

Feed-Forward Networks

Figure 13-7 showed a neural network with no apparent structure. A key feature of deep learning is that we arrange our neurons into *layers*. Typically, the neurons on each layer get their inputs only from the previous layer and send their outputs only to the following layer, and neurons do not communicate with other neurons on the same layer (there are, as always, exceptions to these rules).

This organization allows us to process data in stages, with each layer of neurons building on the work done by the previous stage. By analogy, consider an office tower of many floors. The people on any given floor receive their work only from the people on the floor immediately below them, and they pass their work on only to the people on the floor immediately above them. In this analogy, each floor is a layer, and the people are the neurons on that layer.

We say that this type of arrangement processes the data *hierarchically*. There is some evidence that the human brain is structured to handle some tasks hierarchically, including the processing of sensory data like vision and hearing (Meunier et al. 2009; Serre 2014). But here again, the connection

between our computer models and real biology is much closer to inspiration than emulation.

It's amazing that hooking up neurons in a series of layers produces anything useful. As we saw earlier, a single artificial neuron can hardly manage to do anything. It takes a bunch of numerical inputs, weights them, adds the results together, and then passes that result through a little function. This process can identify a straight line that splits a couple of clumps of data, and not much else. But if we assemble many thousands of these little units into layers and use some clever ideas to train them, then, working together, they're capable of recognizing speech, identifying faces in photographs, and even beating humans at games of logic and skill.

The key to this is organization. Over time people have developed a number of ways to organize layers of neurons, resulting in a collection of common layer structures. The most common network structure arranges the neurons so that information flows in only one direction. We call this a *feed-forward network* because the data is flowing forward, with earlier neurons feeding, or delivering values to, later neurons. The art of designing a deep learning system lies in choosing the right sequence of layers, and the right hyperparameters, to create the basic architecture. To build a useful architecture for any given application, we need to understand how the neurons relate to one another. Let's now look at how collections of neurons communicate, and how to set up the initial weights before learning begins.

Neural Network Graphs

We usually represent neural networks as *graphs*. The study of graphs is so large that it is considered a field of mathematics in its own right, called *graph theory* (Trudeau 1994). Here, we're going to stick to the basic ideas of graphs, because that's all we need to organize our neural networks. Though we know we'll usually be working with layers, let's start out with some general graphs first, such as those shown in Figure 13-9.

A graph is made up of *nodes* (also called *vertices* or *elements*), here shown as circles. In this book, nodes are usually neurons, and throughout this book, we occasionally refer to one or more neurons in a network like this as nodes. The nodes are connected by arrows called *edges* (also called *arcs*, *wires*, or simply *lines*). The arrowhead is often left off when the direction of information flow is consistent in the drawing, which is almost always left to right or bottom to top. Information flows along the edges, carrying the output of one node to the inputs of others. Since information flows in only one direction on each edge, we sometimes call this kind of graph a *directed graph*.

The general idea is that we start things off by putting data into the input node or nodes, and then it flows through the edges, visiting nodes where it is transformed or changed, until it reaches the output node or nodes. No data ever returns to a node once it has left. In other words, information only flows forward, and there are no loops, or *cycles*. This kind of graph is like a little factory. Raw materials come in one end, and pass through machines that manipulate and combine them, ultimately producing one or more finished products at the end.

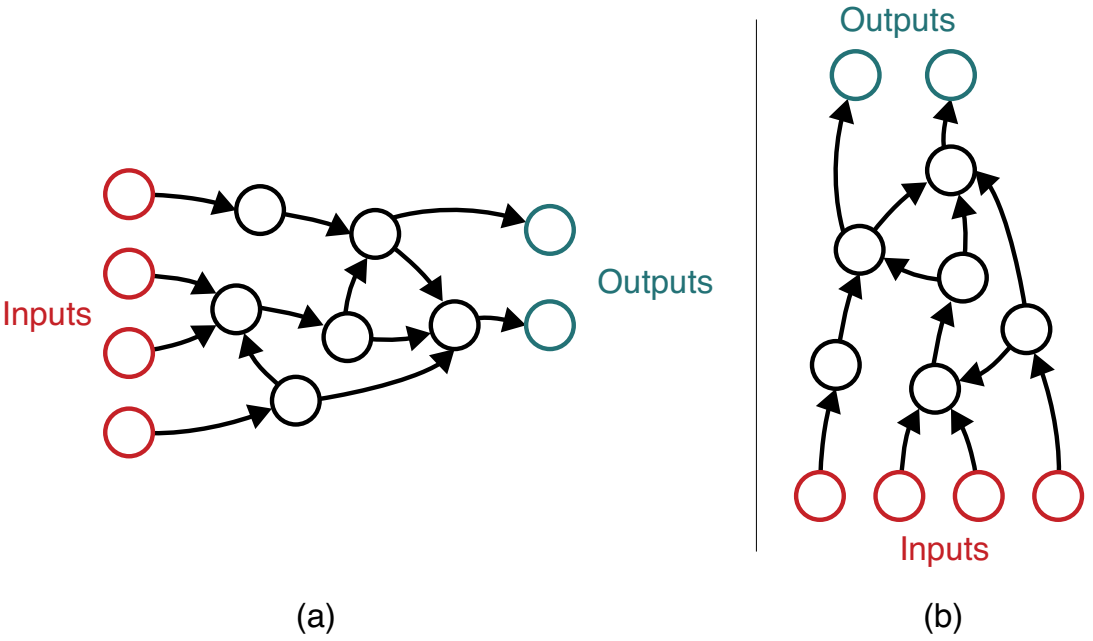


Figure 13-9: Two neural networks drawn as graphs. Data flows from node to node along the edges, following the arrows. When the edges are not labeled with an arrow, data usually flows left-to-right or bottom-to-top. (a) Mostly left-to-right flow. (b) Mostly bottom-to-top flow.

We say that a node near the inputs in Figure 13-9(a) is *before* a node nearer to the outputs, which comes *after* it. In Figure 13-9(b), we'd say a node near the inputs is *below* a node near the outputs, which is *above* it. Sometimes this below/above language is used even when the graph is drawn left to right, which can be confusing. It can help to think of *below* as "closer to the inputs," and *above* as "closer to the outputs."

We also sometimes say that if data flows from one node to another (let's say it flows from A to B), then node A is an *ancestor* or *parent* of B, and node B is a *descendant* or *child* of A.

A common rule in neural networks is that there are no loops. This means that data coming out of a node can never make its way back into that same node, no matter how circuitous a path it follows. The formal name for this kind of graph is a *directed acyclic graph* (or *DAG*, pronounced to rhyme with "drag"). The word *directed* here means that the edges have arrows (which may only be implied, as we mentioned earlier). The word *acyclic* means there are no *cycles*, or loops. As always, there are exceptions to the rules, but they're rare. We'll see one such exception when we discuss recurrent neural networks (RNNs) in Chapter 19.

DAGs are popular in many fields, including machine learning, because they are significantly easier to understand, analyze, and design than arbitrary graphs that have loops. Including loops can introduce *feedback*, where a node's output is returned to its input. Anyone who's moved a live

microphone too close to a speaker is familiar with how quickly feedback can grow out of control. The acyclic nature of a DAG naturally avoids the feedback problem, which saves us from dealing with this complex issue.

Recall that a graph or network in which data only flows forward from inputs to outputs is called *feed-forward*. In Chapter 14, we'll see that a key step in training neural networks involves temporarily flipping the arrows around, sending a particular type of information from the output nodes back to the input nodes. Although the normal flow of data is still feed-forward, when we push data through it backward, generally we call that a *feed-backward*, *backward-flow*, or *reverse-feed* algorithm. We reserve the word *feedback* for situations in which a loop in the graph can enable a node to receive its own output as input. As we've said, we generally avoid feedback in neural networks.

Interpreting graphs like those in Figure 13-9 usually means picturing the information as it flows along the edges, from one node to the next. But this picture only makes sense if we make some conventional assumptions. Let's look at those now.

Though we often use the word *flow* in various forms when referring to how data moves through the graph, this isn't like the flow of water through pipes. Water flowing through pipes is a *continuous* process: new molecules of water flow through the pipes at every moment. The graphs we work with (and the neural networks they represent) are *discrete*: information arrives one chunk at a time, like text messages.

Recall from Figure 13-5 that we can draw a neural network by placing a weight on each edge (rather than inside a neuron). We call this style of the network a *weighted graph*. As we saw in Figure 13-6, we rarely draw the weights explicitly, but they are implied. It is always the case that in any neural network graph, even if no weights are explicitly shown, we are to understand that a unique weight is on each edge and as a value moves from one neuron to another along that edge that value is multiplied by the weight.

Initializing the Weights

Teaching a neural network involves gradually improving the weights. The process begins when we assign initial values to the weights. How should we pick these starting values? It turns out that, in practice, how we initialize the weights can have a big effect on how quickly our network learns.

Researchers have developed theories for good ways to pick the initial values for the weights, and the various algorithms that have proved most useful are each named after the lead authors on the publications that describe them. The *LeCun Uniform*, *Glorot Uniform* (or *Xavier Uniform*), and *He Uniform* algorithms are all based on selecting initial values from a uniform distribution (LeCun et al. 1998; Glorot and Bengio 2010; He et al. 2015). It probably won't be much of a surprise that the similarly named *LeCun Normal*, *Glorot Normal* (or *Xavier Normal*), and *He Normal* initialization methods draw their values from a normal distribution.

We don't need to get into the math behind these algorithms. Happily, modern deep learning libraries offer each of these schemes, plus variations on them. Often the technique used by the library by default works great, so we rarely need to explicitly choose how to initialize the weights.

Deep Networks

Of the many possible ways to organize neurons in a network, placing them in a series of layers has proven to be both flexible and extremely powerful. Typically, neurons within a layer aren't connected to one another. Their inputs come from the previous layer, and their outputs go to the next layer.

In fact, the phrase *deep learning* comes from this structure. If we imagine many layers drawn side by side, we might call the network "wide." If they were drawn vertically and we stood at the bottom looking up, we might call it "tall." If we stood at the top and looked down, we might call it "deep." And that's all that *deep learning* means: a network made of a series of layers that we often draw vertically.

A result of organizing neurons in layers is that we can analyze data hierarchically. The early layers process the raw input data, and each subsequent layer is able to use information from neurons on the previous layer to process larger chunks of data. For example, when considering a photograph, the first layer usually looks at the individual pixels. The next layer looks at groups of pixels, the one after that looks at groups of those groups, and so on. Early layers might notice that some pixels are darker than others, whereas later layers might notice that a clump of pixels looks like an eye, and a much later layer might notice the collection of shapes that reveal that the whole image shows a tiger.

Figure 13-10 shows an example of a deep learning architecture using three layers.

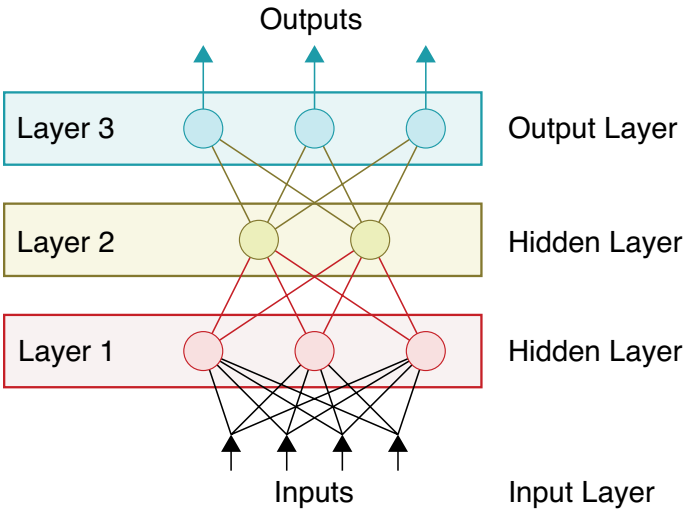


Figure 13-10: A deep learning network

When we draw the layers vertically, as in Figure 13-10, the inputs are almost always drawn at the bottom, and the outputs where we collect our results are almost always drawn at the top.

In Figure 13-10, all three layers contain neurons. In practical systems, we usually use lots of other kinds of layers, which we might group together as *support layers*. We'll see many such layers in later chapters. When we count the number of layers in a network, we usually don't count these support layers. Figure 13-10 would be described as a deep network of three layers.

The topmost layer that contains neurons (Layer 3 in Figure 13-10) is called the *output layer*.

We would probably expect that Layer 1 in Figure 13-10 would be called the *input layer*, but it's not. In a quirk of terminology, the term *input layer* is applied to the bottom of the network, labeled "Inputs" in Figure 13-10. There's no processing in this "layer." Instead it's just the memory where the input values reside. The input layer is an example of a support layer because it has no neurons, and therefore isn't included when we count the layers in a network. The number of layers we count is called the network's *depth*.

If we imagine standing above the top of Figure 13-10 and looking down, we only see the output layer. If we imagine we are below the bottom and looking up, we only see the input layer. The layers in between aren't visible to us. Each of these layers between the input and output is called a *hidden layer*.

Sometimes the stack is drawn left to right, as in Figure 13-11.

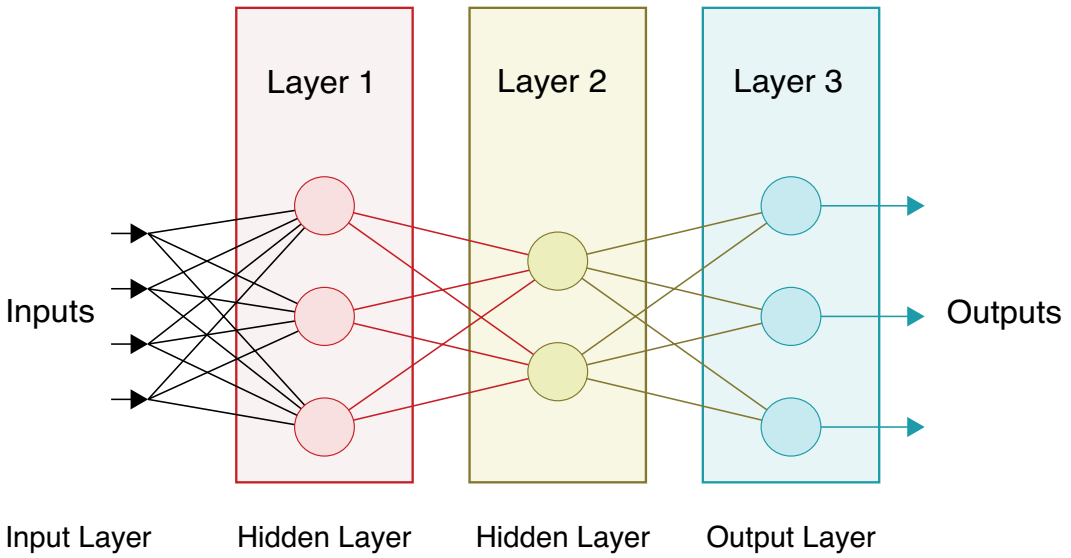


Figure 13-11: The same deep network of Figure 13-10, but drawn with data flowing left to right

Even when drawn this way, we still use terms that refer to the vertical orientation. Authors might say that Layer 2 is "above" Layer 1, and "below" Layer 3. We can always keep things straight regardless of how the diagram is drawn if we think of "above" or "higher" as referring to a layer closer to the outputs, and "below" or "lower" as meaning closer to the inputs.

Fully Connected Layers

A *fully connected layer* (also called an *FC*, *linear*, or *dense* layer) is a set of neurons that each receive an input from *every* neuron on the previous layer. For example, if there are three neurons in a dense layer, and four neurons in the preceding layer, then each neuron in the dense layer has four inputs, one from each neuron in the preceding layer, for a total of $3 \times 4 = 12$ connections, each with an associated weight.

Figure 13-12(a) shows a diagram of a fully connected layer with three neurons, coming after a layer with four neurons.

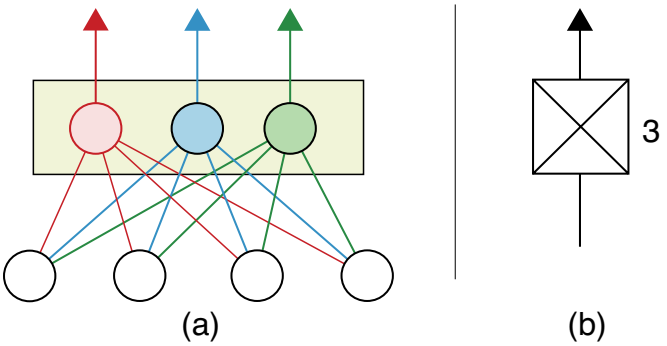


Figure 13-12: A fully connected layer. (a) The colored neurons make up a fully connected layer. Each of the neurons in this layer receives an input from every neuron in the previous layer. (b) Our schematic symbol for a fully connected layer.

Figure 13-12(b) shows a schematic shorthand that we'll use for fully connected layers. The idea is that two neurons are at the top and bottom of the symbol, and the vertical and diagonal lines are the four connections between them. Next to the symbol, we identify how many neurons are in the layer, as we've done here with the number 3. When it's relevant, this is also where we identify that layer's activation function. If a layer is made up of only dense layers, it is sometimes called a *fully connected network*, or, in a throwback to earlier terminology, a *multilayer perceptron (MLP)*.

In later chapters, we'll see many other types of layers that help us organize our neurons in useful ways. For example, *convolution layers* and *pooling layers* have proven very useful for image processing tasks, and we'll give them a lot of attention.

Tensors

We've seen that a deep learning system is built from a sequence of layers. And though the output of any neuron is a single number, we often want to talk about the output of an entire layer at once. The key idea that characterizes this collection of output numbers is its shape. Let's see what that means.

If the layer contains a single neuron, the layer's output is just a single number. We might describe this as an array, or a list, with one element. Mathematically, we can call this a *zero-dimensional array*. The number of

dimensions in an array tells us how many indices we need to use to identify an element. Since a single number needs no indices, that array has zero dimensions.

If we have multiple neurons in a layer, then we can describe their collective output as a list of all the values. Since we need one index to identify a particular output value in this list, this is a one-dimensional (1D) array. Figure 13-13(a) shows such an array containing 12 elements.

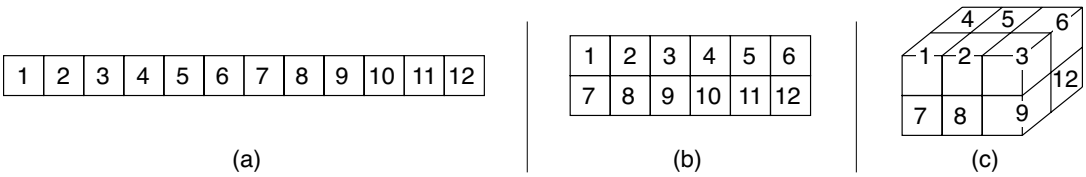


Figure 13-13: Three tensors, each with 12 elements. (a) A 1D tensor is a list. (b) A 2D tensor is a grid. (c) A 3D tensor is a volume. In all cases, and in higher-dimensional cases as well, there are no holes and no elements stick out from the block.

We frequently organize our data into other box-like shapes. For instance, if the input to our system is a black and white image, it can be represented as a 2D array, as in Figure 13-13(b), indexed by x and y positions. If it's a color image, then it can be represented as a 3D array, indexed by x position, y position, and color channel. A 3D shape is shown in Figure 13-13(c).

We frequently call a 1D shape an *array*, *list*, or *vector*. To describe a 2D shape we often use the terms *grid* or *matrix*, and we can describe a 3D shape as a *volume* or *block*. We will often use arrays with even more dimensions. Rather than create a mountain of new terms, we use a single term for any collection of numbers arranged in a box shape with any number of dimensions: a *tensor* (pronounced ten'-sir).

A tensor is merely a block of numbers with a given number of dimensions and a size in each dimension. It has no holes and no bits sticking out. The term *tensor* has a more complex meaning in some fields of math and physics, but in machine learning, we use this word to mean a collection of numbers organized into a multidimensional block. Taken together, the number of dimensions and the size in each dimension provide the *shape* of the tensor.

We often refer to a network's *input tensor* (meaning all the input values), and its *output tensor* (meaning all the output values). The outputs of internal (or hidden) layers have no special name, so we usually say something like "the tensor produced by layer 3" to refer to the multidimensional array of numbers coming out of the neurons on layer 3.

Preventing Network Collapse

Earlier we promised to return to activation functions. Let's look at them now.

Each activation function, while a small piece of the overall structure, is critical to a successful neural network. Without activation functions, the neurons in a network combine, or *collapse*, into the equivalent of a single neuron. And, as we saw earlier, one neuron has very little computational power.

Let's see how a network collapses when it doesn't have activation functions. Figure 13-14 shows a little network with two inputs (A and B), and five neurons (E through G) on three layers. Every neuron receives an input from every neuron on the previous layer, and each connection has a weight, giving us a total of ten weights, shown in red.

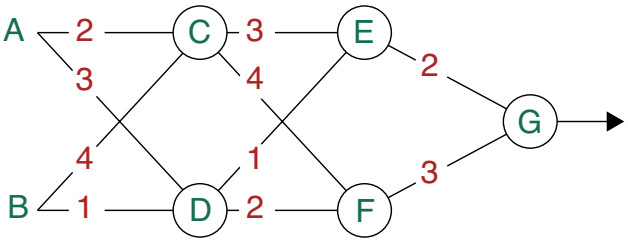


Figure 13-14: A little network of two inputs, five neurons, and ten weights

Let's suppose for the moment that these neurons don't have activation functions. Then we can write the output of each neuron as a weighted sum of its inputs, as in Figure 13-15. In this figure, we're using the mathematical convention of leaving out the multiplication sign when possible, so 2A is shorthand for 2 × A.

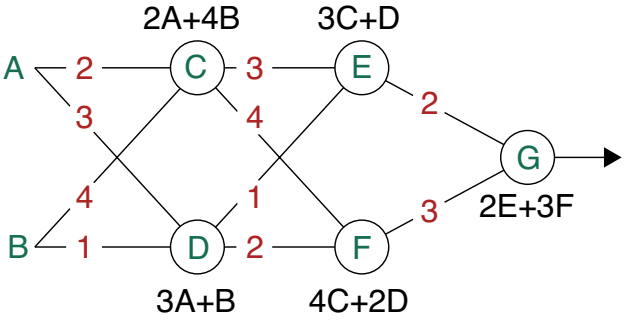


Figure 13-15: Each neuron is labeled with the value of its output.

The outputs of C and D depend only on A and B. Similarly, the outputs of E and F only depend on the outputs of C and D, which means that they, too, ultimately depend only on A and B. The same argument holds for G. If we start with the expression for G, plug in the values for E and F, and then plug in the values for C and D, we get a big expression in terms of A and B. If we do that and simplify, we find that the output of G is 78A + 86B. We can write this as a single neuron with two new weights, as shown in Figure 13-16.

This output of G in Figure 13-16 is exactly the same as the output of G in Figure 13-14. Our whole network has collapsed into a single neuron!

No matter how big or complicated our neural network is, if it has no activation functions, then it will always be equivalent to a single neuron. This is bad news if we want our network to be able to do anything more than what one neuron can do.

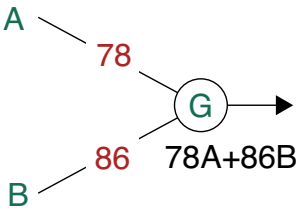


Figure 13-16: This network's output is exactly the same as the output in Figure 13-14.

In mathematical language, we say that our fully connected network collapsed because it only used addition and multiplication, which are in the category of *linear functions*. Linear functions can combine as we just saw, but *nonlinear functions* are fundamentally different and don't combine this way. By designing activation functions to use *nonlinear* operations, we prevent this kind of collapse. We sometimes call an activation function a *nonlinearity*.

There are many different types of activation functions, each producing different results. Generally speaking, the variety is there because in some situations, some functions can run into numerical trouble, making training run more slowly than it should, or even cease altogether. If that happens, we can substitute an alternative activation function that avoids the problem (though of course it has its own weak points).

In practice, a handful of activation functions are all we usually use. When reading the literature and looking at other people's networks, we sometimes see the rarer activation function. Let's survey the functions that by most major libraries usually provide and then gather together the most common ones.

Activation Functions

An *activation function* (sometimes also called a *transfer function*, or a *nonlinearity*) takes a floating-point number as input and returns a new floating-point number as output. We can define these functions by drawing them as little graphs, without any equations or code. The horizontal, or X, axis is the input value, and the vertical, or Y, axis is the output value. To find the output for any input, we locate the input along the X axis, and move directly upward until we hit the curve. That's the output value.

In theory, we can apply a different activation function to every neuron in our network, but in practice, we usually assign the same activation function to all the neurons in each layer.

Straight-Line Functions

Let's first look at activation functions that are made up of one or more straight lines. Figure 13-17 shows a few "curves" that are just straight lines.

Let's look at the leftmost example in Figure 13-17. If we pick any point on the X axis, and go vertically up until we hit the line, the value of that intersection on the Y axis is the same as the value on the X axis. The output, or y value, of this curve is always the same as the input, or x value. We call this the *identity function*.

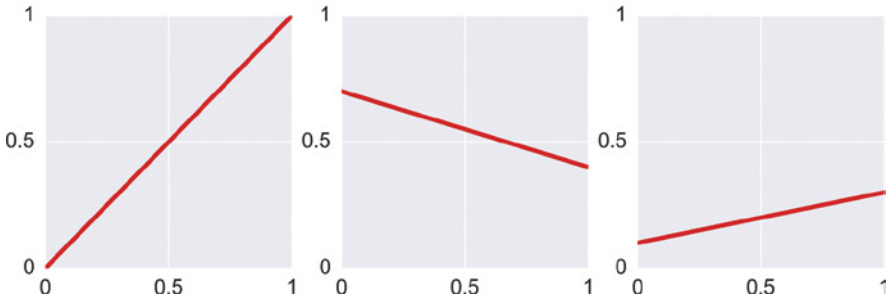


Figure 13-17: Straight-line functions. The leftmost function is called the identity function.

The other curves in Figure 13-17 are also straight lines, but they're tilted to different slopes. We call any curve that's just a single straight line a *linear function*, or even (slightly confusingly) a *linear curve*.

These activation functions do not prevent network collapse. When the activation function is a single straight line, then mathematically, it's only doing multiplication and addition, and that means it's a linear function and the network can collapse. These straight-line activation functions usually appear only in two specific situations.

The first application is on a network's output neurons. There's no risk of collapse since there are no neurons after the output. The top of Figure 13-18 shows the idea.

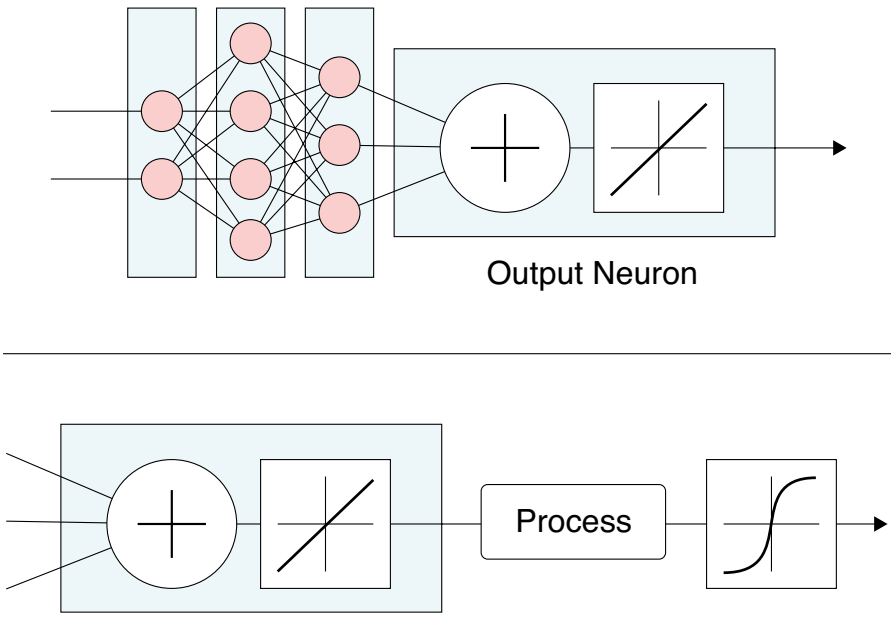


Figure 13-18: Using the identity as an activation function. Top: The identity function on an output neuron. Bottom: Using an identity function to insert a step of processing between the summation step and a nonlinear activation function for any neuron.

The second situation where we use a straight-line activation function is when we want to insert some processing between the summation step in a neuron and its activation function. In this case, we apply the identity function to the neuron, perform the processing step, and then perform the nonlinear activation function, as shown at the bottom of Figure 13-18.

Since we generally want nonlinear activation functions, we need to get away from a single straight line. All of the following activation functions are nonlinear and prevent network collapse.

Step Functions

We don't want a straight line, but we can't pick just any curve. Our curve needs to be single-valued. As we discussed in Chapter 5, this means that if we look upward from any value of x along the X axis, there's only one value of y above us. An easy variation on a linear function is to start with a straight line and break it up into several pieces. They don't even have to join. In the language of Chapter 5, this means that they don't have to be continuous.

Figure 13-19 shows an example of this approach. We call this a *stair-step function*. In this example, it outputs the value 0 if the input is from 0 to just less than 0.2, but then the output is 0.2 if the input value is from 0.2 to just less than 0.4, and so on. These abrupt jumps don't violate our rule that the curve has only one y output value for each input x value.

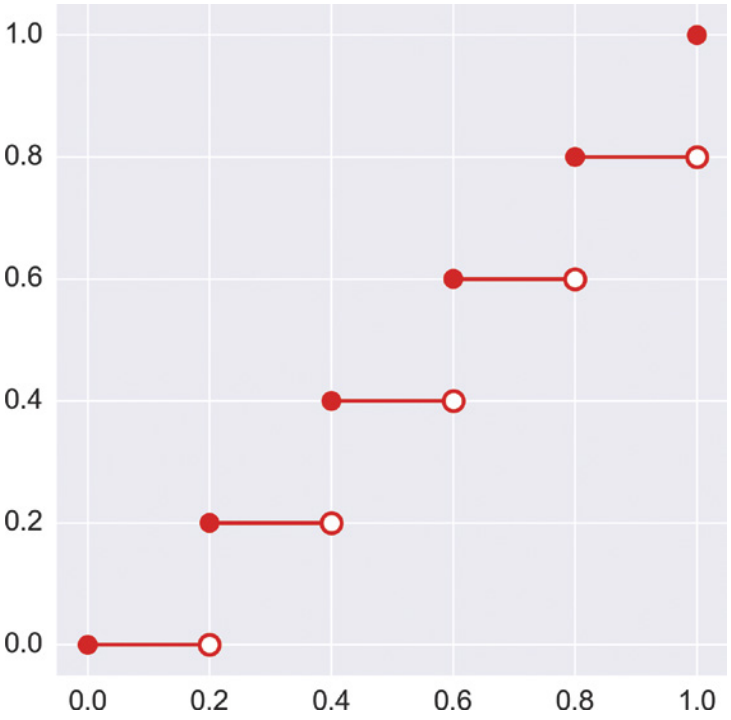


Figure 13-19: This curve is made up of multiple straight lines. A filled circle tells us that the y value there is valid, whereas an open circle tells us that there is no curve at that point.

The simplest stair-step function has only a single step. This is a frequent special case, so it gets its own name: the *step function*. The original perceptron of Figure 13-2 used a step function as its activation function. A step function is usually drawn as in Figure 13-20(a). It has one value until some *threshold* and then it has some other value.

Different people have different preferences for what happens when the input has precisely the value of the threshold. In Figure 13-20(a) we're showing that the value at the threshold is the value of the right side of the step, as shown by the solid dot.

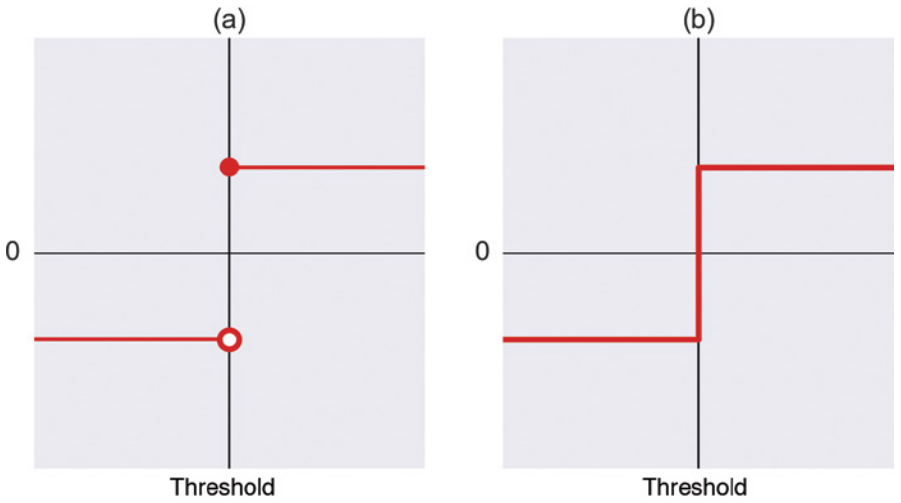


Figure 13-20: A step function has two fixed values, one each to the left and right of a threshold value of x.

Often authors are casual about what happens when the input is exactly at the transition, and draw the picture as in Figure 13-20(b) in order to stress the "step" of the function. This is an ambiguous way to draw the curve because we don't know what value is intended when the input is precisely at the threshold, but it's a common kind of drawing (often we don't care which value is used at the threshold, so we can choose whatever we prefer).

A couple of popular versions of the step have their own names. The *unit step* is 0 to the left of the threshold, and 1 to the right. Figure 13-21 shows this function.

If the threshold value of a unit step is 0, then we give it the more specific name of the *Heaviside step*, also shown in Figure 13-21.

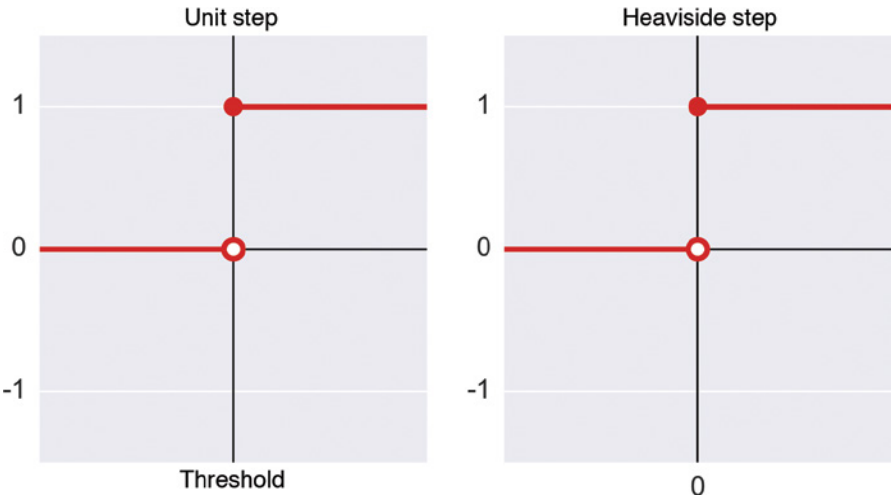


Figure 13-21: Left: The unit step has a value of 0 to the left of the threshold, and 1 to the right. Right: The Heaviside step is a unit step where the threshold is 0.

Finally, if we have a Heaviside step (so the threshold is at 0) but the value to the left is –1 rather than 0, we call this the *sign function*, shown in Figure 13-22. There's a popular variation of the sign function where input values that are exactly 0 are assigned an output value of 0. Both variations are commonly called "the sign function," so when the difference matters, it's worth paying attention to figure out which one is being referred to.

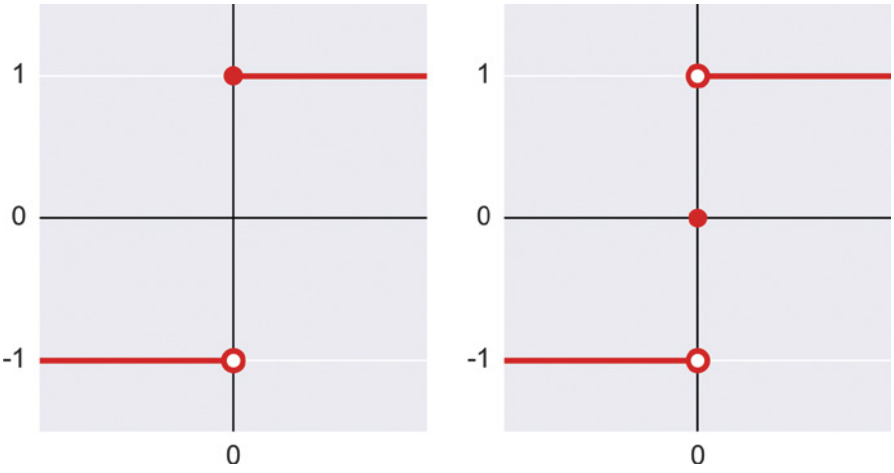


Figure 13-22: Two versions of the sign function. Left: Values less than 0 are assigned an output of –1, all others are 1. Right: Like the left, except that an input of exactly 0 gets the value 0.

Piecewise Linear Functions

If a function is made up of several pieces, each of which is a straight line, we call it *piecewise linear*. This is still a nonlinear function as long as the pieces, taken together, don't form a single straight line.

Perhaps the most popular activation function is a piecewise linear function called a *rectifier*, or *rectified linear unit*, which is abbreviated *ReLU* (note that the e is lowercase). The name comes from an electronics part called a rectifier, which can be used to prevent negative voltages from passing from one part of a circuit to another (Kuphaldt 2017). When the voltage goes negative, the physical rectifier clamps it to 0, and our rectified linear unit does the same thing with the numbers that are fed into it.

The ReLU's graph is shown in Figure 13-23. It's made up of two straight lines, but thanks to the kink, or bend, this is not a linear function. If the input is less than 0, then the output is 0. Otherwise, the output is the same as the input.

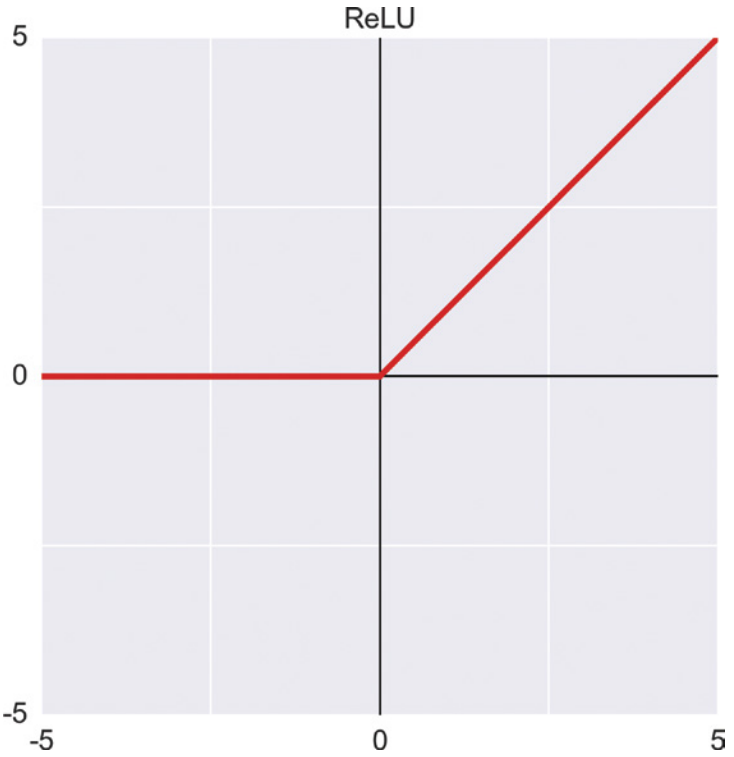


Figure 13-23: The ReLU, or rectified linear unit. It outputs 0 for all negative inputs, otherwise the output is the input.

The ReLU activation function is popular because it's a simple and fast way to include a nonlinearity at the end of our artificial neurons. But there's a potential problem. As we'll see in Chapter 14, if changes in the input don't lead to changes in the output, a network can stop learning. And

the ReLU has an output of 0 for every negative value. If our input changes from, say, –3 to –2, then the output of ReLU stays at 0. Fixing this problem has led to the development of the ReLU variations that follow.

Despite this issue, ReLU (or leaky ReLU, which we'll see next) often performs well in practice, and people often use it as their default choice when building a new network, particularly for fully connected layers. Beyond the fact that these activation functions work well in practice, there are good mathematical reasons for wanting to use ReLU (Limmer and Stanczak 2017), though we won't explore them here.

The *leaky ReLU* changes the response for negative values. Rather than output a 0 for any negative value, this functions outputs the input, scaled down by a factor of 10. Figure 13-24 shows this function.

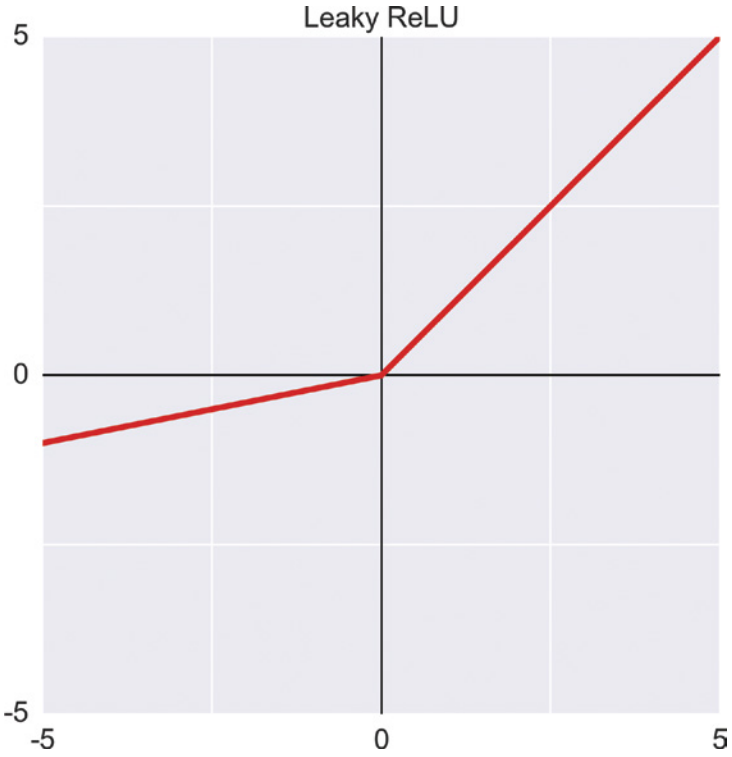


Figure 13-24: The leaky ReLU is like the ReLU, but it returns a scaled-down value of x when x is negative.

Of course, there's no need to always scale down the negative values by a factor of 10. A *parametric ReLU* lets us choose by how much negative amounts are scaled, as shown in Figure 13-25.

When using a parametric ReLU, the essential thing is to never select a factor of exactly 1.0, because then we lose the kink, the function becomes a straight line, and any neuron we apply this to collapses with those that immediately follow it.

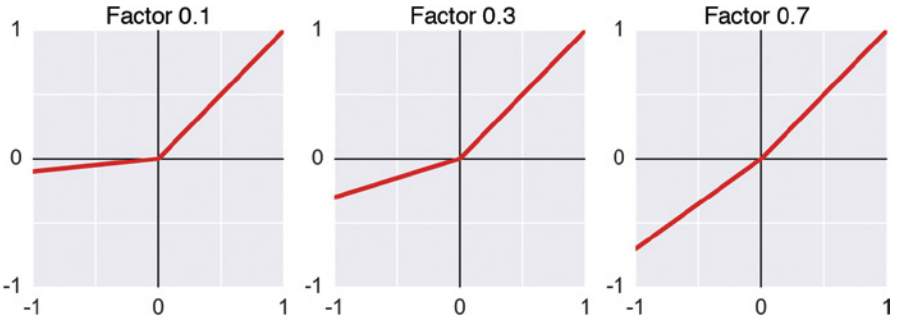


Figure 13-25: A parametric ReLU is like a leaky ReLU, but the slope for values of x that are less than 0 can be specified.

Another variation on the basic ReLU is the *shifted ReLU*, which just moves the bend down and left. Figure 13-26 shows an example.

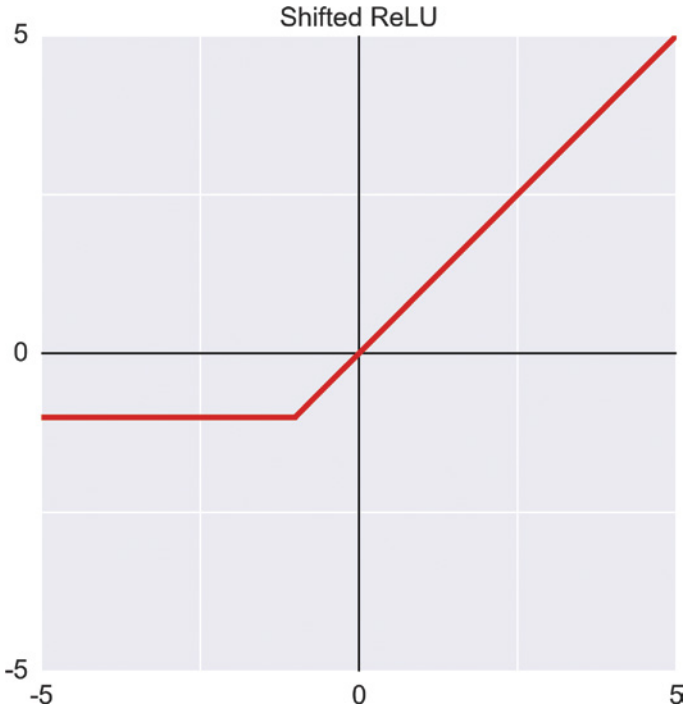


Figure 13-26: The shifted ReLU moves the bend in the ReLU function down and left.

We can generalize the various flavors of ReLU with an activation function called *maxout* (Goodfellow et al. 2013). Maxout allows us to define a set of lines. The output of the function at each point is the largest value

among all the lines, evaluated at that point. Figure 13-27 shows maxout with just two lines, forming a ReLU, as well as two other examples that use more lines to create more complex shapes.

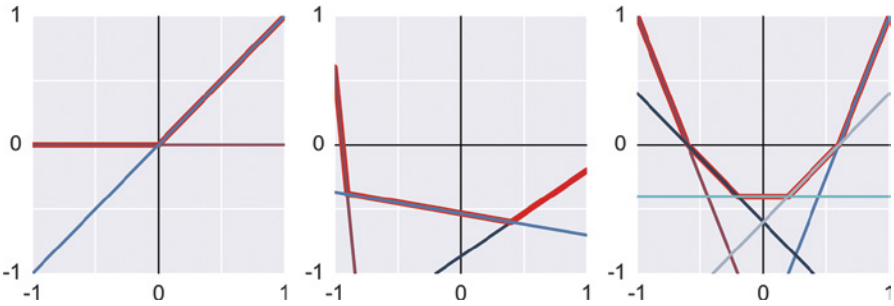


Figure 13-27: The maxout function lets us build up a function from multiple straight lines. The heavy red line is the output of maxout for each set of lines.

Another variation on the basic ReLU is to add a small random value to the input before running it through a standard ReLU. This function is called a *noisy ReLU*.

Smooth Functions

As we'll see in Chapter 14, a key step in teaching neural networks involves computing derivatives for the outputs of neurons, which necessarily involve their activation functions.

The activation functions that we saw in the last section (except for the linear functions) create their nonlinearities by using multiple straight lines with at least one kink in the collection. Mathematically, there is no derivative at the kink between a pair of straight lines, and therefore the function is not linear.

If these kinks prevent the computation of derivatives, which are necessary for teaching a network, why are functions like ReLU useful at all, let alone so popular? It turns out that standard mathematical tools can finesse the sharp corners like those in ReLU and still produce a derivative (Oppenheim and Nawab 1996). These tricks don't work on all functions, but one of the principles that guided the development of the functions we saw earlier is that they allow these methods to be used.

An alternative to using multiple straight lines and then patching up the problems is to use smooth functions that inherently have a derivative everywhere. That is, they're smooth everywhere. Let's look at a few popular and smooth activation functions.

The *softplus* function simply smooths out the ReLU, as shown in Figure 13-28.

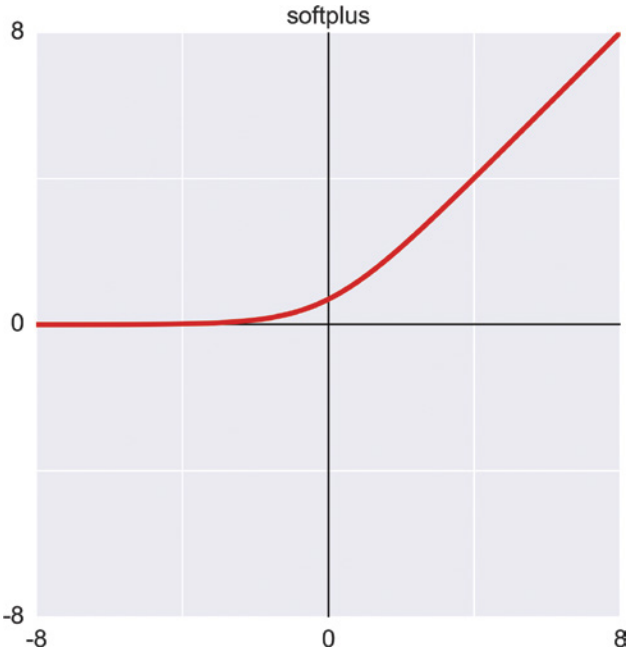


Figure 13-28: The softplus function is a smoothed version of the ReLU.

We can smooth out the shifted ReLU as well. This is called the *exponential ReLU*, or *ELU* (Clevert, Unterthiner, and Hochreiter 2016). It's shown in Figure 13-29.

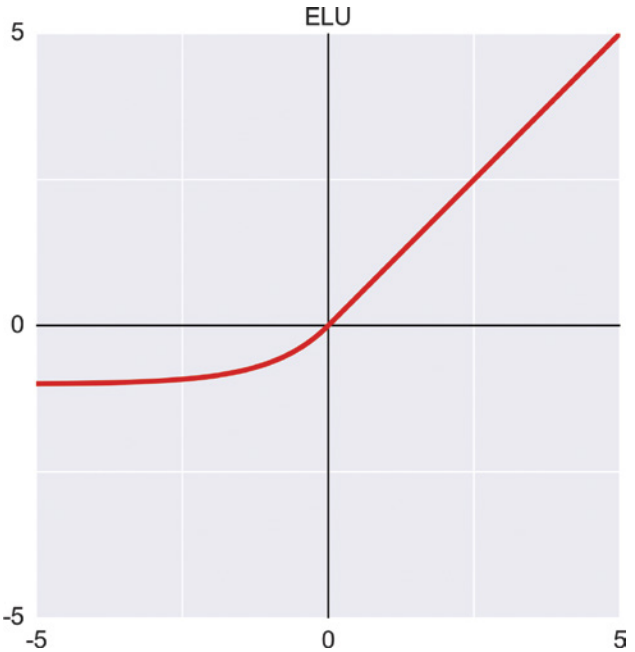


Figure 13-29: The exponential ReLU, or ELU

Another way to smooth out the ReLU is called *swish* (Ramachandran, Zoph, and Le 2017). Figure 13-30 shows what this looks like. In essence it's a ReLU, but with a small, smooth bump just left of 0, which then flattens out.

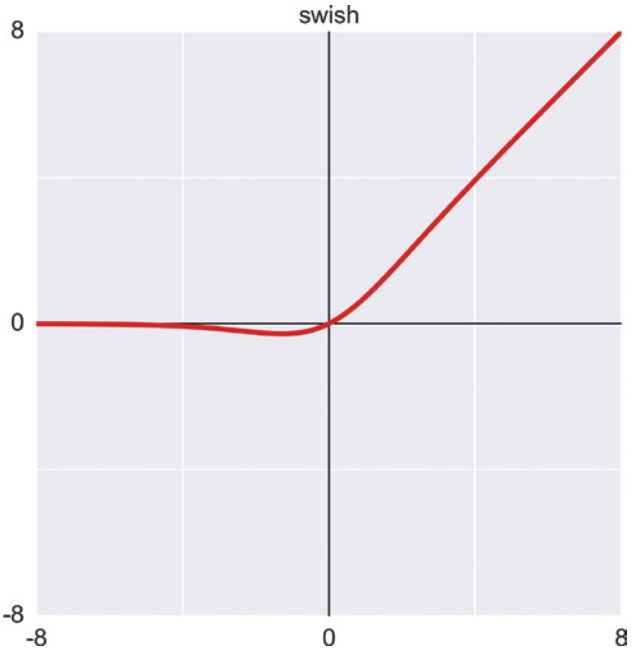


Figure 13-30: The swish activation function

Another popular smooth activation function is the *sigmoid*, also called the *logistic function* or *logistic curve*. This is a smoothed-out version of the Heaviside step. The name *sigmoid* comes from the resemblance of the curve to an S shape, while the other names refer to its mathematical interpretation. Figure 13-31 shows this function.

Closely related to the sigmoid is another mathematical function called the *hyperbolic tangent*. It's much like the sigmoid, only negative values are sent to –1 rather than to 0. The name comes from the curve's origins in trigonometry. It's a big name, so it's usually written simply as *tanh*. This is shown in Figure 13-32.

We say that the sigmoid and tanh functions both *squash* their entire input range from negative to positive infinity into a small range of output values. The sigmoid squashes all inputs to the range [0, 1], while tanh squashes them to [–1, 1].

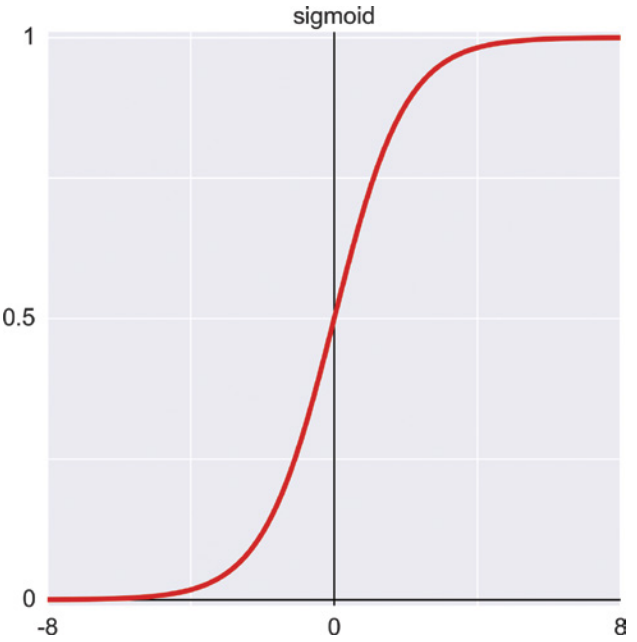


Figure 13-31: The S-shaped sigmoid function is also called the logistic function or logistic curve. It has a value of 0 for very negative inputs, and a value of 1 for very positive inputs. For inputs in the range of about –6 to 6, it smoothly transitions between the two.

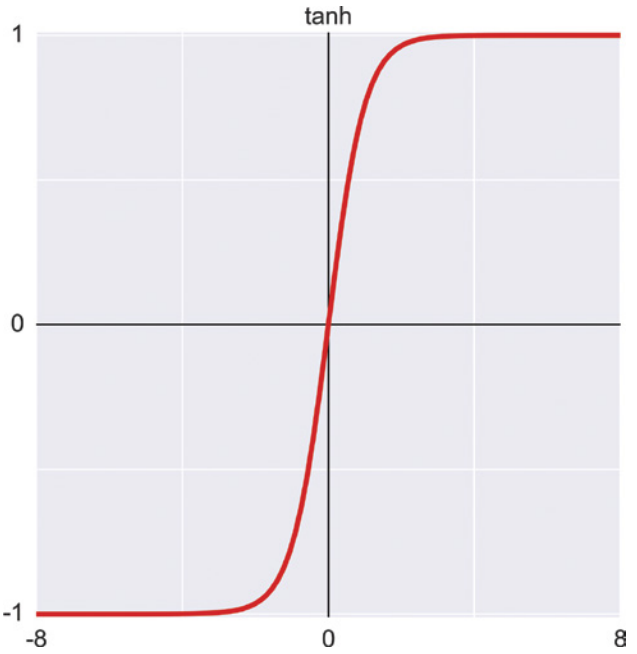


Figure 13-32: The hyperbolic tangent function, written tanh, is S-shaped like the sigmoid of Figure 13-31. The key differences are that it returns a value of –1 for very negative inputs, and the transition zone is a bit narrower.

The two are shown on top of one another in Figure 13-33.

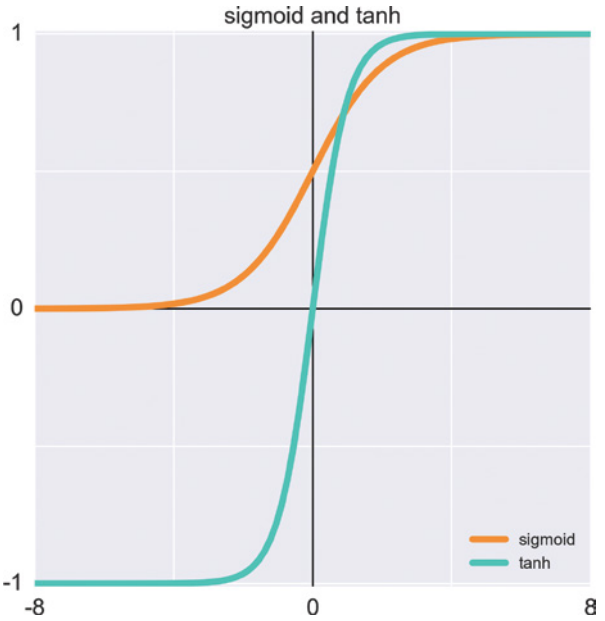


Figure 13-33: The sigmoid function (orange) and tanh function (teal), both plotted for the range –8 to 8

Another smooth activation function uses a sine wave, as shown in Figure 13-34 (Sitzmann 2020). This squashes the outputs to the range [–1, 1] like tanh, but it doesn't saturate (or stop changing) for inputs that are far from 0.

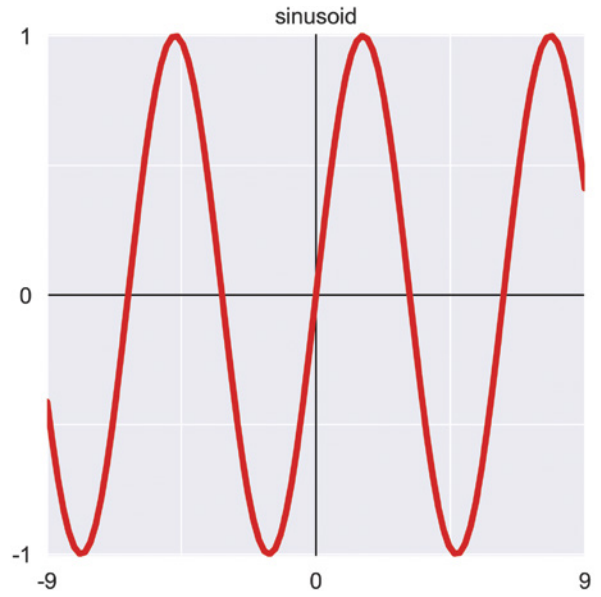


Figure 13-34: A sine wave activation function

Activation Function Gallery

Figure 13-35 summarizes the activation functions we've discussed.

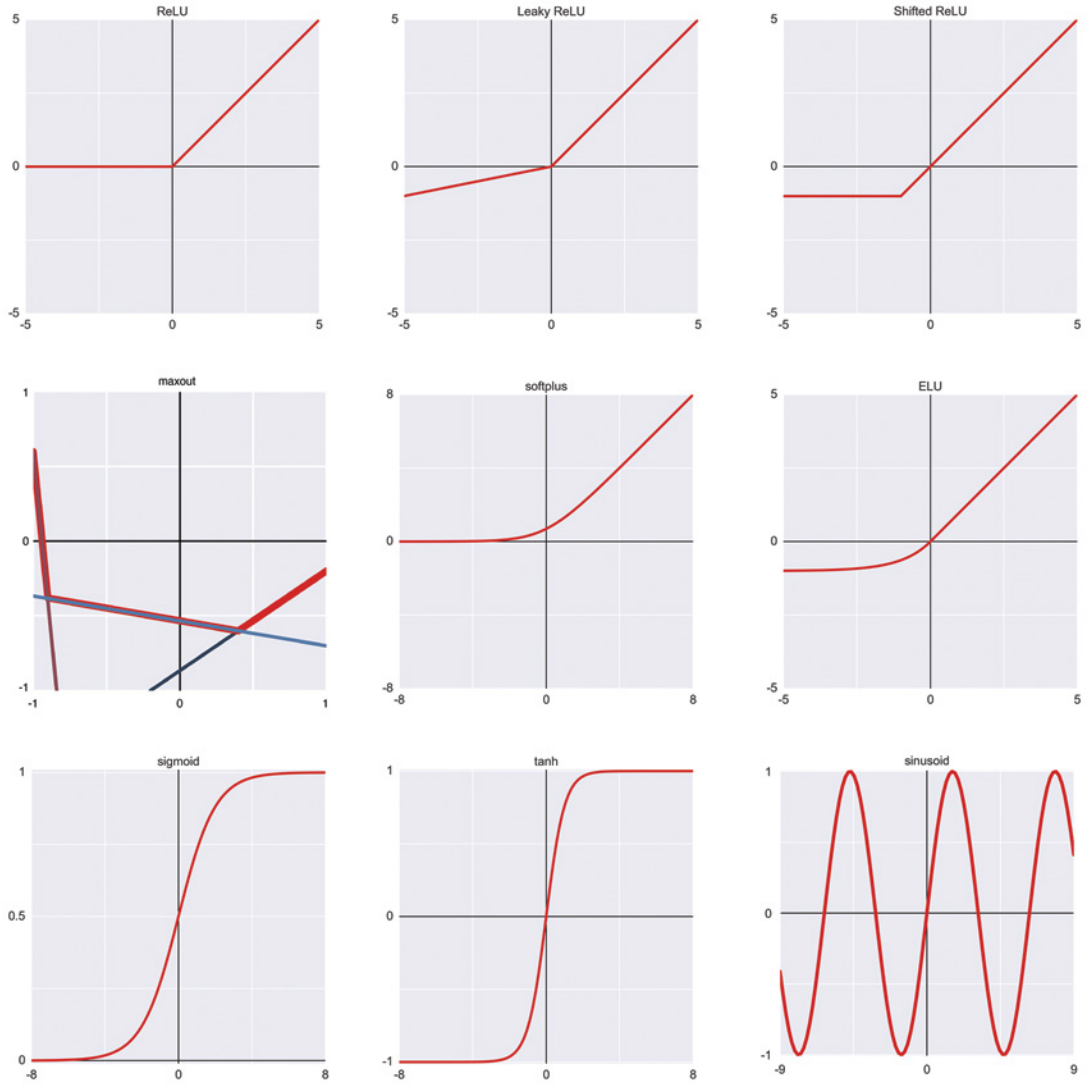


Figure 13-35: A gallery of popular activation functions

Comparing Activation Functions

ReLU used to be the most popular activation function, but in recent years, the leaky ReLU has been gaining in popularity. This is a result of practice: networks with leaky ReLU often learn faster.

The reason is that ReLU has a problem, which we mentioned earlier. When a ReLU's input is negative, its output is 0. If the input is a large negative number, then changing it by a small amount still results in a negative

input to ReLU and an unchanged output of 0. This means that the derivative is also zero. As we'll see in Chapter 14, when a neuron's derivative goes to zero, not only does it stop learning, but it also makes it more likely that the neurons that precede it in the network will stop learning as well. Because a neuron whose output never changes no longer participates in learning, we sometimes use rather drastic language and say that the neuron has *died*. The leaky ReLU has been gaining in popularity over ReLU because, by providing an output that isn't the same for every negative input, its derivative is not 0, and thus it does not die. The sine wave function also has a non-zero derivative almost everywhere (except at the very top and bottom of each wave).

After ReLU and leaky ReLU, sigmoid and tanh are probably the next most popular functions. Their appeal is that they're smooth, and the outputs are bounded to [0, 1] or [–1, 1]. Experience has shown that networks learn most efficiently when all the values flowing through it are in a limited range.

There is no firm theory to tell us which activation function works best in a specific layer of a specific network. We usually start by making the same choices that have worked in other, similar networks that we've seen, and then we try alternatives if learning goes too slowly.

A few rules of thumb give us a good starting point in many situations. Generally speaking, we often apply ReLU or leaky ReLU to most neurons on hidden layers, particularly fully connected layers. For regression networks, we often use no activation function on the final layer (or if we must supply one, we use a linear activation function, which amounts to the same thing), because we care about the specific output value. When we're classifying with just two classes, we have just a single output value. Here we often apply a sigmoid to push the output clearly to one class or the other. For classification networks with more than two classes, we almost always use a somewhat different kind of activation function, which we'll look at next.

Softmax

There's an operation that we typically apply only to the output neurons of a classifier neural network, and even then, only if there are two or more output neurons. It's not an activation function in the sense that we've been using the term because it takes as input the outputs of *all* the output neurons simultaneously. It processes them together and then produces a new output value for each neuron. Though it's not quite an activation function, it's close enough in spirit to activation functions to merit including it in this discussion.

The technique is called *softmax*. The purpose of softmax is to turn the raw numbers that come out of a classification network into class probabilities.

It's important to note that softmax takes the place of any activation function we'd otherwise apply to those output neurons. That is, we give them no activation function (or, equivalently, apply the linear function) and then run those outputs into softmax.

The mechanics of this process are involved with the mathematics of how the network computes its predictions, so we won't go into those details here. The general idea is shown in Figure 13-36: *scores* come in, and *probabilities* come out.

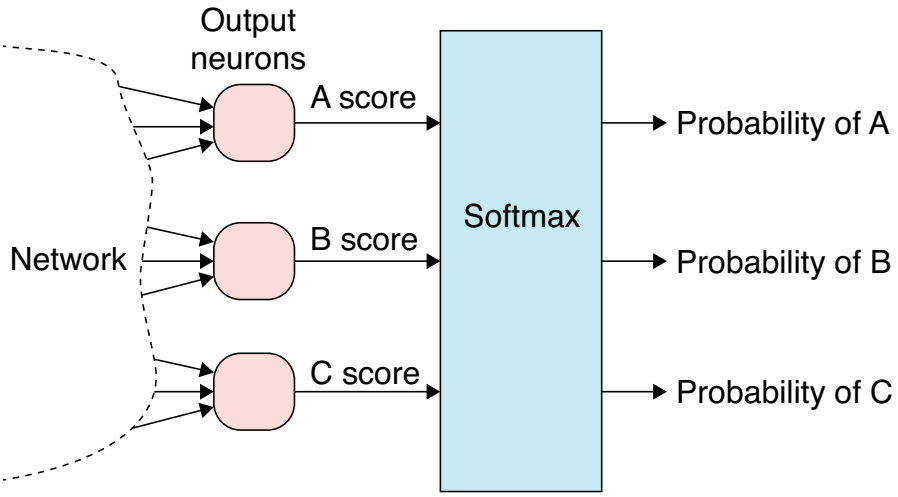


Figure 13-36: The softmax function takes all the network's outputs and modifies them simultaneously. The result is that the scores are turned into probabilities.

Each output neuron presents a value, or score, that corresponds to how much the network thinks the input is of that class. In Figure 13-36 we're assuming that we have three classes in our data, named A, B, and C, so each of the three output neurons gives us a score for its class. The larger the score, the more certain the system is that the input belongs to that class.

If one class has a larger score than some other class, it means the network thinks that class is more likely. That's useful. But the scores aren't designed to be compared in any other convenient way. For instance, if the score for A is twice that of B, it doesn't mean that A is twice as likely as B. It just means that A is more likely. Because making comparisons like "twice as likely" is so useful, we use softmax to turn the output scores into probabilities. Now, if the softmax output of A is twice that of B, then indeed A is twice as probable as B. That's such a useful way to look at the network's output that we almost always use softmax at the end of a classification network.

Any set of numbers that we want to treat as probabilities must satisfy two criteria: the values all lie between 0 and 1, and they add up to 1. If we just modify each output of the network independently, we don't know the other values, so we can't make sure they added up to anything in particular. When we hand all the outputs to softmax, it can simultaneously adjust all the values so that they sum to 1.

Let's look at softmax in action. Consider the top-left graph of Figure 13-37.

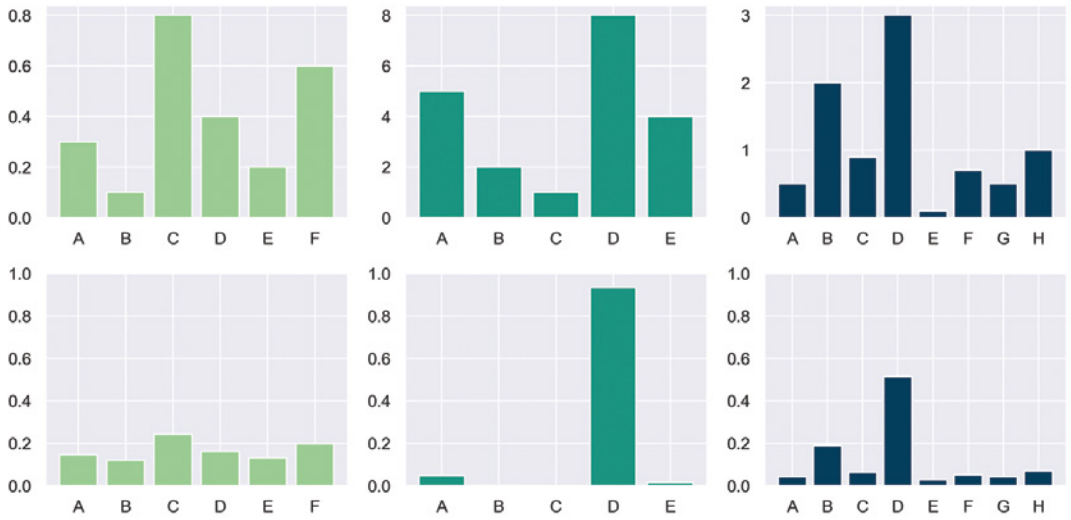


Figure 13-37: The softmax function takes all the network's outputs and modifies them simultaneously. The result is that the scores are turned into probabilities. Top row: Scores from a classifier. Bottom row: Results of running the scores in the top row through softmax. Note that the graphs in the upper row use different vertical scales.

The top left of Figure 13-37 shows the outputs for a classifier with six output neurons, which we've labeled A through F. In this example, all six of these values are between 0 and 1. From this graph, we can see that the value for class B is 0.1 and the value for class C is 0.8. As we've discussed, it is a mistake to conclude from this that the input is 8 times more likely to be in class C than class B, because these are scores and not probabilities. We can say that class C is more likely than class B, but anything more requires some math. To usefully compare these outputs to one another, we can apply softmax to carry out that math, and change them into probabilities.

We show the output of softmax in the graph in the lower left. These are the probabilities of the input belonging to each of the six classes. It's interesting to note that the big values, like C and F, get scaled down by a lot, but the small values, like B, are hardly scaled at all. This is a natural result of how scores between 0 and 1 turn into probabilities. But the ordering of the bars by size is still the same as it was for the scores (with C the largest, then F, then D, and so on). From the probabilities produced by softmax in the lower figure, we can see that class C has a probability of about 0.25, and class B has a probability of about 0.15. We can conclude that the input is a little more than 1.5 times more probable to be in class C than class B.

The middle and right columns of Figure 13-37 show the outputs for two other hypothetical networks and inputs, before and after softmax. The three examples show that the output of softmax depends on whether the inputs are all less than 1. The input ranges in Figure 13-37, reading left to right, are [0, 0.8], [0, 8], and [0, 3]. Softmax always preserves the

ordering of its inputs (that is, if we sort the inputs from largest to smallest, they match a similar sort on the outputs). But when some input values are greater than 1, the largest value tends to stand out more. We say that softmax *exaggerates* the influence of the output with the largest value. Sometimes we also say that softmax *crushes* the other values, making the largest one dominate the others more obviously.

Figure 13-37 shows that the input range makes a big difference in the output of softmax. Softmax also has an interesting behavior depending on whether the inputs values are all less than 1, all greater than 1, or mixed.

At the far left of Figure 13-37 all of the inputs are all less than 1, in the range [0, 0.8].

In the middle column, the inputs are all greater than 1, in the range [0, 8]. Notice that in the output, the value of D (corresponding to the 8) clearly dominates all of the other values. Softmax has exaggerated the differences among the outputs, making it easier to pick out D as the largest.

On the far right of Figure 13-37 we have values both less and greater than 1, in the range [0, 3]. Here the exaggeration effect is somewhere between the left column, where all inputs are less than 1, and the middle column, where all inputs are greater than 1.

In all cases, though, softmax gives us back probabilities that are each between 0 and 1, and sum up to 1. The ordering of the inputs is always preserved, so the sequence of largest to smallest input is also the largest to smallest output.

Summary

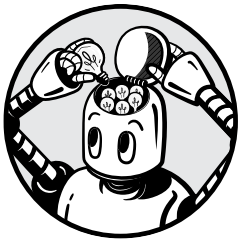
Real, biological neurons are sophisticated nerve cells that process information using a fabulously complex range of chemical, electrical, and mechanical processes. They serve as the inspiration for a simple bit of computation that we call an artificial neuron, despite the enormous gulf between the computer version and its biological namesake. An artificial neuron multiplies each input value by a corresponding weight, adds the results, then passes that through an activation function. We can assemble artificial neurons into networks. Typically, those networks are DAGs: they are directed (information flows in only one direction), they are acyclic (no neuron ever receives its own output as an input), and they are graphs (the neurons are connected to one another). Input data enters at one end, and the network's results appear at the other.

We saw how, if we're not careful in constructing our network, the entire network can collapse into a single neuron. We prevent this by using the activation function, a small function that takes each neuron's output and turns it into a new number. These functions are designed to be nonlinear, meaning that they cannot be described merely by operations such as addition and multiplication. It is this nonlinearity that prevents the network from being equivalent to a single neuron. We concluded the chapter by looking at some of the more common activation functions, and how softmax can turn the numbers we get from a neural network into class probabilities.

The only difference between untrained deep learning systems and those that have been trained and are ready for deployment is in the value of the weights. The goal of training, or learning, is to find values for the weights so that the network's output is correct for as many samples as possible. Since the weights start out with random numbers, we need some principled way to find these new, useful values. In Chapters 14 and 15, we'll see how neural networks learn by looking at the two key algorithms that gradually improve the starting weights, transforming a network's outputs into accurate, useful results.

14

BACKPROPAGATION



As we've seen, a neural network is just a collection of neurons, each doing its own little calculation and then passing on its results to other neurons. How can we train such a thing to produce the results we want? And how can we do it efficiently?

The answer is *backpropagation*, or simply *backprop*. Without backprop, we wouldn't have today's widespread use of deep learning because we wouldn't be able to train big networks in reasonable amounts of time. Every modern deep learning library provides a stable and efficient implementation of backprop. Even though most people will never implement backprop, it's important to understand the algorithm because so much of deep learning depends on it.

Most introductions to backprop are presented mathematically, as a collection of equations with associated discussion (Fullér 2010). As usual, we skip the mathematics here and focus instead on the concepts. The middle of this chapter, where we discuss the core of backprop, is the most detailed part of this book. You might want to read it lightly the first time to get the big

picture for what's going on and how the pieces fit together. Then, if you like, you can come back and take it more slowly, following the individual steps.

A High-Level Overview of Training

Networks learn by minimizing their mistakes. The process begins with a number called a *cost*, *loss*, or *penalty*, for each mistake. During training, the network reduces the cost, resulting in outputs closer to what we want.

Punishing Error

Suppose we have a classifier that identifies each input as one of five classes, numbered 1 to 5. The class that has the largest value is the network's prediction for each input's class. Our classifier is brand-new and has had no training, so all of the weights have small random values. Figure 14-1 shows the network classifying its first input sample.

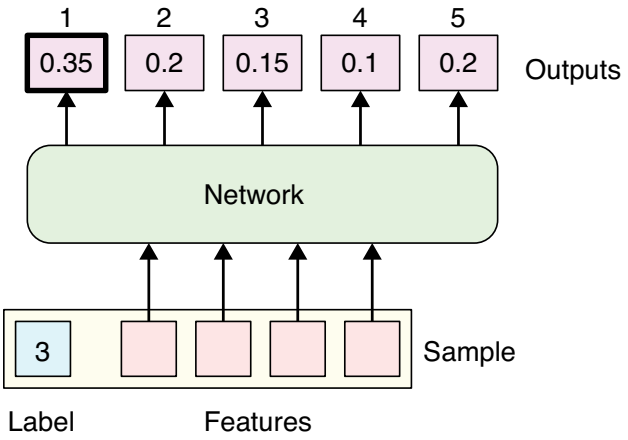


Figure 14-1: A neural network processing a sample and assigning it to class 1. We want it to be assigned to class 3.

In this example, the network has decided that the sample belongs to class 1 because the largest output, 0.35, is from output 1 (we're assuming we have a softmax layer at the end of the network, so the outputs add up to 1). Unfortunately, the sample was labeled as belonging to class 3. We shouldn't have expected the right answer. The network can easily have thousands, or even millions, of weights, and they currently all have their initial, random values. Therefore, the outputs are just random values as well. Even if the network had predicted class 3 for this sample, it would have been pure luck.

When a prediction doesn't match that sample's label, we can come up with a single number to tell us just how wrong this answer is. For example, if class 3 were to get almost the largest score, we say the network would be more correct (or less wrong) relative to assigning class 3 the smallest score. We call this number describing the mismatch between the label and the prediction the *error score*, or *error*, or sometimes the *penalty*, or *loss* (if the word

loss seems like a strange synonym for "error," it may help to think of it as describing how much information is "lost" because we have a wrong answer).

The error (or loss) is a floating-point number that can take on any value, though often we set things up so that it's always positive. The larger the error, the more "wrong" our network's prediction is for the label of this input. An error of zero means that the network predicted the sample's label correctly. In a perfect world, we'd get the error down to zero for every sample in the training set. In practice, we usually settle for getting as close as we can.

Although in this chapter we focus on reducing the error on specific samples (or groups of samples), our overall goal is to minimize the total error for the entire training set, which is usually just the sum of the individual errors.

The way we choose to determine the error gives us tremendous flexibility in guiding the network's learning process. The thinking can seem a little backward, however, because the error tells the network what *not* to do. It's like the apocryphal quote about sculpting: to carve an elephant, you simply start with a block of stone and chip away everything that doesn't look like an elephant (Quote Investigator 2020).

In our case, we start with an initialized network and then use the error term to chip away all the behavior we don't want. In other words, we don't really teach the network to find the correct answers. Instead, we penalize incorrect answers by assigning them a positive amount of error. The only way the network can reduce the overall error is to avoid incorrect answers, so that's what it learns to do. This is a powerful idea: to get the behavior we want, we penalize the behavior we don't want.

If we want to penalize several things at once, we compute a *value*, or *term*, for each one and add them up to get the total error. For instance, we might want our classifier to predict the correct class *and* assign it a score that is at least twice as large as the score for the next-closest class. We can compute numbers representing both desires and use their sum as our error term. The only way for the network to drive the error down to zero (or as close to zero as it can get) is to change its weights to achieve both goals.

A popular error term comes from the observation that learning is often the most efficient when the weights in the network are all in a small range, such as [–1, 1]. To enforce this, we can include an error term that has a large value when the weights get too far from this range. This is called *regularization*. In order to minimize the error, the network learns to keep the weights small.

All of this raises the natural question of how on earth the network is able to accomplish the goal of minimizing the error. That's the point of this chapter.

To keep things simple, we'll use an error measure with just one term that punishes a mismatch between the network's prediction and the label. Everything we see in the rest of this chapter works identically when there are more terms in the error. Our first algorithm for teaching the network is just a thought experiment since it would be absurdly slow on today's computers. But the ideas resulting from this experiment form the conceptual basis for the more efficient techniques we discuss later in this chapter.

A Slow Way to Learn

Let's stick with our running example of a classifier trained with supervised learning. We'll give the network a sample and compare the system's prediction with the sample's label. If the network gets it right and predicts the correct label, we won't change anything and we'll move on to the next sample (as the proverb goes, "If it ain't broke, don't fix it" [Seung 2005]). But if the result for a particular sample is incorrect, we'll try to improve things.

Let's make this improvement in a simple way. We'll pick one weight at random from the whole network and *freeze* all the other values so they can't change. We already know the error associated with the weight's current value, so we create a small random value centered around zero, which we'll call m, add that to that weight, and reevaluate the same sample again. This change to one weight causes a ripple effect through the rest of the network, as every neuron that depends on a computation involving that neuron's output also changes. The result is a new set of predictions, and thus a new error for that sample.

If the new error is less than the previous error, then we've made things better and we keep this change. If the results didn't get better, then we need to undo the change. Now we pick another weight at random, modify it by another random amount, reevaluate the network to see if we want to keep that change, pick another weight, modify it, and so on, again and again.

We can continue nudging weights until the results improve by a certain amount, we decide we've tried enough times, or we decide to stop for any other reason. At this point, we select the next sample and tune lots of weights again. When we've used all the samples in our training set, we just go through them all again (maybe in a different order), over and over. The idea is that each little improvement brings us closer to a network that accurately predicts the label for every sample.

With this technique, we expect the network to slowly improve, though there may be setbacks along the way. For example, later samples may cause changes that ruin the improvements we just made for earlier samples.

Given enough time and resources, we expect that the network will eventually improve to the point where it's predicting every sample as well as it can. The important word in that last sentence is *eventually*. As in, "The water will boil, eventually," or "The Andromeda galaxy will collide with our Milky Way galaxy, eventually" (NASA 2012). Although the concepts are right, this technique is definitely not practical. Modern networks can have millions of weights. Trying to find the best values for all those weights with this algorithm is just not realistic.

Our goal in the rest of this chapter is to take this rough idea and restructure it into a vastly more practical algorithm.

Before we move on, it's worth noting that because we're focusing on weights, we're automatically adjusting the influence of each neuron's bias, thanks to the bias trick we saw in Chapter 13. That means we don't have to think about the bias terms, which makes everything simpler.

Let's now consider how we might improve our incredibly slow weight-changing algorithm.

Gradient Descent

The algorithm of the last section improved our network, but at a glacial pace. One big source of inefficiency was that half of our adjustments to the weights were in the wrong direction: we added a value when we should have subtracted it, and vice versa. That's why we had to undo our changes when the error went up. Another problem is that we tuned each weight one by one, which required us to evaluate an immense number of samples. Let's solve these problems.

We can double our training speed if we know beforehand whether we want to nudge each weight in a positive or negative direction. We can get exactly that information from the gradient of the error with respect to that weight. Recall that we met the gradient in Chapter 5, where it told us how the height of a surface changes as each of its parameters changes. Let's narrow that down for the present case.

As before, we're going to freeze the whole network except for one weight. If we plot the value of that weight on a horizontal axis, we can plot the network's error for that weight vertically. The errors, taken together, form a curve called the *error curve*. In this situation, we can find the gradient (or derivative) of the error at any particular value of the weight by finding the slope of the error curve above that weight.

If the gradient directly above the weight is positive (that is, the line goes up as we move to the right), then increasing the value of the weight (moving it to the right) causes the error to go up. Similarly, and more usefully for us, decreasing the value of the weight (moving it to the left) causes the error to go down. If the slope of the error is negative, the situations are reversed.

Figure 14-2 shows two examples.

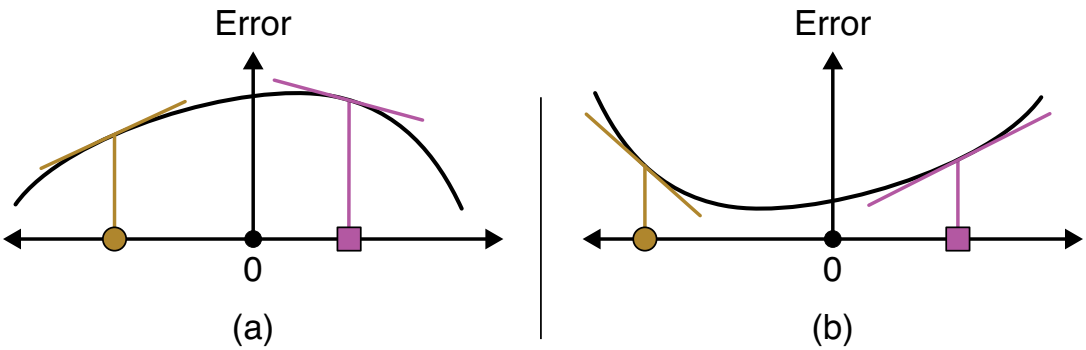


Figure 14-2: The gradient tells us what happens to the error (the black curves) if we make a weight smaller or larger, for two different error curves. Each figure shows the gradient at two weights.

The error curve for every weight in the network is different because every weight has a different effect on the final error. But if we can find the gradient for a specific weight, we've solved the problem of guessing whether it needs to increase or decrease in order to reduce the error. If we can find the gradients for all the weights, we can adjust them all at once, rather than one by one. If we can adjust every weight simultaneously, using its own

specific gradient to tell us whether to make it bigger or smaller, we have an efficient way to improve our network.

This is just what we do. Because we use the gradient to move each weight to produce a lower value on the error curve, we call the algorithm *gradient descent*.

Before we dig into gradient descent, notice that this algorithm makes the assumption that tweaking all the weights independently and simultaneously after we evaluate an incorrect sample leads to a reduction in the error, not just for that sample, but for the entire training set, and by extension, all data that we see after the network is released. This is a bold assumption because we've already noted how changing one weight can cause ripple effects through the rest of the network. Changing the output of one neuron changes the inputs, and thus the outputs, of all neurons that use that value, which in turn changes their gradients. If we're unlucky, some weights that had a positive gradient might now have a negative gradient, or vice versa. That means if we stick with the gradients we computed, changing those weights makes the error bigger, not smaller. To control this problem, we usually make small changes to every weight in the hopes that any such mistakes won't drown out our improvements.

Getting Started

Let's reduce the overall error by adjusting the network's weights in two steps. In the first step, called *backpropagation* or *backprop*, we visit each neuron where we calculate and store a number that is related to the network's error. Once we have this value for every neuron, we use it to update every weight coming into that neuron. This second step is called the *update step*, or the *optimization step*. It's not typically considered part of backpropagation, but sometimes people casually roll the two steps together and call the whole thing backpropagation. This chapter focuses on just the first step. Chapter 15 focuses on optimization.

In this discussion, we're going to ignore activation functions. Their nonlinear nature is essential to making neural networks work, but that same nature introduces a lot of detail that isn't relevant to understanding the essence of backprop. Despite this simplification for the point of a clearer discussion, activation functions are definitely accounted for in any implementation of backprop.

With this simplification in place, we can make an important observation: When any neuron output in our network changes, the final output error changes by a proportional amount.

Let's unpack that. We only care about two types of values in a neural network: weights (which we can set and change as we please), and neuron outputs (which are computed automatically and are beyond our direct control). Except for the very first layer, a neuron's input values are each the output of a previous neuron times the weight of the edge that output travels on. Each neuron's output is just the sum of all of these weighted inputs. Without an activation function, every change in a neuron's output

is proportional to the changes in its inputs, or the weights on those inputs. If the inputs themselves are constant, the only way for a neuron's output to change (and thus affect the final error) is if the weights on its inputs change.

Imagine we're looking at a neuron whose output has just changed. What happens to the network's error as a result? Without activation functions, the only operations in our network are multiplication and addition. If we write down the math (which we won't do), it turns out that the change in the final error is always proportional to the change in the neuron's output.

The connection between any change in the neuron's output and the resulting change in the final error is just the neuron's change multiplied by some number. This number goes by various names, but the most popular is the lowercase Greek letter δ (delta), though sometimes the uppercase version, Δ, is used. Mathematicians often use the delta character to mean "change" of some sort, so this was a natural (if terse) choice of name.

Every neuron has a delta, or δ, associated with it as a result of evaluating the current network with the current sample. This is a real number that can be big or small, positive or negative. Assuming the network's input doesn't change and the rest of the network is frozen, if a neuron's output changes by a particular amount, we can multiply that change by the neuron's delta to see how the entire network's output will change.

To illustrate the idea, let's focus just on one neuron's output for a moment. Let's add some arbitrary number to its output just before that value emerges. Figure 14-3 shows the idea graphically, where we use the letter m (for "modification") for this extra value.

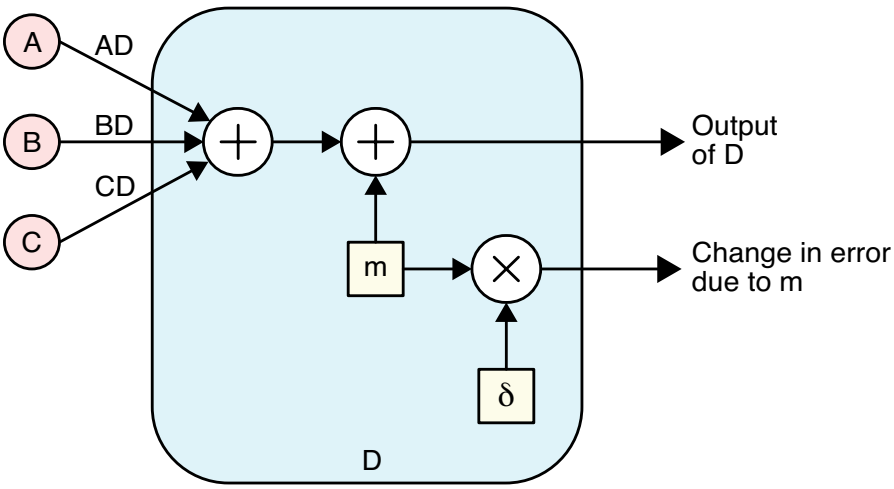


Figure 14-3: Computing the change in the network's final error due to a change in a neuron's output

Because the output will change by m, we know the change in the final error is m times the neuron's δ.

In Figure 14-3, we changed the output directly by placing the value m inside the neuron. Alternatively, we can cause a change in the output by

changing one of the inputs. Let's change the value that's coming in from any other neuron. The same logic holds as for Figure 14-3 and is shown in Figure 14-4. We can instead add m to the value coming in from neurons A or C if we prefer; all that matters is that the output of D changes by m. Since we're still just changing the output by m, we find the change in the final error by multiplying it by the same value of δ as in Figure 14-3.

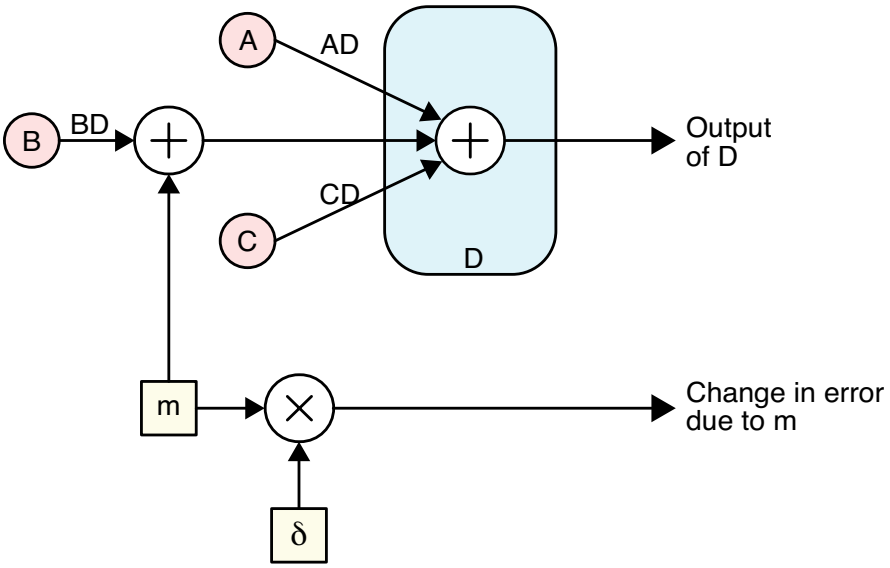


Figure 14-4: A variation of Figure 14-3, where we add m *to the output of B (after it has been multiplied by the weight BD)*

Figure 14-3 and Figure 14-4 illustrate that the change in the network's final output can be predicted from either a change to any neuron's output or any weight in the network.

We can use the delta associated with each neuron to tell us whether each of its incoming weights should be nudged in the positive or negative direction.

Let's walk through an example.

This is where things start to get detailed. The basic idea is that the error will give us a gradient for every weight, and then we can use this gradient to adjust each weight by a little bit so that the overall error decreases. The mechanics for this aren't super complicated, but there are some new ideas, some new names, and a bunch of details to keep straight. If it feels like too much to take in, you might want to skim this part of the chapter on first reading (up to, say, the section "Backprop on a Larger Network"), and return here later for a more complete understanding of the process.

Backprop on a Tiny Neural Network

To get a handle on backprop, we'll use a tiny network that classifies 2D points into two categories, which we'll call class 1 and class 2. If the points

can be separated by a straight line, then we can do this job with just one neuron, but let's use a little network because it lets us see the general principles. Let's begin by looking at the network and giving a label to everything we care about. This will make later discussions simpler and easier to follow. Figure 14-5 shows our little network, along with a name for each of its eight weights. For simplicity, we'll leave out the usual softmax step after neurons C and D.

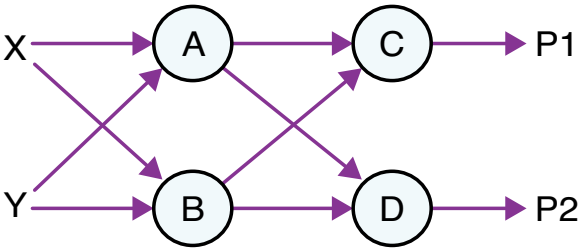


Figure 14-5: A simple neural network with four neurons

Finally, we want to refer to the output and delta for every neuron. For this, let's make little two-letter names by combining the neuron's name with the value we want to refer to. So *Ao* and *Bo* are the names of the outputs of neurons A and B, and *Aδ* and *Bδ* are the delta values for those two neurons.

Figure 14-6 shows these values stored with their neurons.

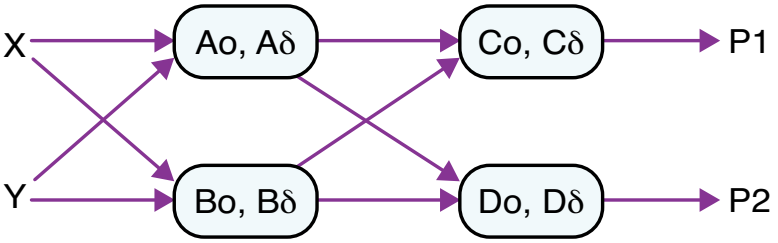


Figure 14-6: Our simple network with the output and delta values for each neuron

We can watch what happens when neuron outputs change, causing changes to the error. Let's label the change in the output of neuron A due to a change by an amount m as Am, the network's final error as E, and the resulting change to the error as Em.

Now we can be more precise about what happens to the error when a neuron's output changes. If we have a change Am in the output of neuron A, then multiplying that change by $A\delta$ gives us the change in the error. That is, the change Em is given by $Am \times A\delta$. We think of the action of $A\delta$ as multiplying, or scaling, the change in the output of neuron A, giving us the corresponding change in the error. Figure 14-7 shows the schematic setup we use in this chapter for visualizing the way changes in a neuron's output are scaled by its delta to produce changes to the error.

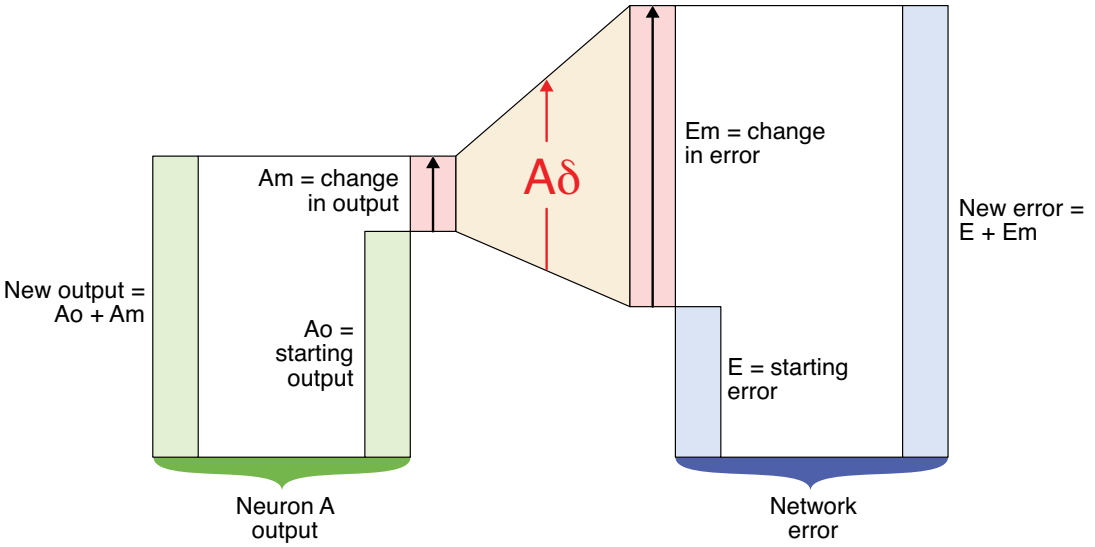


Figure 14-7: Our schematic for visualizing how changes in a neuron's output can change the network's error

At the left of Figure 14-7 we start with neuron A. We see the starting output of A, or *Ao*, a change in output *Am*, and its new output *Ao* + *Am*. The arrow inside the box for *Am* shows that this change is positive. This change is multiplied by *Aδ* to give us *Em*, the change in the error. We show this operation as a wedge, illustrating the amplification of *Am*. Adding *Em* to the previous value of the error, *E*, gives us the new error *E* + *Em*. In this case, both *Am* and *Aδ* are positive, so the change in the error *Am* × *Aδ* is also positive, increasing the error. When either (but not both) of *Am* or *Aδ* is negative, the error decreases.

Now that we've labeled everything, we're finally ready to look at the backpropagation algorithm.

Finding Deltas for the Output Neurons

Backpropagation is all about finding the delta value for each neuron. To do that, we find gradients of the error at the end of the network and then propagate, or move, those gradients backward to the start. So, we begin at the end: the output layer.

Calculating the Network Error

The outputs of neuron C and D in our tiny network give us the probabilities that the input is in class 1 or class 2, respectively. In a perfect world, a sample that belongs to class 1 would produce a value of 1.0 for *P1* and 0.0 for *P2*, meaning that the system is certain that it belongs to class 1 and simultaneously certain that it does not belong to class 2. If the system's a little less certain, we might get *P1* = 0.8 and *P2* = 0.2, telling us that it's much more likely that the sample is in class 1.

We want to come up with a single number to represent the network's error. To do that, we compare the values of *P1* and *P2* with the label for this sample. The easiest way to make that comparison is if the label is one-hot encoded, as we saw in Chapter 10. Recall that one-hot encoding makes a list of zeros as long as the number of classes, except for a 1 in the entry corresponding to the correct class. In our case, we have only two classes, so our labels are (1, 0) for a sample in class 1, and (0, 1) for a sample in class 2. Sometimes this form of label is also called a *target*.

Let's put the predictions *P1* and *P2* into a list as well: (*P1*, *P2*). Now we can just compare the lists. We almost always use cross entropy for this, as discussed in Chapter 6. Figure 14-8 shows the idea.

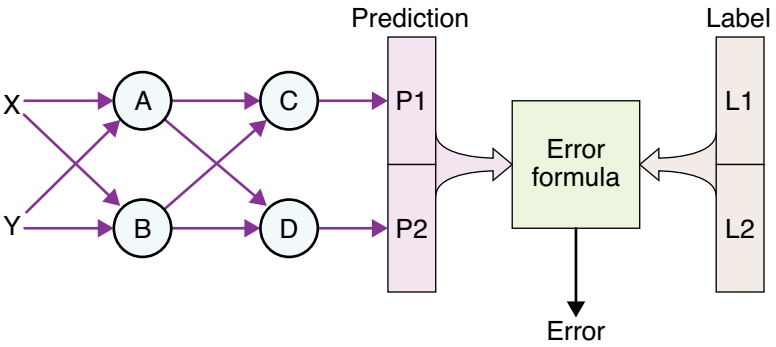


Figure 14-8: Finding the error from a sample

Every deep learning library provides a built-in cross entropy function to help us find the error in a classifier such as this one. In addition to computing the network's error, the function also provides a gradient to tell us how the error will change if we increase any one of its four inputs.

Using the error gradient, we can look at the value coming out of every neuron in the output layer, and determine if we'd like that value to become more positive or more negative. We will later nudge each neuron in the direction that causes the error to decrease.

Drawing Our Error

Let's look at an error curve. We'll also draw the gradient with respect to one particular output or weight in the network. Remember that this is just the slope of the error at that point.

Let's look at how the error varies with changes in the prediction *P1*, shown in Figure 14-9. Suppose *P1* has the value –1.

In Figure 14-9 we've marked the value *P1* = –1 with an orange dot, and we've drawn the derivative at the location on the curve directly above this value of *P1* with a green line. That derivative (or gradient) tells us that if we make *P1* more positive (that is, we move right from –1), the error in the network will decrease.

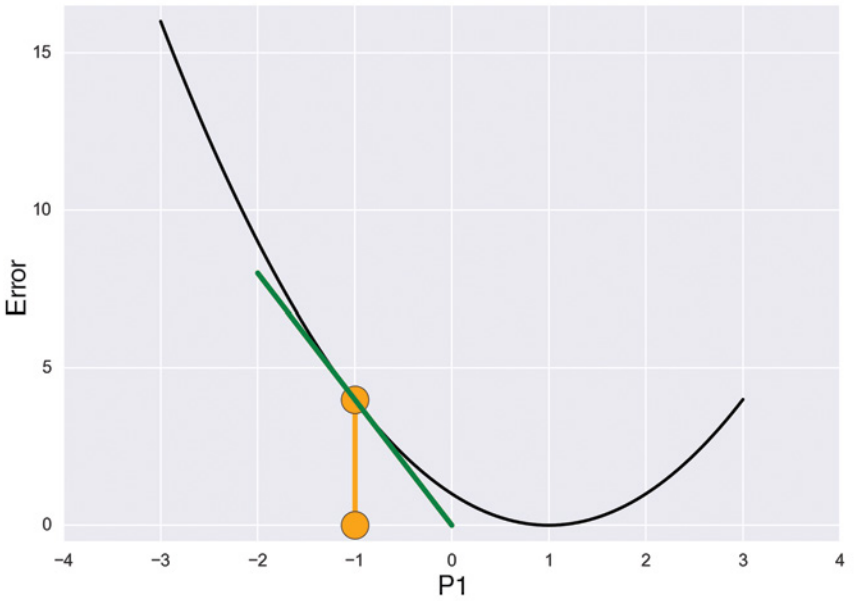


Figure 14-9: How the error depends on different values of P1

If we knew the black curve representing the error, we wouldn't need the gradient, since we'd just find the curve's minimum. Unfortunately, the math doesn't give us the black curve (we're drawing it here just for reference). But the day is saved because the math gives us enough information to find the curve's derivative at any location.

The derivative in Figure 14-9 tells us what happens to the error if we increase or decrease *P1* by a little bit. After we've changed *P1*, we can find the derivative at its new location and repeat. The derivative, or gradient, accurately predicts the new error after each change to *P1* as long as we keep that change small. The bigger the change, the less accurate the prediction is.

We can see this characteristic in Figure 14-9. Suppose we move *P1* by one unit to the right from –1. According to the derivative, we now expect an error of 0. But at *P1* = 0, the error (the value of the black curve) is really about 1. We moved *P1* too far. In the interests of clear figures that are easy to read, we'll sometimes make large moves, but in practice, we change our weights by small amounts.

Let's use the derivative to predict the numerical change in the error due to a change in *P1*. What's the slope of the green line in Figure 14-9? The left end is at about (–2, 8), and the right end is at about (0, 0). Thus, the line descends about four units for every one unit we move to the right, for a slope of –4/1 or –4. If *P1* changed by 0.5 (that is, it changed from –1 to –0.5), we'd predict that the error would go down by $0.5 \times -4 = -2$.

Remember that our goal is to find $C\hat{o}$. We've just done it! *P1* in this discussion is just another name for *Co,* the output of neuron C. We've found that when P1 = –1, a change of 1 in *Co* (or *P1*) would result in a change of

–4 in the error. As we discussed, we shouldn't have too much confidence in this prediction after such a big change in *P1*. But for small moves, the proportion is right. For instance, if we increase *P1* by 0.01, then we expect the error to change by $-4 \times 0.01 = -0.04$, and for such a small change in *P1*, the predicted change in the error should be pretty accurate. If we increase *P1* by 0.02, then we expect the error to change by $-4 \times 0.02 = -0.08$.

The same thinking holds if we decrease the value of *P1*, or move it to the left. If *P1* changes from –1 to, say, –1.1, we expect the error to change by $-0.1 \times -4 = 0.4$, so the error would increase by 0.4.

We've found that for any amount of change in *Co*, we can predict the change in the error by multiplying *Co* by –4. That's exactly what we've been looking for! The value of $C\delta$ is –4. Note that as soon as the value of *P1* changes, for any reason, the error curve changes and the value of $C\delta$ has to be computed all over again.

We've just found our first delta value, which tells us how much the error will change if there's a change to the output of C. It's just the derivative of the error function measured at *P1* (or *Co*). Figure 14-10 shows all of this visually using our error diagram.

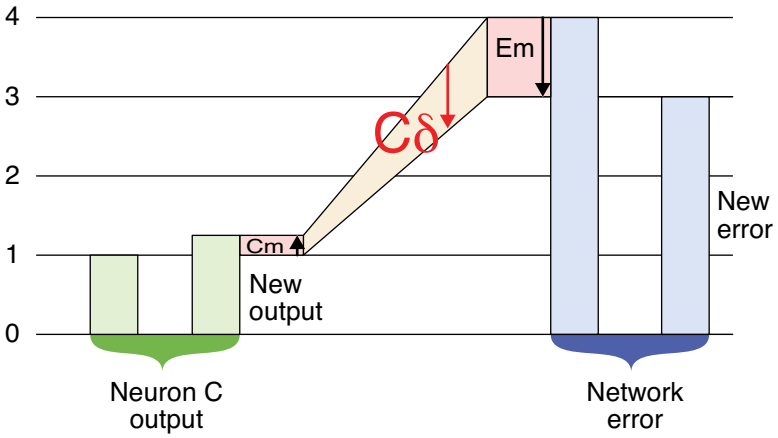


Figure 14-10: Our error diagram illustrating the change in the error from a change in the output of neuron C due to a small increase, Cm

The original output is the green bar at the far left of Figure 14-10. We imagine that due to a change in one of the input weights, the output of C increases by an amount *Cm*. This is amplified by multiplying it by $C\delta$, which gives us the change in the error, *Em*. That is, $Em = Cm \times C\delta$. Here the value of *Cm* is about 1/4 (the upward arrow in the box for *Cm* tells us that the change is positive), and the value of $C\delta$ is –4 (the arrow in that box tells us the value is negative). So $Em = -4 \times 1/4 = -1$. The new error, at the far right, is the previous error plus *Em*, or $4 + (-1) = 3$.

Remember that at this point, we're not yet doing anything with this delta value. Our goal right now is just to find the deltas for our neurons. We'll use them later to change the weights.

Finding Dδ

Let's repeat this whole process for *P2*, to get the value of *Dδ*, or the delta for neuron D.

Let's start with a recap of *Cδ*. On the left of Figure 14-11 we show the error curve for *P1*. As a result of also moving all of the other weights to better values, the error curve for *P1* now has a minimum of around 2.

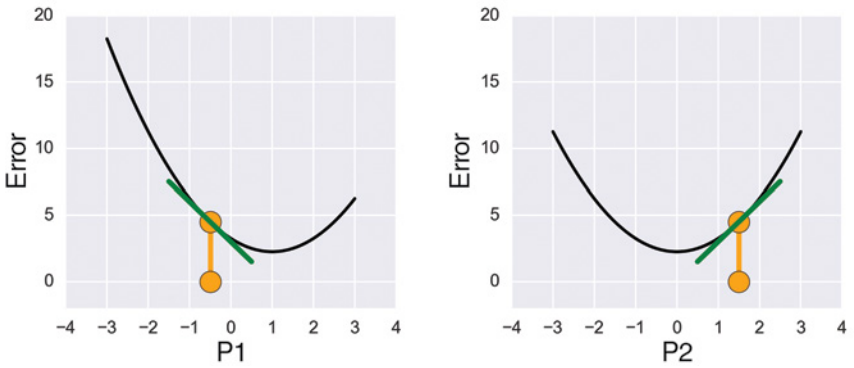


Figure 14-11: Left: The error for different values of P1. *Right: The error for different values of* P2.

If we use the new value and error curve for *P1*, it looks like a change of about 0.5 in *P1* will result in a change of about –1.5 in the error, so *Cδ* is about –1.5 / 0.5 = –3. Instead of changing *P1*, what if we change *P2*?

Take a look at the graph on the right of Figure 14-11. A change of about –0.5 (moving left this time, toward the minimum of the bowl) results in a change of about –1.25 in the error, so *Dδ* is about 1.25 / –0.5 = 2.5. The positive result here tells us that moving *P2* to the right causes the error to go up, so we want to move *P2* to the left.

There are some interesting things to observe here. First, although both curves are bowl shaped, the bottoms of the bowls are at different weight values. Second, because the current values of *P1* and *P2* are on opposite sides of the bottom of their respective bowls, their derivatives have opposite signs (one is positive, the other is negative).

The most important observation is that we cannot currently get the error down to 0. In this example, the curves never get lower than about 2. That's because each curve looks at changing just one value, while the other is fixed. So even if *P1* got to a value of 1, where its curve is a minimum, there would still be error in the result because *P2* is not at its ideal value of 0, and vice versa. This means that if we change just one of these two values, we can't get down to the minimum error of 0. Getting an error of 0 is ideal, but, more generally, our goal is to move each weight, a little bit at a time, until we've pushed the error to as small a value as possible. For some networks, we may never be able to get to 0.

Note that we may not want to get the error to 0, even if we could. As we saw in Chapter 9, when a network is overfitting, its training error continues to decrease, but its ability to handle new data gets worse. We really want to minimize the error as much as possible without overfitting. In casual

discussions, we usually say that we want to get down to zero error, with the understanding that it's better to stop with some error than to keep training and overfit.

We'll see later that we can improve all the weights in the network at the same time, as long as we take very small steps. Then we have to evaluate the errors again to find new curves and then new derivatives and deltas before we can make another adjustment. Rather than take many steps after each sample, we usually adjust the weights only once, and then evaluate another sample, adjust the weights once again, and so on.

Measuring Error

We mentioned earlier that we often compute the error in a classifier using cross entropy. For this discussion, let's use a simpler formula that makes it easy to find the delta for each output neuron. This error measure is called the *quadratic cost function*, or the *mean squared error (MSE)* (Nielsen 2015). As usual, we won't get into the mathematics of this equation. We chose it because it lets us find the delta for an output neuron as the difference between the neuron's value and the corresponding label entry (Seung 2005). Figure 14-12 shows the idea graphically.

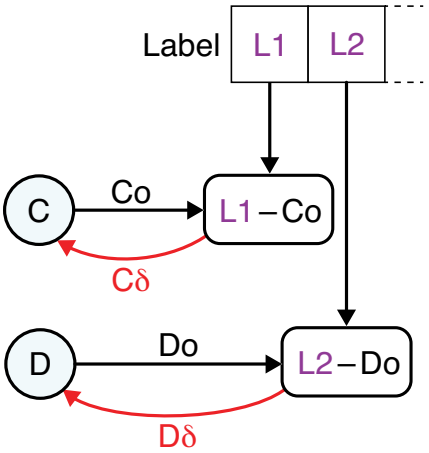


Figure 14-12: When we use the quadratic cost function, the delta for any output neuron is just the value in the label minus the output of that neuron. As shown in red, we save that delta value with its neuron.

Remember that *Co* and *P1* are two names for the same value, as are *Do* and *P2*.

Let's consider *Co* (or *P1*) when the first label is 1. If $Co = 1$, then the value of $C\delta = 1 - Co = 0$, so any change in *Co* gets multiplied by 0, resulting in no change to the output error.

Now suppose that $Co = 2$. Then the difference is $C\delta = 1 - Co = -1$, telling us that a change to *Co* changes the error by the same amount, but with the opposite sign. If *Co* is much larger, say $Co = 5$, then $1 - Co = -4$, which tells us that any change to *Co* is amplified by a factor of −4 in the change to

the error. We've been using large numbers for convenience, but remember that the derivative only accurately predicts what happens if we take a very small step.

The same thought process holds for neuron D, and its output *Do* (or *P2*).

We've now completed the first step in backpropagation: we found the delta values for all the neurons in the output layer. We know from Figure 14-12 that the delta for an output neuron depends on the value in the label and the neuron's output. When we change the values of the weights going into that neuron, its delta changes as well. The delta is a temporary value that changes with every change to the network or its inputs. This is another reason we adjust the weights only once per sample. Since we have to recompute all the deltas after each update, we might as well evaluate a new sample first, and make use of the extra information it provides us.

Remember that our big goal is to find changes for the weights. When we know the deltas for all the neurons in a layer, we can update all the weights feeding into that layer. Let's see how that's done.

Using Deltas to Change Weights

We've seen how to find a delta value for every neuron in the output layer. We know that a change to the neuron's output must come from a change in an input, which in turn can come from either a change in a previous neuron's output or the weight connecting that output to this neuron. Let's look at these cases.

For convenience, let's say that a neuron's output or a weight is changed by a value of 1. Figure 14-13 shows that every change of 1 in the weight AC, which multiplies the output of neuron A before it's received by neuron C, leads to a corresponding change of $Ao \times C\delta$ in the network error in our network. Subtracting that value leads to a change of $-Ao \times C\delta$ in the error. So, if we want to reduce the network's error by subtracting $Ao \times C\delta$ from it, we can change the value of weight AC by –1 to accomplish this.

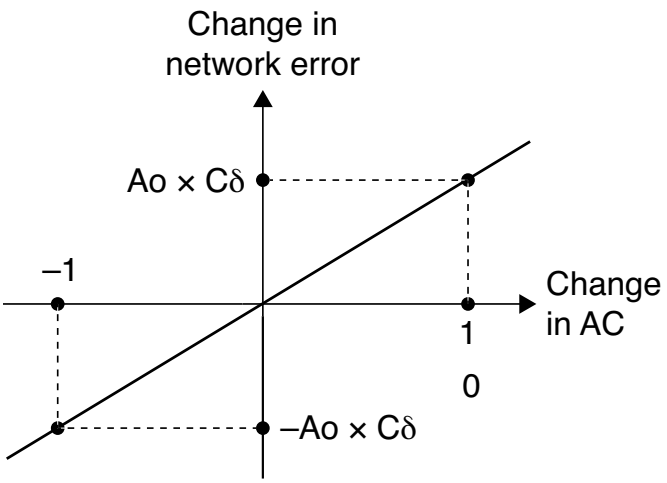


Figure 14-13: When AC *changes by 1, the network error changes by* Ao × Cδ.

We can summarize this process visually with an additional convention for our diagrams. We've been drawing the outputs of neurons as arrows coming out of a circle to the right. Let's draw deltas using arrows coming out of the circles to the left, as in Figure 14-14.

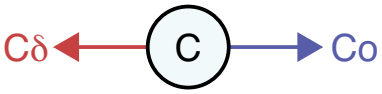


Figure 14-14: Neuron C has an output Co*, drawn with an arrow pointing right, and a delta* Cδ*, drawn with an arrow pointing left.*

With this convention, the whole process for finding the updated value for weight *AC*, or $AC - (Ao \times C\delta)$, is summarized in Figure 14-15. Showing subtraction in a diagram like this is hard, because if we have a "minus" node with two incoming arrows, it's not clear which value is being subtracted from the other (that is, if the inputs are x and y, are we computing $x - y$ or $y - x$?). To sidestep that problem, we compute $AC - (Ao \times C\delta)$ by finding $Ao \times C\delta$, multiplying that by –1, and then adding that result to *AC*.

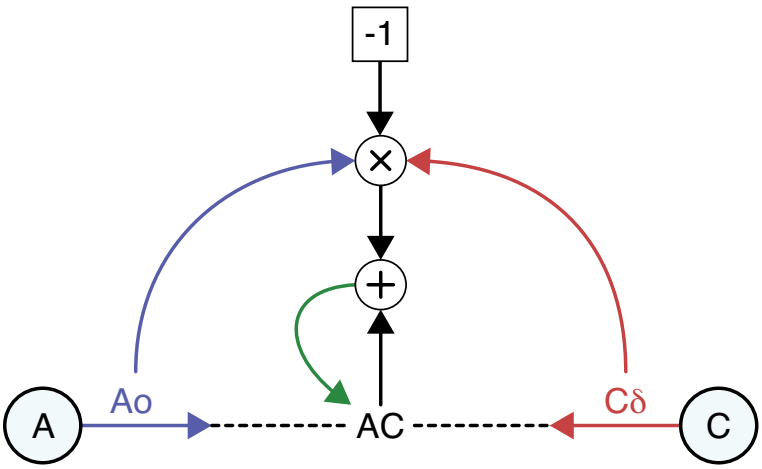


Figure 14-15: Updating the value of weight AC *to the new value* AC – (Ao × Cδ)

Let's walk through this figure. We start with the output Ao from neuron A and the delta $C\delta$ from output neuron C, and multiply them together (at the top of the figure). We want to subtract this from the current value of *AC*. To show this clearly in the diagram, we multiply the product by –1 and then add it to the weight *AC*. The green arrow is the update step, where this result becomes the new value of *AC*.

Figure 14-15 is big news! We've found out how to change the weights coming into the output neurons in order to reduce the network's error. We can apply this to all four weights going into the output neurons (that is, *AC*, *BC*, *AD*, and *BD*). We've just trained our neural network a little bit by improving four of its weights.

Sticking with the output layer, if we change the weights for both output neurons C and D to reduce the error by 1 from each neuron, we'd expect the error to go down by –2. We can predict this because the neurons sharing the same layer don't rely on each other's outputs. Since C and D are both in the output layer, C doesn't depend on *Do* and D doesn't depend on *Co*. They do depend on the outputs of neurons on previous layers, but right now we're just focusing on the effect of changing weights for C and D.

It's wonderful that we know how to adjust the weights on edges going into the output layer, but how about all the other weights? Our next goal is to figure out the deltas for all the neurons in all the preceding layers. Once we have a delta for every neuron in the network, we can use Figure 14-15 to adjust every weight in the network to reduce the error.

And this brings us to the remarkable trick of backpropagation: we can use the neuron deltas at one layer to find the neuron deltas for its preceding layer. Let's see how.

Other Neuron Deltas

Now that we have the delta values for the output neurons, we can use them to compute the deltas for neurons on the layer just before the output layer. In our simple model, that layer is the hidden layer containing neurons A and B. Let's focus for the moment just on neuron A and its connection to neuron C.

What happens if *Ao*, the output of A, changes for some reason? Let's say it goes up by *Am*. Figure 14-16 follows the chain of actions using arbitrary values for *AC* and *Cδ*.

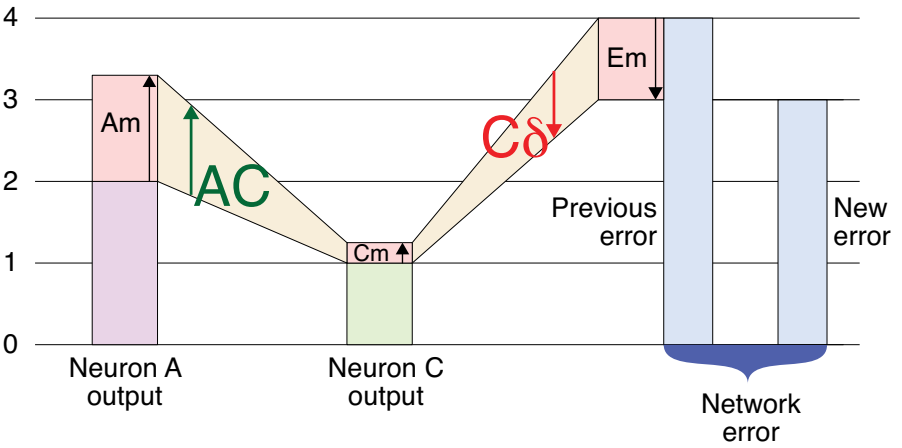


Figure 14-16: Following the results if we change the output of neuron A

If we read the diagram in Figure 14-16 from left to right, the change to A, shown as *Am*, is multiplied by the weight *AC* and then added to the values accumulated by neuron C. This raises the output of C by *Cm*. As we know, this change in C can be multiplied by *Cδ* to find the change in the network error.

So now we have a chain of operations from neuron A to neuron C and then to the error. The first step of the chain says that if we multiply the change in *Ao* (that is, *Am*) by the weight *AC*, giving us $Am \times AC$, we get *Cm*, the change in the output of C. And we know from earlier that if we multiply this value of *Cm* by *Cδ*, forming $Cm \times C\delta$, we get the change in the error.

So, mushing this all together, we find that the change in the error due to a change *Am* in the output of A is $Am \times AC \times C\delta$. We just found the delta for A! It's just $A\delta = AC \times C\delta$.

Figure 14-17 shows this visually.

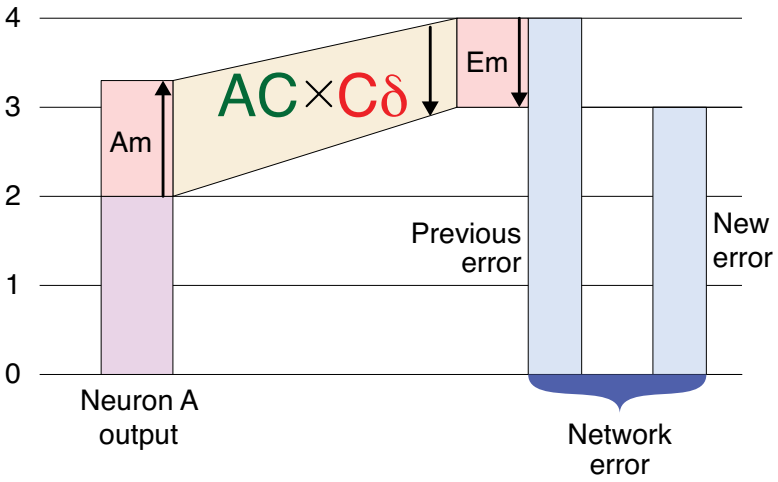


Figure 14-17: We can mush together the operations in Figure 14-16 into a more succinct diagram.

This is kind of amazing. Neuron C has disappeared. It's literally out of the picture in Figure 14-17. All we needed was its delta, *Cδ*, and from that we could find *Aδ*, the delta for A. And now that we know *Aδ*, we can update all of the weights that feed into neuron A, and then . . . no, wait a second.

We don't really have *Aδ* yet. We just have one piece of it.

At the start of this discussion we said we'd focus on neurons A and C, and that was fine. But if we now remember the rest of the network in Figure 14-8, we can see that neuron D also uses the output of A. If *Ao* changes due to *Am*, then the output of D changes as well, and that also affects the error.

To find the change in the error due to neuron D caused by a change in the output of neuron A, we can repeat the analysis we just went through by just replacing neuron C with neuron D. If *Ao* changes by *Am*, and nothing else changes, the change in the error due to the change in D is given by $AD \times D\delta$.

Figure 14-18 shows these two outputs from A at the same time. This figure is set up slightly differently from previous figures of this type that we've seen earlier. Here, the effect of a change in A on the error due to a change in C is shown by the path from the center of the diagram moving to the right. The effect of a change in A on the error due to a change in D is shown by the path from the center of the diagram and moving left.

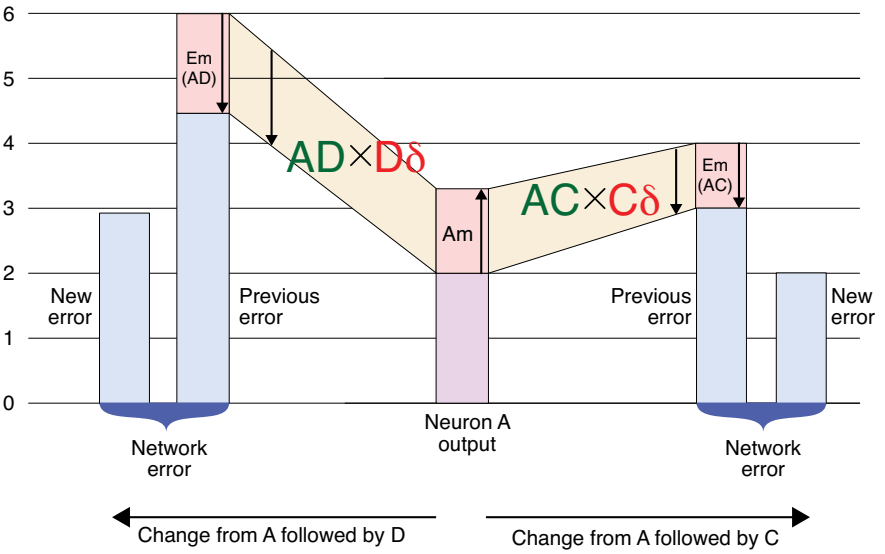


Figure 14-18: The output of neuron A is used by both neuron C and neuron D.

Figure 14-18 shows two separate changes to the error. Since neurons C and D don't influence each other, their effects on the error are independent. To find the total change to the error, we just add up the two changes. Figure 14-19 shows the result of adding the change in error via neuron C and the change via neuron D.

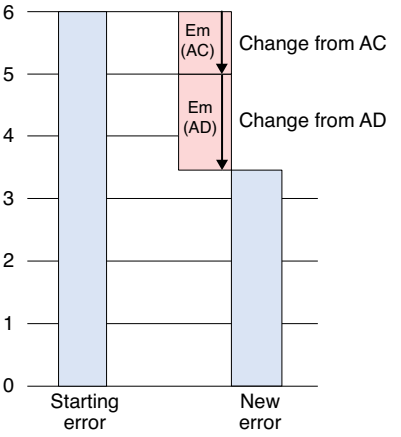


Figure 14-19: When the output of neuron A is used by both neuron C and neuron D, the resulting changes to the error add together.

Now that we've handled all the paths from A to the outputs, we can finally write the value for $A\delta$. Since the errors add together, as in Figure 14-19, we can just add up the factors that scale Am. If we write it out, this is $A\delta = (AC \times C\delta) + (AD \times D\delta)$.

Now that we've found the value of delta for neuron A, we can repeat the process for neuron B to find its delta.

What we've just done is actually far better than finding the delta for just neurons A and B. We've found out how to get the value of delta for every neuron in *any* network, no matter how many layers it has or how many neurons there are! That's because everything we've done involves nothing more than a neuron, the deltas of all the neurons in the next layer that use its value as an input, and the weights that join them. With nothing more than these values, we can find the effect of a neuron's change on the network's error, even if the output layer is dozens of layers away.

To summarize this visually, let's expand on our convention for drawing outputs and deltas as right-pointing and left-pointing arrows to include the weights, as in Figure 14-20. Let's say that the weight on a connection multiplies either the output moving to the right, or the delta moving to the left, depending on which step we're thinking about.

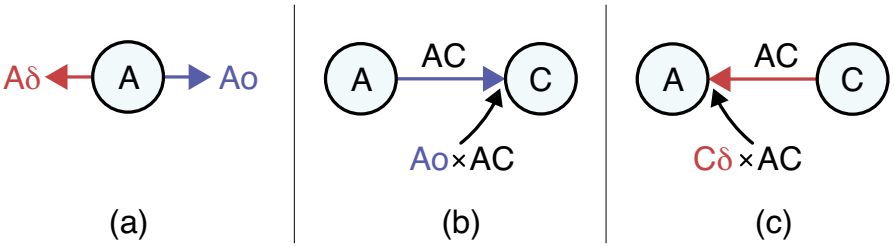


Figure 14-20: Drawing the values associated with neuron A. (a) The output Ao *is an arrow coming out of the right of the neuron, and the delta* Aδ *as an arrow coming out of the left. (b)* Ao *is multiplied by* AC *on its way to being used by C. (c)* Cδ *is multiplied by* AC *on its way to being used by A.*

Here's a way to think about Figure 14-20. There is one connection with one weight joining neurons A and C. If the arrow points to the right, then the weight multiplies Ao, the output of A, as it heads into neuron C. If the arrow points to the left, the weight multiplies $C\delta$, the delta of C, as it heads into neuron A.

When we evaluate a sample, we use the feed-forward, left-to-right style of flow, where the output value from neuron A to neuron C travels over a connection with weight AC. The result is that the value $Ao \times AC$ arrives at neuron C where it's added to other incoming values, as in Figure 14-20(b).

When we later want to compute $A\delta$, we follow the flow from right to left. Then the delta leaving neuron C travels over a connection with weight AC. The result is that the value $C\delta \times AC$ arrives at neuron A where it's added to other incoming values, as in Figure 14-20(c).

Now we can summarize both the processing of a sample input and the computation of the deltas for some arbitrary neuron named H (remember, we're ignoring the activation function), as in Figure 14-21.

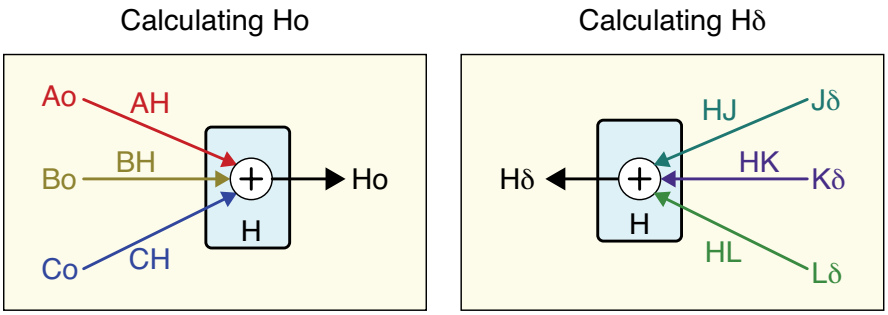


Figure 14-21: Left: To calculate Ho*, we scale the output of each preceding neuron by the weight of its connection and add the results together. Right: To calculate* Hδ*, we scale the delta of each following neuron by the connection's weight and add the results together. As usual, we're ignoring activation functions.*

This is pleasingly symmetrical. It also reveals an important practical result: calculating deltas is often as efficient as calculating output values. Even when the number of incoming connections is different from the number of outgoing connections, the amount of work involved is still close in both directions.

Note that Figure 14-21 doesn't require anything of neuron H except that it has inputs from a neighbor layer that travel on connections with weights and deltas. We can apply the left half of Figure 14-21 and calculate the output of neuron H as soon as the outputs from the previous layer are available. We can apply the right half of Figure 14-21 and calculate the delta of neuron H as soon as the deltas from the following layer are available.

The dependence of *Hδ* on the deltas of the following neurons shows why we had to treat the output layer neurons as special cases: there are no "next layer" deltas to be used.

Throughout this discussion we've left out activation functions. It turns out that we can fit them into Figure 14-21 without changing the basic approach. Though the process is conceptually straightforward, the mechanics involve a lot of details, so we won't go into them here.

This process of finding the delta for every neuron in the network is the heart of the backpropagation algorithm. Let's get a feeling for how backprop works in a larger network.

Backprop on a Larger Network

In the last section we saw the backpropagation algorithm, which lets us compute the delta for every neuron in a network. Because that calculation depended on the deltas in the following neurons and the output neurons don't have any of those, and because the changes to the output neurons are driven directly by the loss function, we treat the output neurons as a special case. Once all the neuron deltas for any layer (including the output layer) have been found, we can then step backward one layer (toward the inputs), and find the deltas for all the neurons on that layer. Then we step backward

again, compute all the deltas, step back again, and so on, until we reach the input layer. Once we have the delta for every neuron, we can adjust the values of the weights going into that neuron, thereby training our network.

Let's walk through the process of using backprop to find the deltas for all the neurons in a slightly larger network.

In Figure 14-22 we show a network with four layers. There are still two inputs and outputs, but now we have three hidden layers of two, four, and three neurons.

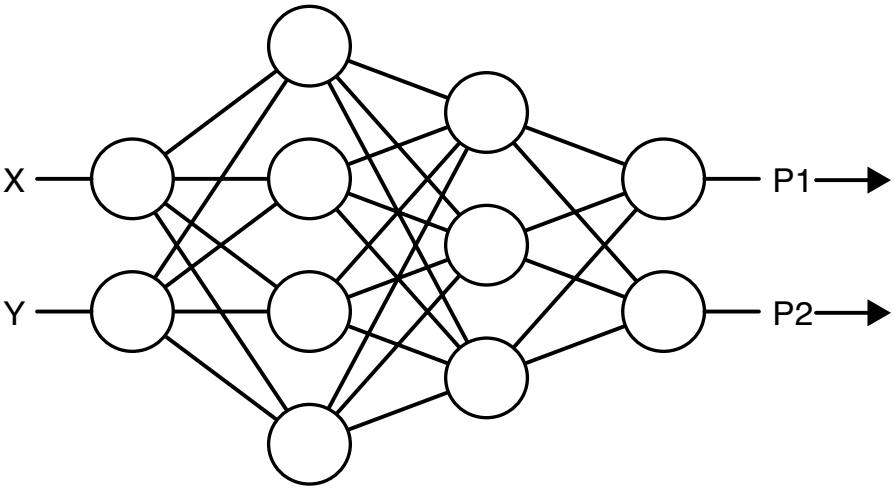


Figure 14-22: A new classifier network with two inputs, two outputs, and three hidden layers

We start things off by evaluating a sample. We provide the values of its X and Y features to the inputs, and eventually the network produces the output predictions *P1* and *P2*.

Now we can start backpropagation by finding the error in the first of the output neurons, as shown in the upper part of Figure 14-23.

We've begun arbitrarily with the upper neuron, which gives us the prediction we've labeled *P1* (the probability that the sample is in class 1). From the values of *P1* and *P2* and the label, we can compute the error in the network's output. Let's suppose the network didn't predict this sample perfectly, so the error is greater than zero.

Using the error, the label, and the values of *P1* and *P2*, we can compute the value of delta for this neuron. If we're using the quadratic cost function, this delta is just the value of the label minus the value of the neuron, as we saw in Figure 14-12. But if we're using some other function, it might be more complicated, so we'll discuss the general case.

We save this delta with its neuron, and then repeat this process for all the other neurons in the output layer (here we have only one more), as shown in the lower part of Figure 14-23. That finishes up the output layer, since we now have a delta for every neuron in the output layer.

At this point, we could start adjusting the weights coming into the output layer, but we usually first find all the neuron deltas first, and then adjust all the weights. Let's follow that typical sequence here.

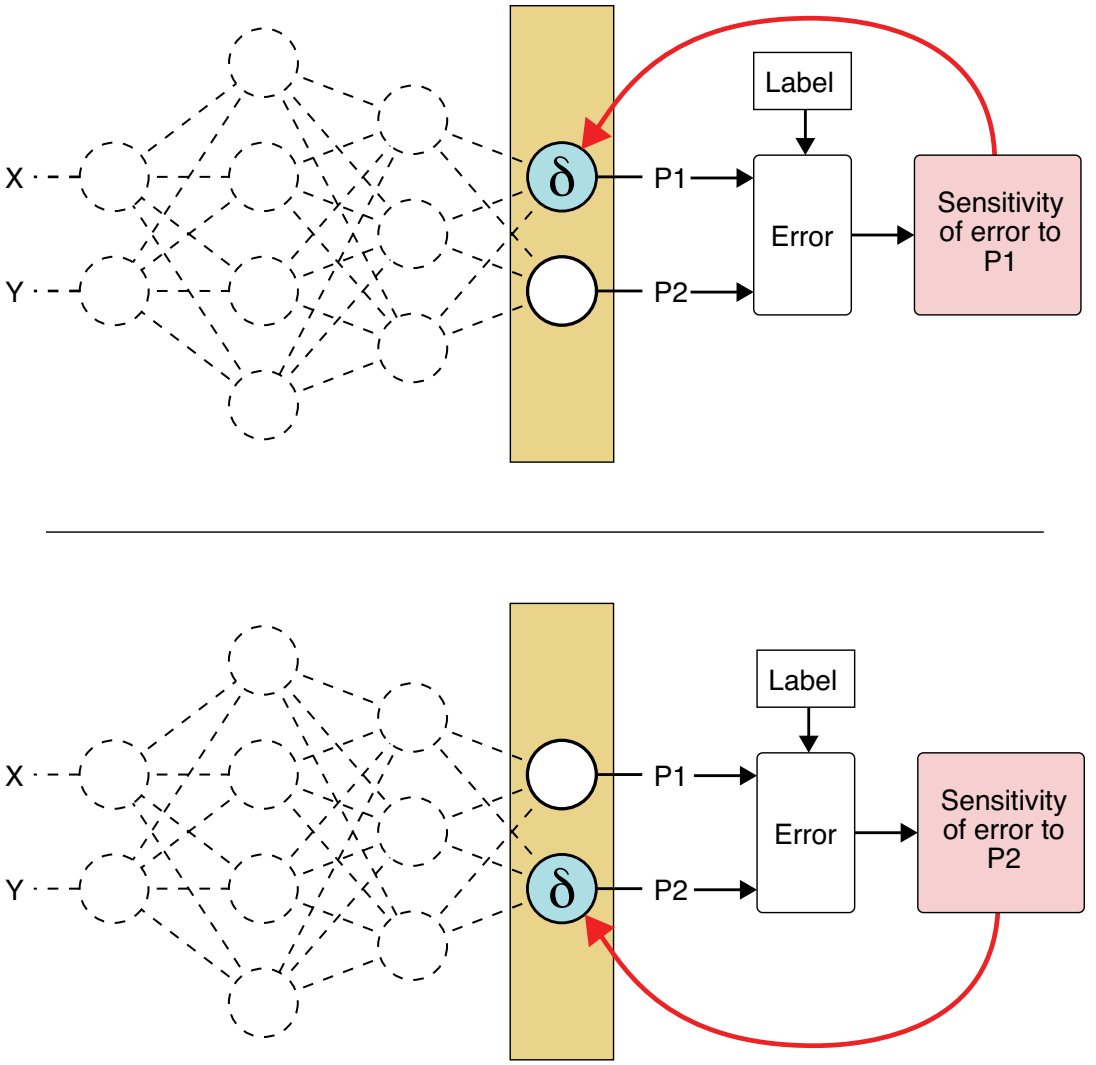


Figure 14-23: Summarizing the steps for finding the delta for both output neurons

We move backward one step to the third hidden layer (the one with three neurons). Let's consider finding the value of delta for the topmost of these three, as in the left image of Figure 14-24.

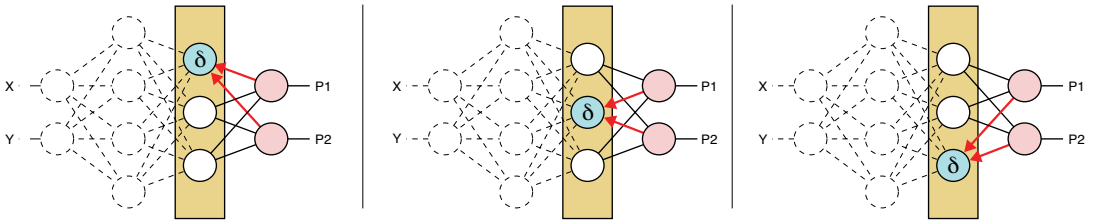


Figure 14-24: Using backpropagation to find the deltas for the next-to-last layer of neurons

To find the delta for this neuron, we follow the recipe of Figure 14-18 to get the individual contributions, and then the recipe of Figure 14-19 to add them together to get the delta for this neuron.

Now we just work our way through the layer, applying the same process to each neuron. When we've completed all the neurons in this three-neuron layer, we take a step backward and start on the preceding hidden layer with four neurons. This is where things really become beautiful. To find the deltas for each neuron in this layer, we need only the weights to each neuron that uses that neuron's output and the deltas for those neurons, which we just computed.

The other layers are irrelevant. We don't care about the output layer anymore now.

Figure 14-25 shows how we compute the deltas for the four neurons in the second hidden layer.

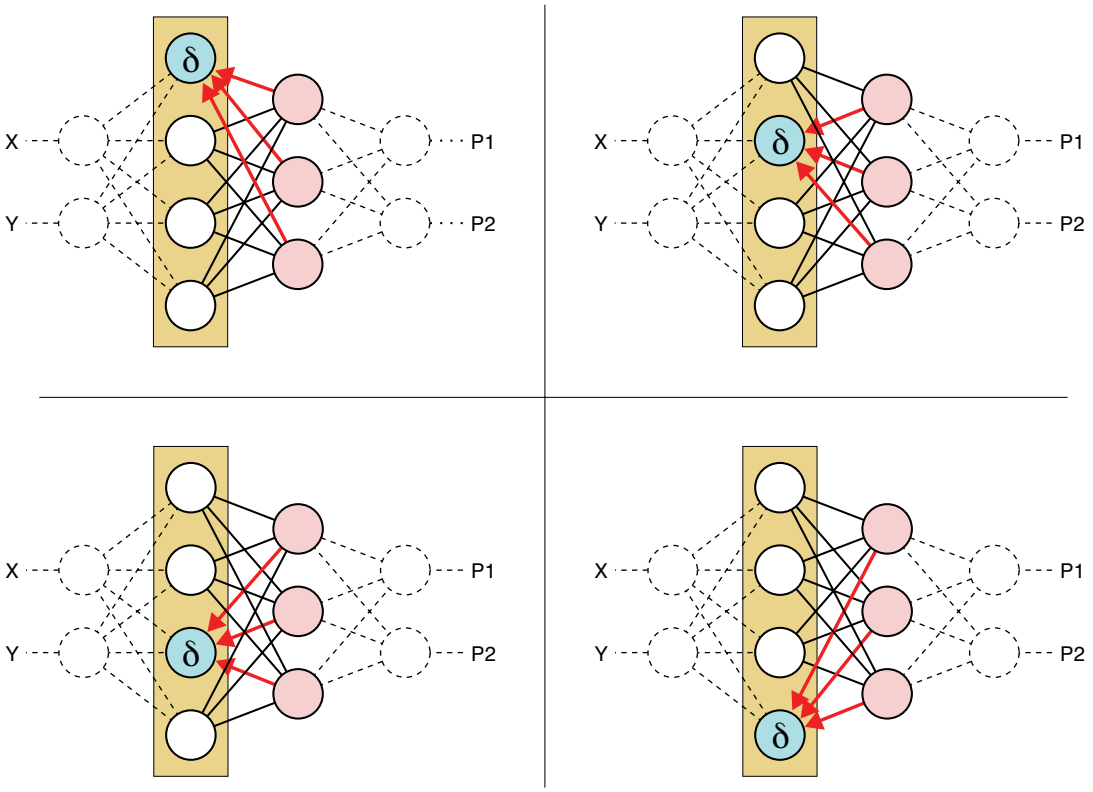


Figure 14-25: Using backprop to find the delta values for the second hidden layer

When all four neurons have had deltas assigned to them, that layer is finished, and we take another step backward. Now we're at the first hidden layer with two neurons. Each of these connects to the four neurons on the next layer. Once again, all we care about are the deltas in that next layer and the weights that connect the two layers. For each neuron, we find the deltas for all the neurons that consume that neuron's output, multiply those by the weights, and add up the results, as shown in Figure 14-26.

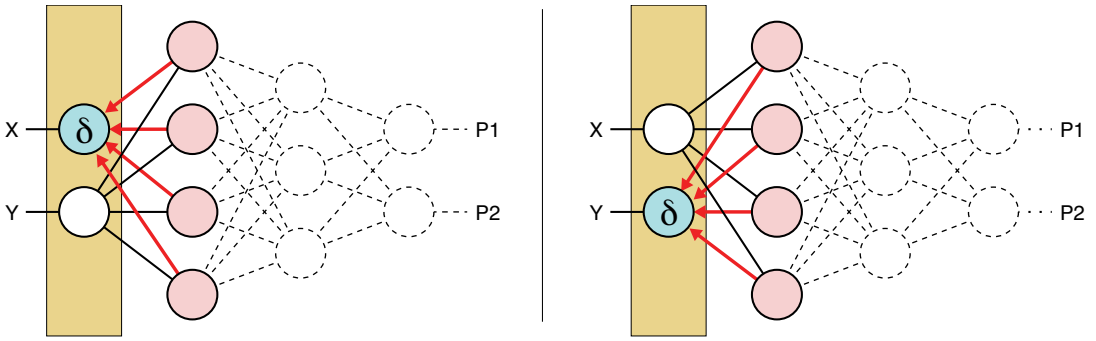


Figure 14-26: Using backprop to find the deltas for the first hidden layer

When Figure 14-26 is complete, we've found the delta for every neuron in the network.

Now let's adjust the weights. We can run through the connections between neurons and use the technique we saw in Figure 14-15 to update every weight to a new and improved value.

Figure 14-23 through Figure 14-26 show why the algorithm is called *backward propagation*. We're taking the deltas from any layer and *propagating*, or moving, their delta (or gradient) information *backward* one layer at a time, modifying it as we go. As we've seen, computing each of these delta values is fast. Even when we put the activation function steps in, that doesn't add much to the computational cost.

Backprop becomes highly efficient when we use parallel hardware like a GPU, because we can use a GPU to multiply all the deltas and weights for an entire layer simultaneously. The tremendous efficiency boost that comes from this parallelism is a key reason why backprop has made learning practical for huge neural networks.

Now we have all of the deltas, and we can update the weights. That's the core process of training a neural network.

Before we leave the discussion, though, let's return to the issue of how much we should move each weight.

The Learning Rate

As we've mentioned, changing a weight by a lot in a single step is often a recipe for trouble. The derivative is only an accurate predictor of the shape of a curve for very tiny changes in the input value. If we change a weight by too much, we can jump right over the smallest value of the error and even find ourselves increasing the error.

On the other hand, if we change a weight by too little, we might see only the tiniest bit of learning, requiring us to spend more time learning than we should actually require. Still, that inefficiency is usually better than a system that's constantly overreacting to errors.

In practice, we control the amount of change to the weights during every update with a hyperparameter called the *learning rate*, usually symbolized by the lowercase Greek letter η (eta). This is a number between 0 and 1, and it tells the weights how much of each neuron's newly computed change to use when it updates.

When we set the learning rate to 0, the weights don't change at all. Our system never changes and never learns. If we set the learning rate to 1, the system applies big changes to the weights and may cause them to increase the error, not decrease it. If this happens a lot, the network can spend its time constantly overshooting and then compensating, with the weights bouncing around and never settling into their best values. Therefore, we usually set the learning rate somewhere between these extremes. In practice, we usually set it to be only slightly larger than 0.

Figure 14-27 shows how the learning rate is applied. Starting with Figure 14-15, we insert an extra step to scale the value of $-(Ao \times C\delta)$ by η before adding it back in to *AC*.

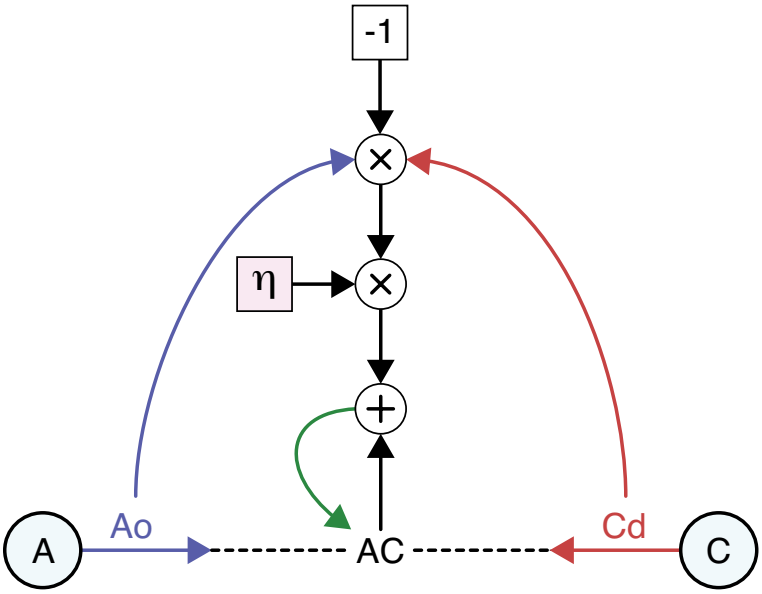


Figure 14-27: The learning rate helps us control how fast the network learns by controlling the amount by which weights change on each update.

The best value to use for the learning rate is dependent on the specific network we've built and the data we're training on. Finding a good choice of learning rate can be essential to getting the network to learn at all. Once the system is learning, changing this value can affect whether that process goes quickly or slowly. Usually we have to hunt for the best value of η using trial and error. Happily, some algorithms automate the search for a good starting value for the learning rate and others fine-tune the learning rate as learning progresses. As a general rule of thumb, and if none of our other choices direct us to a particular learning rate,

we often start with a value around 0.001 and then train the network for a while, watching how well it learns. Then we raise or lower it from that value and train again, over and over, hunting for the value that learns most efficiently. We'll look at techniques for controlling the learning rate more closely in Chapter 15.

Let's see how the choice of learning rate affects the performance of backprop, and thus learning.

Building a Binary Classifier

Let's build a classifier to find the boundary between two crescent moons. We will use about 1,500 points of training data, shown in Figure 14-28.

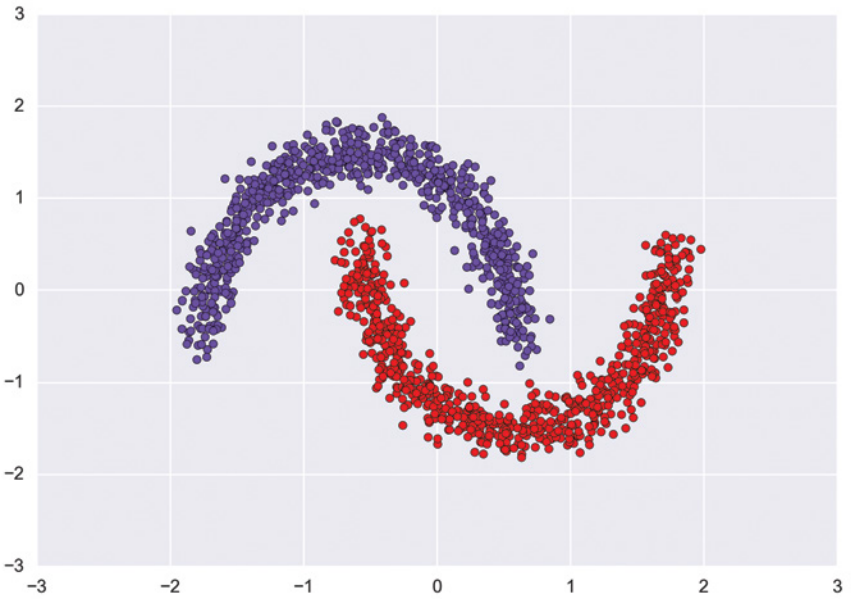


Figure 14-28: About 1,500 points assigned to two classes

Because we have only two classes, we only need a binary classifier. This lets us skip the whole one-hot encoding of labels and dealing with multiple outputs and instead lets us use just one output neuron. If the value is near 0, the input is in one class. If the output is near 1, the input is in the other class.

Our classifier will have just two hidden layers, each with four neurons. These are essentially arbitrary choices we've made that give us a network that's just complex enough for our discussion. As shown in Figure 14-29, both layers are fully connected.

This network uses ReLU activation functions for the neurons in the hidden layers and a sigmoid activation function on the output neuron.

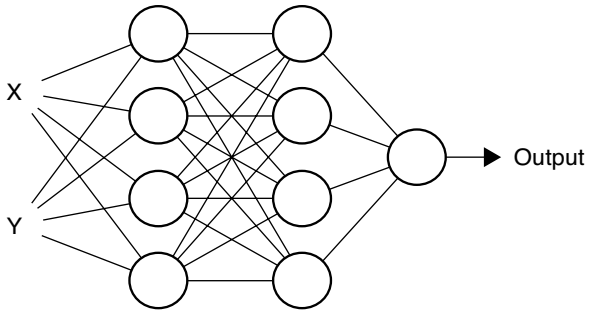


Figure 14-29: Our binary classifier with two inputs, four neurons in each of the two hidden layers, and a single output neuron

How many weights are in our network? There are four coming out of each of the two inputs, then four times four between the layers, and then four going into the output neuron. That gives us $(2 \times 4) + (4 \times 4) + 4 = 28$. Each of the nine neurons also has a bias term, so our network has a total of $28 + 9 = 37$ weights. They are all initialized as small random numbers. Our goal is to use backprop to adjust those 37 weights so that the number that comes out of the final neuron always matches the label for that sample.

As we discussed earlier, we evaluate one sample, calculate the error, and if the error is not zero, we compute the deltas with backprop and then update the weights using the learning rate. Then we move on to the next sample. Note that if the error is 0, then we don't change anything, since the network gave us the answer we wanted. Each time we process all the samples in the training set, we say we've completed one *epoch* of training.

Running backprop successfully relies on making small changes to the weights. There are two reasons for this. The first, which we've discussed, is because the gradient is only accurate very near the point we're evaluating. If we move too far, we may find ourselves increasing the error rather than decreasing it.

The second reason for taking small steps is that changes in weights near the start of the network cause changes in the outputs of neurons in later layers, which change their deltas. To prevent everything from turning into a terrible snarl of conflicting changes, we adjust the weights only by small amounts.

But what is "small"? For every network and dataset, we have to experiment to find out. As we saw earlier, the size of our step is controlled by the learning rate, or eta (η). The bigger this value, the more each weight moves toward its new value.

Picking a Learning Rate

Let's start with an unusually large learning rate of 0.5. Figure 14-30 shows the boundaries computed by our network for our test data, using a different background color for each class.



Figure 14-30: The boundaries computed by our network using a learning rate of 0.5

This is terrible: there don't seem to be any boundaries at all! Everything is being assigned to a single class, shown by the light orange background. If we look at the accuracy and error (or loss) after each epoch, we get the graphs of Figure 14-31.



Figure 14-31: Accuracy and loss for our half-moons data with a learning rate of 0.5

Things are looking bad. As we'd expect, the accuracy is just about 0.5, meaning that half the points are being misclassified. This makes sense, since the red and blue points are roughly evenly divided. If we

assign them all to one class, as we're doing here, half of those assignments will be wrong. The loss, or error, starts high and doesn't fall. If we let the network run for hundreds of epochs, it continues on in this way, never improving.

What are the weights doing? Figure 14-32 shows the values of all 37 weights during training.

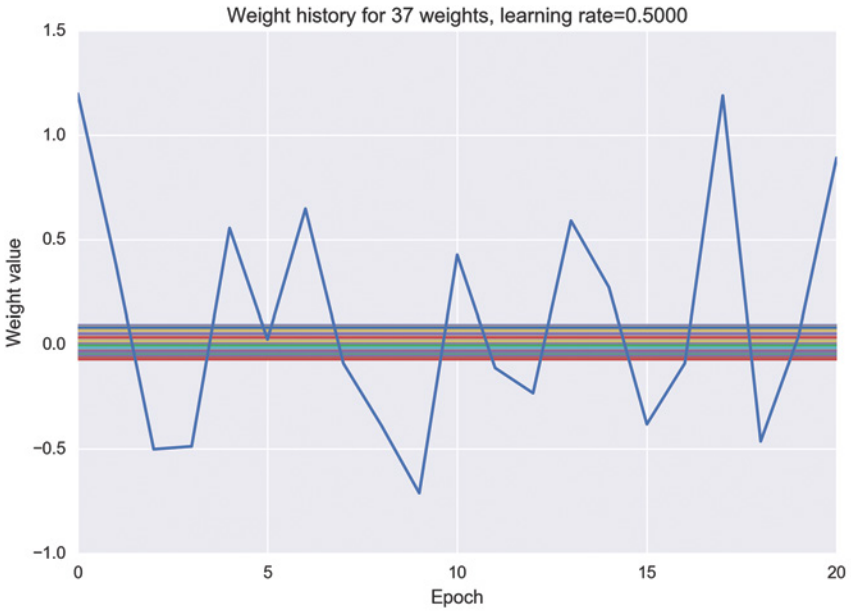


Figure 14-32: The weights of our network when using a learning rate of 0.5. One weight is constantly changing and overshooting its goal, while the others are making changes too small to show on this graph.

The graph is dominated by one weight that's jumping all over. That weight is one of those going into the output neuron, trying to move its output around to match the label. That weight goes up, then down, then up, jumping too far almost every time, then overcorrecting by too much, then overcorrecting for that, and so on. The other neurons are changing too, but at too small a scale to see in this graph.

These results are disappointing, but they're not shocking, because a learning rate of 0.5 is *big*. That's what's causing all the erratic bouncing around in Figure 14-32.

Let's reduce the training rate by a factor of 10 to a more reasonable (though still big) value of 0.05. We'll change absolutely nothing else about the network or the data, and we'll even reuse the same sequence of pseudo-random numbers to initialize the weights. The new boundaries are shown in Figure 14-33.

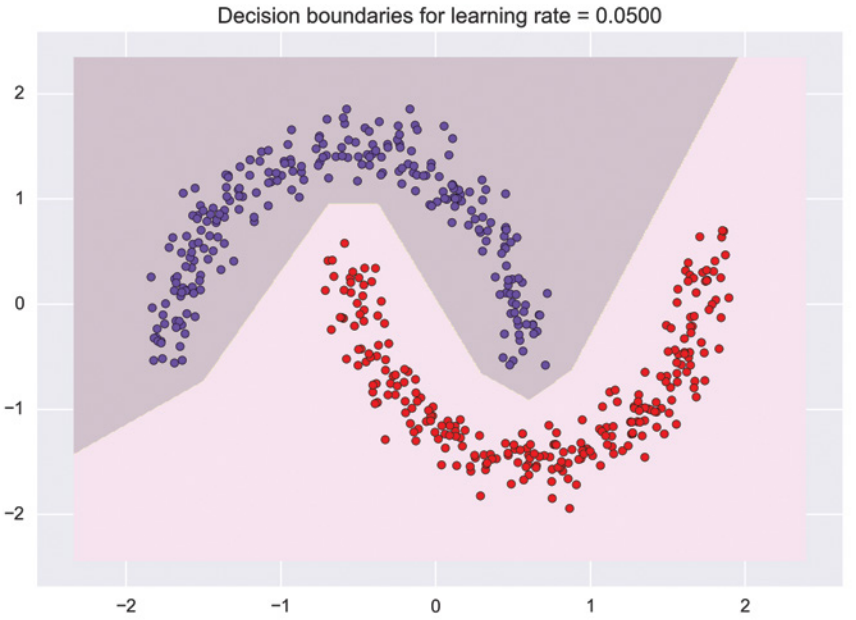


Figure 14-33: The decision boundaries when we use a learning rate of 0.05

This is much better! Looking at the graphs in Figure 14-34 reveals that we've reached 100 percent accuracy on both the training and test sets after about 16 epochs. Using a smaller learning rate gave us a huge improvement.

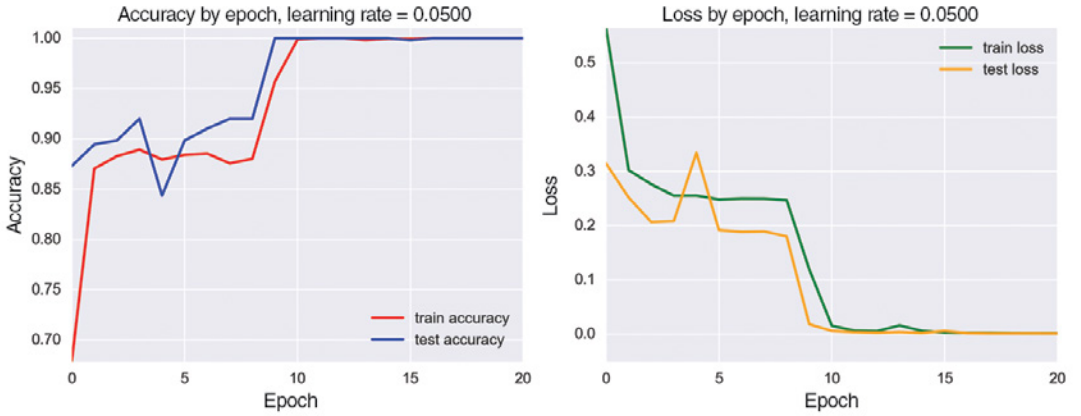


Figure 14-34: Accuracy and loss for our network when using a learning rate of 0.05

This shows us the importance of tuning the learning rate for every new combination of network and data. If a network refuses to learn, we can sometimes make things better by simply reducing the learning rate.

What are the weights doing now? Figure 14-35 shows us their history.

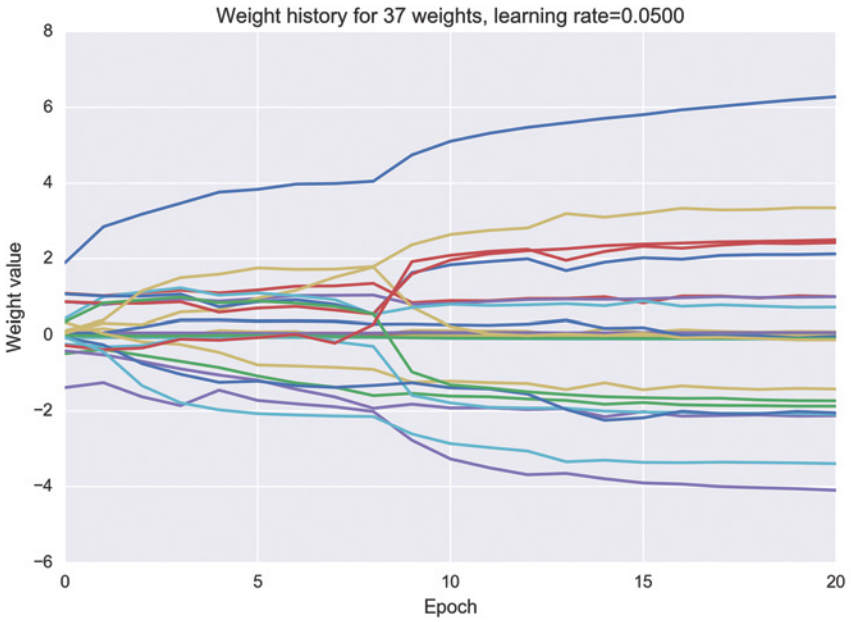


Figure 14-35: The weights in our network over time, using a learning rate of 0.05

Overall, this is way better, because lots of weights are changing. They're getting pretty large, which can itself inhibit or slow down learning. We usually want our weights to be in a small range, typically [–1, 1]. We'll see some ways to control weight values when we discuss regularization in Chapter 15.

Figure 14-33 and Figure 14-34 are pictures of success. Our network has learned to perfectly classify the data, and it did it in only 16 epochs, which is nice and fast (in fact, the graphs show us that it really took only 10 epochs). On a late 2014 iMac without GPU support, the whole training process for 16 epochs took less than 10 seconds.

An Even Smaller Learning Rate

What if we lower the learning rate down to 0.01? Now the weights change even more slowly. Does this produce better results?

Figure 14-36 shows the decision boundary resulting from these tiny steps. The boundary seems simpler than the boundary in Figure 14-33, but both boundaries separate the sets perfectly.



Figure 14-36: The decision boundaries for a learning rate of 0.01

Figure 14-37 shows our accuracy and loss graphs. Because our learning rate is so much slower, our network takes around 170 epochs to get to 100 percent accuracy, rather than the 16 in Figure 14-35.

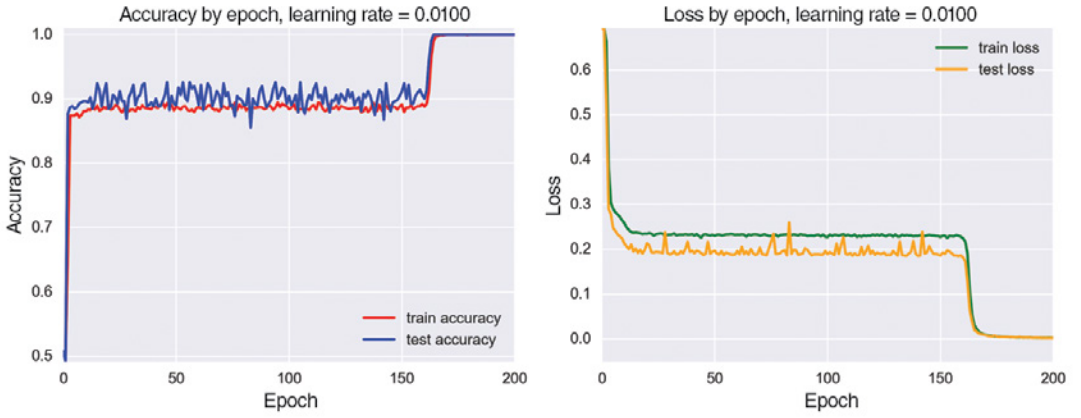


Figure 14-37: The accuracy and learning rate for our network using a learning rate of 0.01

These graphs show an interesting learning behavior. After an initial jump, both the training and test accuracies reach about 90 percent and plateau there. At the same time, the losses plateau as well. Then around epoch 170, things improve rapidly again, with the accuracy climbing to 100 percent and the errors dropping to zero.

This pattern of alternating improvement and plateaus is not unusual, and we can even see a hint of a plateau in Figure 14-34 between epochs 3 and 8. These plateaus come from the weights finding themselves on nearly flat regions of the error surface, resulting in near-zero gradients, and thus their updates are very small.

Though our weights might be getting stuck in a local minimum, it's more common for them to get caught in a flat region of a saddle, like those we saw in Chapter 5 (Dauphin et al. 2014). Sometimes it takes a long time for one of the weights to move into a region where the gradient is large enough to give it a good push. When one weight gets moving, it's common to see the others kick in as well, thanks to the cascading effect of that weight's changes on the rest of the network.

The values of the weights follow almost the same pattern over time, as shown in Figure 14-38. The interesting thing is that at least some of the weights are not flat or on a plateau near the middle of our training process. They're changing, but slowly. The system is getting better, but in tiny steps that don't show up in the performance graphs until the changes become bigger around epoch 170.

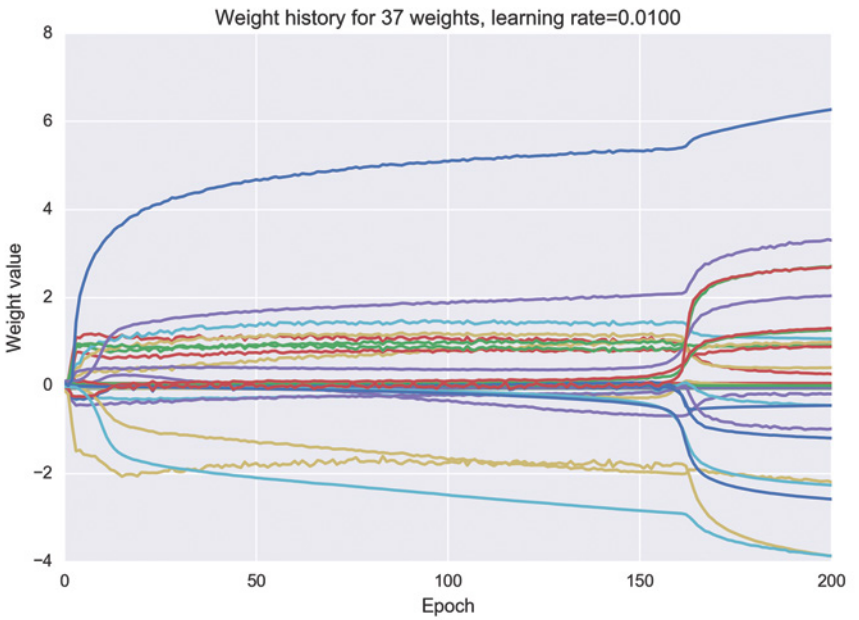


Figure 14-38: The history of our weights using a learning rate of 0.01

So, was there any benefit to lowering the learning rate down to 0.01? In this case, not really. Even at 0.05, the classification was already perfect on both the training and test data. For this network and this data, the smaller learning rate just meant the network took longer to learn. This investigation has shown us how sensitive the network is to our choice of learning rate. We want to find a value that's not too big, or too small, but just right (Pyle 1918).

We usually do this kind of experimenting with the learning rate as part of developing nearly every deep learning network. We need to find a value that does the best job on each specific network and data. Happily, in Chapter 15 we'll see algorithms that can automatically adjust the learning rate for us in sophisticated ways.

Summary

This chapter was all about backpropagation. We saw that we can predict how the error of a network will change in response to a change in each weight. If we can determine whether each weight should increase or decrease in value, we can reduce the error.

To find how we should change each weight, we started by assigning a delta value to each neuron. This value tells us the relationship between a change in a weight's value and a change in the final error. This enabled us to determine how to change each weight in order to reduce the error.

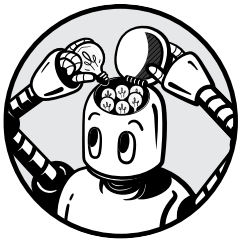
The computation of these deltas proceeds from the final layer backward to the first. Because the gradient information needed to compute the delta for each neuron is propagated backward one layer at a time, we get the name *backpropagation*. Backprop can be implemented on a GPU, where we can carry out the calculations for many neurons simultaneously.

It's important to keep in mind that backprop is propagating the gradient of the error, which is the information that tells us how the error changes when the weight changes. Some authors casually speak of backprop as propagating the error, but that's a misleading simplification. We're propagating the gradient, which tells us how to manipulate the weights to improve the network's output.

Now that we know whether each weight should be adjusted to be larger or smaller, we need to decide how big a change to actually make. That's exactly what we'll figure out in the next chapter.

15

OPTIMIZERS



Training neural networks is frequently a time-consuming process. Anything that speeds it up is a welcome addition to our toolkit. This chapter is about a family of tools that are designed to speed up learning by improving the efficiency of gradient descent. The goals are to make gradient descent run faster and avoid some of the problems that can cause it to get stuck. These tools also automate some of the work of finding the best learning rate, including algorithms that can adjust that rate automatically over time. Collectively, these algorithms are called *optimizers*. Each optimizer has its strengths and weaknesses, so it's worth becoming familiar with them so we can make good choices when training a neural network.

Let's begin by drawing some pictures that enable us to visualize error and how it changes as we learn. These pictures will help us build some intuition for the algorithms yet to come.

Error as a 2D Curve

It's often helpful to think of the errors in our systems in terms of geometrical ideas. We frequently plot error as a 2D curve.

To get familiar with this 2D error, let's consider the task of splitting two classes of samples represented as dots arranged on a line. Dots at negative values are in one class, and dots at zero and above are in the other, as shown in Figure 15-1.

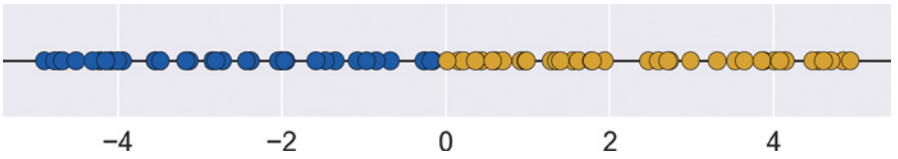


Figure 15-1: Two classes of dots on a line. Dots to the left of 0 are in class 0, shown in blue, and the others are in class 1, shown in beige.

Let's build a classifier for these samples. In this example, the boundary consists of just a single number. All samples to the left of that number are assigned to class 0, and all those to the right are assigned to class 1. If we imagine moving this dividing point along the line, we can count up the number of samples that are misclassified and call that our error. We can summarize the results as a plot, where the X axis shows us each potential splitting point, and the error associated with that point is plotted as a dot above it. Figure 15-2 shows the result.

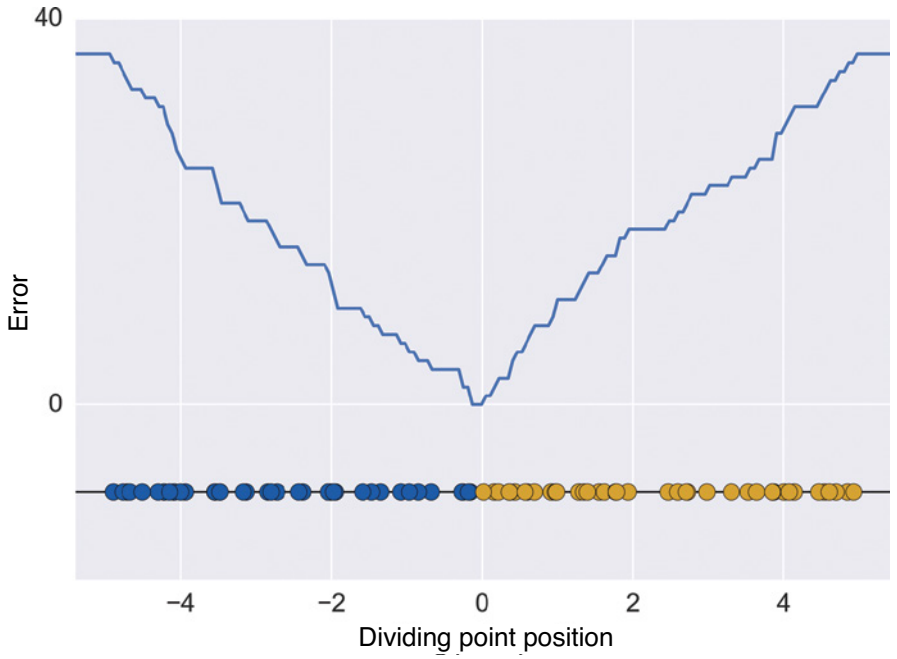


Figure 15-2: Plotting the error function for a simple classifier

We can smooth out the error curve of Figure 15-2 as shown in Figure 15-3.

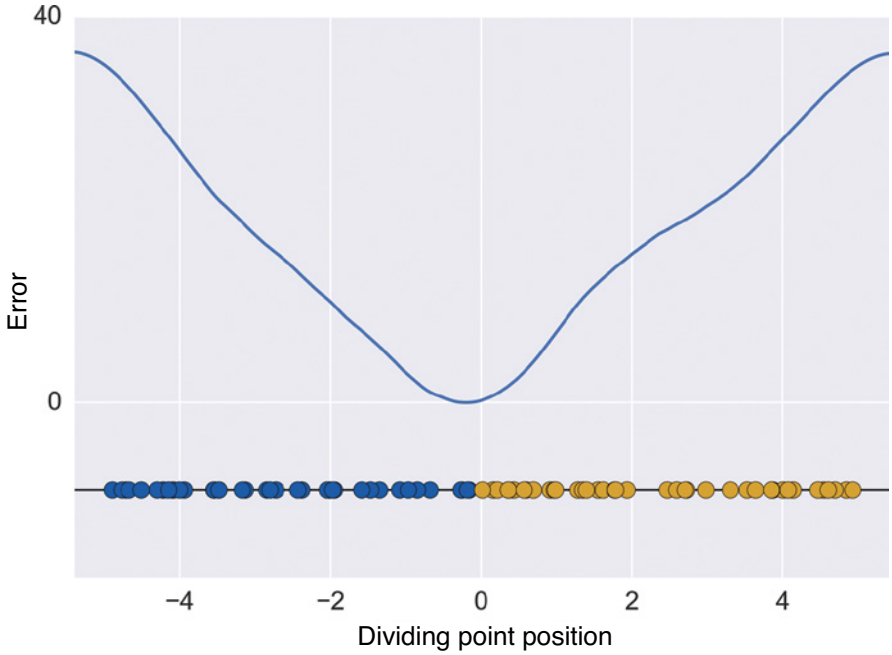


Figure 15-3: A smoothed version of Figure 15-2

For this particular set of random data, we see that the error is 0 when we're at 0, or just a little to the left of it. This tells us that regardless of where we start, we want to end up with our divider just to the left of 0.

Our goal is to find a way to locate the smallest value of any error curve. When we can do that, we can apply the technique to all the weights of a neural network and thus reduce the whole network's error.

Adjusting the Learning Rate

When we teach a system using gradient descent, the critical parameter is the learning rate, usually written with the lowercase Greek letter η (eta). This is often a value in the range 0.01 to 0.0001. Larger values lead to faster learning, but they can lead us to miss valleys by jumping right over them. Smaller values of η (nearing 0, but always positive) lead to slower learning and can find narrow valleys, but they can also get stuck in gentle valleys even when there are much deeper ones nearby. Figure 15-4 recaps these phenomena graphically.

An important idea shared by many optimizers is that we can improve learning by changing the learning rate as we go. The general thinking is analogous to hunting for buried metal on a beach using a metal detector. We start by taking big steps as we walk across the beach, but when the detector goes off, we take smaller and smaller steps to pinpoint the metal object's

location. In the same way, we usually take big steps along the error curve early in the learning process while we're hunting for a valley. As time goes on, we hope that we found that valley, and we can now take smaller and smaller steps as we approach its lowest point.

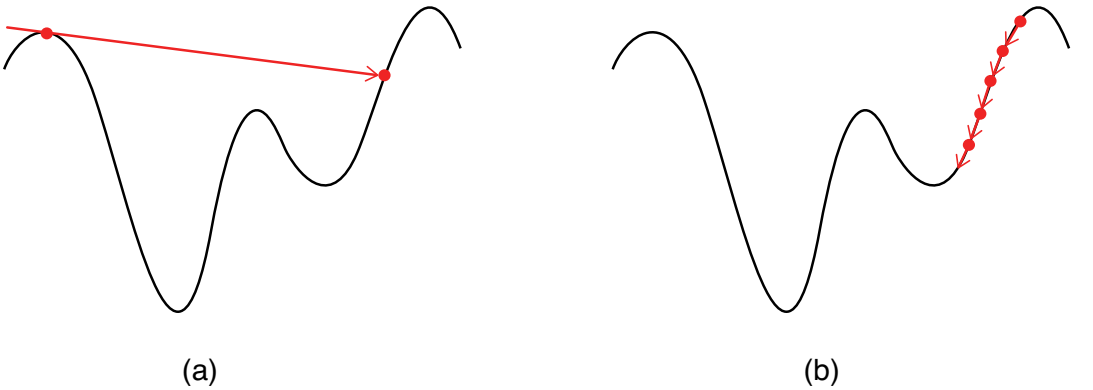


Figure 15-4: The influence of the learning rate, η. (a) When η is too large, we can jump right over a deep valley and miss it. (b) When η is too small, we can slowly descend into a local minimum, and miss the deeper valley.

We can illustrate our optimizers with a simple error curve containing a single isolated valley with the shape of a negative Gaussian, shown in Figure 15-5.

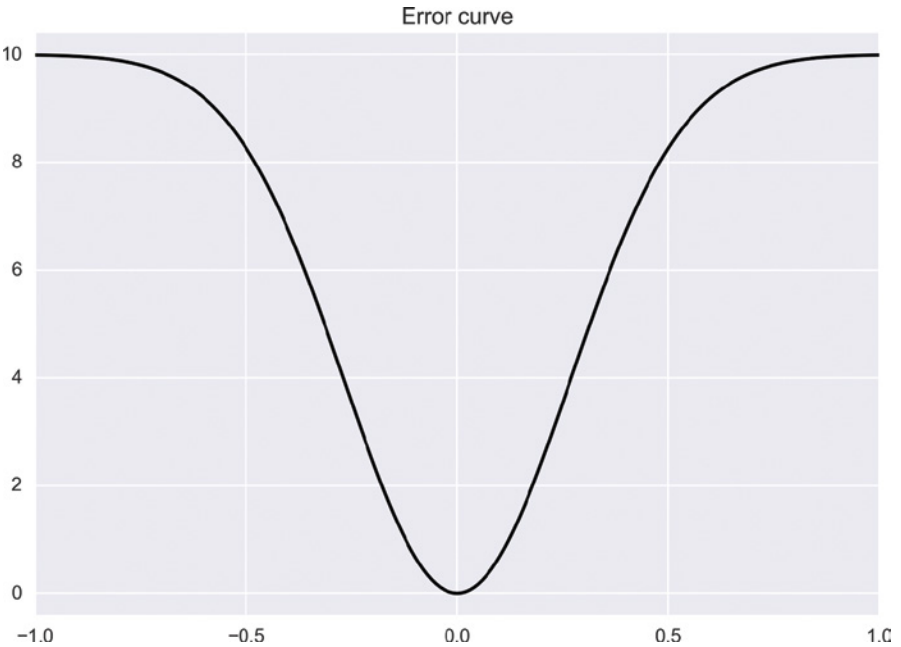


Figure 15-5: Our error curve for looking at optimizers

Some gradients for this error curve are shown in Figure 15-6 (we're actually showing the negative gradients).

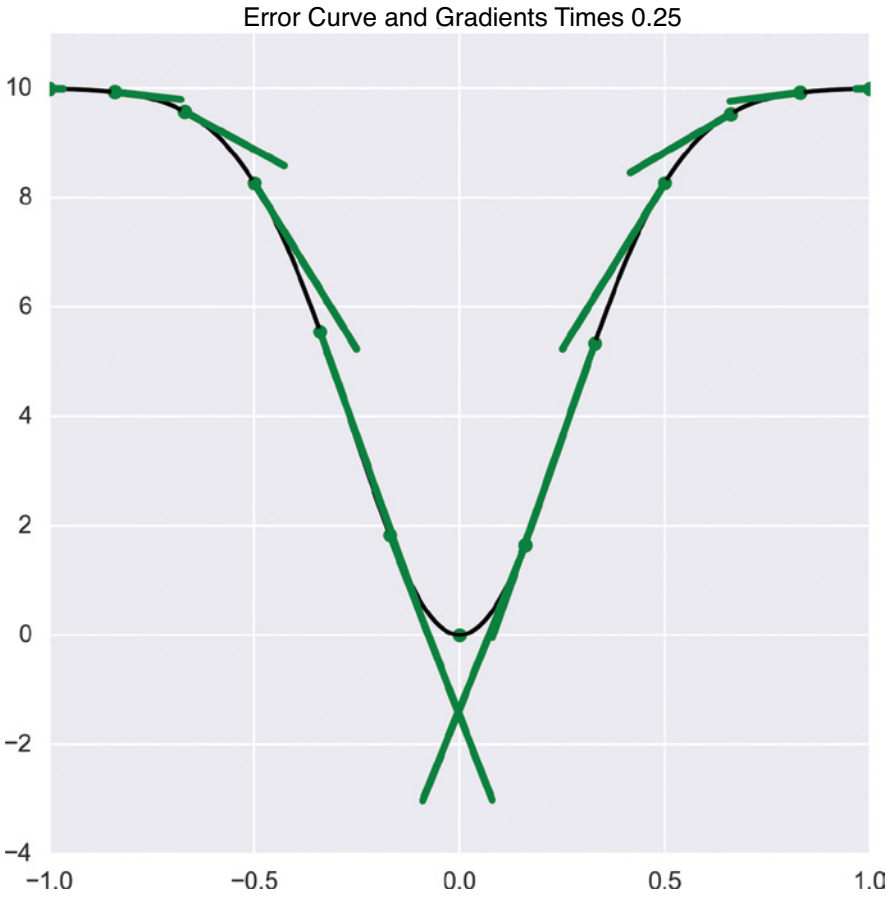


Figure 15-6: Our error curve and its negative gradients (scaled down by a factor of 0.25) at some locations

The gradients in Figure 15-6 have been scaled down to 25 percent of their actual length for clarity. We can see that for this curve, the gradient is negative for input values that are less than 0 and positive for input values that are greater than 0. When the input is 0, we're at the very bottom of the bowl, so the gradient there is 0, drawn as just a single dot.

Constant-Sized Updates

Let's start our investigation of the effect of the learning rate by seeing what happens when we use a constant learning rate. In other words, we always scale the gradient by a value of η that stays fixed, or constant, during the whole training process.

Figure 15-7 shows the basic steps of updating with a fixed η.

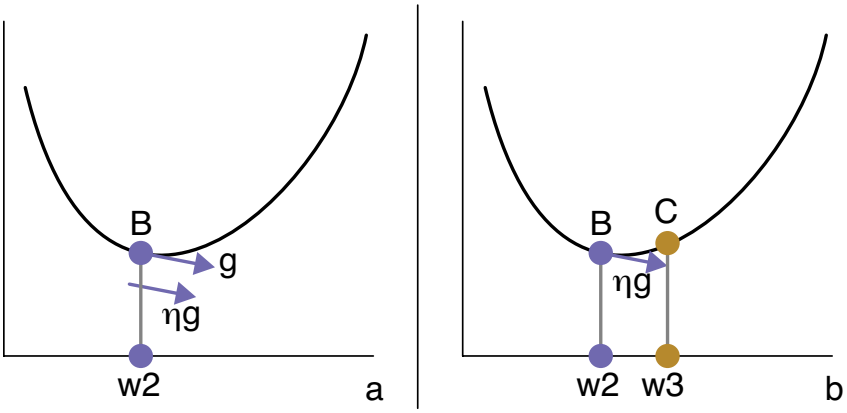


Figure 15-7: Finding the step for basic gradient descent

Suppose we're looking at a particular weight in a neural network. Let's pretend that the weight begins with a value of w1, and we updated it once, so it now has the value w2, shown in Figure 15-7(a). Its corresponding error is the point on the error curve directly above it, marked B. We want to update the weight again to a new and better value that we'll call w3.

To update the weight, we find its gradient on the error surface at the point B, shown as the arrow labeled *g*. We scale the gradient by the learning rate η to get a new arrow labeled ηg. Because η is between 0 and 1, ηg is a new arrow that points in the same direction as *g* but is either the same size as *g* or smaller.

In Figure 15-7, the arrow we show for the gradient *g* is actually the *opposite*, or negative, of the gradient. The positive and negative gradients point in opposite directions along the same line, so people tend to refer to simply *the gradient* when the choice of positive or negative can be understood from context. We'll follow that convention in this chapter.

To find w3, the new value of the weight, we add the scaled gradient to w2. In pictures, this means we place the tail of the arrow ηg at B, as in Figure 15-7(b). The horizontal position of the tip of that arrow is the new value of the weight, w3, and its value, directly above it on the error surface, is marked C. In this case, we stepped a bit too far and increased our error by a little.

Let's look at this technique in practice using an error curve with a single valley. Figure 15-8 shows a starting point in the upper left. The gradient here is small, so we move to the right a small amount. The error at that new point is a little less than the error we started with.

For these figures, we've chosen $\eta = 1/8$, or 0.125. This is an unusually large value of η for constant-sized gradient descent, where we often use a value of 1/100 or less. We chose this large value because it makes for clearer pictures. Smaller values work in similar ways, just more slowly. We

aren't showing values on the axes for these graphs to avoid visual clutter, since we're more interested in the nature of what happens rather than the numbers.

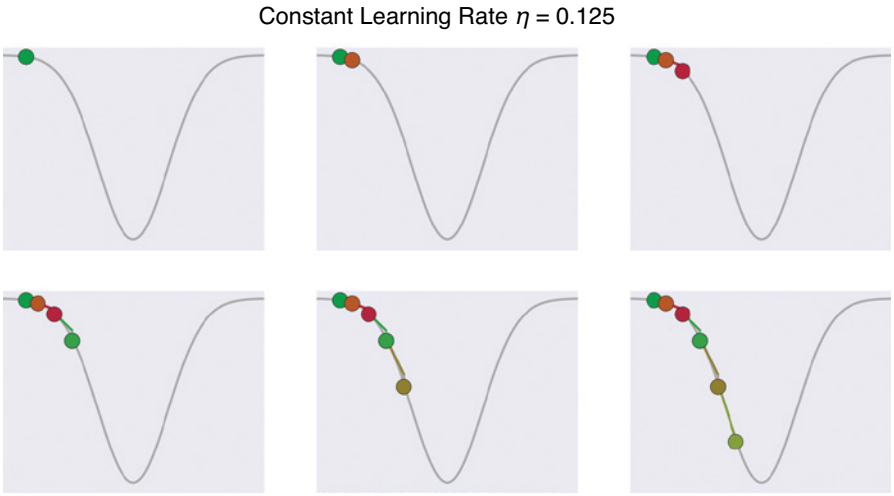


Figure 15-8: Learning with a constant learning rate

Rather than move from our first point by the entire gradient, we're moving only 1/8 of its length. This move takes us to a steeper part of the curve, where the gradient is larger, so the next update moves a little farther. Each step of learning is shown with a new color, which we use to draw the gradient from the previous location and then the new point.

We show a close-up of six steps in Figure 15-9, starting after the first step in Figure 15-8. We also show the error for each point.

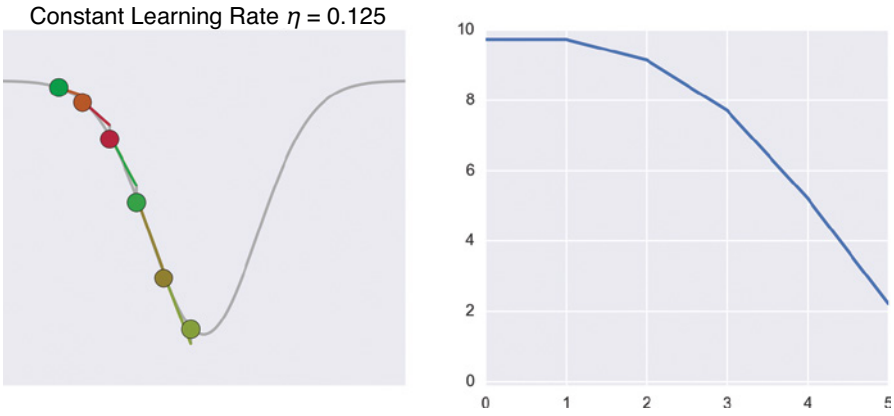


Figure 15-9: Left: A close-up of the final image in Figure 15-8. Right: The error associated with each of these six points.

Will this process ever reach the bottom of the bowl and get down to 0 error? Figure 15-10 shows the first 15 steps in this process.

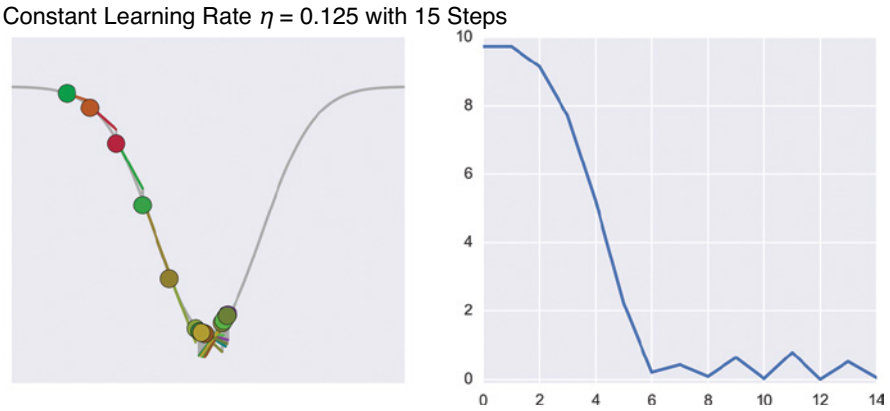


Figure 15-10: Left: The first 15 steps of learning with a constant learning rate. Right: The errors of these 15 points.

We get near the bottom and then head up the hill on the right side. But that's okay because the gradient here points down and to the left, so we head back down the valley until we overshoot the bottom again, and end up somewhere on the left side, then we turn around and overshoot again and end up on the right side, and so on, back and forth. We're *bouncing around* the bottom of the bowl.

It doesn't look like we'll ever get to 0. The problem is particularly bad in this symmetrical valley, as the error jumps back and forth between the left and right sides of the minimum. But this type of behavior happens a lot when we use a constant learning rate. The bouncing around is happening because when we're near the bottom of a valley, we want to take small steps, but because our learning rate is a constant, we're taking steps that are too big.

We might wonder if the bouncing problem of Figure 15-10 was caused by too large a learning rate. Figure 15-11 shows how things go for the first 15 steps of some smaller values of η.

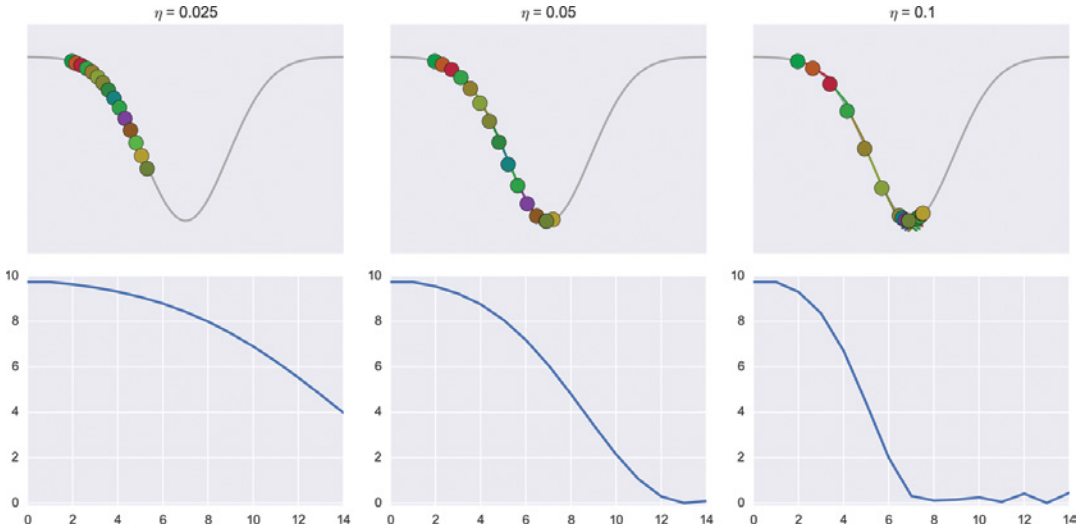


Figure 15-11: Taking 15 steps with our small learning rates. Top row: Learning rates of 0.025 (left column), 0.05 (middle column), and 0.1 (right column). Bottom row: Errors for the points in the top row.

As we can see from Figure 15-11, taking smaller steps doesn't solve the bouncing problem, though the bounces are smaller. On the other hand, increasing the learning rate makes the bouncing problem worse, as shown in Figure 15-12.

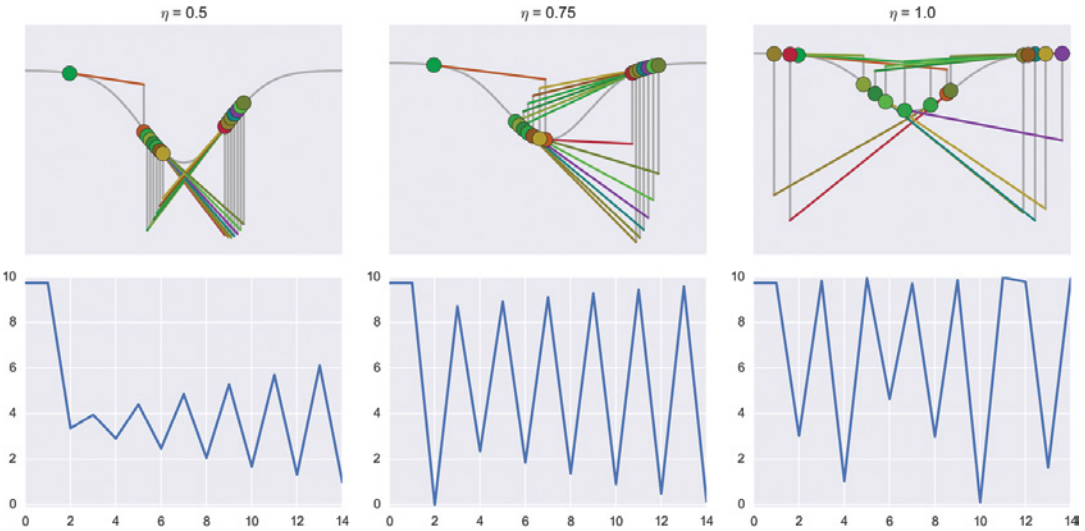


Figure 15-12: Top row: Learning rates of 0.5 (left column), 0.75 (middle column), and 1.0 (right column). Bottom row: Errors for the points in the top row.

Larger learning rates can also cause us to jump out of a nice valley with a low minimum. In Figure 15-13, starting at the green dot, we jump right over the rest of the valley we're in (and would like to stay in) and into a new valley with a much larger minimum.

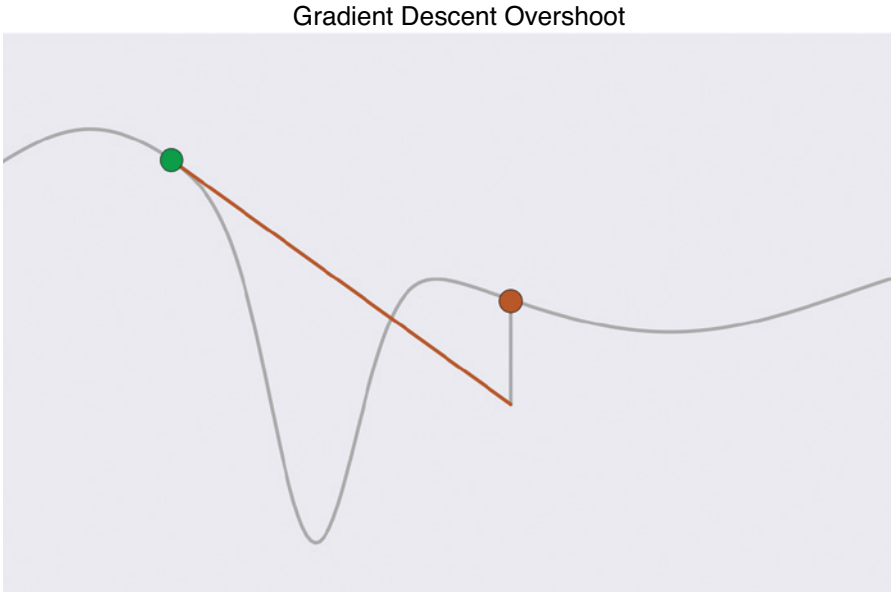


Figure 15-13: A large step overshoots the valley and ends up in a different valley with a higher minimum.

Sometimes a big jump like this can help us move from a shallow valley to a deeper one, but for such a large learning rate, we'll probably jump around valleys a lot, never finding a minimum. It seems like a challenge to find just one learning rate that moves at a reasonable speed but won't overshoot valleys or get trapped bouncing around in the bottom. A nice alternative is to change the learning rate as we go.

Changing the Learning Rate over Time

We can use a large value of η near the start of our learning so we don't crawl along, and a small value near the end so we don't end up bouncing around the bottom of a bowl.

An easy way to start big and gradually get smaller is to multiply the learning rate by some number that's almost 1 after every update step. Let's use 0.99 as a multiplier and suppose that the starting learning rate is 0.1. Then after the first step, it will be $0.1 \times 0.99 = 0.099$. On the next step, it would be $0.099 \times 0.99 = 0.09801$. Figure 15-14 shows what happens to η when we do this for many steps using a few different values for the multiplier.

The easiest way to write the equation of these curves involves using exponents, so this kind of curve is called an *exponential decay* curve. The value by which we multiply η on every step is called the *decay parameter*. This is usually a number very close to 1.

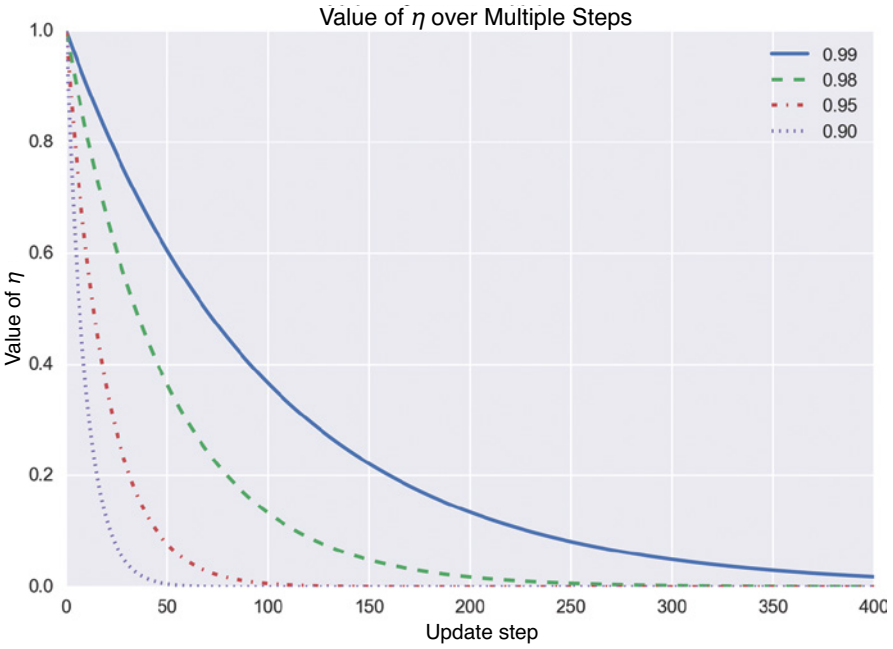


Figure 15-14: Starting with a learning rate of η = 1, the various curves show how the learning rate drops after multiplying it by a given value after each update.

Let's apply this gradual reduction of the learning rate to gradient descent on our error curve. Once again, we start with a learning rate of 1/8. To make the effect of the decay parameter easily visible, let's set it to the unusually low value of 0.8. This means each step will only be 80 percent as long as the step before it. Figure 15-15 shows the result for the first 15 steps.

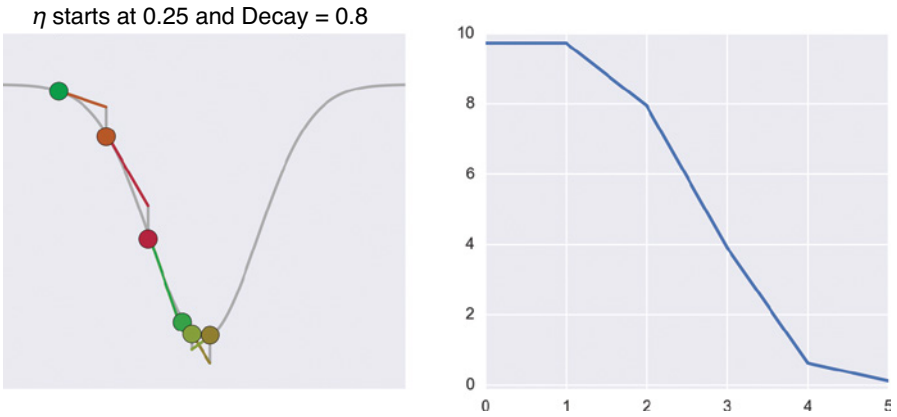


Figure 15-15: The first 15 steps using a shrinking learning rate

Let's compare this with our "bouncing" result from using a constant step size. Figure 15-16 shows the results for the constant and shrinking step sizes together for 15 steps.

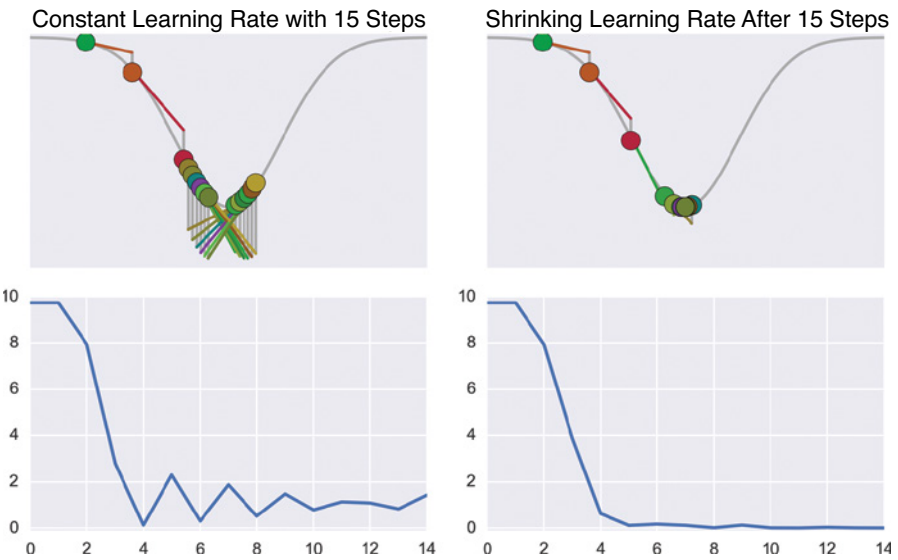


Figure 15-16: On the left is the constant step size from Figure 15-10, and on the right is the decaying step size from Figure 15-15. Notice how the shrinking learning rate helps us efficiently settle into the minimum of the valley.

The shrinking step size does a beautiful job of landing us in the bottom of the bowl and keeping us there.

Decay Schedules

The decay technique is attractive, but it comes with some new challenges. First, we have to choose a value for the decay parameter. Second, we might not want to apply the decay after every update. To address these issues, we can try some other strategies for reducing the learning rate.

Any given approach to changing the learning rate over time is called a *decay schedule* (Bengio 2012).

Decay schedules are usually expressed in epochs, rather than samples. We train on all the samples in our training set, and only then consider changing the learning rate before we train on all the samples again.

The simplest decay schedule is to always apply decay to the learning rate after every epoch, as we just saw. Figure 15-17(a) shows this schedule.

Another common scheduling method is to put off any decay at all for a while so our weights have a chance to get away from their starting random values and into something that might be close to finding a minimum. Then we apply whatever schedule we've picked. Figure 15-17(b) shows this *delayed exponential decay* approach, putting off the exponential decay schedule of Figure 15-17(a) for a few epochs.

Another option is to apply the decay only every once in a while. The *interval decay* approach shown in Figure 15-17(c), also called *fixed-step decay*, reduces the learning rate after every fixed number of epochs, say every 4th or 10th. This way we don't risk getting too small too fast.

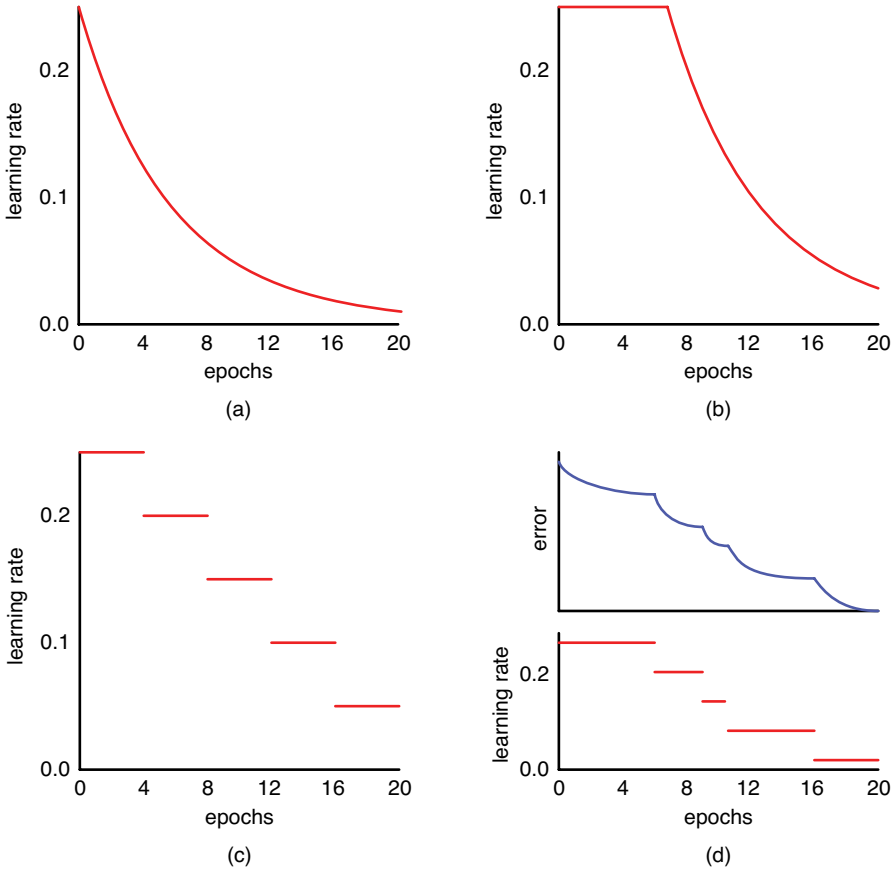


Figure 15-17: Decay schedules for reducing the size of the learning rate over time. (a) Exponential decay, where the learning rate is reduced after every epoch. (b) Delayed exponential decay. (c) Interval decay, where the learning rate is reduced after every fixed number of epochs (here, 4). (d) Error-based decay, where the learning rate is reduced when the error stops dropping.

Yet another option is to monitor the error of our network. As long as the error is going down, we stick with whatever learning rate we have now. When the network stops learning, we apply the decay so it can take smaller steps and hopefully work its way into a deeper part of the error landscape. This *error-based decay* is shown in Figure 15-17(d).

We can easily cook up a lot of alternatives, such as applying decay only when the error decreases by a certain amount or certain percentage, or perhaps updating the learning rate by just subtracting a small value from it rather than multiplying it by a number close to 1 (as long as we stop at some

positive value—if the learning rate went to 0, the system would stop learning, and if the learning rate went negative, the system would increase the error, rather than decrease it).

We can even increase the learning rate over time if we want. The *bold driver* method looks at how the total loss is changing after each epoch (Orr 1999a; Orr 1999b). If the error is going down, then we *increase* the learning rate a little, say 1 percent to 5 percent. The thinking is that if things are going well, and the error is dropping, we can take big steps. But if the error has gone up by more than just a little, then we slash the learning rate, cutting it by half. This way we can stop any increases immediately, before they can carry us too far away from the decreasing error we were previously enjoying.

Learning rate schedules have the drawback that we have to pick their parameters in advance (Darken, Chang, and Moody 1992). We think of these parameters as *hyperparameters*, just like the learning rate itself. Most deep learning libraries offer routines that automatically search ranges of values for us to help us find the best values of one or more hyperparameters.

Generally speaking, simple strategies for adjusting the learning rate usually work well, and most machine-learning libraries let us pick one of them with little fuss (Karpathy 2016).

Some kind of learning rate reduction is a common feature in most machine learning systems. We want to learn quickly in the early stages, moving in big steps over the landscape, looking for the lowest minimum we can find. Then we reduce the learning rate, enabling us to gradually take smaller steps and land in the very lowest part of whatever valley we've found.

It's natural to wonder if there's a way to control the learning rate that doesn't depend on a schedule that we set up before we start training. Surely, we can somehow detect when we're near a minimum, or in a bowl, or bouncing around, and automatically adjust the learning rate in response.

An even more interesting question is to consider that maybe we don't want to apply the same learning rate adjustments to all of our weights. It would be nice to be able to tune our updates so that each weight is learning at a rate that works best for it.

Let's look at some variations on gradient descent that address those ideas.

Updating Strategies

In the following sections, we compare the performance of three different ways to enhance gradient descent. In these examples, we use a small, but real, two-class classification problem.

Figure 15-18 shows our familiar dataset of two fuzzy crescent moons. The classes for these points are shown by color. These 300 samples are our reference data for the rest of this chapter.

In order to compare different networks, we need to train them until the error has reached a minimum, or seems to have stopped improving. We can show the results of our training with plots that graph the error after each epoch. Because of the wide variation in algorithms, the number of epochs in these graphs vary over a large range.

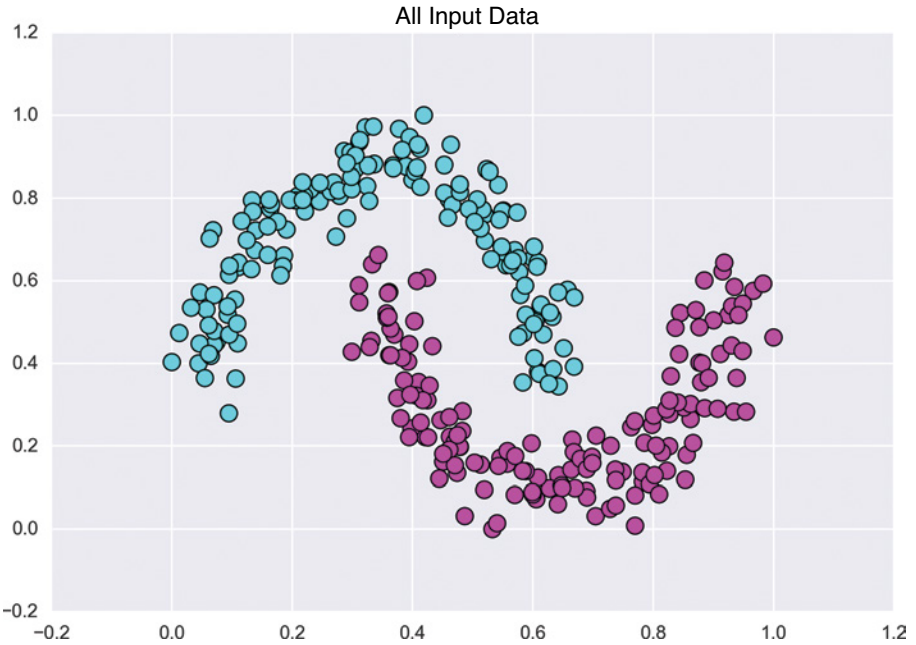


Figure 15-18: The data we use for the rest of this chapter. The 300 points form two classes of 150 points each.

To classify our points, we'll use a neural network with three fully connected hidden layers (of 12, 13, and 13 nodes), and an output layer of 2 nodes, giving us the probability for each of the two classes. We'll use ReLU on each hidden layer, and softmax at the end. Whichever class has the larger probability at the output is taken as our network's prediction. For consistency, when we need a constant learning rate, we use a value of $\eta = 0.01$. The network is shown in Figure 15-19.

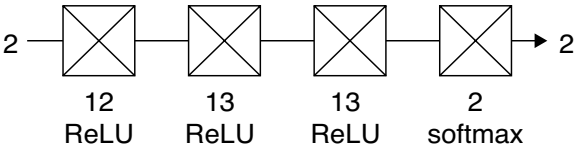


Figure 15-19: Our network of four fully connected layers

Batch Gradient Descent

Let's begin by updating the weights just once per epoch, after we've evaluated all the samples. This is *batch gradient descent* (also called *epoch gradient descent*). In this approach, we run the entire training set through our system, accumulating the errors. Then we update all of the weights once using the combined information from all the samples.

Figure 15-20 shows the error from a typical training run using batch gradient descent.

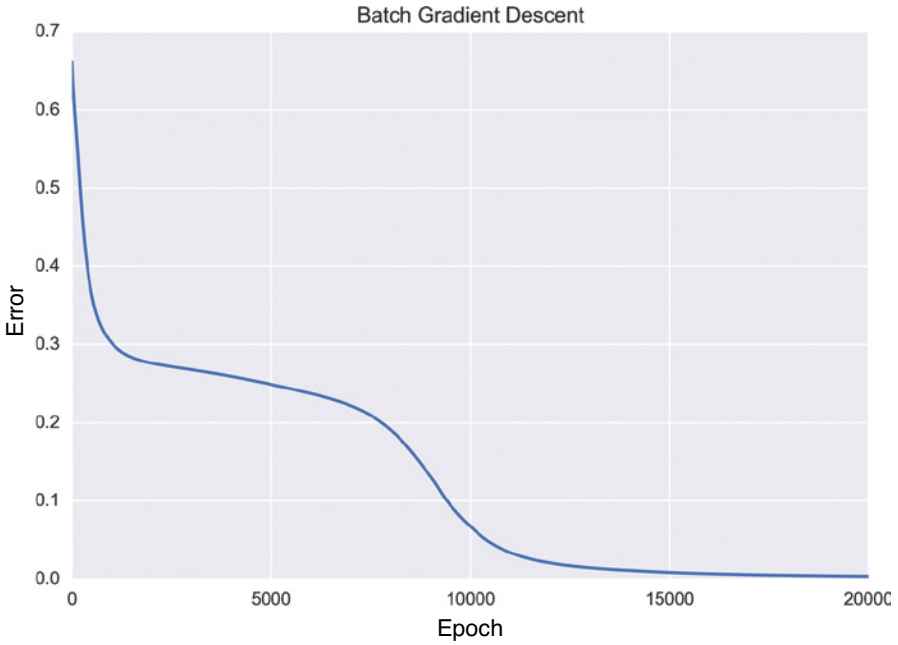


Figure 15-20: The error for a training run using batch gradient descent

The broad features are reassuring. The error drops quite a bit at the beginning, suggesting that the network is starting on a steep section of the error surface. Then the curve becomes much shallower. The error surface here might be a nearly flat region of a shallow saddle or a region that's nearly a plateau but has just a bit of slope to it, because the error does continue to drop slowly. Eventually the algorithm finds another steep region and follows it all the way down to 0.

Batch gradient descent looks very smooth, but to get down to near 0 error for this network and data requires about 20,000 epochs, which can take a long time. Let's get a closer look at what happens from one epoch to the next by zooming in on the first 400 epochs, shown in Figure 15-21.

It seems that batch gradient descent really is moving smoothly. That makes sense, because it's using the error from all the samples on each update.

Batch gradient descent usually produces a smooth error curve, but it has some issues in practice. If we have more samples than can fit in our computer's memory, then the costs of *paging*, or retrieving data from slower storage media, can become substantial enough to make training impractically slow. This can be a problem in some real situations when we work with enormous datasets of millions of samples. It can take a great deal of time to read samples from slower memory (or even a hard drive) over and over. There are solutions to this problem, but they can involve a lot of work.

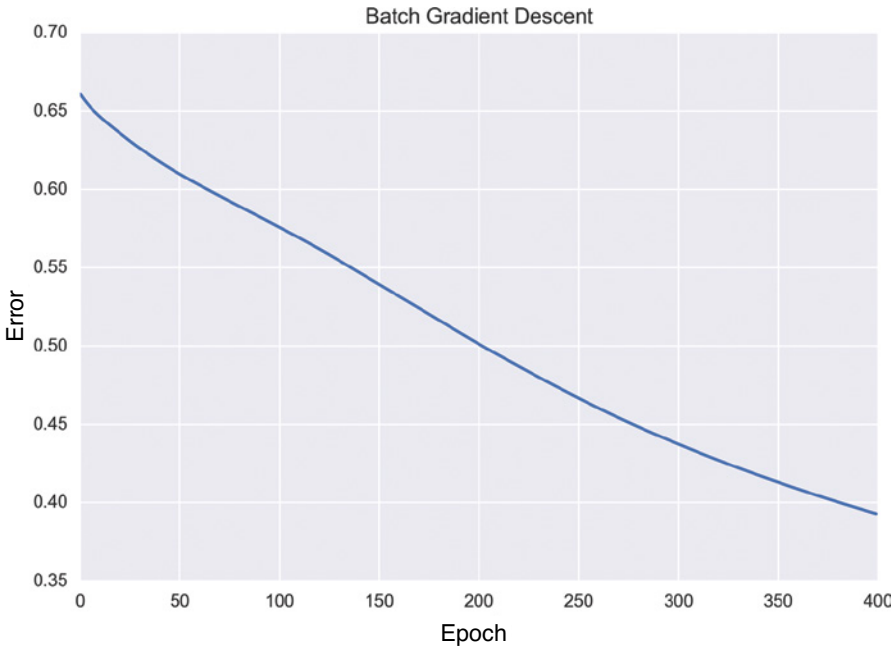


Figure 15-21: A close-up of the first 400 epochs of batch gradient descent shown in Figure 15-20

Closely related to this memory issue is that we must keep all the samples around and available so that we can run through them over and over, once per epoch. We sometimes say that batch gradient descent is an *offline algorithm*, meaning that it works strictly from information that it has stored and has access to. We can imagine disconnecting the computer from all networks, and it could continue to learn from all of our training data.

Stochastic Gradient Descent

Let's go to the other extreme and update our weights after every sample. This is called *stochastic gradient descent*, or, more commonly, just *SGD*. Recall that the word *stochastic* is roughly a synonym for *random*. This word is used because we present the network with the training samples in a random order, so we can't predict how the weights are going to change from one sample to the next.

Since we update after every sample, our dataset of 300 samples requires us to update the weights 300 times over the course of each epoch. This is going to cause the error to jump around a lot as each sample pulls the weights one way and then another. Since we're only plotting the error on an epoch-by-epoch basis, we don't see this small-scale wiggling. But we still see a lot of variation epoch by epoch.

Figure 15-22 shows the error of our network learning from this data using SGD.

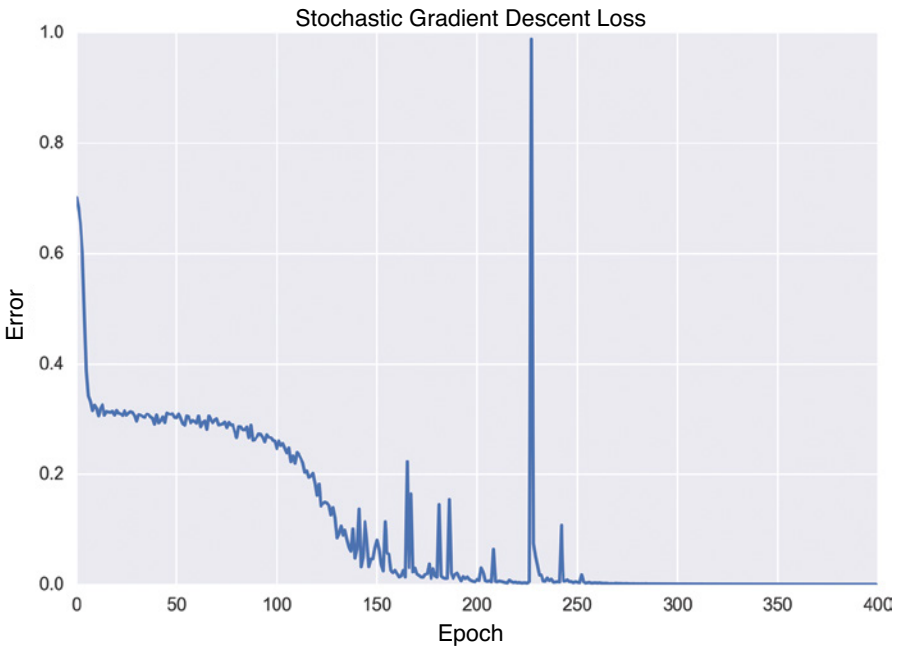


Figure 15-22: Stochastic gradient descent, or SGD

The graph has the same general shape as the one for batch gradient descent in Figure 15-20, which makes sense since both training runs use the same network and data.

The huge spike at around epoch 225 shows just how unpredictable SGD can be. Something in the sequencing of the samples and the way the network's weights were updated caused the error to soar from nearly 0 to nearly 1. In other words, it went from finding the right class for almost every sample to being dead wrong on almost every sample, and then back to being right again (though this recovery took a few epochs, as shown by the small curve to the right of the spike). If we were watching the errors as learning progresses, we might be inclined to stop the training session at the spike. If we use an automatic algorithm to watch the error, it may also stop it there. Yet just a few epochs after that spike, the system has recovered and we're back to nearly 0. The algorithm has definitely earned the word *stochastic* in its name.

We can see from the plot that SGD got down to about 0 error in just 400 epochs. We cut off Figure 15-22 after that, since the curve stayed at 0 from then on. Compare this to the roughly 20,000 epochs required by batch gradient descent in Figure 15-20. This increase in efficiency over batch gradient descent is typical (Ruder 2017).

But let's compare apples to apples. How many times did each algorithm update the weights? Batch gradient descent updates the weights after each batch, so the 20,000 epochs means it did 20,000 updates. SGD does an update after every one of our 300 samples. So in 400 epochs it performed $300 \times 400 = 120{,}000$ updates, six times more than batch gradient descent. The moral is that the amount of time we actually spend waiting for results isn't completely predicted by the number of epochs, since the time per epoch can vary considerably.

We call SGD an *online algorithm*, because it doesn't require the samples to be stored or even to be consistent from one epoch to the next. It just handles each sample as it arrives and updates the network immediately.

SGD produces noisy results, as we can see in Figure 15-22. This is both good and bad. The upside of this is that SGD can jump from one region of the error surface to another as it searches for minima. But the downside is that SGD can overshoot a deep minimum and spend its time searching around inside of some valley with a larger error. Reducing the learning rate over time definitely helps with the jumping problem, but the progress is still typically noisy.

Noise in the error curve can be a problem because it makes it hard for us to know when the system is learning and when it starts overfitting. We can look at a sliding window of many epochs, but we may only know that we've overshot the minimum error long after it happened.

Mini-Batch Gradient Descent

We can find a nice middle ground between the extremes of batch gradient descent, which updates once per epoch, and stochastic gradient descent, which updates after every sample. This compromise is called *mini-batch gradient descent*, or sometimes *mini-batch SGD*. Here, we update the weights after some fixed number of samples has been evaluated. This number is almost always considerably smaller than the batch size (the number of samples in the training set). We call this smaller number the *mini-batch size*, and a set of that many samples drawn from the training set is a *mini-batch*.

The mini-batch size is frequently a power of 2 between about 32 and 256, and often it is chosen to fully use the parallel capabilities of our GPU, if we have one. But that's just for efficiency purposes. We can use any size of mini-batch that we like.

Figure 15-23 shows the results of using a mini-batch of 32 samples.

This is indeed a nice blend of the two algorithms. The curve is smooth, like batch gradient descent, but not perfectly so. It drops down to 0 in about 5,000 epochs, between the 400 needed by SGD and the 20,000 of batch gradient descent. Figure 15-24 shows a close-up of the first 400 steps.

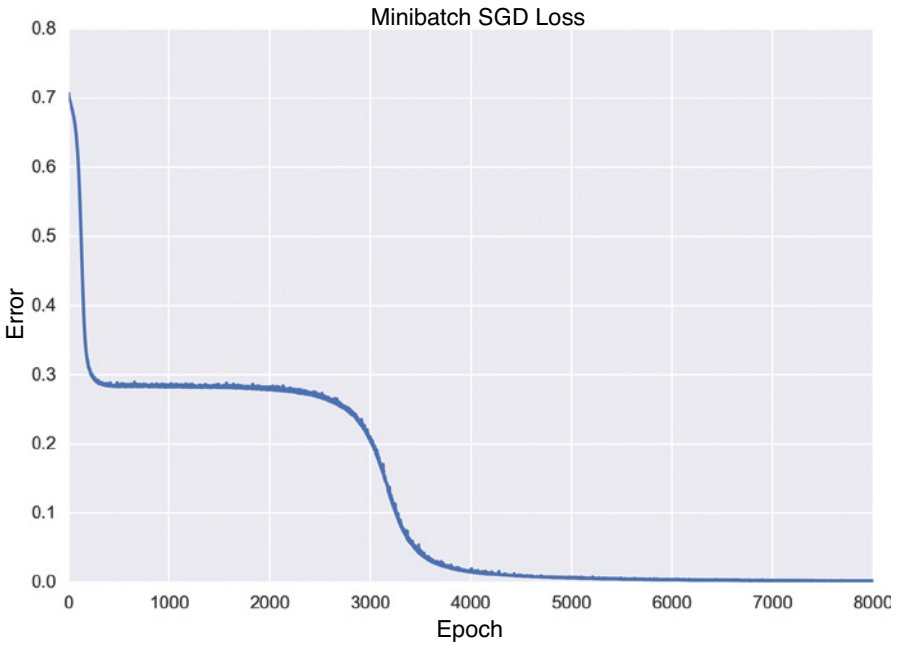


Figure 15-23: Mini-batch gradient descent

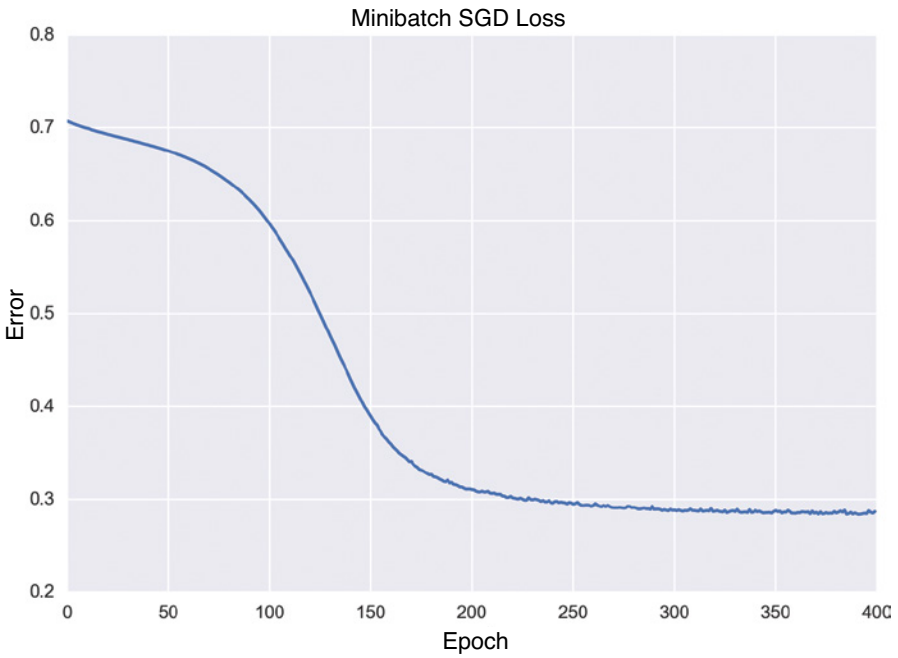


Figure 15-24: A close-up of the first 400 epochs of Figure 15-23, showing the deep plunge at the very beginning of training

How many updates did mini-batch SGD perform? We have 300 samples, and we used a mini-batch size of 32, so there are 10 mini-batches per epoch. (Ideally, we'd like the mini-batches to precisely divide the size of the input, but in practice, we can't control the size of our datasets. This often leaves us with a partial mini-batch at the end.) So 10 updates per epoch, times 5,000 epochs, gives us 50,000 updates. This is also nicely between the 20,000 updates of batch gradient descent and the 120,000 updates of SGD.

Mini-batch gradient descent is less noisy than SGD, which makes it attractive for tracking the error. The algorithm can take advantage of huge efficiency gains by using the GPU for calculations, evaluating all the samples in a mini-batch in parallel. It's faster than batch gradient descent, and more attractive in practice than SGD.

For all these reasons, mini-batch SGD is popular in practice, with "plain" SGD and batch gradient descent being used relatively infrequently. In fact, most of the time when the term *SGD* is used in the literature, or even just *gradient descent*, it's understood that the authors mean mini-batch SGD (Ruder 2017). To make things a little more confusing, the term *batch* is often used instead of *mini-batch*. Because epoch-based gradient descent is used so rarely these days, references to batch gradient descent and batches almost always refer to mini-batch gradient descent and mini-batches.

Gradient Descent Variations

Mini-batch gradient descent is an important algorithm, but it's not perfect. Let's review some of the challenges of mini-batch gradient descent, and a few ways to address them. Following convention, from here on we refer to mini-batch gradient descent as SGD. (The organization of this section is inspired by Ruder 2017.)

Our first challenge is to specify what value of the learning rate η we want to use, which is notoriously hard to pick ahead of time. As we've seen, a value that's too small can result in long learning times and getting stuck in shallow local minima, but a value that's too big can cause us to overshoot deep local minima and then get stuck bouncing around inside a minimum when we do find it. If we try to avoid the problem by using a decay schedule to change η over time, we still have to pick the starting value of η and then the schedule's hyperparameters as well.

We also have to pick the size of the mini-batch. This is rarely an issue, because we usually choose whatever value produces calculations that are most closely matched to the structure of our GPU or other hardware.

Let's consider some improvements. Right now, we're updating all the weights with a one-update-rate-fits-all approach. Instead, we can find a unique learning rate for each weight in the system so we're not just moving it in the best direction, but we're moving it by the best amount. We'll see examples of this in the following pages.

Another improvement begins with the recognition that sometimes when the error surface forms a saddle, the surface can be shallow in all directions, so locally, it's almost (but not quite) a plateau. This can slow our progress to

an excruciating crawl. Research has shown that deep learning systems often have plenty of saddles in their error landscapes (Dauphin et al. 2014). It would be nice if there was a way to get unstuck in these situations, or better yet, to avoid getting stuck in them in the first place. The same thing goes for plateaus: we'd like to avoid getting stuck in the flat regions where the gradient drops to 0. To do so, we want to avoid the regions where the gradient drops to 0, except of course for the minima we're seeking.

Let's look at some variations of gradient descent that address these issues.

Momentum

Let's consider two weights at the same time. We can plot their values on an XY plane, and above them show the error that results from training the system with those values for those weights. Let's think of our error surface as a landscape. Now we can picture our task of minimizing error as following a drop of water that's looking for the lowest point.

Figure 15-25 repeats a figure from Chapter 5 that shows an example of this way of thinking about the training process.

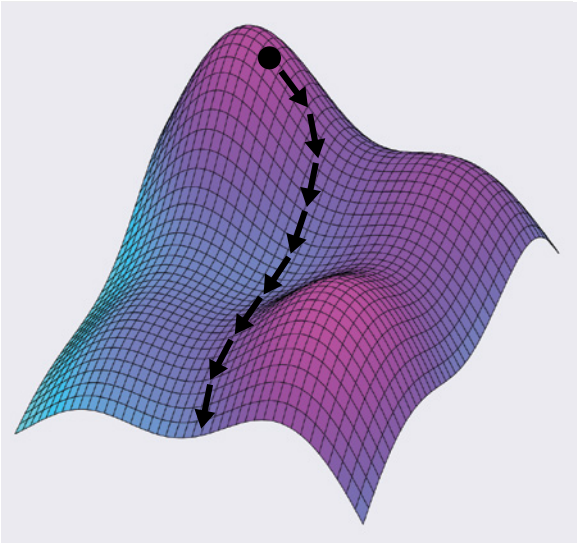


Figure 15-25: A drop of water rolling down an error surface. This is a repeat of a figure from Chapter 5.

Instead of water, let's think of this as a little ball rolling down the error surface. We know from the physical world that a real ball rolling down a hill in this way has some *inertia*, which describes its resistance to a change in its motion. If it's rolling along in a given direction at a certain speed, it will continue to move that way unless something interferes with it.

A related idea is the ball's *momentum*, which is a bit more abstract from a physical point of view. Although they're distinct ideas, sometimes deep learning discussions casually refer to inertia as momentum, and the algorithm we're about to look at uses that language.

This idea is what keeps the ball in Figure 15-25 moving across the plateau after it has come down from the peak and passed into the saddle near the middle of the figure. If the ball's motion was determined strictly by the gradient, when it hit the plateau near the middle of the figure, it would stop (or if it was a near-plateau, the ball would slow to a crawl). But the ball's momentum (or more properly, its inertia) keeps it rolling onward.

Suppose we're near the left side of Figure 15-26. As we roll down the hill, we reach the plateau starting at around −0.5.

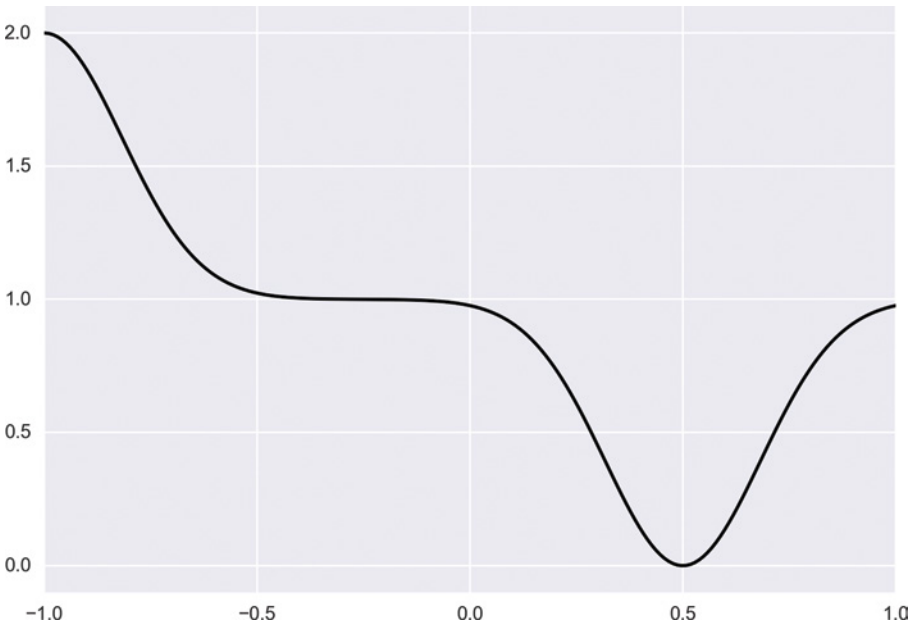


Figure 15-26: An error curve with a plateau between a hill and valley

With regular gradient descent, we stop on the plateau since the gradient is zero, as shown in the left of Figure 15-27. But if we include some momentum, as shown on the right, the ball keeps going for a while. It does slow down, but if we're lucky, it continues to roll far enough to find the next valley.

The technique of *momentum gradient descent* (Qian 1999) is based on this idea. For each step, once we calculate how much we want each weight to change, we add in a small amount of its change from the previous step. If the change on a given step is 0, or nearly 0, but we had some larger change on the last step, we use some of that prior motion now, which pushes us along over the plateau.

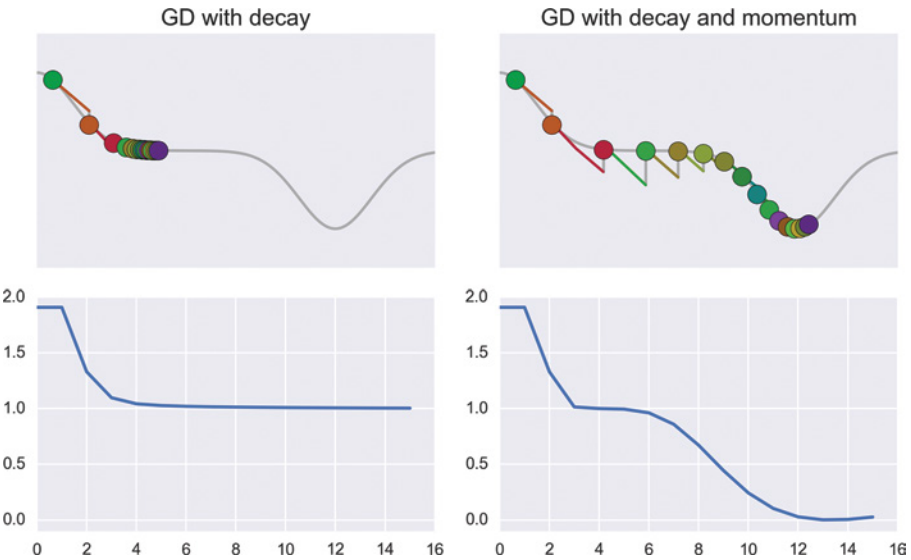


Figure 15-27: Gradient descent on the error curve of Figure 15-26. Left: Gradient descent with decay. Right: Gradient descent with decay and momentum.

Figure 15-28 shows the idea visually.

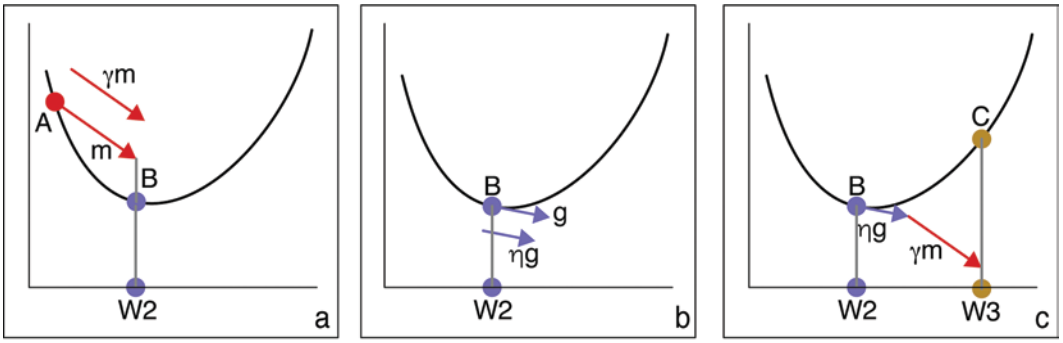


Figure 15-28: Finding the step for gradient descent with momentum

We suppose that some weight had an error A. We updated that weight to value w2, with an error B. We now want to find the next value for the weight, w3, which will have error C. To find C, we find the change that we applied to point A. That is, we find the previous motion applied to A. This is the momentum, labeled m, shown in Figure 15-28(a).

We multiply the momentum, m, by a scaling factor usually referred to with the lowercase Greek letter γ (gamma). Sometimes this is called the *momentum scaling factor*, and it's a value from 0 to 1. Multiplying m by this value gives us a new arrow γm that points in the same direction as m but

is the same length or shorter. We then find the scaled gradient, ηg, at B, as we did before, shown in Figure 15-28(b). Now we have all we need. We add together the scaled momentum, γm, and the scaled gradient, ηg, to B, which we do graphically by placing the tail of γm at the head of ηg, as in Figure 15-28(c).

Let's apply this rule and see how the weight and error change over time. Figure 15-29 shows our symmetrical valley from before, and sequential steps of training. In this figure, we use both an exponential decay schedule and momentum. This is just like our sequence from Figure 15-15, but now the change applied to each step also includes momentum, or a scaled version of the change from the previous step. We can see this by looking at the two lines that emerge from each point (one for the gradient, the other for the momentum). This total then becomes the new change.

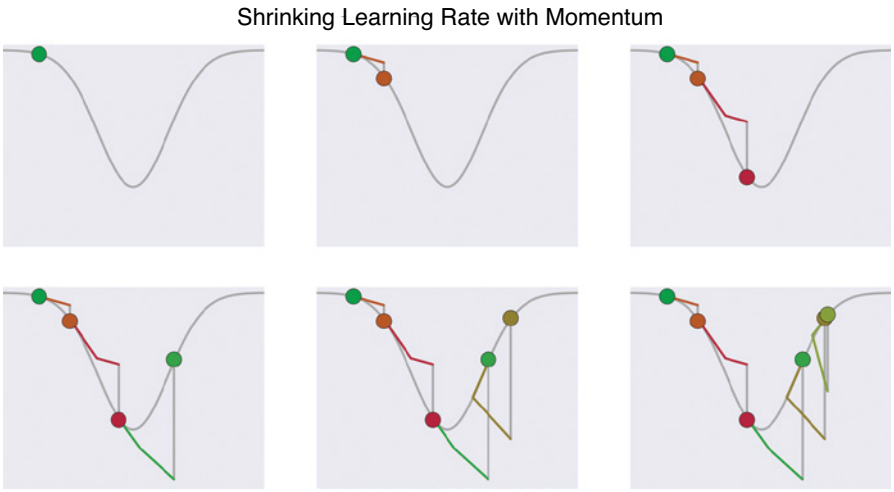


Figure 15-29: Learning with both an exponential decay schedule and momentum

On each step, we first find the gradient and multiply it by the current value of the learning rate η, as before. Then we find the previous change, scale it by γ, and add both of those changes to the current position of the weight. That combination gives us the change in this step.

Figure 15-30 shows a close-up of the sixth step in the grid, along with the error at each point along the way.

An interesting thing happened here: when the ball reached the right side of the valley, it continued to roll up, even though the gradients pointed down. That's just what we'd expect of a real ball. We can see it slowing down, and then eventually it comes back down the slope, overshooting the bottom, but by less than before, then slowing and coming back down again, and so on.

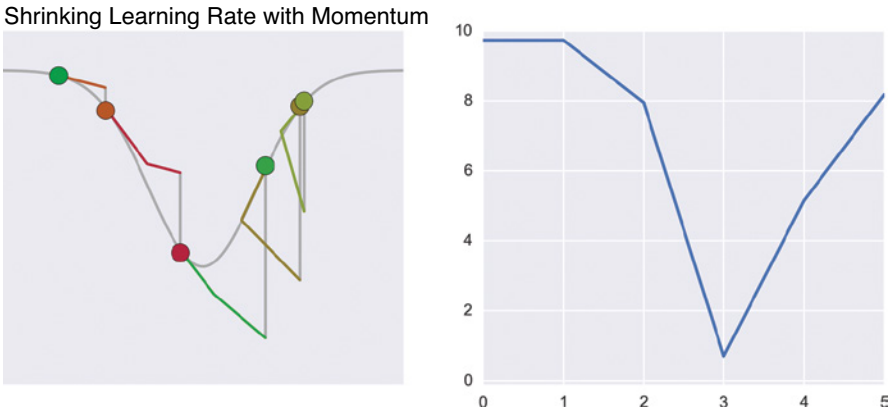


Figure 15-30: The final step in Figure 15-29, along with the error for each point

If we use too much momentum, our ball can fly right up the other side and out of the bowl altogether, but if we use too little momentum, our ball may not get across the plateaus it encounters along the way. Figure 15-31 shows our error curve from Figure 15-26. Here we used trial and error to find a value of γ to scale the momentum so that our ball gets through the plateau but can still settle into the minimum at the bottom of the bowl.

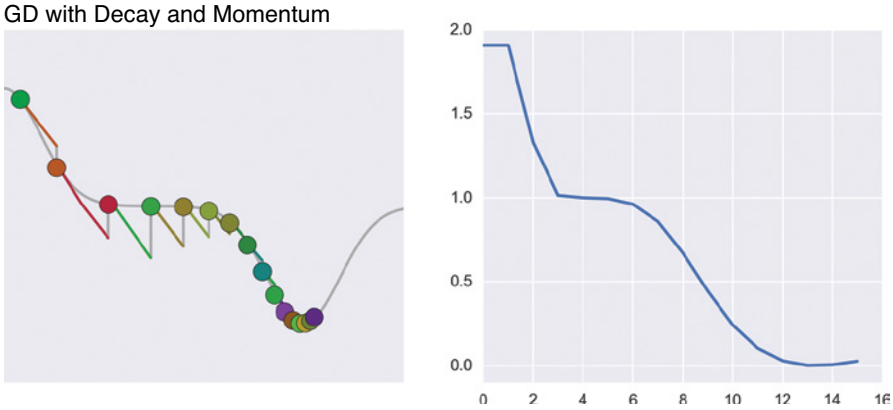


Figure 15-31: Using enough momentum to cross a plateau, but not so much that the ball is unable to settle nicely into the bottom of the minimum

Finding the right amount of momentum to use is another task where we need to use our experience and intuition, along with trial and error, to help us understand the behavior of our specific network and the data we're working with. We can also search for it using hyperparameter searching algorithms.

To put this all together, we find the gradient, scale it by the current learning rate η, add in the previous change scaled by γ, and that gives us our new position. If we set γ to 0, then we add in none of the last step, and we have "normal" (or "vanilla") gradient descent. If γ is set to 1, then we add in the entirety of the last change. Often we use a value of around 0.9. In Figures 15-29 and Figure 15-31, we set gamma to 0.7 to better illustrate the process.

Figure 15-32 shows the result of 15 steps of learning with both learning rate decay and momentum. The ball starts on the left, rolls down and then far up the right side, then it rolls down again and rolls up the left side, and so on, climbing a little less each time.

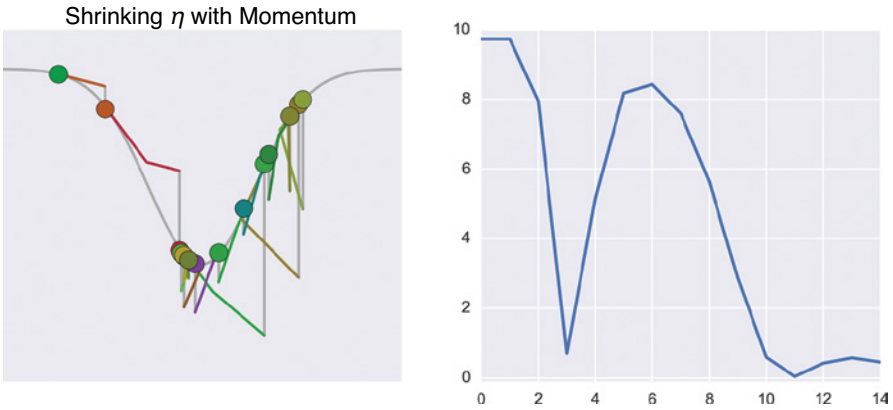


Figure 15-32: Learning with momentum and a decaying learning rate for 15 points

Momentum helps us get over flat plateaus and out of shallow places in saddles. It has the additional benefit of helping us zip down steep slopes, so even with a small learning rate, we can pick up some efficiency.

Figure 15-33 shows the error for a training run on our dataset of Figure 15-18 consisting of two crescent moons.

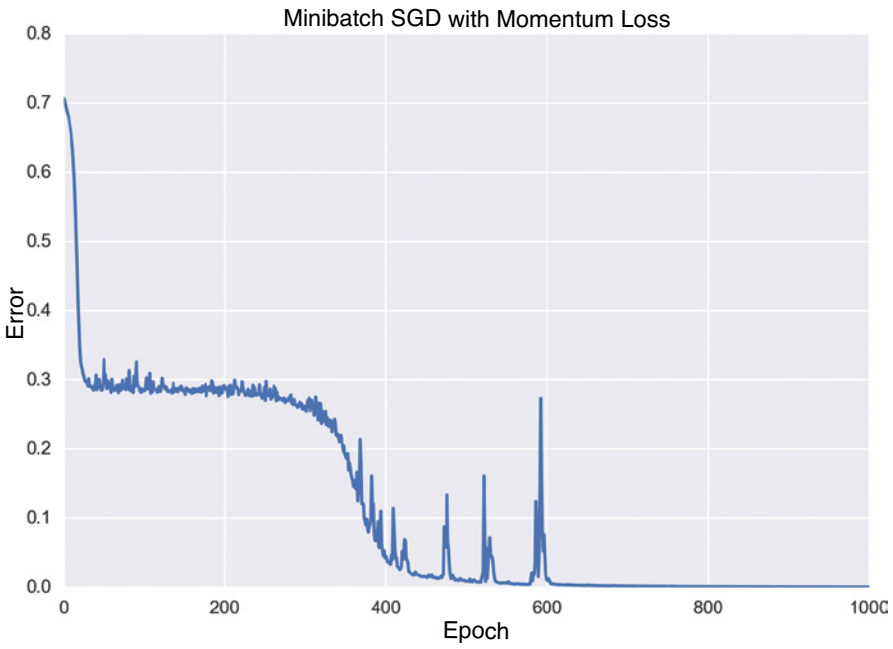


Figure 15-33: Error curve for training with our two-crescent data using mini-batch gradient descent with momentum. We got to zero error in a little more than 600 epochs.

Here we're using mini-batch gradient descent (or SGD) with momentum. It's noisier than the mini-batch curve of Figure 15-23 because the momentum sometimes carries us past where we want to be, causing a spike in the error. The error when using mini-batch SGD alone in Figure 15-23 for our data took about 5,000 epochs to reach about 0 error. With momentum, we get there in a little over 600 epochs. Not bad!

Momentum clearly helps us learn more quickly, which is a great thing. But momentum brings us a new problem: choosing the momentum value γ. As we mentioned, we can pick this value using experience and intuition or use a hyperparameter search for the value that gives us the best results.

Nesterov Momentum

Momentum let us reach into the past for information to help us train. Now let's reach into the future. The key idea is that instead of using only the gradient at the location where we currently are, we also use the gradient at the location where we expect that we're *going to be*. Then we can use some of that "gradient from the future" to help us now.

Because we can't really predict the future, we estimate where we're going to be on the next step and use the gradient there. The thinking is that if the error surface is relatively smooth, and our estimate is pretty good, then the gradient we find at our estimated next position is close to the gradient where we'd actually end up if we just moved using standard gradient descent, with or without momentum.

Why is it useful to use the gradient from the future? Suppose we're rolling down one side of a valley and approaching the bottom. On the next step, we overshoot the bottom and end up somewhere on the other wall. As we saw before, momentum carries us up that wall for a few steps, slowing as we lose momentum, until we turn around and come back down. But if we can predict that we'll be on the far side, we can include some of the gradient at that point in our calculations now. So instead of moving so far to the right and up the hill, that leftward push from the future causes us to move by a little less distance, so we don't overshoot so far and end up closer to the bottom of the valley.

In other words, if the next move we're going to make is in the same direction as the last one, we take a larger step now. If the next move is going to move us backward, we take a smaller step.

Let's break this down into steps so we don't get mixed up between estimates and realities. Figure 15-34 shows the process.

As before, we imagine that we started with our weight at position A, and after the most recent update, we ended up at B, as shown in Figure 15-34(a). As with momentum, we find the change applied at point A to bring us to B (the arrow m) and we scale that by γ.

Now comes the new part, starting in Figure 15-34(b). Rather than finding the gradient at B, we first add the scaled momentum to B to get the "predicted" error P. This is our guess for where we will end up on the error

surface after the next step. As shown Figure 15-34(c), we find the gradient g at point P and scale it as usual to get ηg. Now we find the new point C in Figure 15-34(d) by adding the scaled momentum γm and the scaled gradient ηg to B.

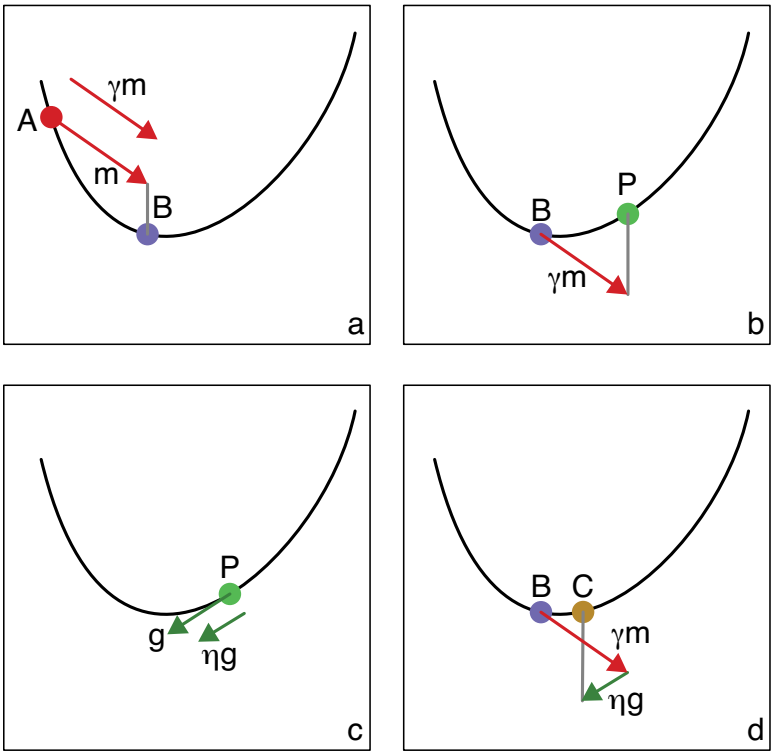


Figure 15-34: Gradient descent with Nesterov momentum

Notice we're not using the gradient at point B at all. We just combine a scaled version of the momentum that got us to B and a scaled version of the gradient at our predicted point P.

Notice too that the point C in Figure 15-34(d) is closer to the bottom of the bowl than point P, where we'd have ended up with normal momentum. By looking into the future and seeing that we'd be on the other side of the valley, we are able to use that left-pointing gradient to prevent rolling far up the far side.

In honor of the researcher who developed this method, it's called *Nesterov momentum*, or the *Nesterov accelerated gradient* (Nesterov 1983). It's basically a souped-up version of the momentum technique we saw earlier. Though we still have to pick a value for γ, we don't have to pick any new parameters. This is a nice example of an algorithm that gives us increased performance without requiring more work on our end.

Figure 15-35 shows the result of Nesterov momentum for 15 steps.

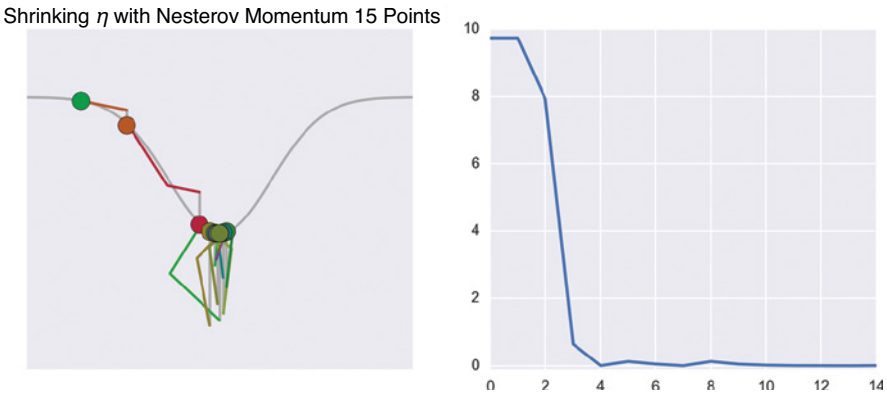


Figure 15-35: Running Nesterov momentum for 15 steps. It finds the bottom of the valley in about seven steps and then stays there.

Figure 15-36 shows the error curve for our standard test case using Nesterov momentum. This uses the exact same model and parameters as the momentum-only results in Figure 15-33, but it's both less noisy and more efficient, getting down to about 0 error at roughly epoch 425, rather than the roughly 600 required by regular momentum alone.

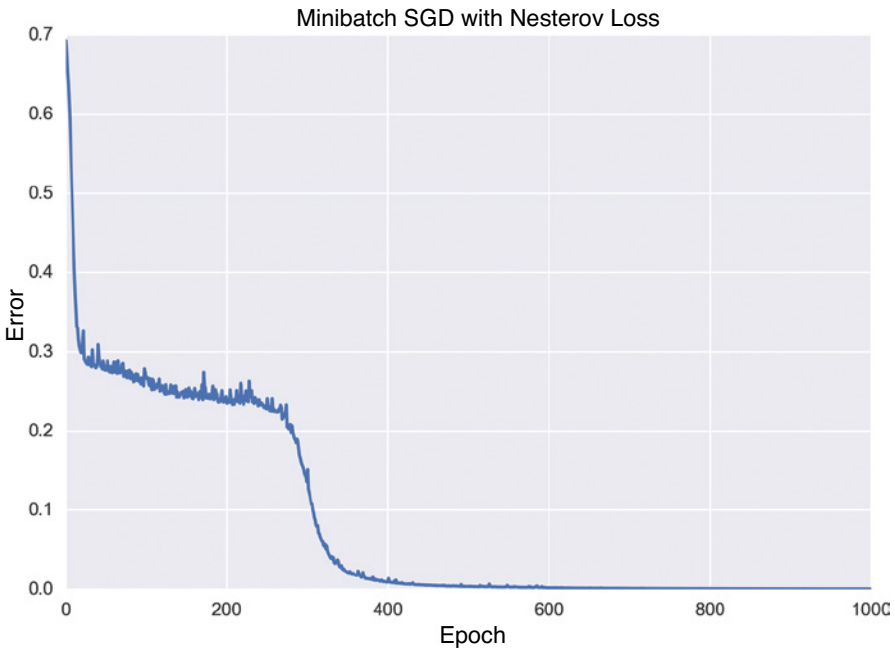


Figure 15-36: Error for mini-batch SGD with Nesterov momentum. The system reaches zero error around epoch 600. The graph shows 1,000 epochs.

Any time we use momentum it's worth considering Nesterov momentum instead. It requires no additional parameters from us, but it usually learns more quickly and with less noise.

Adagrad

We've seen two types of momentum that help push us through plateaus and reduce overshooting. We've been using the same learning rate when we update all the weights in our network. Earlier in this chapter, we mentioned the idea of using a learning rate η that's tailored individually for each weight.

Several related algorithms use this idea. Their names all begin with *Ada*, standing for "adaptive."

Let's start with an algorithm called *Adagrad*, which is short for *adaptive gradient learning* (Duchi, Hazan, and Singer 2011). As the name implies, the algorithm adapts (or changes) the size of the gradient for each weight. In other words, Adagrad gives us a way to perform learning-rate decay on a weight-by-weight basis. For each weight, Adagrad takes the gradient that we use in that update step, squares it, and adds that into a running sum for that weight. Then the gradient is divided by a value derived from this sum, giving us the value that's then used for the update.

Because each step's gradient is squared before it's added in, the value that's added into the sum is always positive. As a result, this running sum gets larger and larger over time. To keep it from growing out of control, we divide each change by that growing sum, so the changes to each weight get smaller and smaller over time.

This sounds a lot like learning rate decay. As time goes on, the changes to the weights get smaller. The difference here is that the slowdown in learning is being computed uniquely for each weight based on its history.

Because Adagrad is effectively automatically computing a learning rate for every weight on the fly, the learning rate we use to kick things off isn't as critical as it was for earlier algorithms. This is a huge benefit, since it frees us from the task of fine-tuning that error rate. We often set the learning rate η to a small value like 0.01 and let Adagrad handle things from there.

Figure 15-37 shows the performance of Adagrad on our test data.

This has the same general shape as most of our other curves, but it takes a very long time to get to 0. Because the sum of the gradients gets larger over time, eventually we'll find that dividing each new gradient by a value related to that sum gives us gradients that approach 0. The increasingly small updates are why the error curve for Adagrad descends so very slowly as it tries to get rid of that last remaining error.

We can fix that without too much work.

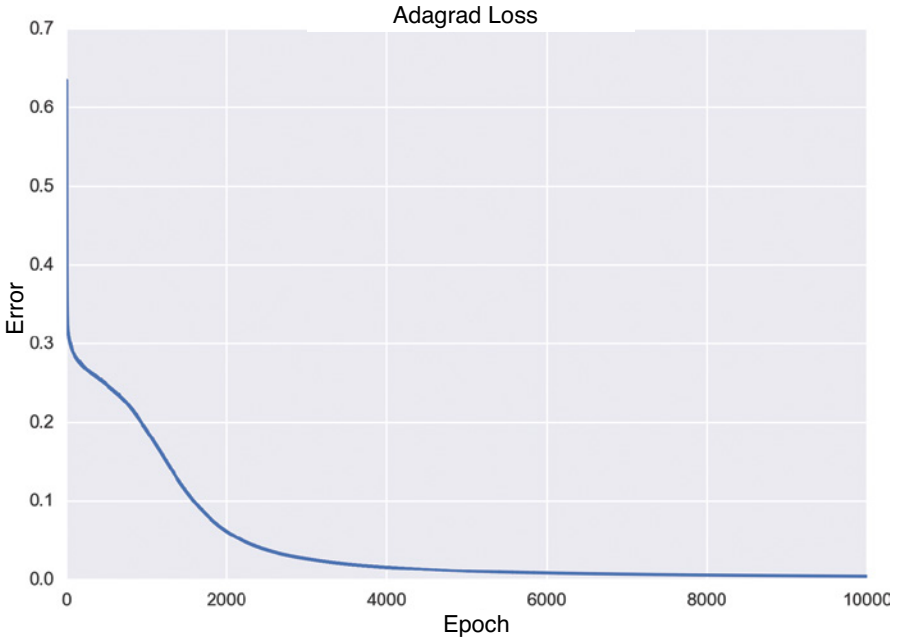


Figure 15-37: The performance of Adagrad on our test setup

Adadelta and RMSprop

The problem with Adagrad is that the gradient we apply to each weight for its update step just keeps getting smaller and smaller. That's because the running sum just gets larger and larger.

Instead of summing up all the squared gradients since the beginning of training, suppose we keep a *decaying sum* of these gradients. We can think of this as a running list of the most recent gradients for each weight. Each time we update the weights, we tack the new gradient onto the end of the list and drop the oldest one off the start. To find the value we use to divide the new gradient, we add up all the values in the list, but we first multiply them all by a number based on their position in the list. Recent values get multiplied by a large value, while the oldest ones get multiplied by a very small value. This way our running sum is most heavily determined by recent gradients, though it is influenced to a lesser degree by the older gradients (Ruder 2017).

In this way, the running sum of the gradients (and thus the value we divided new gradients by) can go up and down based on the gradients we've applied recently.

This algorithm is called *Adadelta* (Zeiler 2012). The name comes from "adaptive," like Adagrad, and the *delta* refers to the Greek letter δ (delta), which mathematicians often use to refer to change. This algorithm adaptively changes how much the weights are updated on each step using each one's weighted running sum.

Since Adadelta adjusts the learning rates on the weights individually, any weight that's been on a steep slope for a while will slow down so it

doesn't go flying off, but when that weight is on a flatter section, it's allowed to take bigger steps.

Like Adagrad, we often start the learning rate at a value around 0.01, and then let the algorithm adjust it from then on.

Figure 15-38 shows the results of Adadelta on our test setup.

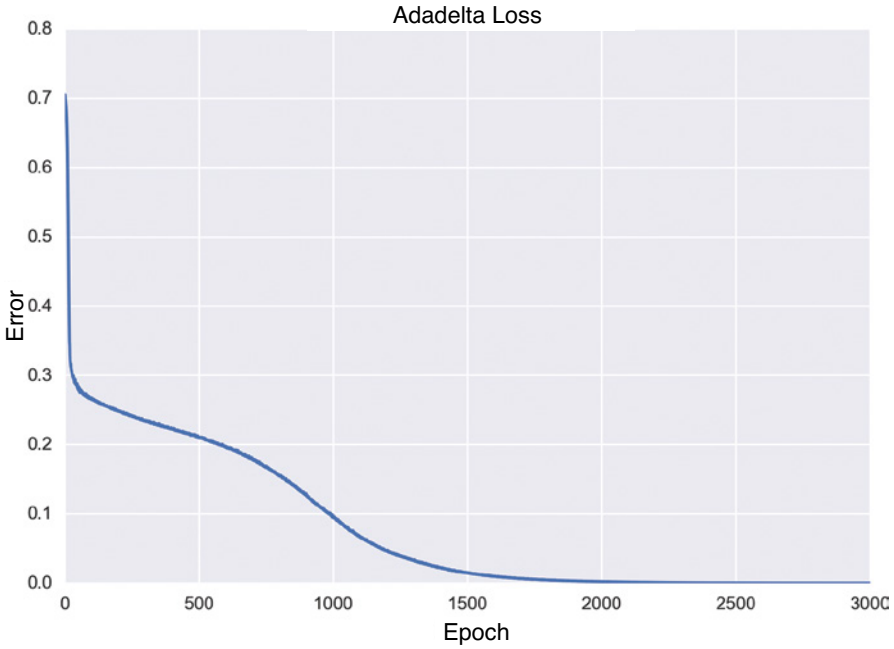


Figure 15-38: The results of training with Adadelta on our test data

This compares favorably to Adagrad's performance in Figure 15-37. It's nice and smooth and reaches 0 at around epoch 2,500, much sooner than Adagrad's 8,000 epochs.

Adadelta has the downside of requiring another parameter, which is unfortunately also called gamma (γ). It's roughly related to the parameter γ used by the momentum algorithms, but they're sufficiently different that it's best to consider them distinct ideas that happen to have been given the same name. The value of γ here tells us how much we scale down the gradients in our history list over time. A large value of γ "remembers" values from farther back than smaller values and will let them contribute to the sum. A smaller value of γ just focuses on recent gradients. Often we set this γ to around 0.9.

There's actually another parameter in Adadelta, named with the Greek letter ε (epsilon). This is a detail that's used to keep the calculations numerically stable. Most libraries will set this to a default value that's carefully selected by the programmers to make things work as well as possible, so it should never be changed unless there's a specific need.

An algorithm that's very similar to Adadelta, but that uses slightly different mathematics, is called *RMSprop* (Hinton, Srivastava, and Swersky 2015). The name comes from the fact that it uses a root-mean-squared

operation, often abbreviated RMS, to determine the adjustment that is added (or *propagated*, hence the "prop" in the name) to the gradients.

RMSprop and Adadelta were invented around the same time, and work in similar ways. RMSprop also uses a parameter to control how much it "remembers," and this parameter, too, is named γ. Again, a good starting value is around 0.9.

Adam

The previous algorithms share the idea of saving a list of squared gradients with each weight. They then create a scaling factor by adding up the values in this list, perhaps after scaling them. The gradient at each update step is divided by this total. Adagrad gives all the elements in the list equal weight when it builds its scaling factor, while Adadelta and RMSprop treat older elements as less important, and thus they contribute less to the overall total.

Squaring the gradient before putting it into the list is useful mathematically, but when we square a number, the result is always positive. This means that we lose track of whether that gradient in our list was positive or negative, which is useful information to have. So, to avoid losing this information, we can keep a second list of the gradients without squaring them. Then we can use both lists to derive our scaling factor.

This is the approach of an algorithm called *adaptive moment estimation*, or more commonly *Adam* (Kingma and Ba 2015).

Figure 15-39 shows how Adam performs.

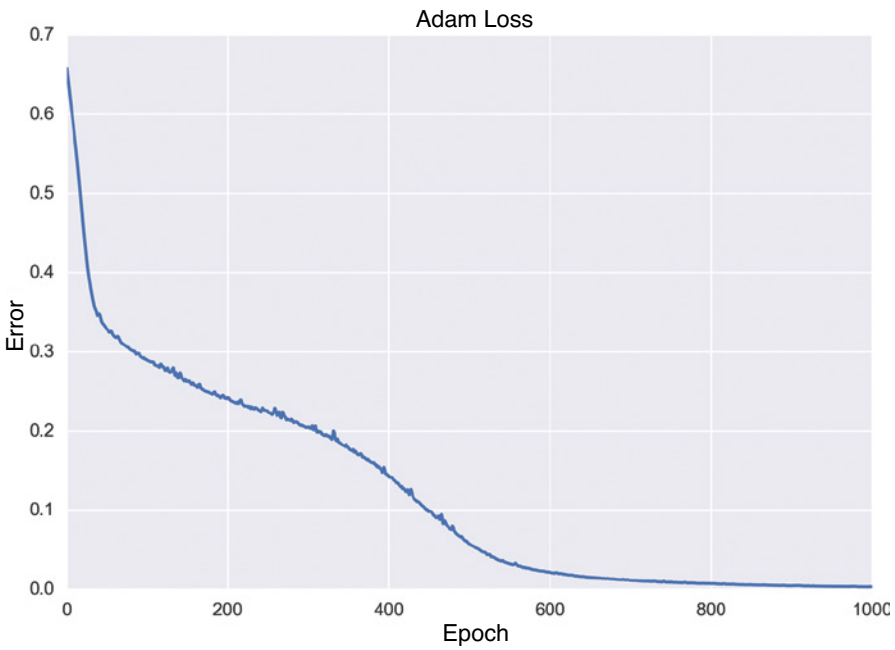


Figure 15-39: The Adam algorithm on our test set

The output is great. It's only slightly noisy and hits about 0 error at around epoch 900, much sooner than Adagrad or Adadelta. The downside is that Adam has two parameters, which we must set at the start of learning. The parameters are named for the Greek letter β (beta) and are called "beta 1" and "beta 2," written $\beta 1$ and $\beta 2$. The authors of the paper on Adam suggest setting $\beta 1$ to 0.9, and $\beta 2$ to 0.999, and these values indeed often work well.

Choosing an Optimizer

This has not been a complete list of all the optimizers that have been proposed and studied. There are many others, with more coming all the time, and each has its own strengths and weaknesses. Our goal was to give an overview of some of the most popular techniques and to understand how they achieve their speedups.

Figure 15-40 summarizes our two-moon results for SGD with Nesterov momentum and the three adaptive algorithms of Adagrad, Adadelta, and Adam.

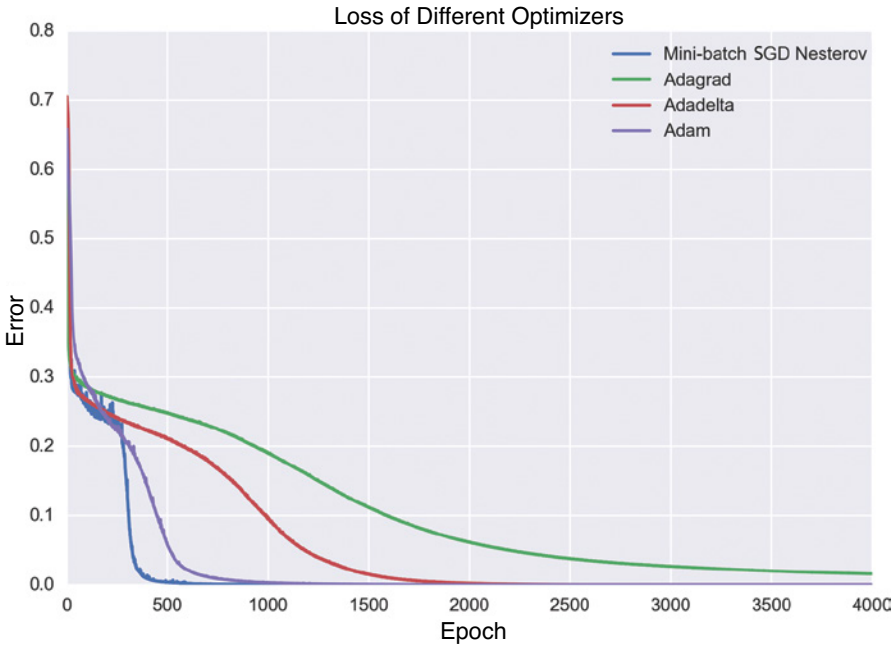


Figure 15-40: The loss, or error, over time for four of the algorithms just covered. This graph shows only the first 4,000 epochs.

In this simple test case, mini-batch SGD with Nesterov momentum is the clear winner, with Adam coming in a close second. In more complicated situations, the adaptive algorithms typically perform better.

Across a wide variety of datasets and networks, the final three adaptive algorithms that we discussed (Adadelta, RMSprop, and Adam) often perform very similarly (Ruder 2017). Studies have found that Adam does a slightly better job than the others in some circumstances, so that's usually a good place to start (Kingma and Ba 2015).

Why are there so many optimizers? Wouldn't it be wise to find the best one and stick with that? It turns out that not only do we not know of a "best" optimizer, but there can't be a best optimizer for all situations. No matter what optimizer we put forth as the "best," we can prove that it's always possible to find some situation in which another optimizer would be better. This result is famously known by its colorful name, the *No Free Lunch Theorem* (Wolpert 1996; Wolpert and Macready 1997). This guarantees us that no optimizer will always perform better than any other.

Note that the No Free Lunch Theorem doesn't say that all optimizers are equal. As we've seen in our tests in this chapter, different optimizers do perform differently. The theorem only tells us that no one optimizer will *always* beat the others.

Though no one optimizer is the best choice for all possible training situations, we can find the best optimizer for any specific combination of network and data. Most deep learning libraries offer routines that carry out an automated search that can try out multiple optimizers and run through multiple parameter choices for each one. Whether we choose our optimizer and its values by ourselves or as the result of a search, we need to keep in mind that the best choices can vary from one network and set of data to the next. As soon as we make a big change to either, we should consider checking to see if a better optimizer would give us more efficient training. As a practical guide, many people start out with Adam, using its default parameters.

Regularization

No matter what optimizer we choose, our network can suffer from overfitting. As we discussed in Chapter 9, overfitting is a natural result of training for too long. The problem is that the network learns the training data so well that it becomes tuned to just that data and performs poorly on new data once it's released.

Techniques that delay the onset of overfitting are called *regularization* methods. They allow us to train for more epochs before overfitting has too great an impact, which means our networks have more training time in which to improve their performance.

Dropout

A popular regularization method is called *dropout*. It is usually applied in a deep network in the form of a *dropout layer* (Srivastava et al. 2014). The dropout layer is called an *accessory layer* or a *supplemental layer*, because it doesn't do any computation of its own. We call it a layer, and draw it as one, because it's convenient conceptually, and lets us include dropout in

drawings of networks. But we don't consider it a real layer (hidden or otherwise), and we don't count it when we describe how many layers make up a particular network.

Dropout is a placeholder that tells the network to run an algorithm on the previous layer. It's also only active during training. When the network is deployed, dropout layers are disabled or removed.

The job of the dropout layer is to temporarily disconnect some of the neurons on the previous layer. We give it a parameter that describes the percentage of neurons that should be affected, and at the start of each batch, it randomly chooses that percentage of neurons on the preceding layer and temporarily disconnects their inputs and outputs from the network. Since they're disconnected, these neurons don't participate in any forward calculations, they're not included in backprop, and the weights coming into them are not updated by the optimizer. When the batch is done and the rest of the weights have been updated, the chosen neurons and all of their connections are restored.

At the start of the next batch, the layer again chooses a new random set of neurons and temporarily removes those, repeating the process for each epoch. Figure 15-41 shows the idea graphically.

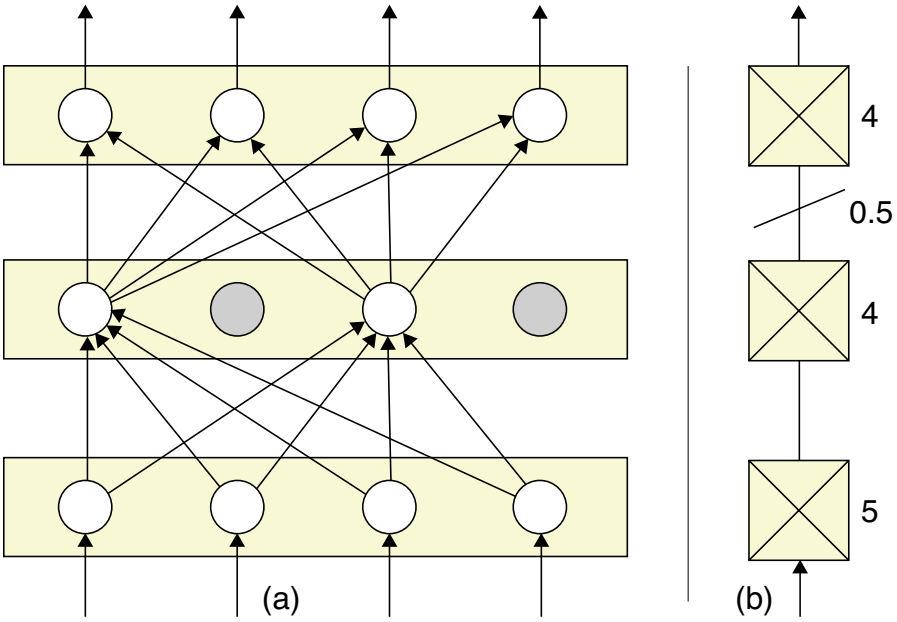


Figure 15-41: Dropout. (a) 50 percent of the four neurons in the middle layer (in gray) are chosen to be disconnected before the batch is evaluated. (b) Our schematic for a single dropout layer is a diagonal slash. To the right, we indicate the proportion of neurons that are selected for disconnection. Since dropout applies to its preceding layer, in this example, we apply it to the middle of the three fully connected layers.

Dropout delays overfitting by preventing any neurons from overspecializing and dominating. Suppose that one neuron in a photo classification system gets highly specialized to detect the eyes of cats. That's useful for

recognizing picture of cats' faces, but useless for all the other photographs the system might be asked to classify. If all the neurons in a network also specialize at finding just one or two features in the training data, then they can perform beautifully on that data because they spot the idiosyncratic details that they're trained to locate. But the system as a whole will then perform badly when presented with new data that's missing the precise cues those neurons became specialized for.

Dropout helps us avoid this kind of specialization. When a neuron is disconnected, the remaining neurons must adjust to pick up the slack. Thus, the specialized neuron is freed up to perform a more generally useful task, and we've delayed the onset of overfitting. Dropout helps us put off overfitting by spreading around the learning among all the neurons.

Batchnorm

Another regularization technique is called *batch normalization*, often referred to simply as *batchnorm* (Ioffe and Szegedy 2015). Like dropout, batchnorm can be implemented as a layer without neurons. Unlike dropout, batchnorm actually does perform some computation, though there are no parameters for us to specify.

Batchnorm modifies the values that come out of a layer. This might seem strange, since the whole purpose of training is to get our neurons to produce output values that lead to good results. Why would we want to modify those outputs?

Recall that many of our activation functions, such as leaky ReLU and tanh, have their greatest effect near 0. To get the most benefit from those functions, we need the numbers flowing into them to be in a small range centered around 0. That's what batchnorm does by scaling and shifting all the outputs of a layer together. Because batchnorm moves the neuron outputs into a small range near 0, we're less prone to seeing any neuron learning one specific detail and producing a huge output that swamps all the other neurons, and thus we are able to delay the onset of overfitting. Batchnorm scales and shifts all the values coming out of the previous layer over the course of an entire mini-batch in just this way. It learns the parameters for this scaling and shifting along with the weights in the network so they take on the most useful values.

We apply batchnorm before the activation function so that the modified values will fall in the region of the activation function where they are affected the most. In practice, this means we place no activation function on the neurons going into batchnorm (or if we must specify a function, it's the linear activation function, which has no effect). Those values go into batchnorm, and then they're fed into the activation function we want to apply.

The process is illustrated in Figure 15-42. Our icon for a regularization step like batchnorm is a black disc inside a circle, suggesting that the values in the circle are transformed into a smaller region. In later chapters, we'll see other, similar regularization steps for which we'll use the same icon. The text (or a nearby label) identifies which variety of regularization is applied.

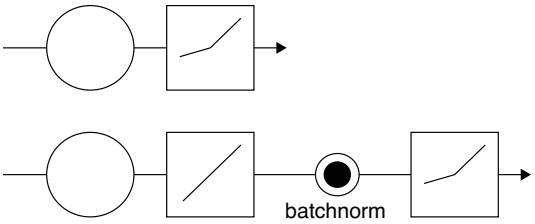


Figure 15-42: Applying a batchnorm layer. Top: A neuron followed by a leaky ReLU activation function. Bottom: The same neuron with batchnorm. The activation function is replaced with the linear function, followed by batchnorm (represented by a circle with a black disc inside) and then the leaky ReLU.

Like dropout, batchnorm defers the onset of overfitting, allowing us to train longer.

Summary

Optimization is the process of adjusting the weights so that our network learns. The core idea begins with the gradient for every weight. We follow that gradient to direct us to a lower point on the error surface, hence the name gradient descent. The most important value in this process is the learning rate. A common technique is to reduce the learning rate over time, according to a decay schedule.

We covered several efficient optimization techniques. We can adjust the weights after every epoch (batch gradient descent), after every sample (stochastic gradient descent, or SGD), or after mini-batches of samples (mini-batch gradient descent or mini-batch SGD). Mini-batch gradient descent is by far the most common technique, and the convention in the field is to refer to it simply as SGD. We can improve the efficiency of every type of gradient descent by using momentum. We can also improve learning by computing a custom, adaptive learning rate for every weight over time with an algorithm such as Adam. Lastly, to prevent overfitting, we can use a regularization technique such as dropout or batchnorm.

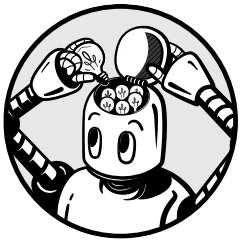
Deep networks that are made up of fully connected layers can do some amazing things. But if we create our layers by structuring the neurons in different ways and add a little bit of supporting computation, their power increases significantly. In the next few chapters, we'll look at these new layers and how they can be used to classify, predict, and even generate images, sounds, and more.

PART IV

BEYOND THE BASICS

16

CONVOLUTIONAL NEURAL NETWORKS



This chapter is all about a deep learning technique called *convolution*. Among its uses, convolution has become the standard method for classifying, manipulating, and generating images. Convolution is easy to use in deep learning because it can be easily encapsulated in a *convolution layer* (also called a *convolutional layer*). In this chapter, we look at the key ideas behind convolution and the related techniques we use to make convolution work in practice. We will see how to arrange a series of these operations to create a hierarchy of operations, which turns a series of simple operations into a powerful tool.

In order to stay specific, in this chapter we focus our discussion of convolution on working with images. Models that use convolution have been spectacularly successful in this domain. For example, they excel at basic classification tasks like determining if an image is a leopard or a cheetah, or a planet or a marble. We can recognize the people in a photograph (Sun, Wang, and Tang 2014); detect and classify different types of skin cancers (Esteva et al. 2017); repair image damage like dust, scratches, and blur (Mao, Shen, and Yang 2016); and classify people's age and gender from their photos (Levi and Hassner 2015). Convolution-based networks are also useful in many other applications, such as natural language processing (Britz 2015), where we can work out the structure of sentences (Kalchbrenner, Grefenstette, and Blunsom 2014) or classify sentences into different categories (Kim 2014).

Introducing Convolution

In deep learning, images are 3D tensors, with a height, width, and number of *channels*, or values per pixel. A grayscale image has only one value per pixel, and thus only one channel. A color image stored as RGB has three channels (with values for red, green, and blue). Sometimes people use the terms *depth* or *fiber size* to refer to the number of channels in a tensor. Unfortunately, *depth* is also used to refer to the number of layers in a deep network, and *fiber size* has not caught on widely. To avoid confusion, we always refer to the three dimensions of an image (and related 3D tensors) as height, width, and channels. Using our deep learning terminology, each image we provide to the network for processing is a sample. Each pixel in an image is a feature.

When a tensor moves through a series of convolution layers, it often changes in width, height, and number of channels. If a tensor happens to have 1 or 3 channels, we can think of it as an image. But if a tensor has, say, 14 or 512 channels, it's probably best not to think of it as an image any more. This suggests that we shouldn't refer to individual elements of the tensor as *pixels*, which is an image-centric term. Instead, we call them *elements*. Figure 16-1 shows these terms visually.

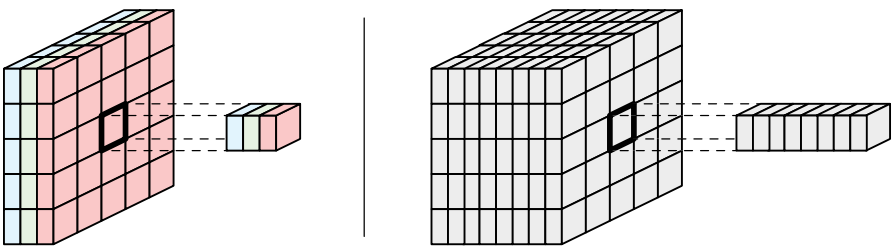


Figure 16-1: Left: When our tensor has one or three channels, we can say that it's made up of pixels. Right: For tensors with any number of channels, we call each slice through the channels an element.

A network in which the convolution layers play a central role is usually called a *convolutional neural network*, *convnet*, or *CNN*. Sometimes people also say *CNN network* (an example of "redundant acronym syndrome syndrome" [Memmott 2015]).

Detecting Yellow

To kick off our discussion of convolution, let's consider processing a color image. Each pixel contains three numbers: one each for red, green, and blue. Suppose we want to create a grayscale output that has the same height and width as our color image, but where the amount of white in each pixel corresponds to the amount of yellow in its input pixel.

For simplicity, let's assume our RGB values are numbers from 0 to 1. Then a pixel that's pure yellow has red and green values of 1, and a blue value of 0. As the red and green values decrease, or the blue value increases, the pixel's color shifts away from yellow.

We want to combine each input pixel's RGB values into a single number from 0 to 1 that represents "yellowness," which is the output pixel's value. Figure 16-2 shows one way to do this.

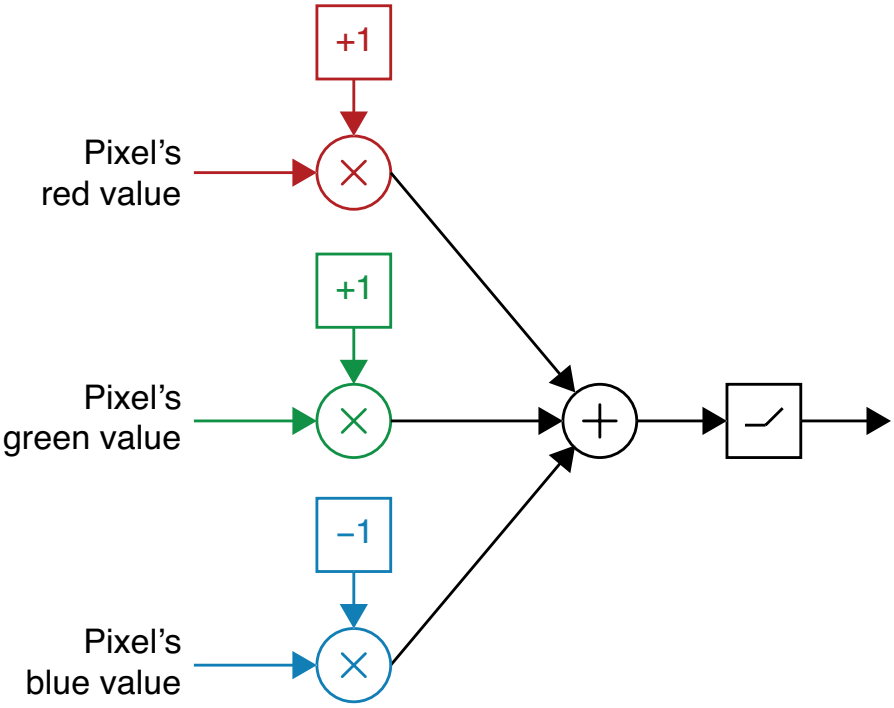


Figure 16-2: Representing our yellow detector as a simple neuron

This sure looks familiar. It has the same structure as an artificial neuron. When we interpret Figure 16-2 as a neuron, +1, +1, and –1 are the three weights, and the numbers associated with the color values are the three inputs. Figure 16-3 shows how to apply this neuron to any pixel in an image.

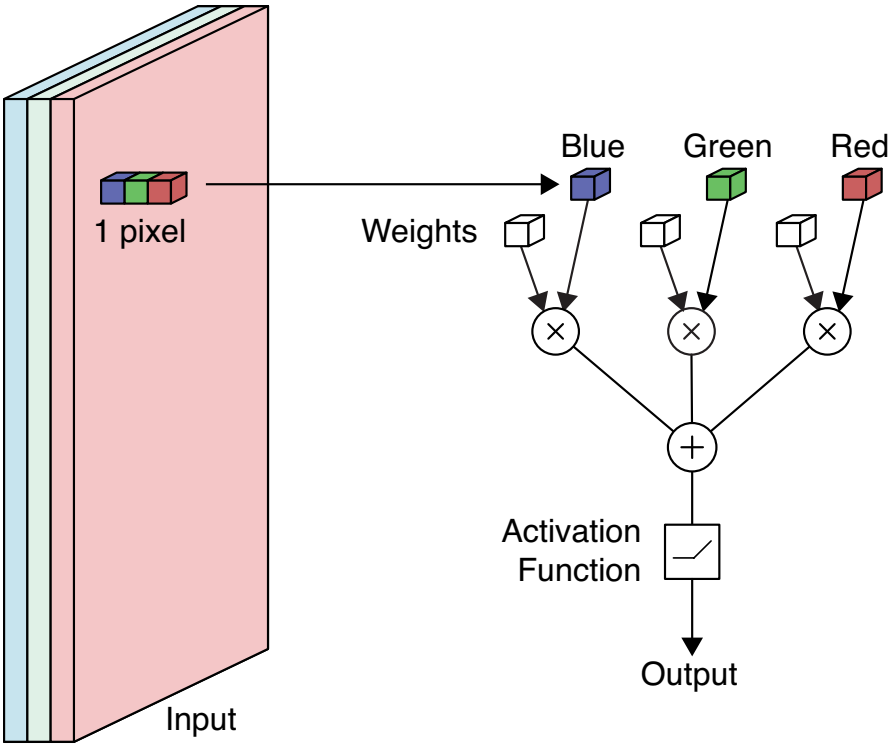


Figure 16-3: Applying our neuron in Figure 16-2 to a pixel in an image

We can apply this operation to every pixel in the input, creating a single output value for every pixel. The result is a new tensor with the same width and height as the input, but only one channel, as shown in Figure 16-4.

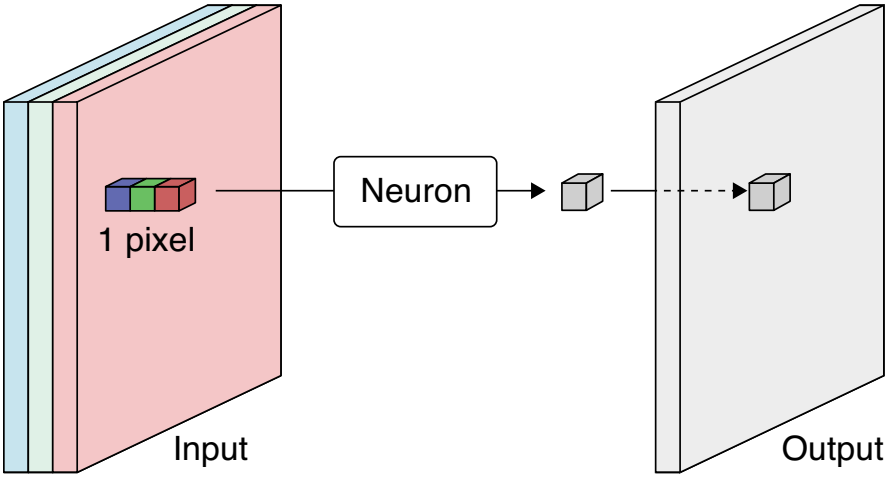


Figure 16-4: Applying our neuron in Figure 16-3 to each pixel in the input produces an output tensor with the same width and height, but only one channel.

We often imagine applying the neuron to the upper-left pixel, then moving it one step at a time to the right until we reach the end of the row, then repeating this for the next row, and the next, until we reach the bottom right pixel. We say that we're *sweeping* the neuron over the input, or *scanning* the input.

Figure 16-5 shows the result of this process on a picture of a yellow frog. As we intended, the more yellow that's present in each input pixel, the more white we see in its corresponding output. We say that the neuron is *identifying* or *detecting* yellow in the input.



Figure 16-5: An application of our yellow-finding operation. The image on the right runs from black to white, depending on the yellowness of the corresponding source pixel in the left image.

Of course there's nothing special about yellow. We can build a little neuron to detect any color. When we use a neuron in this way, we often say that it is *filtering* the input. In this context, the weights are sometimes collectively called the *filter values* or just the *filter*. Inheriting language from their mathematical roots, the weights are also called the *filter kernel* or just the *kernel*. It's also common to refer to the entire neuron as a filter. Whether the word *filter* refers to a neuron, or specifically to its weights, is usually clear from context.

This operation of sweeping the filter over the input corresponds to a mathematical operation called *convolution* (Oppenheim and Nawab 1996). We say that the right side of Figure 16-5 is the result of convolution of the color image with the yellow-detecting filter. We also say that we *convolve* the image with the filter. Sometimes we combine these terms and refer to a filter (whether an entire neuron, or just its weights) as a *convolution filter*.

Weight Sharing

In the last section, we imagined sweeping our neuron over the input image, performing exactly the same operation at every pixel. If we want to go faster, we can create a huge grid of identical neurons and apply them to all the pixels simultaneously. In other words, we process the pixels in parallel.

In this approach, every neuron has identical weights. Rather than repeating the same weights in a separate piece of memory for every neuron, we can imagine that the weights are stored in some shared piece of memory, as in Figure 16-6. We say that the neurons are *weight sharing*.

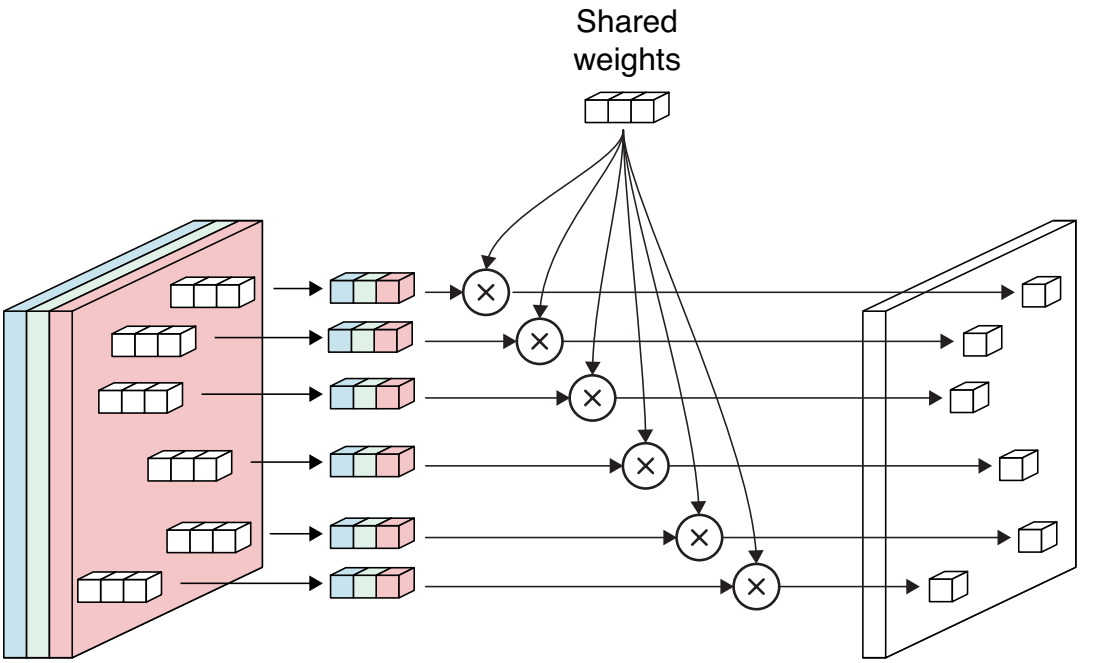


Figure 16-6: We can apply our neuron to every pixel in the input simultaneously. Each neuron uses the same weights, found in a piece of shared memory.

This lets us save on memory. In our yellow detector example, weight sharing also makes it easy to change the color we're detecting. Rather than change the weights in thousands of neurons (or more), we just change the one set in the shared memory.

We can actually implement this scheme on a GPU, which is capable of performing many identical sequences of operations at once. Weight sharing lets us save on precious GPU memory, freeing it up for other uses.

Larger Filters

So far, we've been sweeping our neuron over the image (or applying it in parallel using weight sharing), processing one pixel at a time, using only that pixel's values for input. In many situations, it's also useful to look at the pixels near the one we're processing. Usually we consider a pixel's eight immediate *neighbors*. That is, we use the values in a little three by three box that's centered on the pixel.

Figure 16-7 shows three different operations we can apply using a three by three block of numbers in this way: blurring, detecting horizontal edges, and detecting vertical edges.

To compute each image, we center the block of weights over each pixel in turn and multiply each of the nine values under it by the corresponding weight. We add up the results and use their sum as the output value for that pixel. Let's see how to implement this process with a neuron.

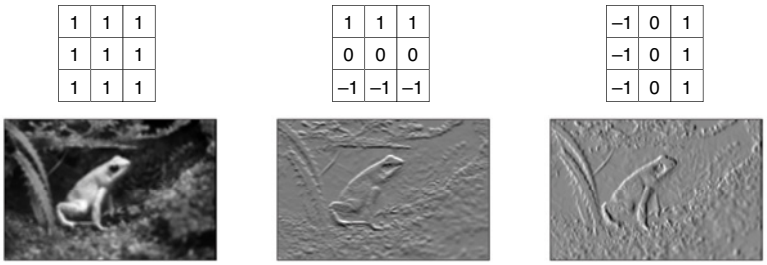


Figure 16-7: Processing a grayscale image of the frog in Figure 16-5 by moving a three by three template of numbers over the image. From left to right, we blur the image, find horizontal edges, and find vertical edges.

For simplicity, we'll stick with a grayscale input for now. We can think of the blocks of numbers in Figure 16-7 as weights, or filter kernels. In this scenario, we have a grid of nine weights that we place over a grid of nine pixel values. Each pixel value is multiplied by its corresponding weight, the results are summed up and run through an activation function, and we have our output. Figure 16-8 shows the idea.

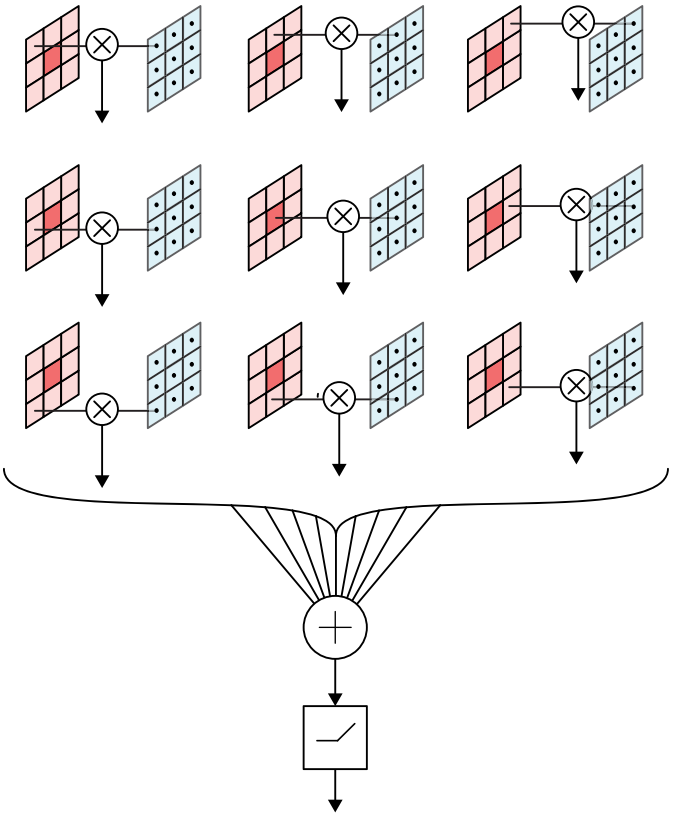


Figure 16-8: Processing a grayscale input (red) with a three by three filter (blue)

This figure shows how to process a single pixel (shown in dark red). We center the filter over the intended pixel and multiply each of the nine values in the input with its corresponding filter value. We add up all nine results and pass that sum through an activation function.

The shape of the pixels that form a neuron's input in this scheme is called that neuron's *local receptive field*, or more simply its *footprint*. In Figure 16-8, the neuron's footprint is a square, three pixels on a side. In our yellow detector, the footprint was a single pixel. When a filter's footprint is larger than a single pixel, we sometimes emphasize that quality by calling a *spatial filter*.

Note that the neuron in Figure 16-8 is just like any other neuron. It receives nine numbers as inputs, multiplies each one by its corresponding weight, adds the results together, and passes that number through an activation function. It doesn't know or care that these nine numbers are coming from a square region of the input, or even that they're coming from an image.

We apply this three by three filter to an image by convolving it with the image, just as before, by sweeping it over each pixel in turn. For each input pixel, we imagine centering the three by three grid of weights over that pixel, applying the neuron, and creating a single output value, as in Figure 16-9. We say that the pixel we're centering the filter over is the *anchor* (or the *reference point* or *zero point*).

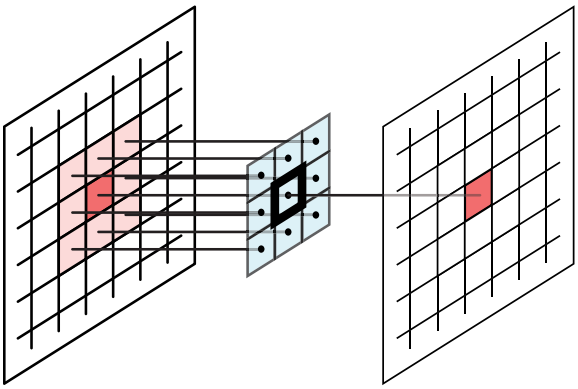


Figure 16-9: Applying a three by three filter (center) to a grayscale image (left), creating a new single-channel image (right)

We can design our filters to have footprints of any size and shape we like. In practice, small sizes are most common, since they are faster to evaluate than larger footprints. We usually use small squares with an odd number of pixels on each side (often between one and nine). Such squares let us place the anchor in the center of the footprint. This keeps everything symmetrical and easier to understand.

Let's put this into practice. Figure 16-10 shows the result of convolving a seven by seven input with a three by three filter. Note that if we were to center the filter over the input's corners or edges, the filter's footprint would extend beyond the input, and the neuron would require input values that aren't present. We address this a little later. For now, let's just limit ourselves

to those locations where the filter sits entirely on top of the image. That means that the output image is only five by five.

We motivated our discussion by looking at spatial filters that can do things like blur an image or detect edges. But why are such things useful for deep learning? To answer this, let's look at filters more closely.

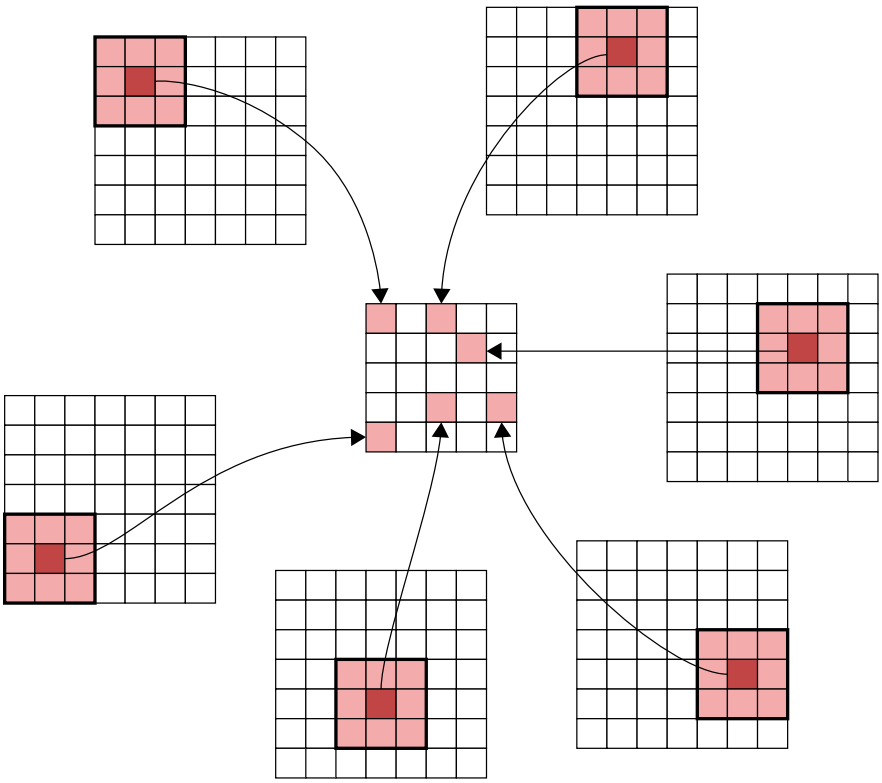


Figure 16-10: To convolve an image with a filter, we move the filter across the image and apply it at each position. We're skipping corners and edges for this figure.

Filters and Features

Some biologists who study toads think that certain cells in the animal's visual system are sensitive to specific types of visual patterns (Ewert et al. 1985). The theory is that a toad is looking for particular shapes corresponding to the creatures it likes to eat and to certain motions that those animals make. People used to think that a toad's eyes absorbed all the light that struck them, sent that mass of information to the brain, and it was the brain's job to sift among the results looking for food. The new hypothesis is that the cells in the eye are doing some early steps in this detection process (such as finding edges) all by themselves, and they only fire and pass on information to the brain if they "think" they're looking at prey.

The theory has been extended to the human visual system, where it has led to the surprising hypothesis that some individual neurons are so precisely fine-tuned that they only fire in response to pictures of specific

people. The original study that led to this suggestion showed people 87 different images, including people, animals, and landmarks. In one volunteer they found a specific neuron that only fired when the volunteer was shown a photo of the actress Jennifer Aniston (Quiroga 2005). Even more curiously, that neuron only fired when Aniston was alone, and not when she was pictured together with other people, including famous actors.

The idea that our neurons are precision pattern-matching devices is not universally accepted, but we're not doing real neuroscience and biology here. We're just looking for inspiration. And this idea of letting neurons perform detection work seems like some pretty great inspiration.

The connection to convolution is that we can use filters to simulate the cells in the toad's eyes. Our filters also pick out specific patterns and then pass on their discoveries to later filters that look for even bigger patterns. Some of the terminology we use for this process echoes terms that we've seen before. Specifically, we've been using the word *feature* to refer to one of the values contained in a sample. But in this context, the word *feature* also refers to a particular structure in an input that the filter is trying to detect, like an edge, a feather, or scaly skin. We say that a filter is *looking for* a stripe feature, or eyeglasses, or a sports car. Continuing this usage, the filters themselves are sometimes called *feature detectors*. When a feature detector has been swept over an entire input, we say that its output is a *feature map* (the word *map* in this context comes from mathematical language). The feature map tells us, pixel by pixel, how well the image around that pixel matched what the filter was looking for.

Let's see how feature detection works. In Figure 16-11 we show the process of using a filter to find short, isolated vertical white stripes in a binary image.

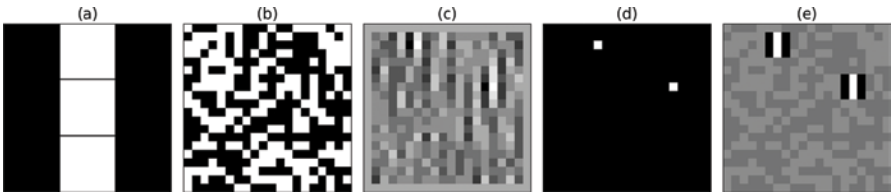


Figure 16-11: 2D pattern matching with convolution. (a) The filter. (b) The input. (c) The feature map, scaled to [0, 1] for display. (d) Feature map entries with value 3. (e) Neighborhoods of (b) around the white pixels in (d).

Figure 16-11(a) shows a three by three filter with values −1 (black) and 1 (white). Figure 16-11(b) shows a noisy input image, consisting only of black and white pixels. Figure 16-11(c) shows the result of applying the filter to each pixel in the input image (except for the outermost border). Here the values range from −6 to +3, which we scaled to [0, 1] for display. The larger the value in this image, the better the match between the filter and the pixel (and its neighborhood). A value of +3 means the filter matched the image perfectly at that pixel.

Figure 16-11(d) shows a thresholded version of Figure 16-11(c), where pixels with a value of +3 are shown in white, and all others are black. Finally, Figure 16-11(e) shows the noisy image of Figure 16-11(b) with the three by

three grid of pixels around the white pixels in Figure 16-11(d) highlighted. We can see that the filter found those places in the image where the pixels matched the filter's pattern.

Let's see why this worked. In the top row of Figure 16-12 we show our filter and a three by three patch of the image, along with the pixel-by-pixel results.

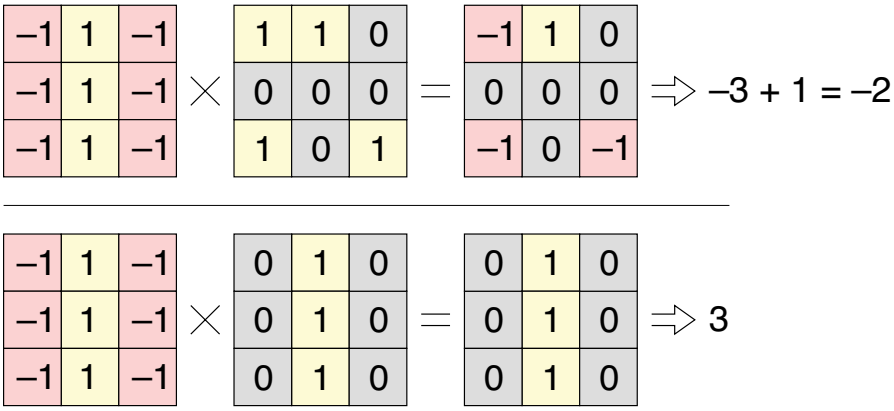


Figure 16-12: Applying a filter to two image fragments. From left to right, each row shows the filter, an input, and the result. The final number is the sum of the rightmost three by three grid.

Consider the pixels shown in the middle of the top row. The black pixels (shown in gray here), with a value of 0, don't contribute to the output. The white pixels (shown in light yellow here), with a value of 1, get multiplied by either 1 or –1, depending on the filter value. In the top row of pixels, only one of the white pixels (the top center) is matched by a 1 in the filter. This gives a result of $1 \times 1 = 1$. The other three white pixels are matched up with –1, giving three results of $-1 \times 1 = -1$. Adding these gives us $-3 + 1 = -2$.

In the lower row, our image matches the filter. All three weights of 1 on the filter are sitting on white pixels, and there are no other white pixels in the input. The result is a score of 3, indicating a perfect match.

Figure 16-13 shows another filter, this time looking for diagonals. Let's run it over the same image. This diagonal of three white pixels surrounded by black is present in two places.

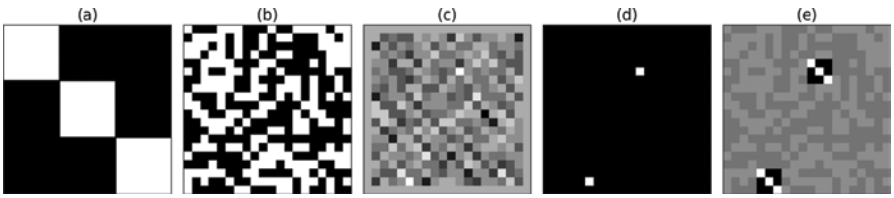


Figure 16-13: Another filter and its result on our random image. (a) The filter. (b) The input. (c) The feature map. (d) Feature map entries with value 3. (e) Neighborhoods of (b) around the white pixels in (d).

By sweeping a filter over the image and computing the output value at each pixel, we can hunt for lots of different simple patterns. In practice, our filter and pixel values are all real numbers (not just 0 and 1), so we can make much more complex patterns that find more complex features (Snavely 2013).

If we take the output of a set of filters and feed them to another set of filters, we can look for patterns of patterns. If we feed that second set of outputs to a third set of filters, we can look for patterns of patterns of patterns. This process lets us build up from, say, a collection of edges, to a set of shapes, such as ovals and rectangles, to ultimately matching a pattern corresponding to some specific object, such as a guitar or bicycle.

Applying successive groups of filters in this way, in concert with another technique we will soon discuss called *pooling*, enormously expands the sorts of patterns that we can detect. The reason is that the filters operate *hierarchically*, where each filter's patterns are combinations of the patterns found by earlier filters. Such a hierarchy allows us to look for features of great complexity, such as the face of a friend, the grain of a basketball, or the eye on the end of a peacock's feather.

If we had to work out these filters by hand, classifying images would be impractical. What are the proper weights in a chain of eight filters that tell us if a picture shows a kitten or an airplane? How could we even go about working out that problem? And how would we know when we found the best filters? In Chapter 1 we discussed expert systems, in which people tried to do this kind of feature engineering by hand. It's a formidable task for simple problems, and it grows in complexity so quickly that really interesting problems, such as distinguishing cats from airplanes, seem entirely out of reach.

The beauty of CNNs is that they carry out the goals of expert systems, but we don't have to figure out the values of the filters by hand. The learning process that we've seen in previous chapters, involving measuring error, backpropagating the gradients, and then improving the weights, teaches a CNN to find the filters it needs. The learning process modifies the kernel of each filter (that is, the weights in each neuron), until the network is producing results that match our targets. In other words, training tunes the values in the filters until they find the features that enable it to come up with the right class for the object in the image. And this can happen for hundreds or even thousands of filters, all at once.

This can seem like magic. Starting with random numbers, the system learns what patterns it needs to look for in order to distinguish a piano from an apricot from an elephant, and then it learns what numbers to put into the filter kernels in order to find those patterns.

That this process can even come close in one situation is remarkable. The fact that it often produces highly accurate results in a vast range of applications is one of the great discoveries in deep learning.

Padding

Earlier, we promised to return to the issue of what happens when a convolution filter is centered over an element in a corner or on an edge of an input tensor. Let's look at that now.

Suppose that we want to apply a 5 by 5 filter to a 10 by 10 input. If we're somewhere in the middle of the tensor, as in Figure 16-14, then our job is easy. We pull out the 25 values from the input, and apply them to the convolution filter.

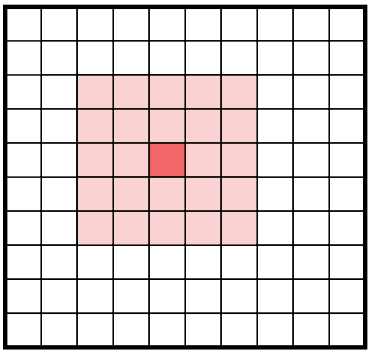


Figure 16-14: A five by five filter located somewhere in the middle of a tensor. The bright red pixel is the anchor, while the lighter ones make up the receptive field.

But what if we're on, or near, an edge, as in Figure 16-15?

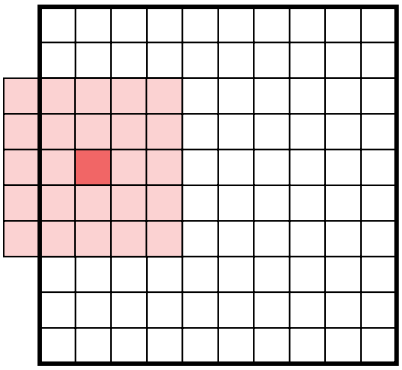


Figure 16-15: Near the edge, the filter's receptive field can fall off the side of the input. What values do we use for these missing elements?

The footprint of the filter is hanging off the edge of the input. There aren't any input elements there. How do we compute an output value for the filter when it's missing some of its inputs?

We have a few choices. One is to disallow this case so we can only place the footprint where it is entirely within the input image. The result is an output that's smaller in height and width. Figure 16-16 shows this idea.

While simple, this is a lousy solution. We said that we often apply many filters in sequence. If we sacrificed one or more rings of elements each time, we would lose information with every step we take through the network.

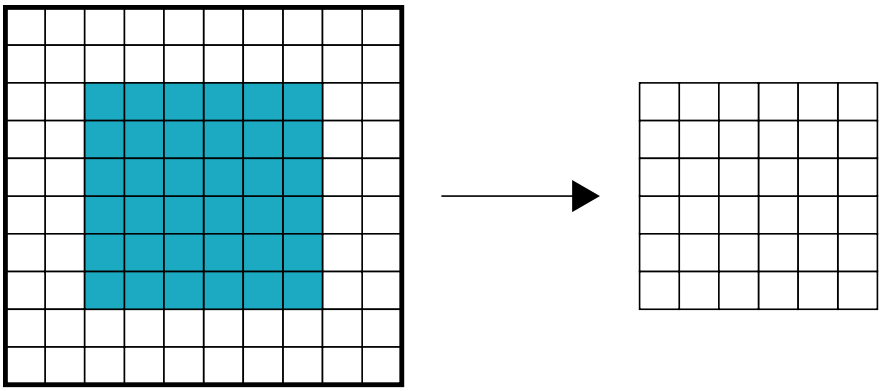


Figure 16-16: We can avoid the "falling off the edge" problem by never letting our filter get that far. With a 5 by 5 filter, we can only center the filter over the elements marked here in blue, reducing our 10 by 10 input to a 6 by 6 output.

A popular alternative is to use a technique called *padding*, which lets us create an output image of the same width and height as the input. The idea is that we add a border of extra elements around the outside of the input, as in Figure 16-17. All of these elements have the same value. If we place zeros in all the new elements, we call the technique *zero-padding*. In practice, we almost always use zeros, so people often refer to zero-padding as merely padding, with the understanding that if they mean to use any value other than zero, they say so explicitly.

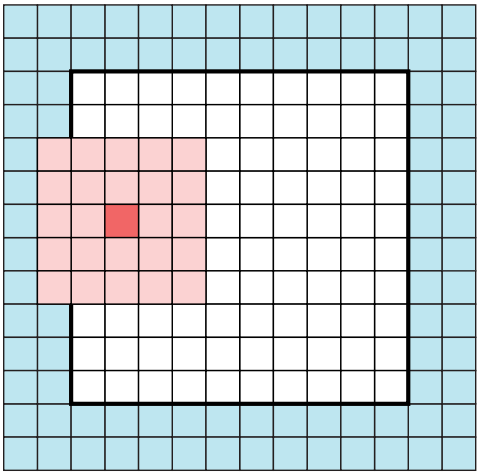


Figure 16-17: A better way to solve the "falling off the edge" problem is to add padding, or extra elements (in light blue), around the border of the input.

The thickness of the border depends on the size of the filter. We usually use just enough padding so that the filter can be centered on every element of the input. Every filter needs to have its input padded if we don't want to lose information from the sides.

Most deep learning libraries automatically calculate the necessary amount of padding so that our output has the same width and height as our input, and apply it for us as a default.

Multidimensional Convolution

So far in this chapter, we've mostly been considering grayscale images with only one channel of color information. We know that most color images have three channels, representing the red, green, and blue components of each pixel. Let's see how to handle those. Once we can work with images with three channels, we can work with tensors of any number of channels.

To process an input with multiple channels, our filters (which can have any footprint) need to have an identical number of channels. That's because each value in the input needs to have a corresponding value in the filter. For an RGB image, a filter needs three channels. So, a filter with a footprint of three by three needs to have three channels, for a total of 27 numbers, as shown in Figure 16-18.

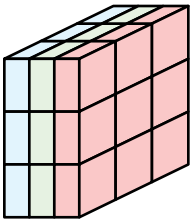


Figure 16-18: A three-channel filter with a three by three footprint. We've colored the values to show which input channel's values they will multiply.

To apply this kernel to a three-channel color image, we proceed much as before, but now we think in terms of blocks (or tensors of three dimensions).

Let's take the filter of Figure 16-18, with a three by three footprint and three channels, and use it to process an RGB image with three color channels. For each input pixel, we center the filter's footprint over that pixel as before, and match up each of the 27 numbers in the image with the 27 numbers in the filter, as in Figure 16-19.

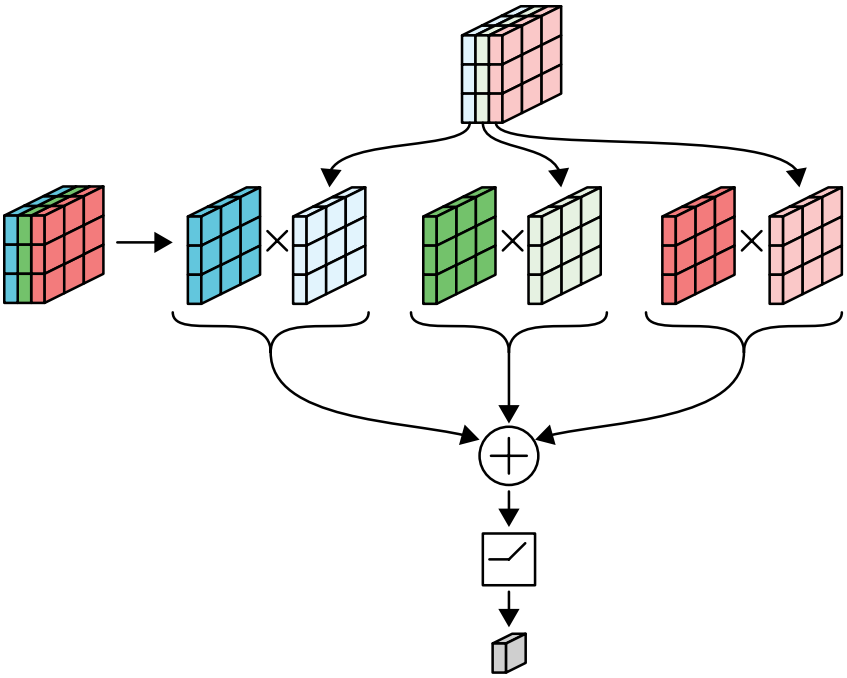


Figure 16-19: Convolving an RGB image with a three by three by three kernel. We can imagine that each channel is filtered by its own channel in the filter.

In Figure 16-19, our input has three channels, so our filter has three channels as well. It may be helpful to think of the red, green, and blue channels as each getting filtered by their corresponding channel in the filter, as shown in Figure 16-19. In practice, we treat the input and the filter as three by three by three blocks, and each of the 27 input values get multiplied with its corresponding filter value.

This idea generalizes to any number of channels. In order to make sure that every input value has a corresponding filter value, we can state the necessary property as a rule: every filter must have the same number of channels as the tensor it's filtering.

Multiple Filters

We've been applying a single filter at a time, but that's rare in practice. Usually we bundle up tens or hundreds of filters into one *convolution layer* and apply them all simultaneously (and independently) to that layer's input.

To see the general picture, imagine that we've been given a black-and-white image, and we want to look for several low-level features in the pixels, such as vertical stripes, horizontal stripes, isolated dots, and plus signs. We can create one filter for each of these features and run each one over the input independently. Each filter produces an output image with one channel. Combining the four outputs gives us one tensor with four channels. Figure 16-20 shows the idea.

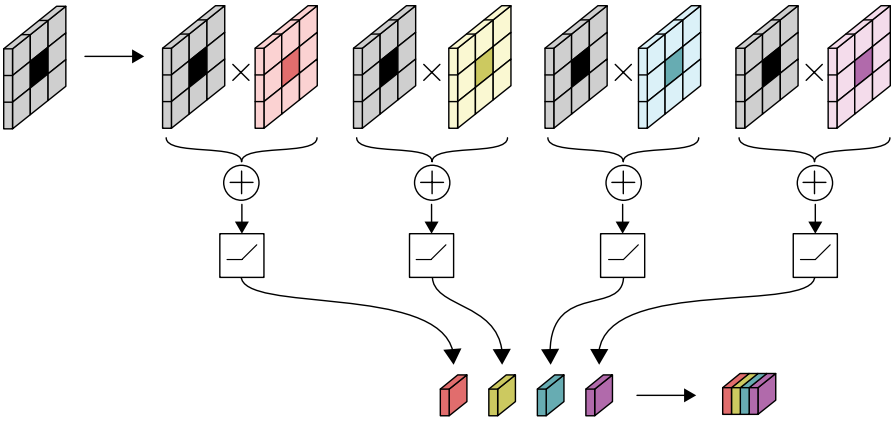


Figure 16-20: We can run multiple filters (in color) over the same input (in gray). Each filter creates its own channel in the output. They are then combined to create a single element in the output tensor with four channels.

Instead of a grayscale image with one channel, or a color image with three channels, we now have an output tensor with four channels. If we used seven filters, then the output is a new image with seven channels. The key thing to note here is that the output tensor has one channel for each filter that's applied.

Generally speaking, our filters can have any footprint, and we can apply as many of them as we like to any input image. Figure 16-21 shows this idea.

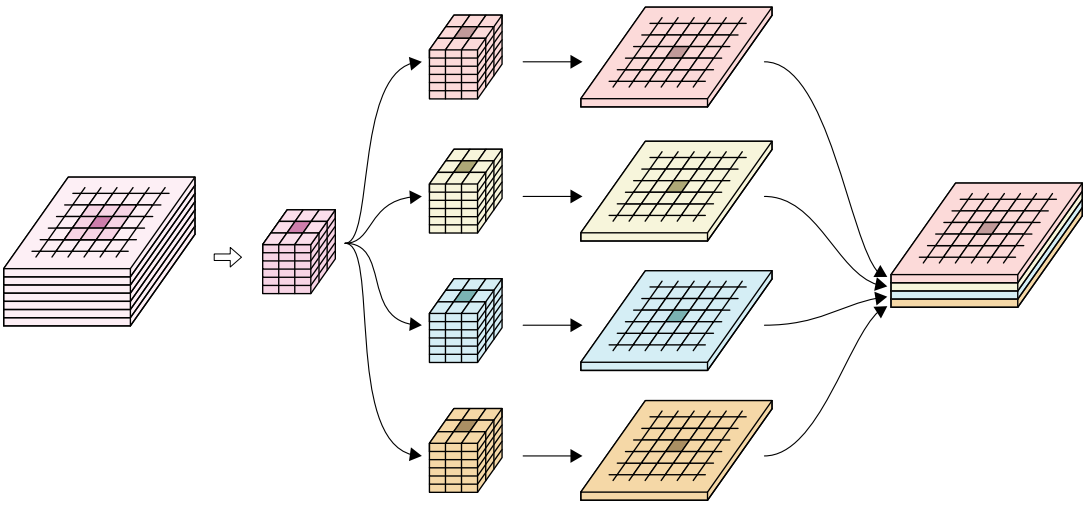


Figure 16-21: When we convolve filters with an input, each filter must have as many channels as the input. The output tensor has one channel for each filter.

The input tensor at the far left has seven channels. We're applying four different filters, each with a three by three footprint, so each filter is a

tensor of size three by three by seven. The output of each filter is a feature map of a single channel. The output tensor is what we get from stacking these four feature maps, so it has four channels.

Although in principle each filter we apply can have a different footprint, in practice we almost always use the same footprint for every filter in any given convolution layer. For example, in Figure 16-21 all the filters have a footprint of three by three.

Let's gather together the two numerical rules from the previous section and this one. First, every filter in a convolution layer must have the same number of channels as that layer's input tensor. Second, a convolution layer's output tensor will have as many channels as there are filters in the layer.

Convolution Layers

Let's take a closer look at the mechanics of convolution layers. A convolution layer is simply a bunch of filters gathered together. They're applied independently to the input tensor, as in Figure 16-21, and their outputs are combined to create a new output tensor. The input is not changed by this process.

When we create a convolution layer in code, we typically tell our library how many filters we want, what their footprint should be, and other optional details like whether we want to use padding and what activation function we want to use—the library takes care of all the rest. Most importantly, training improves the kernel values in each filter, so that the filters learn the values that enable them to produce the best results.

When we draw a diagram of a deep learner, we usually label our convolution layers with how many filters are used, their footprints, and their activation function. Since it's common to use the same padding all around the input, we often just provide a single value rather than two, with the understanding that it applies to both width and height.

Like the weights in fully connected layers, the values in a convolution layer's filters start out with random values and are improved with training. Also like fully connected layers, if we're careful about choosing these random initial values, training usually goes faster. Most libraries offer a variety of initialization methods. Generally speaking, the built-in defaults normally work fine, and we rarely need to explicitly choose an initialization algorithm.

If we do want to pick a method, the He algorithm is a good first choice (He et al. 2015; Karpathy 2016). If that's not available, or doesn't work well in a given situation, Glorot is a good second choice (Glorot and Bengio 2010).

Let's look at a couple of special types of convolution that have their own names.

1D Convolution

An interesting special case of sweeping a filter over an input is called *1D convolution*. Here we sweep over the input as usual in either height or width, but not the other (Snavely 2013). This is a popular technique when

working with text, which can be represented as a grid where each element holds a single letter, and rows contain complete words (or a fixed number of letters) (Britz 2015).

The basic idea is shown in Figure 16-22.

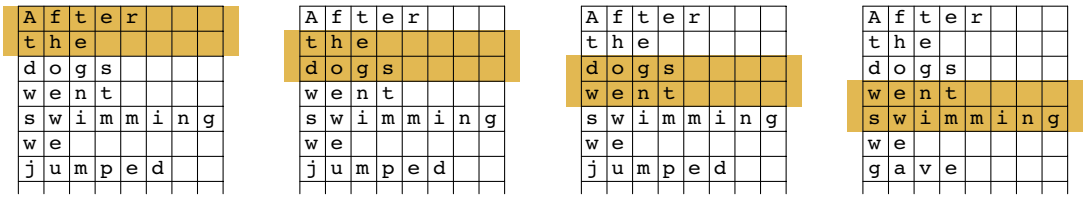


Figure 16-22: An example of 1D convolution. The filter only moves downward.

Here, we've created a filter that is the entire width of the input and two rows high. The first application of the filter processes everything in the first two rows. Then we move the filter down and process the next two rows. We don't move the filter horizontally. The name *1D convolution* comes from this single direction, or dimension, of movement.

As always, we can have multiple filters sliding down the grid. We can perform 1D convolution on an input tensor of any number of dimensions, as long as the filter itself moves in just one dimension. There's nothing otherwise special about 1D convolution: it's just a filter that only moves in one direction. The technique has its own name to emphasize the filter's limited mobility.

The name 1D convolution is almost the same as the name of another, quite different, technique. Let's look at that now.

1×1 Convolutions

Sometimes we want to reduce the number of channels in a tensor as it flows through a network. Often this is because we think that some of the channels contain redundant information. This isn't uncommon. For example, suppose we have a classifier that identifies the dominant object in a photograph. The classifier might have a dozen or more filters that look for eyes of different sorts: human eyes, cat eyes, fish eyes, and so on. If our classifier is going to ultimately lump all living things together into one class called "living things," then there's no need to care about which kind of eye we find. It's enough just to know that a particular region in the input image has an eye.

Suppose that we have a layer containing filters that detect 12 different kinds of eyes. Then the output tensor from that layer will have at least 12 channels, one from each filter. If we only care about whether or not an eye is found, then it would be useful to modify that tensor by combining, or compressing, those 12 channels into just 1 channel representing whether or not an eye is found at each location.

This doesn't require anything new. We want to process one input element at a time, so we create a filter with a footprint of one by one, like we saw in Figure 16-6. We make sure that we have at least 11 fewer filters than

there are input channels. The result is a tensor of the same width and height as the input, but the multiple eye channels get crunched together into just one channel.

We don't have to do anything explicit to make this happen. The network learns weights for the filters such that the network produces the correct output for each input. If that means combining all the channels for eyes, then the network learns to do that.

Figure 16-23 shows how to use these filters to compress a tensor with 300 channels into a new tensor of the same width and height, but with only 175 channels.

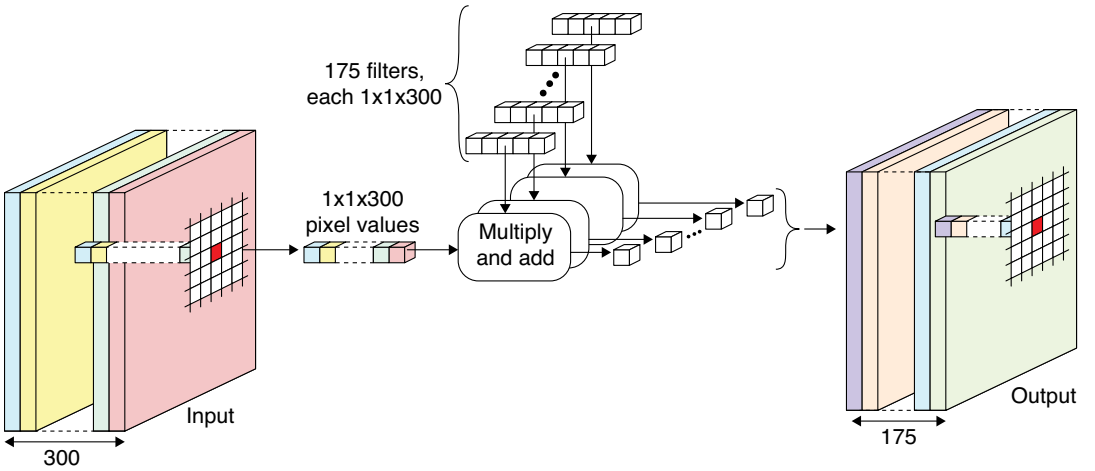


Figure 16-23: Applying 1×1 convolution to perform feature reduction

The technique of using one by one filters has been given its own name. We say that we apply a *one by one filter*, often written as a *1×1 filter*, and use that to perform *1×1 convolution* (Lin, Chen, and Yan 2014).

In Chapter 10 we talked about the value of preprocessing our input data in order to save processing time and memory. Rather than perform this processing once, before the data has entered our system, 1×1 convolution lets us apply this compression and restructuring of the data on the fly, inside of the network. If our network produces information that can be compressed or removed entirely, then 1×1 convolutions can find and then compress or remove that data. We can do this anywhere, even in the middle of a network.

When the channels are correlated, 1×1 convolution is particularly effective (Canziani, Paszke, and Culurciello 2016; Culurciello 2017). This means that the filters on the previous layers have created results that are in sync with one another, so that when one goes up, we can predict by how much the others will go up or down. The better this correlation, the more likely it is that we can remove some of the channels and suffer little to no loss of information. The 1×1 filters are perfect for this job.

The term *1×1 convolution* is uncomfortably close to *1D convolution*, which we discussed in the last section. But these names refer to quite distinct techniques. When encountering either of these terms, it is worth taking a moment to make sure we have the correct idea in mind.

Changing Output Size

We've just seen how to change the number of channels in a tensor by using 1×1 convolution. We can also change the width and height, which is useful for at least two reasons. The first is that if we can make the data flowing through our network smaller, we can use a simpler network and save time, computing resources, and energy. The second is that reducing the width and height can make some operations, like classification, more efficient and even more accurate. Let's see why this is so.

Pooling

In previous sections, we applied each filter to one pixel, or one region of pixels. The filter matches the feature it's looking for if the underlying pixels match the filter's values. But what if some of the elements of the feature are in slightly wrong places? Then the filter won't match. There's no way for the filter to look around and report a match if one or more pieces of the pattern it's looking for are present but slightly out of position. This would be a real problem if we didn't address it. For example, suppose we're looking for a capital T on a page of text. Due to a minor mechanical error during printing, a column of pixels was displaced downward by one pixel.

We still want to find the T. The situation is illustrated in Figure 16-24.

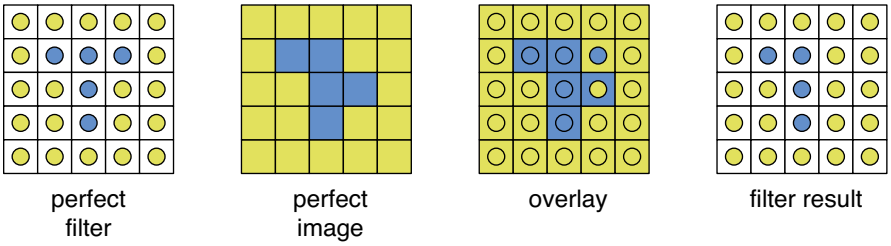


Figure 16-24: From left to right: A five by five filter looking for a letter T, a misprinted T, the filter on top of the image, and the filter's resulting values. The filter would not report a match to the letter T.

We begin with a five by five filter that is looking for a T in the center. We illustrate this using blue for 1 and yellow for 0.

We've labeled this the "perfect filter," a name that will make sense in a moment. To its right is the misprinted text we're going to examine, labeled "perfect image." To the right of that, we overlay the filter on the image. At the far right is the result. Only when the filter and the input are both blue

will the output be blue. Since the filter's upper-right element did not find the blue pixel it was expecting, the filter as a whole reports either no match, or a weak one.

If the upper-right element in the filter could look around and notice the blue pixel just below it, it could match the input. One way to make this happen is to let each filter element "see" more of the input. The most convenient way to do that mathematically is to make the filter a bit blurry.

On the top row of Figure 16-25 we picked out one element of the filter and blurred it. If the filter finds a blue pixel anywhere in this larger, blurry region, it reports finding blue. If we do this for all the entries in the filter, we create a "blurry filter." Thanks to this extended reach, the upper-right blue filter element now overlaps two blue pixels, and since the other blue elements also overlap blue pixels, the filter now reports a match.

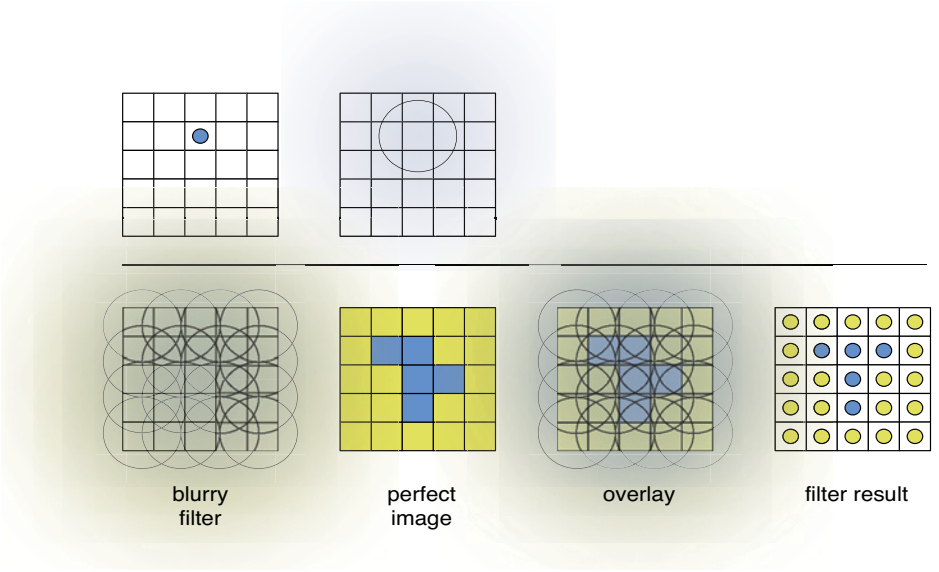


Figure 16-25: Top row: Replacing a filter element with a bigger, blurrier version. Bottom row: Applying the blur to every filter element gives us a blurry filter. Applying this to the image matches the misprinted T.

Unfortunately, we can't blur filters like this. If we modified our filter values by blurring them, our training process would go haywire, since we would be altering the very values we're trying to learn. But there's nothing stopping us from blurring the input! This is particularly easy to see if the input is a picture, but we can blur any tensor. So rather than applying a blurry filter to a perfect input, let's flip that around and apply a perfect filter to a blurry input.

The top row of Figure 16-26 shows a single pixel from the misprinted T, and the version of that pixel after it's been blurred. After we apply this blurring to all the pixels, we can apply the perfect filter to this blurry image. Now every blue dot in the filter sees blue under it. Success!

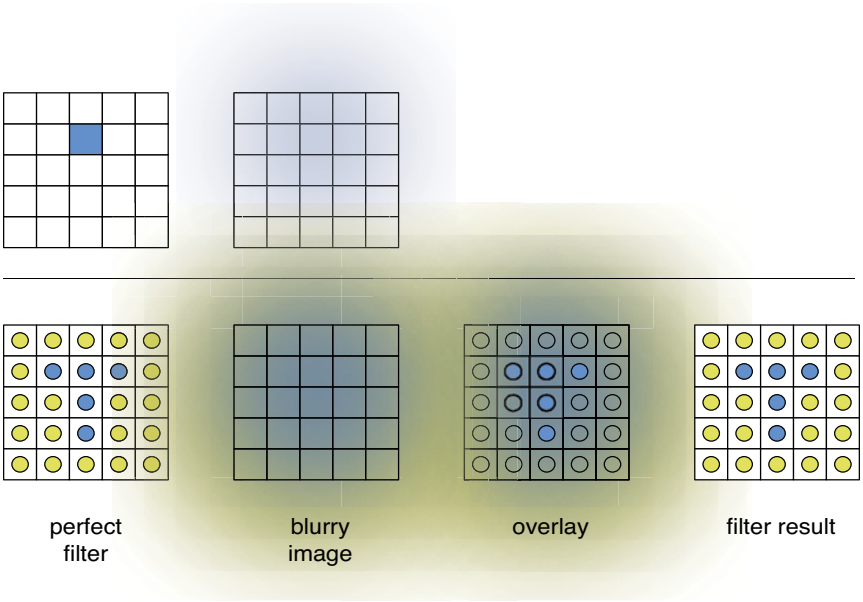


Figure 16-26: Top row: The effect of blurring one pixel in the input. Bottom row: We apply the perfect filter to a blurred version of the image. This matches the misprinted T.

Taking this as our inspiration, we can come up with a technique to blur a tensor. We call the method *pooling*, or *downsampling*. Let's see how pooling works numerically with a small tensor with a single channel. Suppose we start with a tensor that has a width and height of four, as shown in Figure 16-27(a).

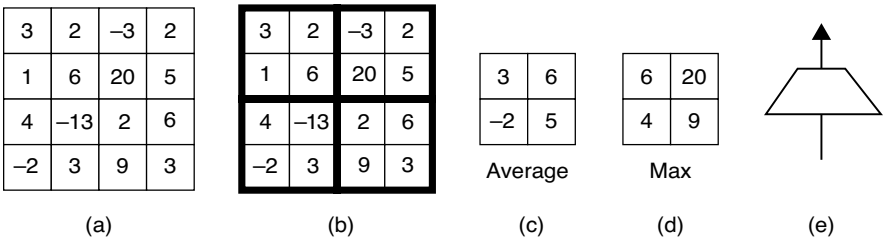


Figure 16-27: Pooling, or downsampling, a tensor. (a) Our input tensor. (b) Subdividing (a) into two by two blocks. (c) The result of average pooling. (d) The result of max pooling. (e) Our icon for a pooling layer.

Let's subdivide the width and height of this tensor into two by two blocks, as in Figure 16-27(b). To blur the input tensor, recall Figure 16-7. We saw that by convolving with a filter whose contents are all 1's, the image got blurry. Such a filter is called a *low-pass filter*, or more specifically, a *box filter*.

To apply a box filter to a tensor, we can use a two by two filter where every weight is a 1. Applying this filter merely means adding up the four

numbers in each two by two block. Because we don't want our numbers to grow without bound, we divide the result by four to get the average value in that block. Since this average now stands in for the entire block, we save it just once. We do the same thing for the other three blocks. The result is a new tensor of size two by two, shown in Figure 16-27(c). This technique is called *average pooling*.

There's a variation on this method: instead of computing the average value, we just use the largest value in each block. This is called *maximum pooling* (or more often, just *max pooling)*, and is shown in Figure 16-27(d). It's common to think of these pooling operations as being carried out by a little utility layer. In Figure 16-27(e) we show our icon for such a *pooling layer*. Experience has shown that networks that use max pooling learn more quickly than those using average pooling, so when people speak of pooling with no other qualifiers, they usually mean max pooling.

The power of pooling appears when we apply multiple convolution layers in succession. Just as with a filter and a blurred input, if the first filter's values aren't in quite the expected locations, pooling helps the second layer's filter still find them. For example, suppose that we have two layers in succession, and Layer 2 has a filter that is looking for a strong match from Layer 1, directly above a match of about half that value (maybe this is characteristic of a particular animal's coloration). Nothing in the original 4 by 4 tensor in Figure 16-27(a) fits that pattern. There's a 20 over a 2, but the 2 isn't close to being half of 20. And there's a 6 over 3, but 6 isn't a very strong output. So Layer 2's filter would fail to find what it was looking for. That's too bad, because there is a 20 that's close to being over a 9, which is what the filter wants to find. The problem is that the 20 and the 9 are not exactly vertical neighbors.

But the max pooling version has the 20 over the 9. The pooling operation is communicating to Layer 2 that there is a strong match of 20 somewhere in the upper right two by two block, and a match of 9 somewhere in the block directly below the 20. That's the pattern we're looking for, and the filter will tell us that it found a match.

We've discussed pooling for just one channel. When our tensors have multiple channels, we apply the same process to each channel. Figure 16-28 shows the idea.

We start with an input tensor of height and width 6 and one channel, padded with a ring of zeros. The convolution layer applies three filters, each producing a feature map of six by six. The output of the convolution layer is a tensor of size six by six by three. The pooling layer then conceptually considers each channel of this tensor, and applies max pooling to it, reducing each feature map to three by three. Those feature maps are then combined as before to produce an output tensor of width and height 3, with three channels.

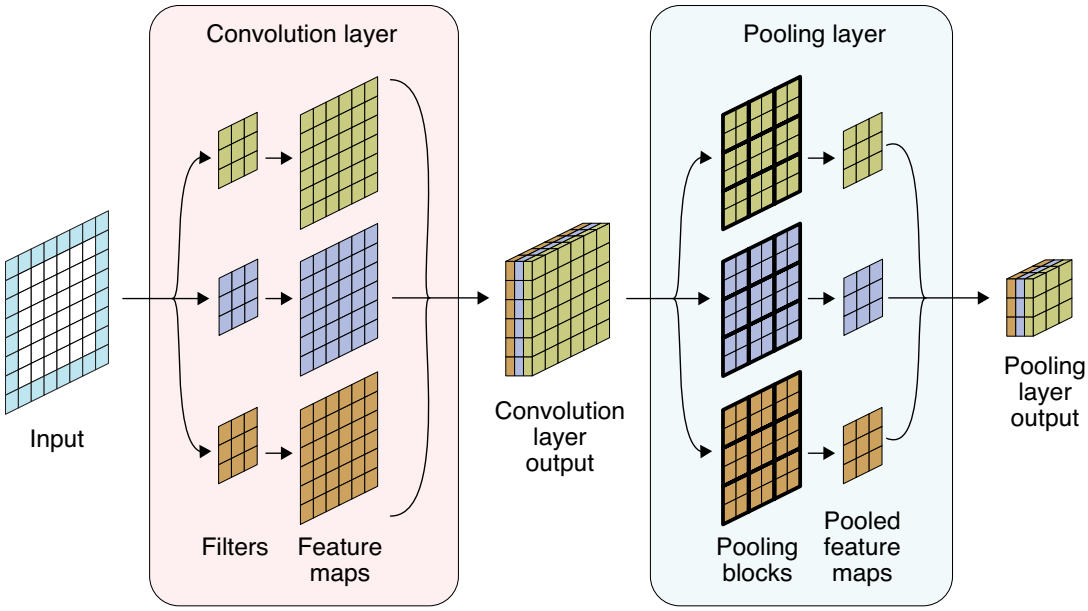


Figure 16-28: Pooling, or downsampling, with multiple filters

We've been using binary images and filters as examples. This means that a feature that straddles cell boundaries could be missed, or wind up in the wrong element in the pooled tensor. When we use real valued inputs and filter kernels, this problem is greatly reduced.

Pooling is a powerful operation that frees filters from requiring their inputs to be in precisely the right place. Mathematicians refer to a change in location as *translation* or *shift*, and if some operation is insensitive to a certain kind of change it's called *invariant* with respect to that operation. Combining these, we sometimes say that pooling allows our convolutions to be *translationally invariant*, or *shift invariant* (Zhang 2019).

Pooling also has the bonus benefit of reducing the size of the tensors flowing through our network, which reduces both memory needs and execution time.

Striding

We've seen how useful pooling is in a convolutional network. Though pooling layers are common, we can save time by bundling the pooling step right into the convolution process. This combined operation is much faster than two distinct layers. The tensors resulting from the two procedures usually contain different values, but experience has shown that the faster, combined operation usually produces results that are just as useful as the slower, sequential operations.

As we saw, during convolution we can imagine starting the filter in the upper-left pixel of the input image (let's assume we have padding). The filter produces an output, then takes one step right, produces another output,

moves another step right, and so on until it reaches the right edge of that row. Then it moves down one row and back to the left side, and the process repeats.

But we don't have to move in single steps. Suppose we move, or *stride*, more than one pixel to the right, or more than one pixel down, as we sweep our filter. Then our output will end up being smaller than the input. We usually use the word *stride* (and the related *striding*) only when we use steps greater than one in any dimension.

To visualize striding, let's see how the filter moves starting from the upper left. As the filter moves left to right, it produces a sequence of outputs, and those get placed one after the other, also left to right, in the output. When the filter moves down, the new outputs go on a new line of cells in the output.

Now suppose that instead of moving the filter to the right by one element on each horizontal step, we moved to the right by three elements. And perhaps on each vertical step we move down by two rows, rather than one. We still grow the output by one element for each output. The idea is shown in Figure 16-29.

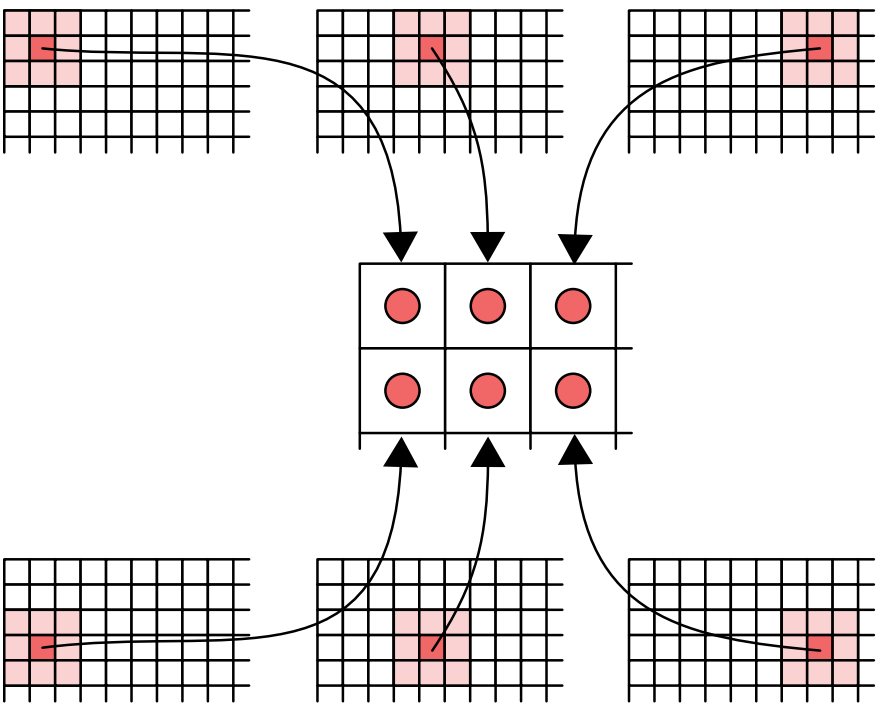


Figure 16-29: Our input scanning can skip over input elements as it moves. Here we move three elements to the right on each horizontal step, and two elements down on each vertical step.

In Figure 16-29 we used a stride of three horizontally, and a stride of two vertically. More often we specify a single stride value for both axes. A stride of two on both axes can be thought of as evaluating every other pixel

both horizontally and vertically. This results in an output that has half the input dimensions as the input, which means the output has the same dimensions as striding by one and then pooling with two by two blocks. Figure 16-30 shows where the filter lands in the input for a couple of different pairs of strides.

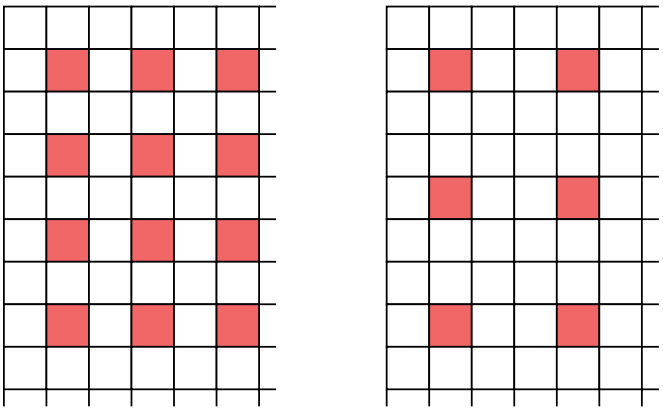


Figure 16-30: Examples of striding. (a) A stride of two in both directions means centering the filter over every other pixel, both horizontally and vertically. (b) A stride of three in both directions means centering over every third pixel.

When we move by one element on every step, a filter with a three by three footprint processes the same input elements multiple times. When we stride by larger amounts, our filter can still process some elements multiple times, as shown in Figure 16-31.

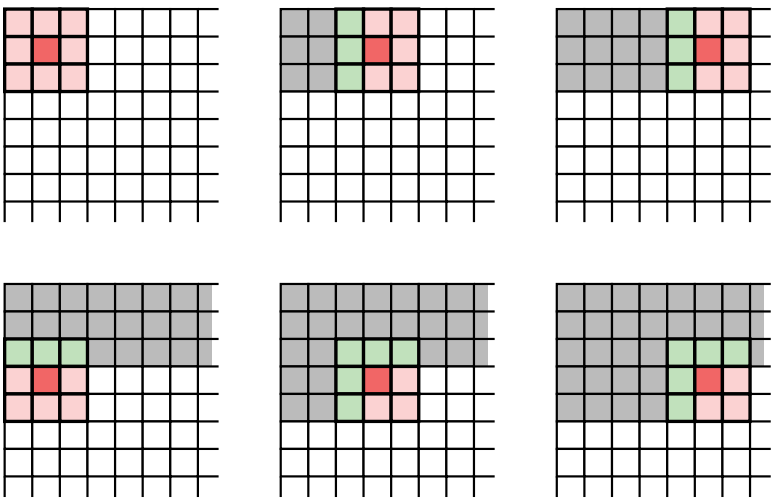


Figure 16-31: This three by three filter is moving with a stride of two in each dimension, reading left to right, top to bottom. The gray elements show what's been processed so far. The green elements are those that have already been used by the filter on previous evaluations but are being used again.

There's nothing wrong with reusing an input value repeatedly, but if we're trying to save time, we might want to do as little computation as possible. Then we can use striding to prevent any input element from being used more than once. For instance, if we're moving a three by three filter over an image, we might use a stride of three in both directions, so that no pixel gets used more than once, as in Figure 16-32.

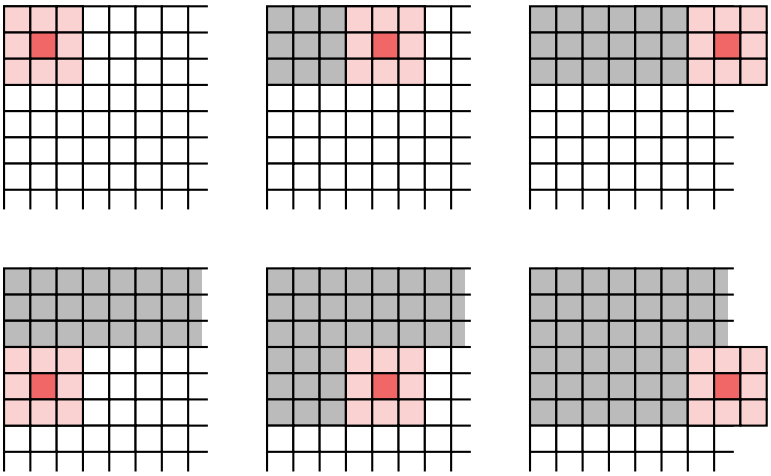


Figure 16-32: Like Figure 16-31, only now we're striding by three in each dimension. Every input element is processed exactly one time.

The striding in Figure 16-32 produces an output tensor with a height and width that are each one-third of the input tensor's height and width. Consider that in Figure 16-32 we processed a nine by six block of input elements with just six filter evaluations. By doing this, we created a three by two block of outputs with no explicit pooling. If we don't stride, and then pool, we need many more filter evaluations to cover the same region, and then we need to run the pooling operation on the filter outputs.

Strided convolutions are faster than convolution without striding followed by pooling for two reasons. First, we evaluate the filter fewer times, and second, we don't have an explicit pooling step to compute. Like padding, striding can (and often is) carried out on any convolutional layer, not just the first.

The filters learned from striding are usually different than those learned from convolution without striding followed by pooling. This means we can't take a trained network and replace pairs of convolution and pooling with strided convolution (or vice versa) and expect things to still work properly. If we want to change our network's architecture, we have to retrain it.

Most of the time, training with strided convolution gives us final results that are roughly the same as those we get from convolution followed by pooling, delivered in less time. But sometimes the slower combination works better for a given dataset and architecture.

Transposed Convolution

We've seen how to reduce the size of the input, or *downsize* it, using either pooling or striding. We can also increase the size of the input, or *upsize* it. As with downsizing, when we upsize a tensor, we increase its width and height, but we don't change the number of channels.

Just as with downsampling, we can upsize with a separate layer or build it into the convolution layer. A distinct upsampling layer usually just repeats the input tensor values as many times as we request. For example, if we upsample a tensor by two in both the width and height, each input element turns into a little two by two square. Figure 16-33 shows the idea.

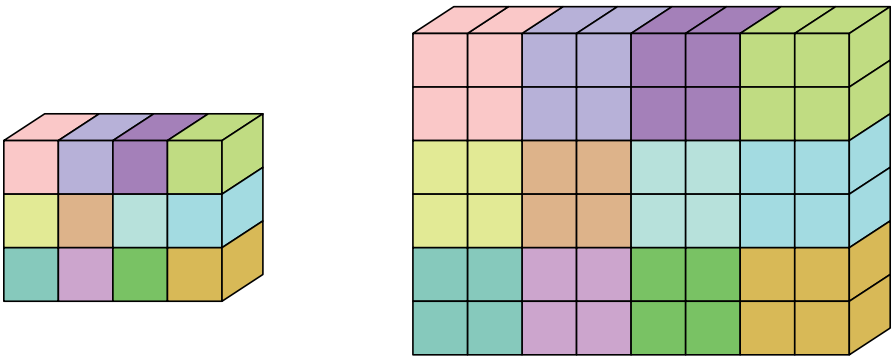


Figure 16-33: Upsampling a tensor by two in each direction. Left: The input tensor. Each element of this tensor is repeated twice vertically and horizontally. Right: The output tensor. The number of channels is unchanged.

We have seen that we can combine downsampling with convolution by using striding. We can also combine upsampling with convolution. This combined step is called *transposed convolution*, *fractional striding*, *dilated convolution*, or *atrous convolution*. The word *transposed* comes from the mathematical operation of transposition, which we can use to write the equation for this operation. The word *atrous* is French for "with holes." We'll see where that term, and the others, come from in a moment. Note that some authors refer to the combination of upsampling and convolution as *deconvolution*, but it's best to avoid that term, since it's already in use and refers to a different idea (Zeiler et al. 2010). Following current practice, we'll use the term *transposed convolution*.

Let's see how transposed convolution works to enlarge a tensor (Dumoulin and Visin 2016). Suppose that we have a starting image of width and height three by three (remember, the number of channels won't be changing), and we'd like to process it with a three by three filter, but we'd like to end up with a five by five image. One approach is to pad the input with two rings of zeros, as in Figure 16-34.

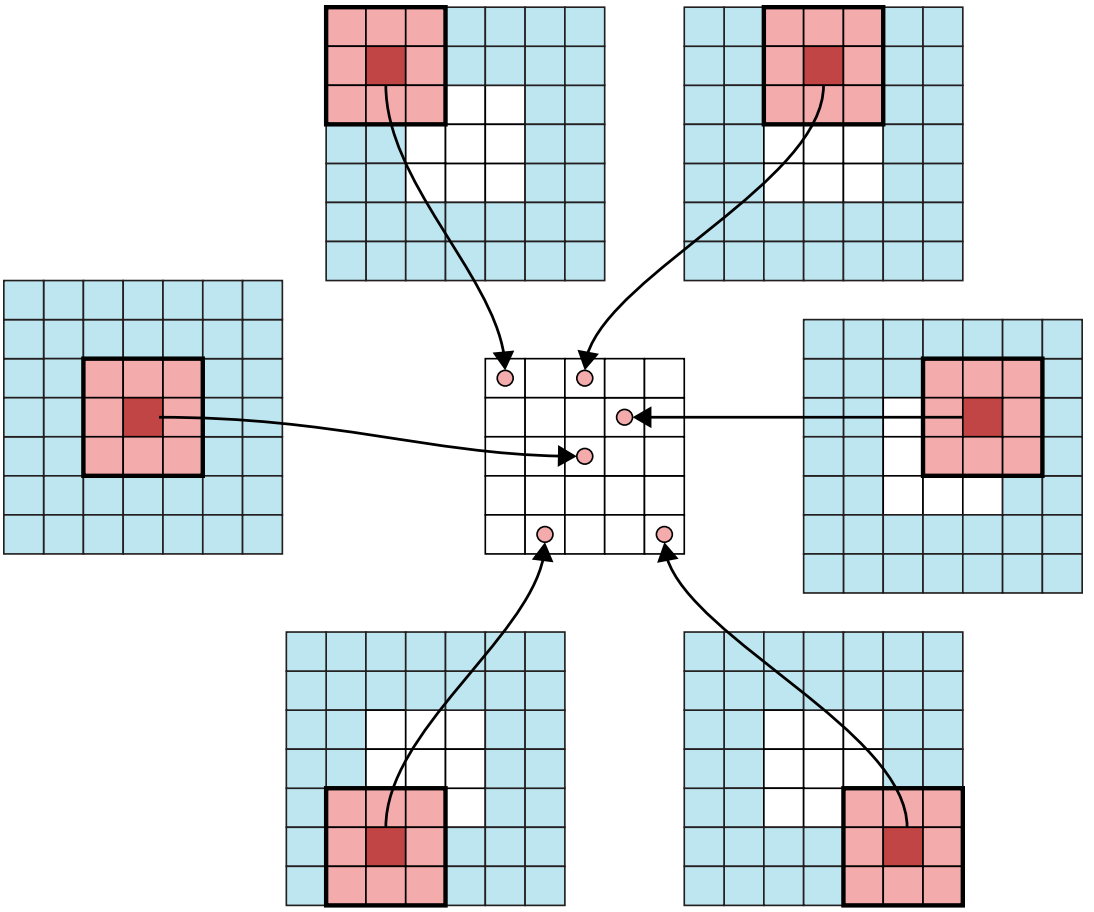


Figure 16-34: Our original three by three input is shown in white in the outer grids, padded with two elements of zeros all around. The three by three filter now produces a five by five result, shown in the center.

If we add more rings of zeros to the input, we get larger outputs, but they will produce rings of zeros around the central five by five core. That's not very useful.

An alternative way to enlarge the input is to spread it out before convolving by inserting padding both around and *between* the input elements. Let's try this out. Let's insert a single row and column of zeros between each element of our starting three by three image, and pad all of that with two rings of zeros around the outside, like before. The result is that our three by three input now has dimensions nine by nine, though a lot of those entries are zero. When we sweep our three by three filter over this grid, we get a seven by seven output, as shown in Figure 16-35.

Our original three by three image is shown in the outer grids with white pixels. We've inserted a row and column of zeros (blue) between each

pixel, and then surrounded the whole thing with two rings of zeros. When we convolve our three by three filter (red) with this grid, we get a seven by seven result, shown in the center.

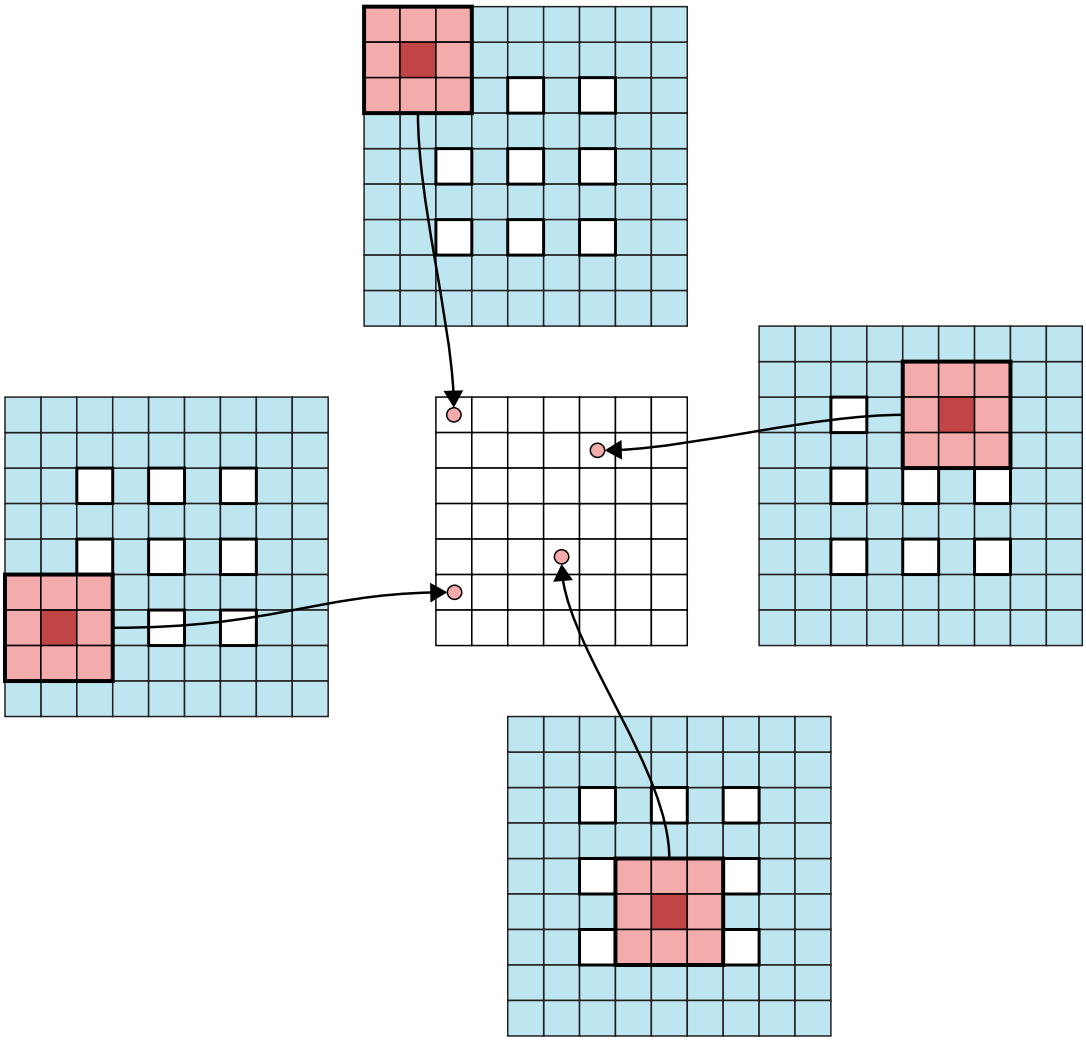


Figure 16-35: Transposed convolution, convolving a three by three filter into a seven by seven result

Figure 16-35 suggests where the names *atrous* (French for "with holes") *convolution* and *dilated convolution* come from. We can make our output even bigger by inserting another row and column between each original input element, as in Figure 16-36. Now our 3 by 3 input has become an 11 by 11 input, and the output is 9 by 9.

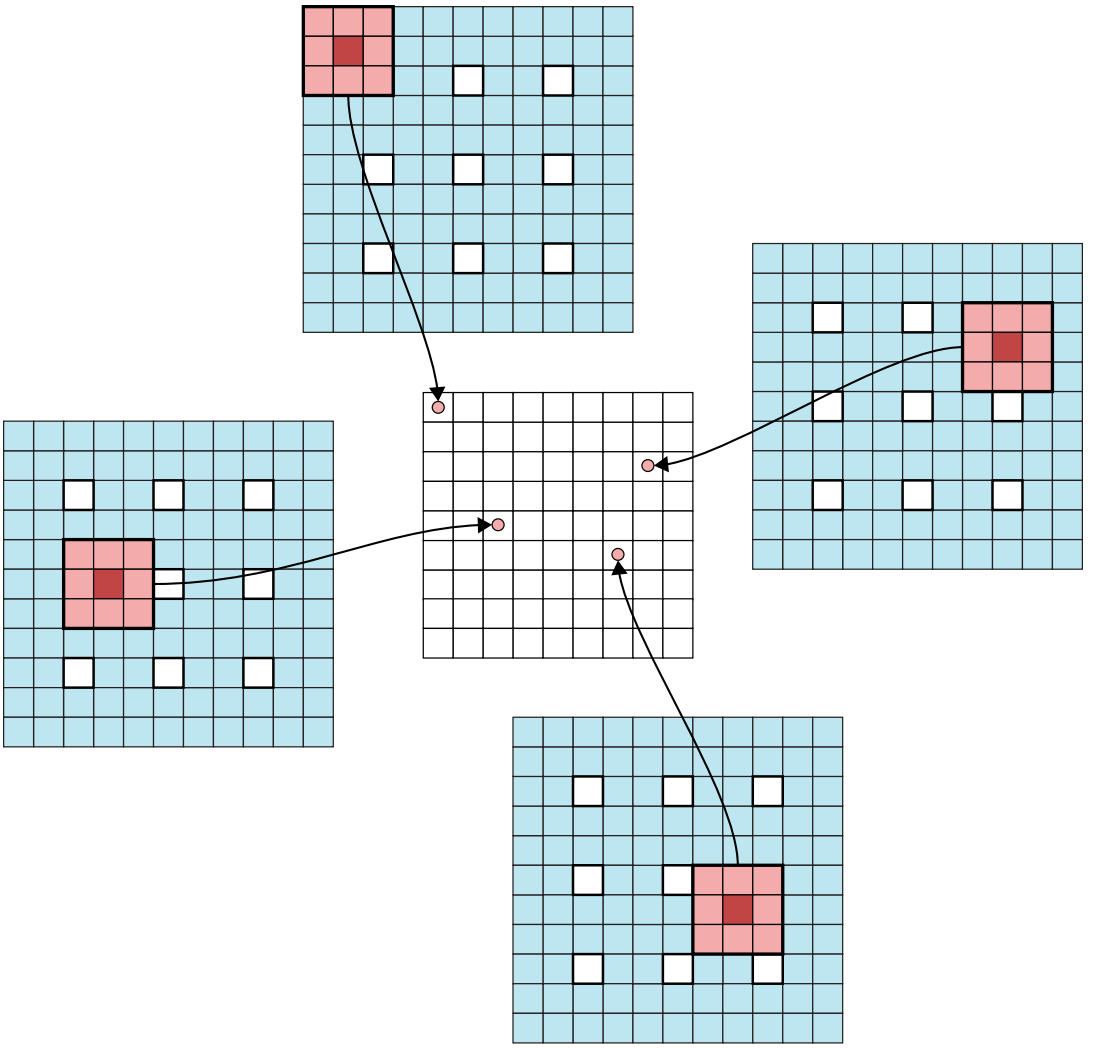


Figure 16-36: The same setup as Figure 16-35, only now we have two rows and columns between our original input pixels, producing the nine by nine result in the center

We can't push this technique any further without producing rows and columns of zeros in the output. The limit of two rows or columns of zeros is due to our filter having a footprint of three by three. If the filter was, say, five by five, we could use up to four rows and columns of zeros. This technique of inserting zeros can create little checkerboard-like artifacts in the output tensors. But library routines can usually avoid these if they take steps to handle the convolution and upsampling carefully (Odena, Dumoulin, and Olah 2018; Aitken et al. 2017).

There is a connection between transposed convolution and striding. With some imagination, we can describe a transposed convolution process like that of Figure 16-36 as using a stride of one-third in each dimension. We don't mean that we literally move one-third of an element, but rather

that we need to take three steps in the 11 by 11 grid to move the equivalent of one step in the original 3 by 3 input. This point of view explains why the method is sometimes called *fractional striding*.

Just as striding combines convolution with a downsampling (or pooling) step, transposed convolution (or fractional striding) combines convolution with an upsampling step. This results in faster execution time, which is always nice. A problem is that there is a limit to how much we can increase the input size. In practice, we commonly double the input dimensions, and use filters with a footprint of three by three, and transposed convolution supports that combination without introducing extraneous zeros in the output.

As with striding, the output of transposed convolution is different than the output of upsampling followed by standard convolution, so if we're given a trained network using upsampling followed by convolution, we can't just replace those two layers with one transposed convolution layer and use the same filters.

Transposed convolution is becoming more common than upsampling followed by convolution because of the increased efficiency, and similarity of the results (Springenberg et al. 2015).

We've covered a lot of basic tools, from different types of convolution to padding and changing the output size. In the next section, we put these all together to create a complete, but simple, convolutional network.

Hierarchies of Filters

Many real visual systems seem to be arranged *hierarchically* (Serre 2014). In broad terms, many biologists think of the processing in the visual system as taking place in a series of layers, with each successive layer working at a higher level of abstraction than the one before.

We've taken inspiration from biology already in this chapter, and we can do it again now.

Let's return to our discussion of the visual system of a toad. The first layer of cells to receive light may be looking for "bug-colored blobs," the next may be looking for "combinations of blobs from the previous layer that form bug-like shapes," the next may be looking for "combinations of bug-like shapes from the previous layer that look like a thorax with wings," and so on, up to the top layer, which looks for "flies" (these features are completely imaginary, and only meant to illustrate the idea).

This approach is nice conceptually because it lets us structure our analysis of an image in terms of a hierarchy of image features and the filters that look for them. It's also nice for implementations because it's a flexible and efficient way to analyze an image.

Simplifying Assumptions

To illustrate the use of hierarchies, let's solve a recognition problem with a convolutional network. To focus this discussion just on the concepts, we'll

make use of some simplifications. These simplifications in no way change the principles we're demonstrating; they just make the pictures easier to draw and interpret.

First, we restrict ourselves to binary images: just black and white, with no shades of gray (though for clarity, we draw them with beige and green for 0 and 1, respectively). In real applications, each channel in our input images is usually either an integer in the range [0, 255], or more commonly a real number in the range [0, 1].

Second, our filters are also binary and look for exact matches in their inputs. In real networks, our filters use real numbers, and they match their inputs to different degrees, represented by different real numbers at their output.

Third, we hand-create all of our filters. In other words, we do our own feature engineering. When we looked at expert systems, we said that their biggest problem was that they required people to manually build features, and here we are, doing just that! We're doing so just for this discussion, however. In practice, our filter values are learned by training. Since we're not interested in the training step right now, we'll use handmade filters (we can think of them as filters that resulted from training).

Fourth, we won't use padding. This also is just to keep things simple.

Finally, our example uses tiny input images that are just 12 pixels on a side. This is large enough to demonstrate the ideas but small enough that we can draw everything clearly on the page.

With these simplifications in place, we're ready to get started.

Finding Face Masks

Let's suppose that we work at a museum that has received a big collection of art, and it's our job to organize it all. One of our tasks is to find all of the drawings of grid-based face masks that are close matches to the simple mask in Figure 16-37.

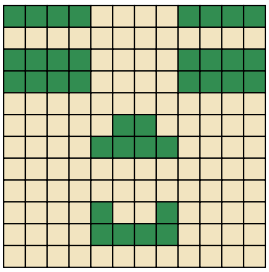


Figure 16-37: A simple binary mask on a 12 by 12 mesh

Suppose we're given the new mask in the middle of Figure 16-37. Let's call this the *candidate*. We want to determine whether it's roughly the "same" as the original mask, which we call the *reference*. We can just overlay the two masks and see if they match up, as in the right of Figure 16-38.

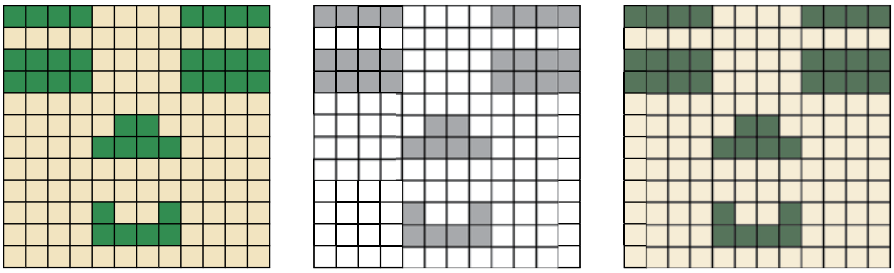


Figure 16-38: Testing for similarity. On the left is our original mask, or reference. In the middle is a new mask, or candidate. To see if they're close to one another, we can overlay them, at the right.

In this case, it's a perfect match, which is easy to detect. But what if a candidate is slightly different than the reference, as in Figure 16-39? Here one eye has moved down by one pixel.

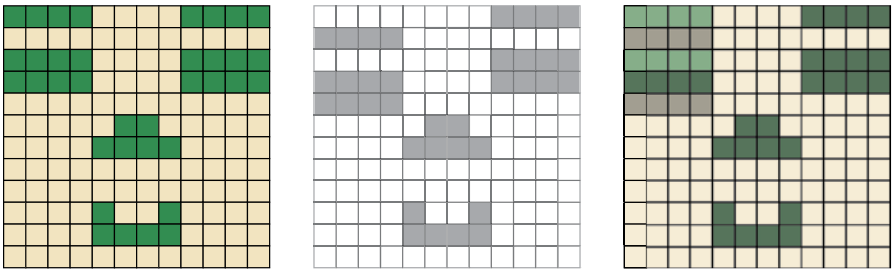


Figure 16-39: Like Figure 16-38, only the candidate's left eye has moved down by one pixel. The overlay is now imperfect.

Let's say that we still want to accept this candidate, since it has all the same features as the reference, and they're mostly in the right places. But the overlay shows that they're not identical, so a simple pixel-by-pixel comparison won't do the job.

In this simple example, we could come up with lots of ways to detect close matches, but let's use convolution to determine that a candidate like the one in Figure 16-39 is "like" the reference. As mentioned earlier, we're going to hand-engineer our filters. To describe our hierarchy, it's easiest to work backward, from the final step of convolution to the first.

Let's begin by describing the reference mask. Then we can determine if a candidate shares its qualities. Let's say that our reference is characterized by having one eye in each of the upper corners, a nose in the middle, and a mouth under the nose. That description applies to all of the masks we saw in Figures 16-38 and 16-39.

We can formalize this description with a three by three filter, as in the top-left grid of Figure 16-40. This will be one of our last filters: if we run a candidate through a series of convolutions, ultimately producing a three by three tensor (we'll see how that happens shortly), then if that tensor matches this filter, we've found a successful match, and an acceptable candidate. The cells with an × in them mean "don't care." For instance, suppose

a candidate has a tattoo on one cheek that falls into the × to the right of the nose. This doesn't affect our decision, so we explicitly don't care about what's in that cell.

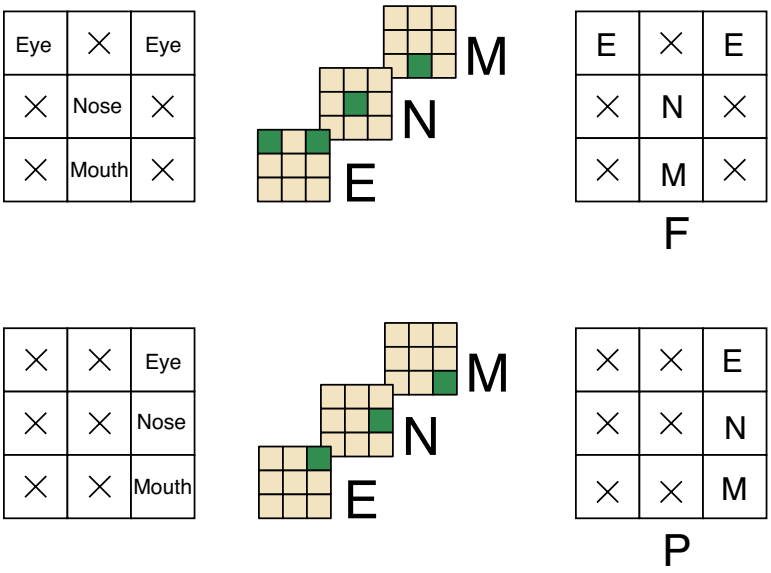


Figure 16-40: Filters for mask recognition. Top and bottom rows: Finding a mask facing forward, or in profile. Left column: Characterizing the reference. Middle: An exploded version of the tensor described by the grid at the left. Right: An X-ray view of the filter (see text).

Since our filters only contain the values 1 (green) and 0 (beige), we can't make a filter like the upper left diagram of Figure 16-40 directly. Instead, since it's looking for three different kinds of features, we need to redraw it as a filter with three channels, which we'll apply to an input tensor with three channels. One input channel tells us all the locations where an eye was located in the input, the next tells us all the locations of a nose, and the last tells us all the locations of a mouth. Our upper-left diagram corresponds, then, to a three by three by three tensor, shown in the upper middle diagram, where we've staggered the channels so we can read each one.

We drew the staggered version because if we drew that tensor as a solid block, we wouldn't be able to see most of the values on the N (nose) and M (mouth) channels. The staggered version is useful, but it will get too complicated when we start comparing tensors in the following discussion. Instead, let's draw an "X-ray view" of the tensor, as in the upper right. We imagine we're looking through the channels of the tensor, and we mark each cell with the names of all the channels that have a 1 in that cell.

Since this filter is looking for a mask facing forward, we label it F. For fun, we can make another mask that's looking for a face in profile, which we can call P. We won't look at any candidates that would be matched by P, but we're including it here to show the generality of this process. The layers to come, which operate before the filters of Figure 16-40, will tell

us where they found an eye, nose, and mouth. We use that information in Figure 16-40 to recognize different arrangements of these facial features just by using different filters.

Finding Eyes, Noses, and Mouths

Let's see how to turn a 12 by 12 candidate picture into the 3 by 3 grid required by the filters of Figure 16-40. We can do that with a series of convolutions, each followed by a pooling step. Since the filters of Figure 16-40 are trying to match eyes, a nose, and a mouth, we know that the convolution layer before these filters has to produce those features. So, let's design filters that search for them.

In Figure 16-41, we show three filters, each with a four by four footprint. They're labeled E4, N4, and M4. They look for an eye, a nose, and a mouth, respectively. The reason for placing the "4" at the end of each name will be clear in a moment.

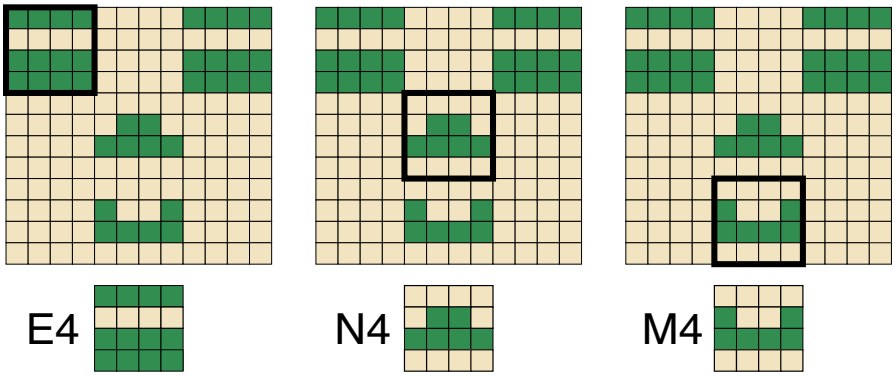


Figure 16-41: Three filters that detect an eye, nose, and mouth

We can jump right in and apply these three filters to any candidate image. Since the images are 12 by 12, and we're not padding, the outputs will be 10 by 10. If we pool those down to 3 by 3, we can then apply the filters of Figure 16-40 to the output of the filters in Figure 16-41 to determine if the candidate is a mask looking forward, or in profile, or neither.

But applying four by four filters requires a lot of computation. Worse, if we want to look for another feature (like a winking eye), we have to build another four by four filter and also apply that to the whole image. We can make our system more flexible, and also faster, by introducing another layer of convolution before this one.

What features can make up our E4, N4, and M4 filters of Figure 16-41? If we think of each four by four filter as a grid of two by two blocks, then we need only four types of two by two blocks to make up all three filters. The top row of Figure 16-42 shows those four little blocks, and the rows below that show how they can be combined to make our eye, nose, and mouth filters. We've called these T, Q, L, and R for top, quartet, lower-left corner, and lower-right corner, respectively.

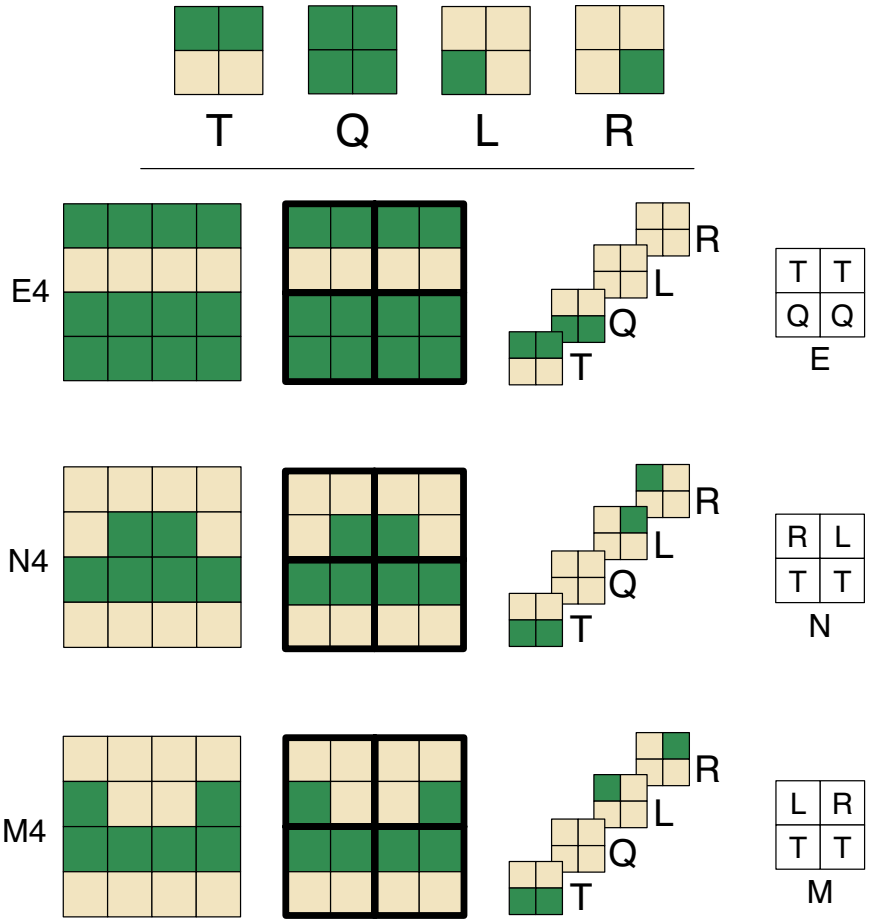


Figure 16-42: Top row: The two by two filters T, Q, L, and R. Second row, left to right: Filter E4, breaking it into four smaller blocks and the tensor form of those blocks. The far right shows the X-ray view of the two by two by four filter E. Third and fourth rows: Filters N4 and M4.

Starting with the eye filter E4, we break the four by four filter into four two by two blocks. The third drawing in the E4 row shows the four channels that we expect as input, one each for T, Q, L, and R, drawn as a single tensor where we staggered the channels. To draw that tensor more conveniently, we use the X-ray convention we saw in Figure 16-40. This gives us a new filter, of size two by two by four. This is the filter we really want to use to detect eyes, so we drop the "4" and just call this E.

The N and M filters are created by the same process of subdivision and assembly from T, Q, L, and R.

Now imagine running the little T, Q, L, and R filters over a candidate image. They're looking for patterns of pixels. Then the E, N, and M filters look for specific arrangements of T, Q, L, and R patterns. And then the F and P filters look for specific arrangements of E, N, and M patterns. Thus,

we have a series of convolution layers, with each output serving as the next layer's input. Figure 16-43 shows this graphically.

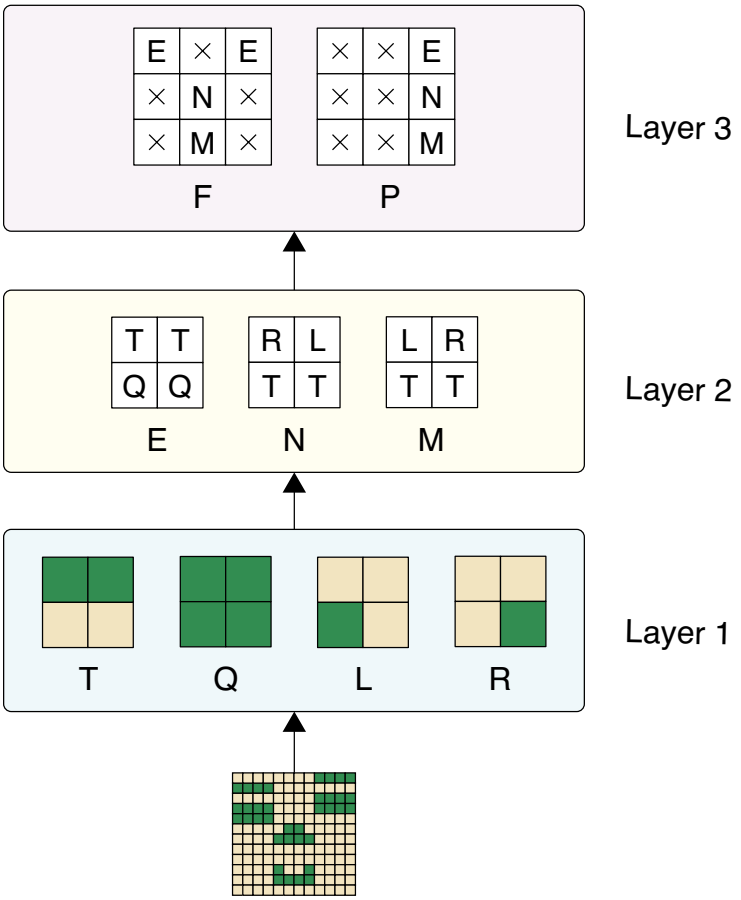


Figure 16-43: Using three layers of convolution to analyze an input candidate

Now that we have our filters, we can start at the bottom and process an input. Along the way, we'll see where to put the pooling layers.

Applying Our Filters

Let's start at the bottom of Figure 16-43 and apply the filters of Layer 1. Figure 16-44 shows the result of sweeping the T filter over the 12 by 12 candidate image. Because T is 2 by 2, it doesn't have a center, so we arbitrarily place its anchor in its upper-left corner. Because we're not padding, and the filter is 2 by 2, the output will be 11 by 11. In Figure 16-44, each location where T finds an exact match is marked in light green; otherwise, it's marked in pink. We'll call this output the T-map.

Now we want to make sure that the E, N, and M filters that are looking for T matches still succeed even if the T matches aren't exactly where our reference mask had them. As we saw in the previous section, the way to make

filters robust to small displacements in their input is to use pooling. Let's use the most common form of pooling: max pooling with two by two blocks.

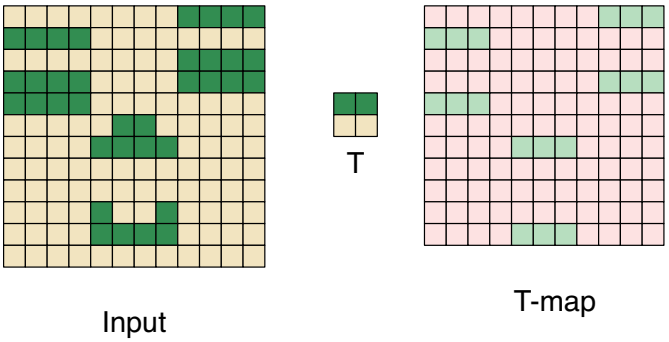


Figure 16-44: Convolving the 12 by 12 input image with the 2 by 2 filter T produces the 11 by 11 output, or feature map, which we call the T-map.

Figure 16-45 shows max pooling applied to the T-map. For each two by two block, if there's at least one green value in the block, the output is green (recall that green elements have a value of 1, and the red are 0). When the pooling blocks fall off the right and bottom sides of the input, we just ignore the missing entries and apply pooling to the values we actually have. We call the result of pooling the T-pool tensor.

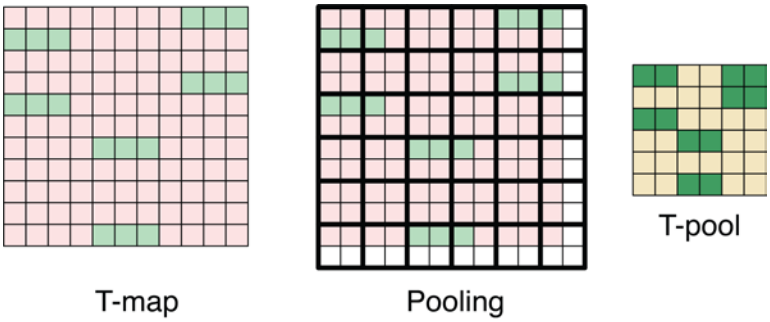


Figure 16-45: Applying two by two max pooling to the T-map to produce the T-pool tensor. Green means 1, and pink means 0.

The upper-left element of T-pool tells us if the T filter matched when placed on top of *any* of the four pixels in the upper left of the input. In this case, it did, so that element is turned green (that is, it's assigned a value of 1).

Let's repeat this process for the other three first-layer filters (Q, L, and R). The results are shown in the left part of Figure 16-46.

The four T, Q, L, and R filters together produce a result with four feature maps, each six by six after pooling. Recall from Figure 16-40 that the E, N, and M filters are expecting a tensor with four channels. To combine these individual outputs into one tensor, we can just stack them up, as in

the center of Figure 16-46. As usual, we then draw this as a 2D grid using our X-ray view convention. This gives us a tensor of four channels, which is just what Layer 2 is expecting as input.

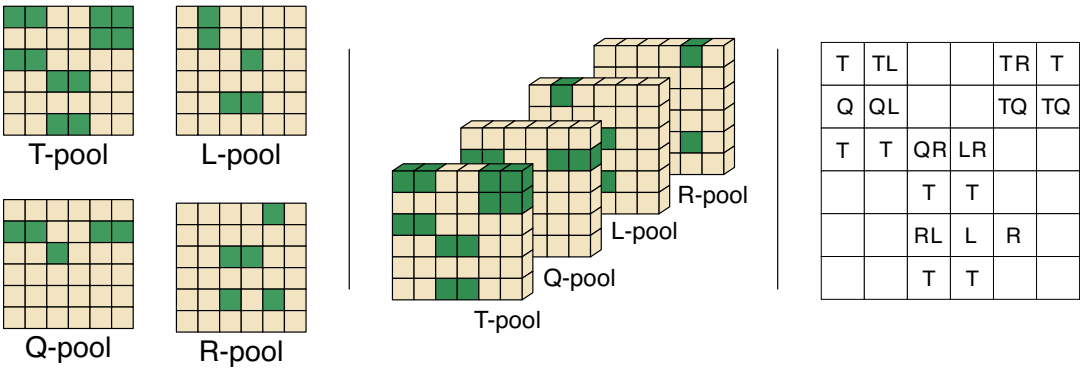


Figure 16-46: Left: The result of applying all four first-level filters to our candidate and then pooling. Center: Stacking up the outputs into a single tensor. Right: Drawing the six by six by four tensor in X-ray view.

Now we can move up to the filters in Layer 2. Let's start with E, in Figure 16-47.

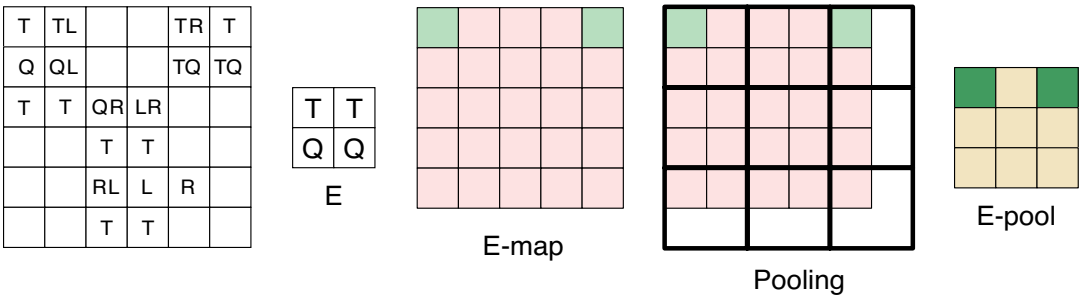


Figure 16-47: Applying the E filter. As before, from left to right, we have the input tensor, the E filter (both in our X-ray view), the result of applying that filter, the pooling grid, and the result of pooling.

Figure 16-47 shows our input tensor (the output of Layer 1) and the E filter, both in X-ray view. To their right, we see the E-map resulting from applying the E filter, the process of applying two by two pooling to the E-map, and finally the E-pool feature map. We can see already how the pooling process allows the next filter to match the locations of the eyes, even though one eye is not located where it was in the reference mask.

We can follow the same process for the N and M filters, producing a new output tensor for the second layer, as shown in Figure 16-48.

Now we have a three by three tensor with three channels, just right for the filters we created for F and P back in Figure 16-40. We're ready to move up another level to Layer 3.

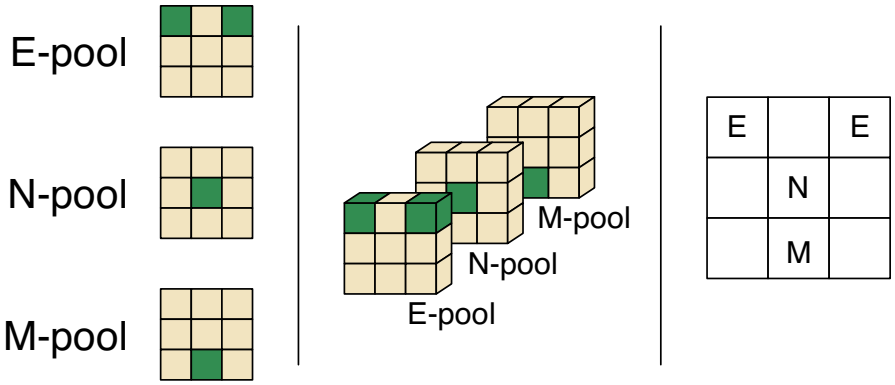


Figure 16-48: Computing outputs for the E, N, and M filters, then stacking them up into a tensor with three channels

This final step is easy: we just apply the F and P filters to their entire input, since their sizes are the same (that is, there's no need to scan the filter over the image). The result is a tensor with shape one by one by two. If the element in the first channel in this tensor is green, then F matches, and the candidate should be accepted as a match to our reference. If it's beige, the candidate's not a match.

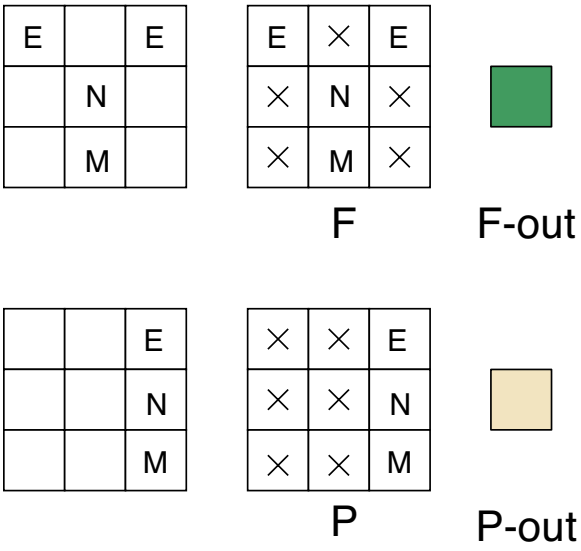


Figure 16-49: Applying the F and P filters to the output tensor of the second layer. In this layer, each filter is the same size as the input, so the layer produces an output tensor of size one by one by two.

And we're done! We used three layers of convolution to characterize a candidate image as being either like, or unlike, a reference image. We found that our candidate with one eye dropped down by one pixel was still close enough to our reference that we should accept it.

We solved this problem by creating not just a sequence of convolutions, but a hierarchy. Each convolution used the results of the previous one. The first layer looked for patterns in the pixels, the second looked for patterns of those patterns, and the third looked for larger patterns still, corresponding to a face looking forward or in profile. Pooling enabled the network to recognize a candidate even though one important block of pixels was shifted a little.

Figure 16-50 shows our whole network at a glance. Since the only layers with neurons are convolution layers, we call this an *all-convolutional network*.

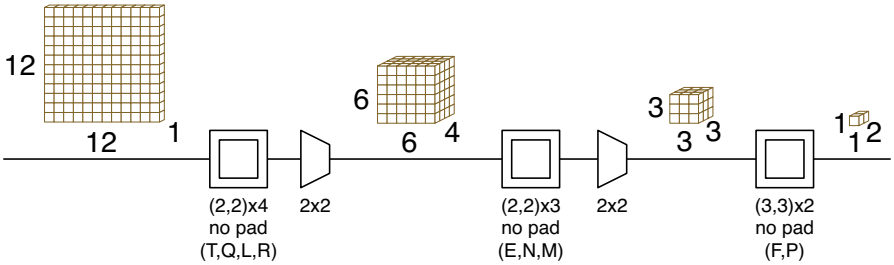


Figure 16-50: Our all-convolutional network for evaluating masks. We're also showing the input, output, and intermediate tensors. The icons with nested boxes are convolution layers, the trapezoids are pooling layers.

In Figure 16-50, the icons with a box in a box represent convolution, and the trapezoids represent pooling layers.

If we want to match even more types of faces, we can just add more filters to the final layer. This lets us match any pattern of eyes, noses, and mouths that we want, with little additional cost. By reducing the size of the tensors in our network, pooling reduces the amount of computation we have to do. This means that not only is the network with pooling more robust than a version without pooling, it also consumes less memory and runs faster.

There's a sense in which our filters are getting more powerful as we work our way up the levels. For example, our eye filter E is processing a four by four region, though it's only two by two itself, because each of its tensor elements is the result of a two by two convolution. In this way, the filters at higher levels in a hierarchy are able to look for large and complex features, even though they use only small (and therefore fast) filters.

Higher levels are able to combine the results of lower levels in multiple ways. Suppose we want to classify a variety of different birds in a photo. Low-level filters may look for feathers or beaks, while higher filters are able to combine different types of feathers or beaks to recognize different species of birds, all in a single pass through a photo. We sometimes say that using this technique of convolution and pooling to analyze an input is applying a *hierarchy of scales*.

Summary

This chapter was all about convolution: the method of taking a filter or kernel (that is, a neuron with a set of weights) and moving it over an input. Each time we apply the filter to the input, we produce a single value of output. The filter may use just a single input element in its calculation, or it may have a larger footprint and use the values of multiple input elements. If a filter has a footprint that is larger than one by one, there will be places in the input where the filter "spills" over the edge, requiring input data that isn't there. If we don't place the filter in such places, the output has a smaller width or height (or both) than the input. To avoid this, we commonly pad the input by surrounding it with enough rings of zeroes so that the filter can be placed over every input element.

We can bundle up many filters into a single convolution layer. In such a layer, typically every filter has the same footprint and the same activation function. Every filter produces one channel per filter. The output of the layer has one channel per filter.

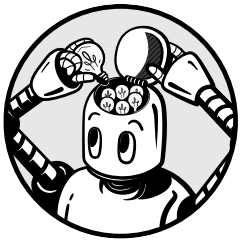
If we want to change the width and height of a tensor, we can perform downsampling (to reduce either or both dimensions) or upsampling (to increase either or both dimensions). To downsample, we can use a pooling layer, which finds the average or maximum value in blocks from the input. To upsample, we can use an upsampling layer, which duplicates input elements. Either of these techniques may be combined with the convolution step itself. To downsample, we use striding, in which the filter is moved by more than one step horizontally, vertically, or both. To upsample, we use fractional striding, or transposed convolution, in which we insert rows and/or columns of zeroes between the input elements.

We saw that by applying convolutions in a series of layers with downsampling, we are able to create a hierarchy of filters that work at different scales. This also means that the system enjoys the property of shift invariance, meaning that it's able to find the patterns it seeks even if they're not exactly where they're expected to be.

In the next chapter, we'll examine real convnets and look at their filters to see how they do their jobs.

17

CONVNETS IN PRACTICE



In the last chapter we discussed convolution, and we wrapped up with a simplified example of a convolutional network, or convnet.

In this chapter, we look at two real convnets designed for image classification. The first identifies grayscale handwritten digits, and the second identifies what object is dominant in a color photograph, choosing from 1,000 different categories.

Categorizing Handwritten Digits

Categorizing handwritten digits is a famous problem in machine learning (LeCun et al. 1989), thanks to a freely available dataset called MNIST (pronounced em´-nist). It contains 60,000 hand-drawn digits from 0 to 9, each a grayscale picture rendered in white on a 28 by 28 black background, with a label identifying the digit. The drawings were collected from census takers and students. Our job is to identify the digit in each image.

We will use a simple convnet designed for this job that is included with the Keras machine learning library (Chollet 2017). Figure 17-1 shows the architecture in our schematic form and in the traditional box-and-label form.

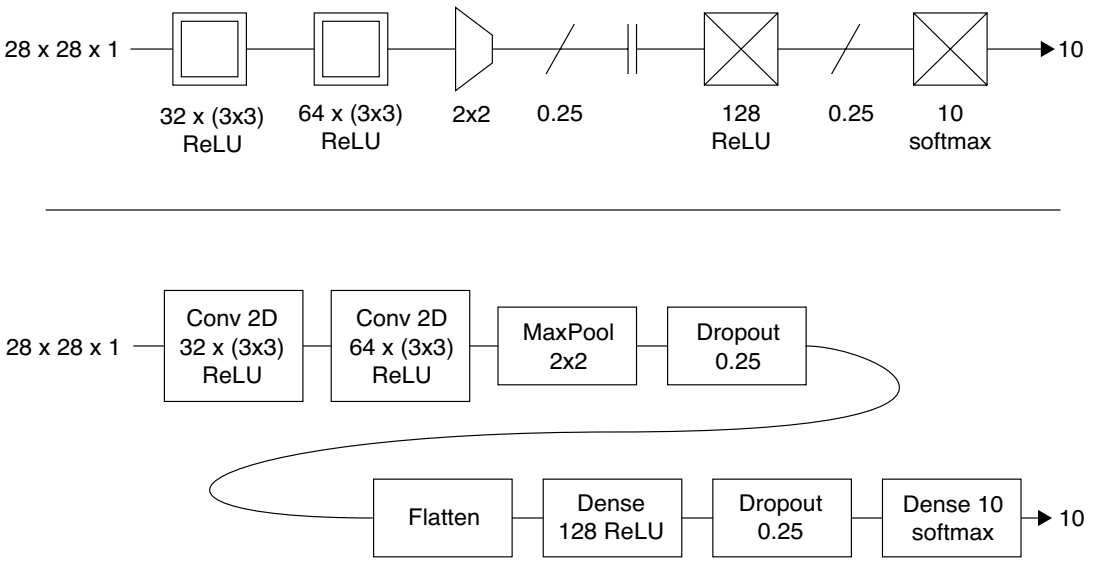


Figure 17-1: A convnet for classifying MNIST digits. The input images are 28 by 28 by 1 channel. Two convolution layers are followed by pooling, dropout, and flatten, then a dense (or fully connected) layer, another dropout, and a final dense layer with 10 outputs followed by softmax. Top: Our schematic version. Bottom: Traditional box-and-label form.

The input to the net is the MNIST image, provided as a 3D tensor of shape 28 by 28 by 1 (the 1 refers to the single grayscale channel). Though there are two fully connected layers at the end, and various helper layers (such as dropout, flatten, and pooling), we still refer to this as a convolutional network, or convnet, because the convolution layers dominate the classification work. The first convolution layer runs 32 filters, each of size 3 by 3, over the input. Each filter's output is run through a ReLU activation function before it leaves the layer.

By not specifying a stride, the filters will move by one element in each direction. We're also not applying any padding. As we saw in Figure 16-10, this means that we lose a ring of elements after each convolution. That's okay in this case because all MNIST images are supposed to have a border of four black pixels around the digit (not all images actually have this border, but most do).

The first layer's input tensor is 28 by 28 by 1, so each filter in the first convolution layer is one channel deep. Because we have 32 filters, we don't have any padding on the input, and the filters have a 3 by 3 footprint, the output of the first convolution layer is 26 by 26 by 32. The second

convolution layer contains 64 filters with 3 by 3 footprints. The system knows that the input has 32 channels (because the previous layer had 32 filters), so each filter is created as a tensor of shape 3 by 3 by 32. Because we're still not using padding, we again lose a ring around the outside of the input, producing an output tensor that's 24 by 24 by 64.

We could have used striding to reduce the size of the output, but here we use an explicit max pooling layer with blocks of size 2 by 2. That means for every nonoverlapping 2 by 2 block in the input, the layer outputs just one value containing the largest value in the block. Thus, the output of this layer is a tensor of size 12 by 12 by 64 (the pooling doesn't change the number of channels).

Next, we come to a dropout layer, represented by a diagonal slash. As we saw in Chapter 15, the dropout layer itself doesn't actually do any processing. Instead, it instructs the system to apply dropout to the nearest preceding layer that contains neurons. The nearest layer preceding the dropout is pooling, but that has no neurons. As we continue to work backward, we find a convolution layer, which does have neurons. During training, the dropout algorithm is applied to this convolution layer (recall that dropout is only applied during training, and is otherwise ignored). Before each epoch of training, one-quarter of the neurons in this convolution layer are temporarily disabled. This should help hold off overfitting. By convention, we usually treat dropout as a layer, even though it does no computation. Note that since the dropout layer looks backward for the nearest layer with neurons, we could have placed it to the left of the pooling layer and nothing about the network would have changed. By convention, when we pool after convolution, we usually place those two layers together.

Now we leave the convolutional part of the network and prepare the values for output. We typically find these steps, or something like them, at the end of classification convnets. The output of the second convolution layer is a 3D tensor, but we want to feed that into a fully connected layer, which expects a list (or 1D tensor). A *flatten* layer, shown as two parallel lines, takes an input tensor of any number of dimensions and reorganizes it into a 1D tensor by placing all the elements together end-to-end. The list is made up starting with the first row in the tensor. We take the first element, and place its 64 values at the head of our list. Then we move to the second element, and place its 64 values at the end of the list. We continue doing this for every element in the row, and then we do it for the next row, and so on. Figure 17-2 shows the process. None of the values in the tensor are lost in this rearrangement.

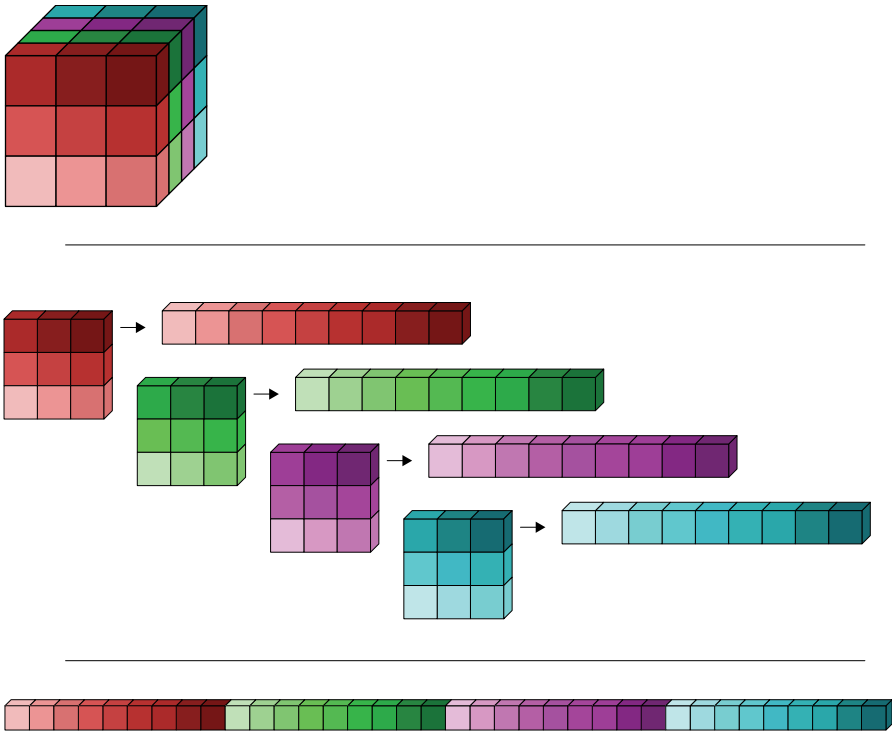


Figure 17-2: The action of a flatten layer. Top: The input tensor. Middle: Turning each channel into a list. Bottom: Placing the lists one after the other to make one large list.

Returning to Figure 17-1, the flatten layer produces a list of 12 × 12 × 64 = 9,216 numbers. That list goes into a fully connected, or dense, layer of 128 neurons. That layer gets affected by dropout, where a quarter of the neurons are temporarily disconnected at the start of each batch during training.

The 128 outputs of this layer go into a final dense layer with 10 neurons. The 10 outputs of this layer go into a softmax step so that they're converted to probabilities. The 10 numbers that come out of this last layer give us the network's prediction of the probability that the input image belongs to each of the 10 possible classes, corresponding to the digits 0 through 9.

We trained this network for 12 epochs using the standard MNIST training data. Its accuracy on the training and validation data sets is shown in Figure 17-3.

The curves show we've achieved about 99 percent accuracy on both the training and validation data sets. Since the curves aren't diverging, we've successfully avoided overfitting.

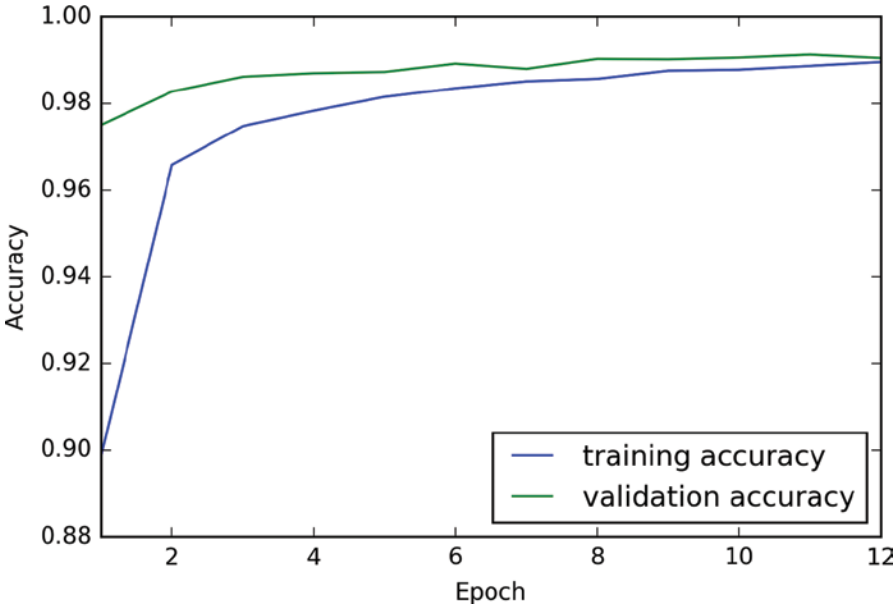


Figure 17-3: The training performance of our convnet in Figure 17-2. We trained for 12 epochs, and since the training and validation curves are not diverging, we've successfully avoided overfitting, while reaching about 99 percent accuracy on both data sets.

Let's look at some predictions. Figure 17-4 shows some images from the MNIST validation set, labeled by the digit that the network gave the largest probability to. On this little set of examples, it did a perfect job.

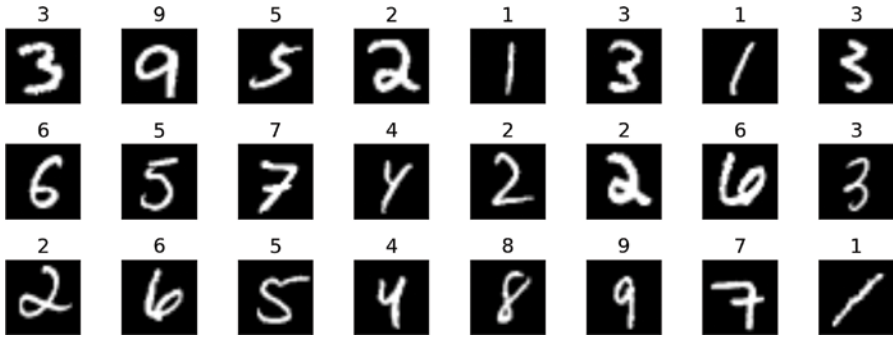


Figure 17-4: These are 24 randomly chosen images from the MNIST validation set. Each image is labeled with the output of the network, showing the digit with the highest probability. The network classified all 24 of these digits correctly.

Just two convolution layers gave this system enough power to achieve 99 percent accuracy.

VGG16

Let's look at a bigger and more powerful convnet, called *VGG16*. It was trained to analyze color photographs and identify the dominant object in each photo by assigning probabilities to 1,000 different classes.

VGG16 was trained on a famous dataset that was used as part of a contest. The ILSVRC2014 competition was a public challenge in 2014. The goal was to build a neural network for classifying photos in a provided database of images (Russakovsky et al. 2015). The acronym ILSVRC stands for ImageNet Large Scale Visual Recognition Challenge, so the database of pictures is often called the ImageNet database. The ImageNet photo database is freely available online and is still widely used for training and testing new networks (newer, bigger versions of ImageNet are also available [Fei-Fei et al. 2020]).

The original ImageNet database contained 1.2 million images, each manually labeled with one of 1,000 labels, describing the object most prominent in the photo. The challenge actually included several subchallenges, each with its own winners (ImageNet 2020). The winner of one of the classification tasks was VGG16 (Simonyan and Zisserman 2014). VGG is an acronym for the Visual Geometry Group, who developed the system. The 16 refers to the network's 16 computational layers (there are also some utility layers, such as dropout and flatten, that don't do computation).

VGG16 broke records for accuracy when it won the contest, and even though years have passed, it remains popular. This is largely because it still does very well at classifying images (even compared to newer, more sophisticated systems), and it has a simple structure that's easy to modify and experiment with. The authors have released all the weights and how they preprocessed the training data. Even better, every deep learning library makes it easy to create a fully trained instance of VGG16 in our own code. Thanks to all of these qualities, VGG16 is a frequent starting point for projects that involve image classification.

Let's look at the VGG16 architecture. Most of the work is done by a series of convolution layers. Utility layers appear along the way, and some flattening and fully connected layers appear at the very end, as they did in Figure 17-1.

Before we feed any data to our model, we must preprocess it in the same way that the authors preprocessed their training data. That involves making sure that each channel has been adjusted by subtracting a specific value from all of its pixels (Simonyan and Zisserman 2014). To better discuss the shapes of the tensors flowing through the network, let's assume each input image has a height and width of 224 to match the dimensions of the Imagenet data the network was trained on, and its colors have been correctly pre-processed. Once that's done, we're ready to feed our image to the network.

We will present the VGG16 architecture as a series of six groups of layers. These groups are strictly conceptual and are just a way of gathering together related layers for our discussion. The first few groups have the same structure: two or three layers of convolution followed by a pooling layer.

Group 1 is shown in Figure 17-5.

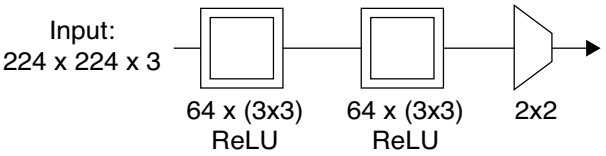


Figure 17-5: Group 1 of VGG16. We convolve the input tensor with 64 filters each of size 3 by 3. Then we convolve again with 64 new filters. Finally, we use max pooling to reduce the output tensor's height and width by half.

The convolutions both apply zero padding to their inputs so there's no loss in width or height. The max pooling step uses nonoverlapping blocks of size 2 by 2.

All of the convolution layers in VGG16 use the default ReLU activation function.

We've seen how useful pooling is for helping our filters recognize patterns even if they've been displaced. For the same reasons that we used pooling when matching masks in Chapter 16, we apply pooling here, too.

The output of the group in Figure 17-5 is a tensor of dimensions 112 by 112 by 64. The values of 112 come from the input dimensions of 224 by 224 that have been halved, and the 64 results from the 64 filters in the second convolution layer.

Group 2 is just like the first, only now we apply 128 filters in each convolution layer. Figure 17-6 shows the layers. The output of this group has size 56 by 56 by 128.

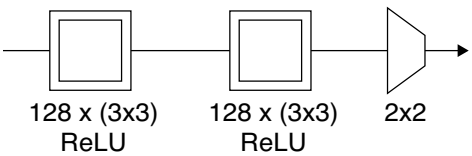


Figure 17-6: Group 2 of VGG16 is just like the first block in Figure 17-5, except that we use 128 filters in each convolution layer rather than 64.

Group 3 continues the pattern of doubling the number of filters in each convolution layer, but it repeats the convolution step three times instead of twice. Figure 17-7 shows Group 3. The tensor after the max pooling step has size 28 by 28 by 256.

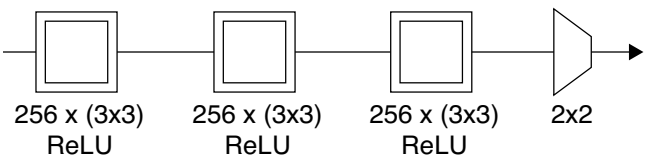


Figure 17-7: Group 3 of VGG16 doubles the number of filters again to 256 and repeats the convolution step three times rather than two as before.

Groups 4 and 5 of the network are the same. Each group is built from three steps of convolution with 512 filters, followed by a max pooling layer. The structure of these layers is shown in Figure 17-8. The tensor coming out of Group 4 has size 28 by 28 by 512, and the tensor after the max pooling layer in Group 5 has dimensions 14 by 14 by 512.

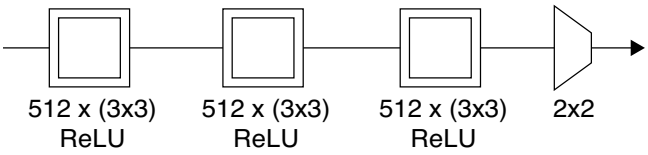


Figure 17-8: Groups 4 and 5 of VGG16 are the same. They each have three convolution layers, followed by a two by two max pooling layer.

This ends the convolution part of the network, and now we come to the wrap-up. As with the MNIST classifier we saw in Figure 17-1, we first flatten the tensor coming out of Group 5. We then run it through two dense layers of 4,096 neurons, each using ReLU, and each followed by dropout with an aggressive setting of 50 percent. Finally, the output goes into a dense layer with 1,000 neurons. The results are fed to softmax, which produces our output of 1,000 probabilities, one for each class that VGG16 was trained to recognize. These final steps, which are typical for classification networks of this style, are shown in Figure 17-9.

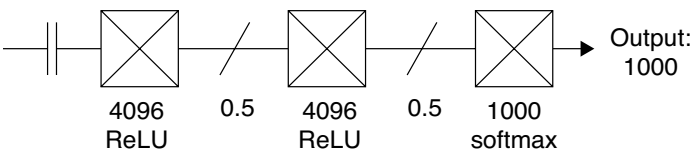


Figure 17-9: The final steps of processing in VGG16. We flatten the image, then run it through two dense layers each using ReLU, followed by dropout, then through a dense layer with softmax.

Figure 17-10 shows the whole architecture in one place.

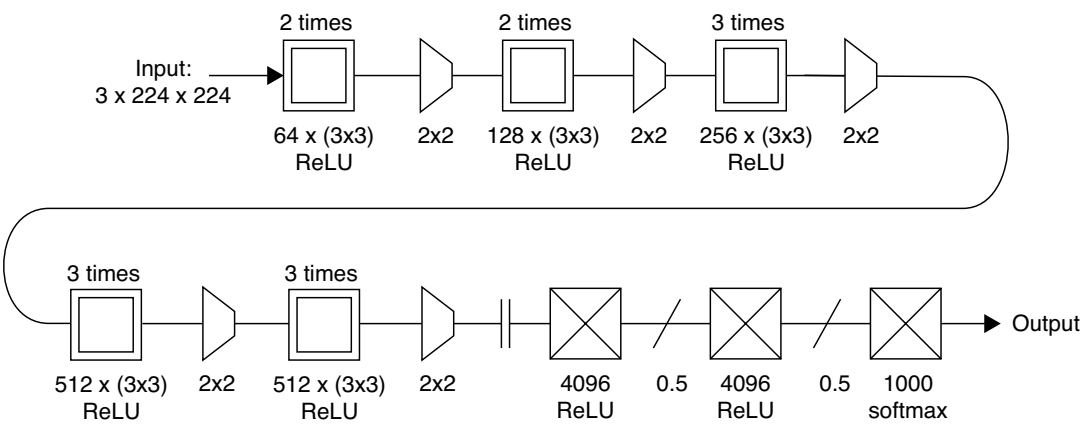


Figure 17-10: The VGG16 architecture in one place

This network works very well. Figure 17-11 shows four pictures shot around Seattle on a phone's camera.

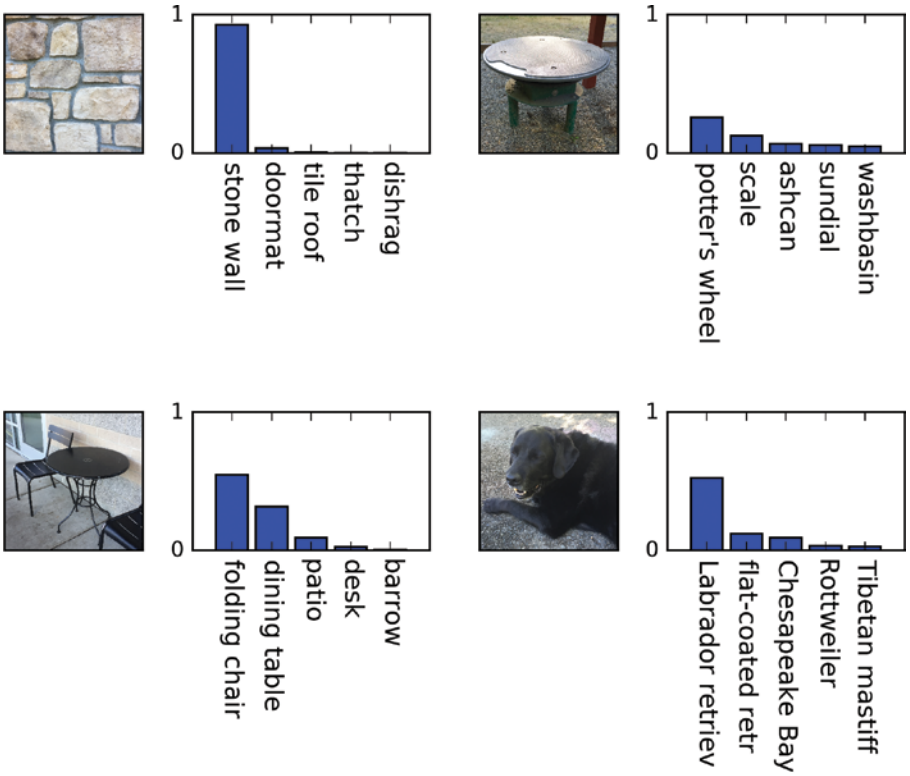


Figure 17-11: Four photos shot around Seattle on a sunny day. The convnet of Figure 17-10 does a great job of identifying each image.

The convnet has never seen these images, but it does a great job with them. Even the ambiguous round object in the upper right is assigned sensible labels.

Let's take a closer look at what's going on inside VGG16 by looking at its filters.

Visualizing Filters, Part 1

VGG16's success in classifying is due to the filters that were learned by its convolution layers. It's tempting to look at the filters and see what they've learned, but the filters themselves are big blocks of numbers, which are hard for us to interpret. Instead of trying to somehow make sense of a block of numbers, we can visualize our filters indirectly by creating images that trigger them. In other words, once we've selected a filter we want to visualize, we can find a picture that causes that filter to output its biggest value. That picture shows us what that filter is looking for.

We can do this with a little trick based on gradient descent, the algorithm that we saw in Chapter 14 as part of backpropagation. We flip gradient descent around to create gradient *ascent*, which we use to climb up the gradient and increase the system's error. Remember from Chapter 14 that during training, we use the system's error to create gradients that we push backward through the network with backprop, enabling us to change the weights in order to reduce that error. For filter visualization, we're going to ignore the network's output and its error entirely. The only output we care about is the feature map that comes out of the particular filter (or neuron) we want to visualize. We know that when the filter sees information that it's looking for, it produces a big output, so if we add up all of the output values of that filter for a given input image, it tells us how much of what the filter is looking for is in that image. We can use the sum of all the values in the feature map as a replacement for the network's error.

Figure 17-12 shows the idea.

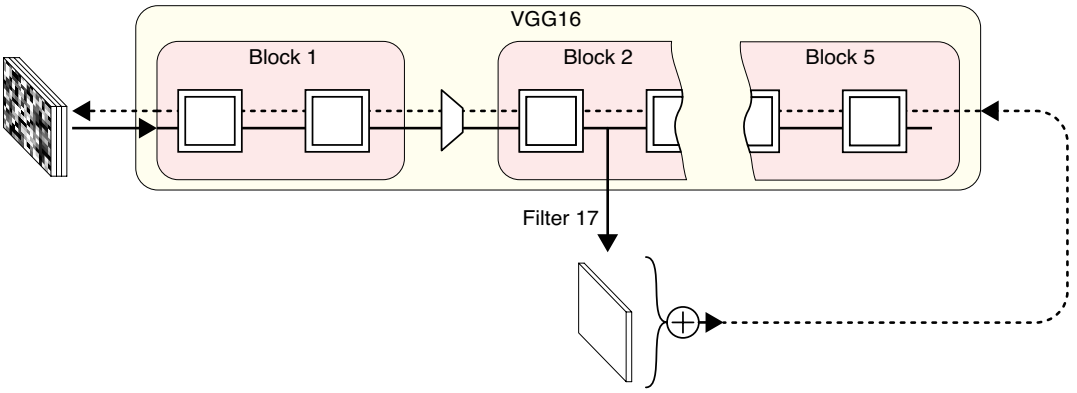


Figure 17-12: Visualizing a filter. The sum of all the values in the feature map serves as the network's error.

We're using VGG16, but for this visualization process we leave off the layers after the last convolution. We feed in a grid of random numbers and extract the filter map for the filter we want to visualize. That becomes our error. Now comes the tricky part: we use this error to compute the gradients, but we don't adjust the weights at all. The network itself and all of its weights are *frozen*. We just keep computing the gradients and pushing them back until we reach the input layer, which holds the pixel values of the input image. The gradients that arrive at this layer tell us how to change those pixel values to decrease the error, which we know is the filter's output. Since we want to stimulate the neuron as much as we can, we want the "error" to be as big as possible, so we change the pixel values to increase, rather than decrease, this error. That makes the picture stimulate our selected neuron a little more than it did before.

After doing this over and over, we will have adjusted our initially random pixel values so that they're making the filter output the biggest values we can get it to produce. When we look at the input after it's been modified

in this way, we see a picture that makes that neuron produce a huge output, so the picture shows us what the filter is looking for (or at least gives us a general idea) (Zeiler and Fergus 2013). We will use this visualization process again in Chapter 23 when we look at the deep dreaming algorithm.

Because we start with random values in the input image, we get a different final image every time we run this algorithm. But each image we make is roughly like the others, since they're all based on maximizing the output of the same filter.

Let's look at some images produced by this method. Figure 17-13 shows pictures produced for the 64 filters in the second convolution layer in the first block, or group, of VGG16 (we use the label `block1_conv2` for this layer and similar names for the other layers we look at). In Figure 17-13 and the others like it to come, we've enhanced the color saturation to make the results easier to interpret.

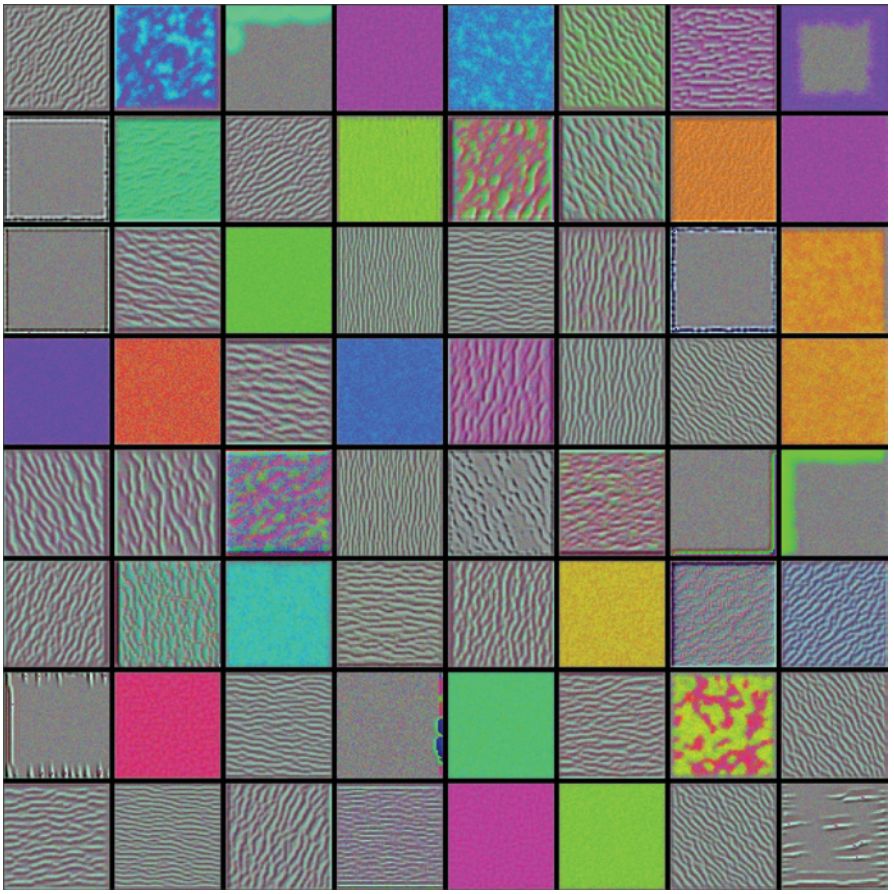


Figure 17-13: Images that get the biggest response from each of the 64 filters in the `block1_conv2` layer of VGG16

It seems that a lot of these layers are looking for edges in different orientations. Some have values that are too subtle for us to interpret easily.

Let's move forward to block 3, and look at the first 64 filters from the first convolution layer there. Figure 17-14 shows images that stimulate these filters the most.

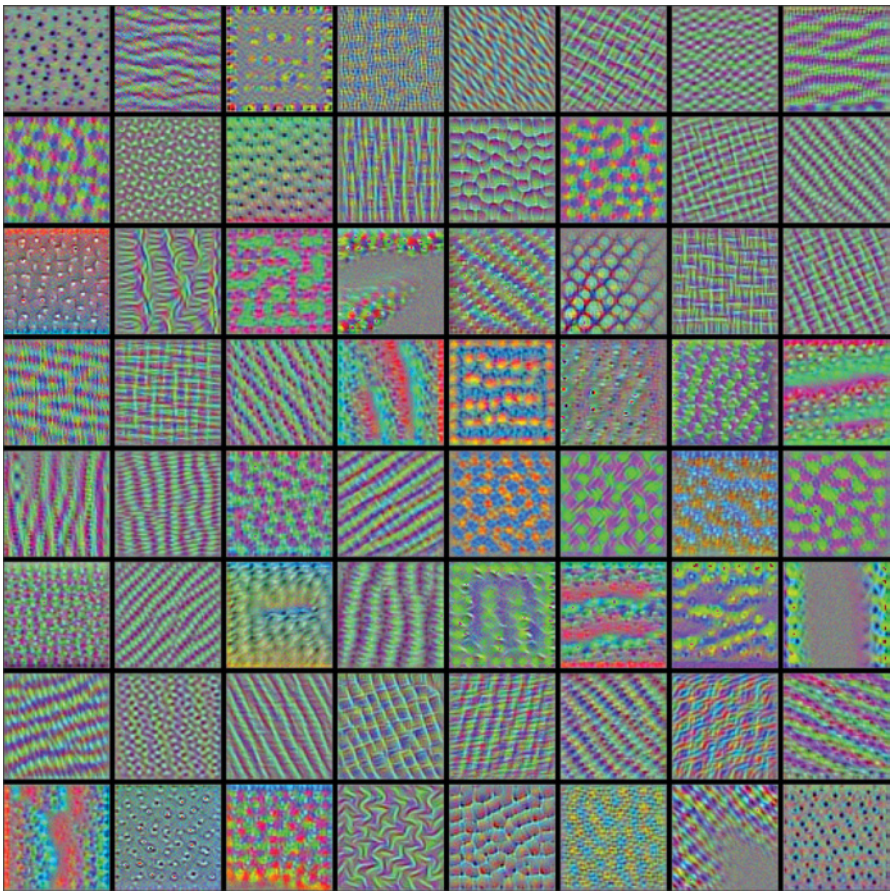


Figure 17-14: Images that get the biggest response from the first 64 filters in the `block3_conv1` layer of VGG16

Now we're talking! As we'd expect, the filters here are looking for more complex textures, combining the simpler patterns found by prior layers. Let's move farther along and look at the first 64 filters from the first convolution layer of block 4, in Figure 17-15.

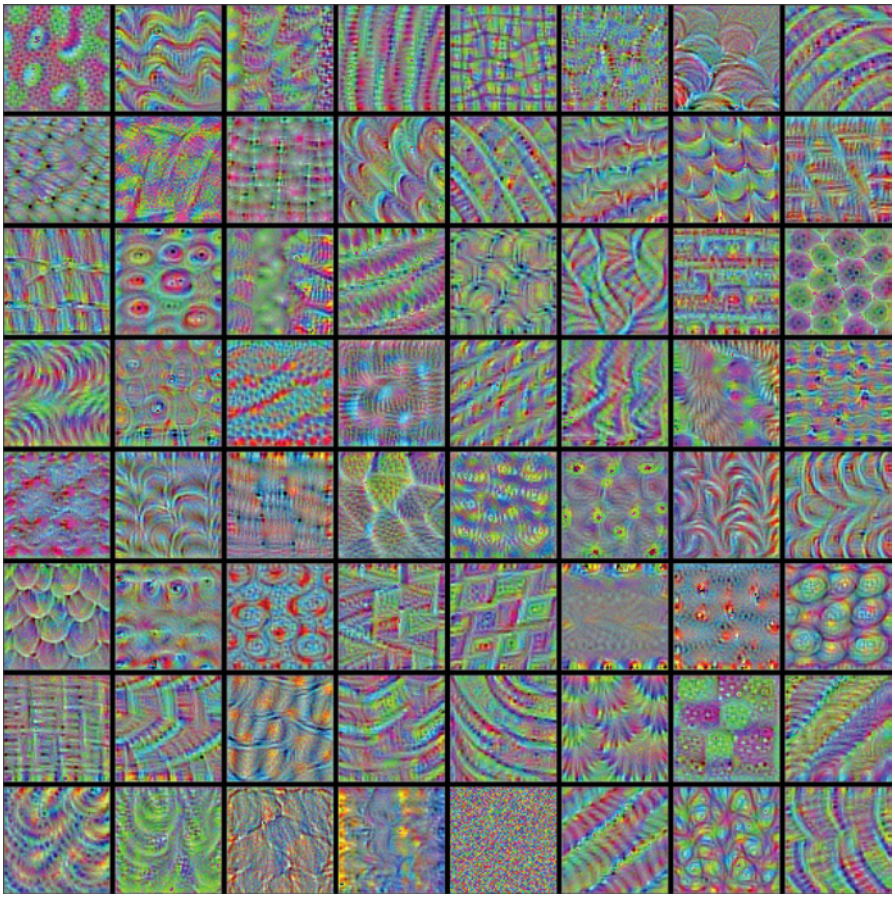


Figure 17-15: Images that get the biggest response from the first 64 filters in the `block4_conv1` layer of VGG16

These are fascinating glimpses into what VGG16 has learned. We can see some of the structures it has found to be useful in order to classify the object in an image. The filters seem to be hunting for patterns that involve a lot of different kinds of flowing and interlocking textures like those we'd find on animals and other surfaces in the world around us.

We can really see the value of the convolution hierarchy here. Each layer of convolution looks for patterns in the output of the previous layer, letting us work our way up from low-level details like stripes and edges to complex and rich geometrical structures.

Just for fun, let's look at close-ups of a few of these filters. Figure 17-16 shows larger views of nine patterns from the first few layers.

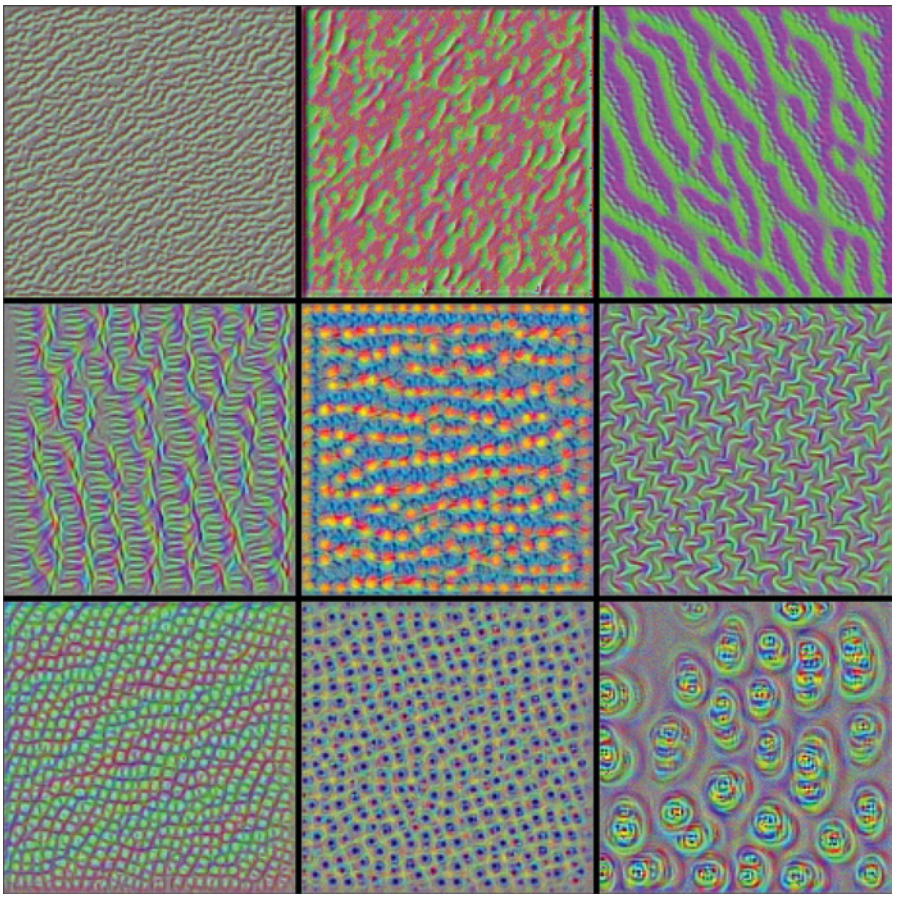


Figure 17-16: Close-ups of some manually selected images that triggered the largest filter responses from the first few layers of VGG16

Figure 17-17 shows patterns that triggered big responses from filters in the last few layers.

These patterns are exciting and beautiful. They also have an organic feeling about them, probably because the ImageNet database contains many images of animals.

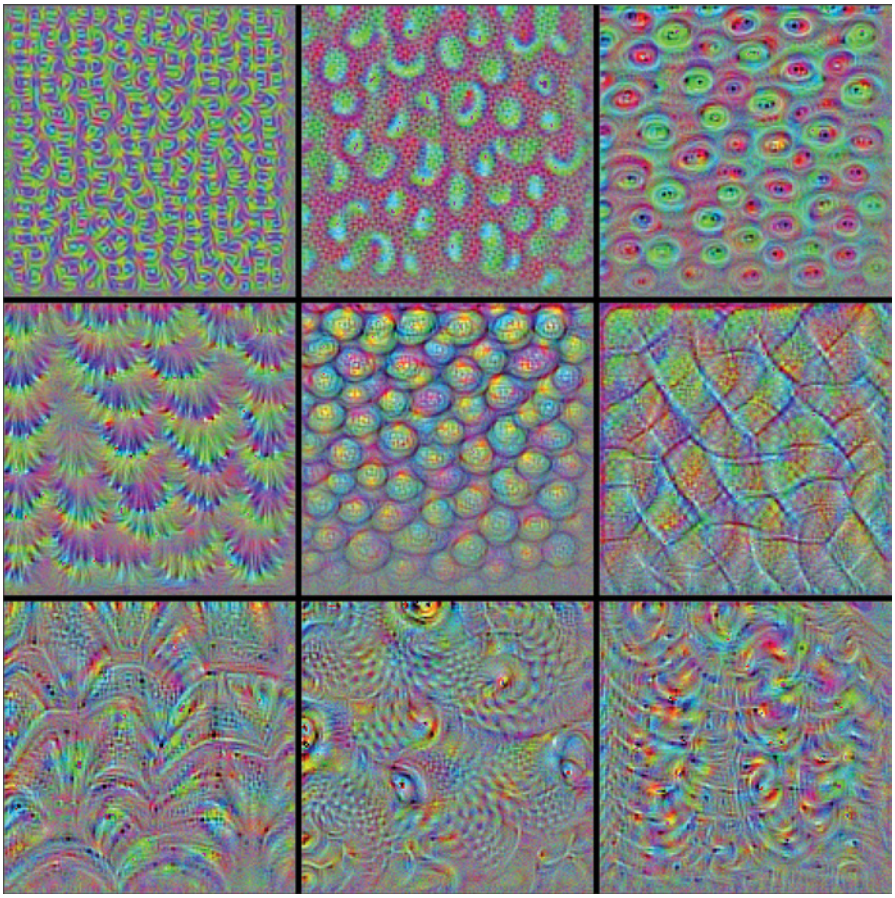


Figure 17-17: Close-ups of some manually selected images that triggered the largest filter responses from the last few layers of VGG16

Visualizing Filters, Part 2

Another way to visualize a filter is to run an image through VGG16, and look at the feature map produced by that filter. That is, we feed an image to VGG16 and let it run through the network, but as before, we ignore the network's output. Instead, we extract the feature map for the filter we're interested in, and draw it like a picture. This is possible because each feature map always has a single channel, so we can draw it as a grayscale image.

Let's give it a spin. Figure 17-18 shows our input image of a drake, or male duck. This is the starting image for all of our visualizations in this section.



Figure 17-18 The drake image that we use to visualize filter outputs

To get a feeling for things, Figure 17-19 shows the response from the very first filter on the very first convolution layer of the network. Since the output of a filter has just one channel, we can draw it in grayscale. We've chosen to instead use a heatmap from black to reds to yellow.

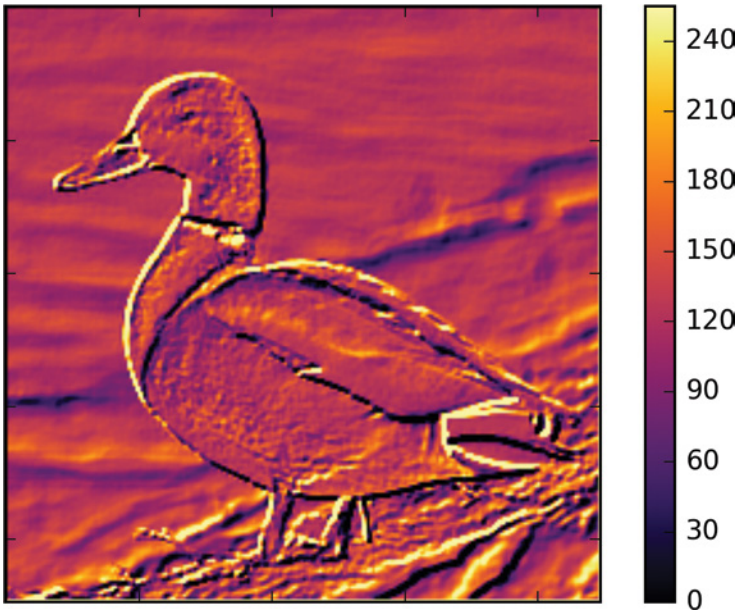


Figure 17-19: The response of filter 0 in layer `block1_conv1` in VGG16 to the duck image in Figure 17-18

This filter is looking for edges. Consider the tail in the lower right. An edge that's light on top and darker below gets a very large output from the filter, whereas an edge in the other direction gets a very low output. Less extreme changes cause smaller outputs, and regions of constant color have middling outputs.

Figure 17-20 shows the responses from the first 32 filters in the first convolution layer of the first block.

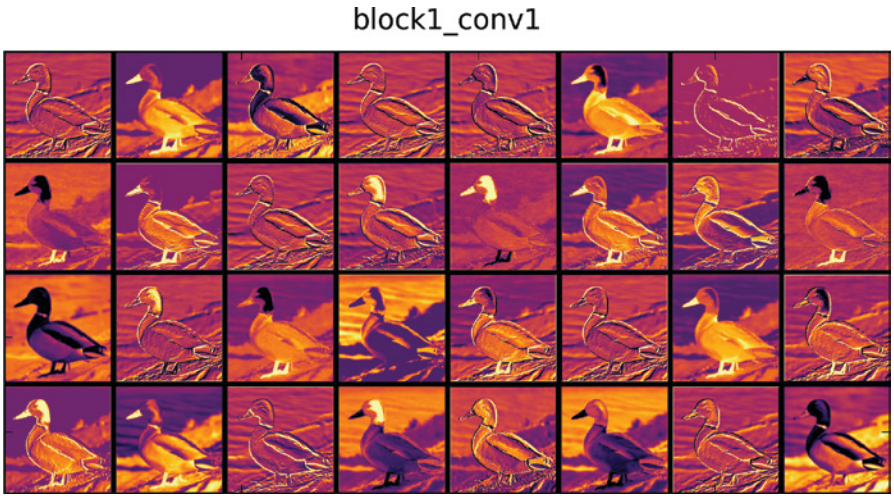


Figure 17-20: The responses of the first 32 filters in VGG convolution layer `block1_conv1`

A lot of these filters seem to be looking for edges, but others seem to be looking for particular features of the image. Let's look at close-ups of 8 manually selected filters chosen from all 64 of the filters on this layer, shown in Figure 17-21.



Figure 17-21: Close-ups of eight manually chosen filter responses from VGG16's first convolution layer, `block1_conv1`

The third image in the top row seems to be looking for the duck's feet, or maybe it's just interested in bright orange things. The left-most image in the bottom row looks like it's searching for the waves and sand behind the duck, though the image to its right appears to be responding most to the blue waves. Some more experimentation with other inputs would help us nail down these interpretations, but it's fun to see how much we can guess from a single image.

Let's move farther into the network, out to the third block of convolution layers. The outputs here are smaller by a factor of four on each side than those coming out of the first block because they've gone through two pooling layers. We expect that they are looking for clusters of features. Figure 17-22 shows the responses for the first convolution layer in block 3.

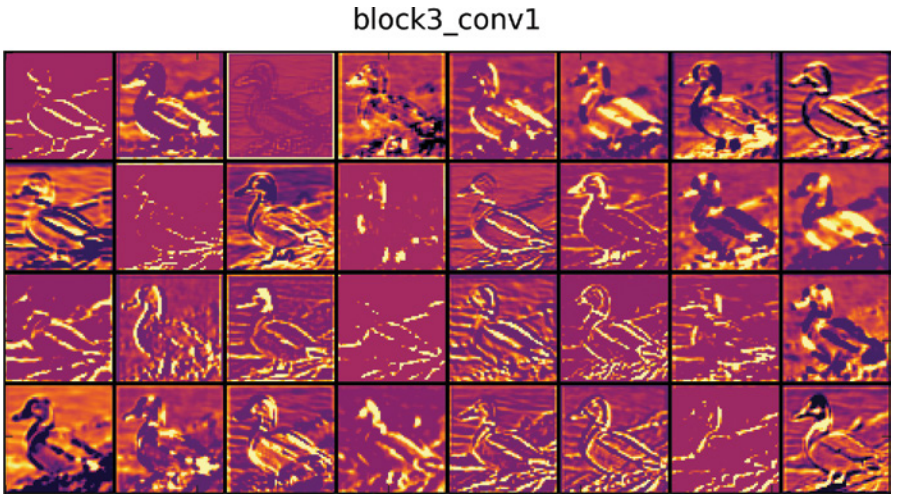


Figure 17-22: The responses of the first 32 filters in the VGG convolution layer `block3_conv1`

It's interesting that a lot of edge finding still seems to be going on. This suggests that strong edges are an important cue for VGG16 as it works to figure out what an image is showing, even in the third set of convolutions. But lots of other regions are also bright.

Let's jump all the way to the last block. Figure 17-23 shows the responses for the first 32 filters for the first convolution layer in block 5.

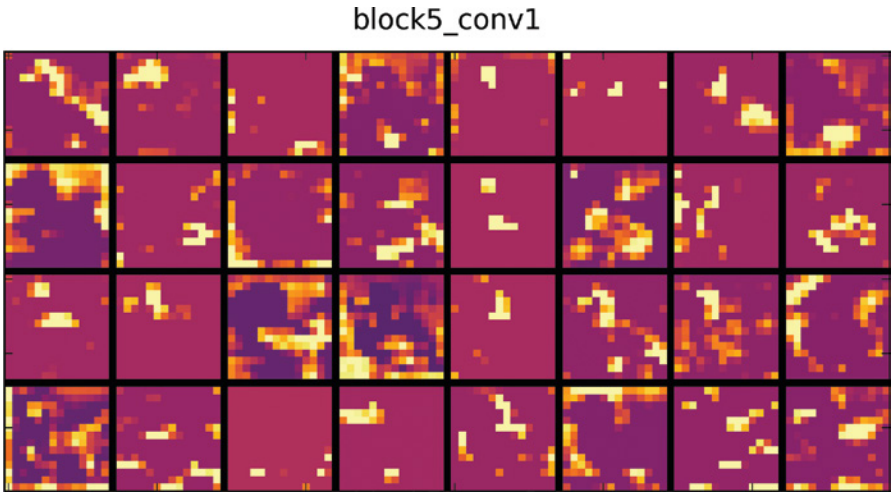


Figure 17-23: Filter responses for the first 32 filters in VGG convolution layer `block5_conv1`

As we'd expect, these images are even smaller, having passed through two more pooling layers that each reduce the size by a factor of two on each side. At this point, the duck is hardly visible because the system is combining features from the previous layers. Some of the filters are barely responding. They are probably responsible for finding high-level features that aren't present in the duck image.

In Chapter 23 we'll look at a couple of creative applications that use the filter responses in a convnet.

Adversaries

Although VGG16 does very well at predicting the correct label for many images, we can change an image in ways so small that they're undetectable to the human eye, but that fools the classifier into assigning the wrong label. In fact, this process can mess up the results of any convolution-based classifier.

The trick to fooling a convnet involves creating a new image called an *adversary*. This image is created from the starting image by adding an *adversarial perturbation* (or more simply, a *perturbation*). The perturbation is another image, the same size as the image we want to classify, typically with very small values. If we add the perturbation to our original image, the changes are usually so small that most people can't detect any difference, even in the finest details. But if we ask VGG16 to classify the perturbed image, it gives us the wrong answer. Sometimes we can find a single perturbation that messes up the results for every image we give to a particular classifier, which we call a *universal perturbation* (Moosavi-Dezfooli et al. 2016).

Let's see this in action. On the left of Figure 17-24 we see an image of a tiger. All of the pixel values in this image are between 0 and 255. The system correctly classifies it as a tiger with about 80 percent confidence, with smaller confidences for related animals such as a tiger cat and a jaguar.

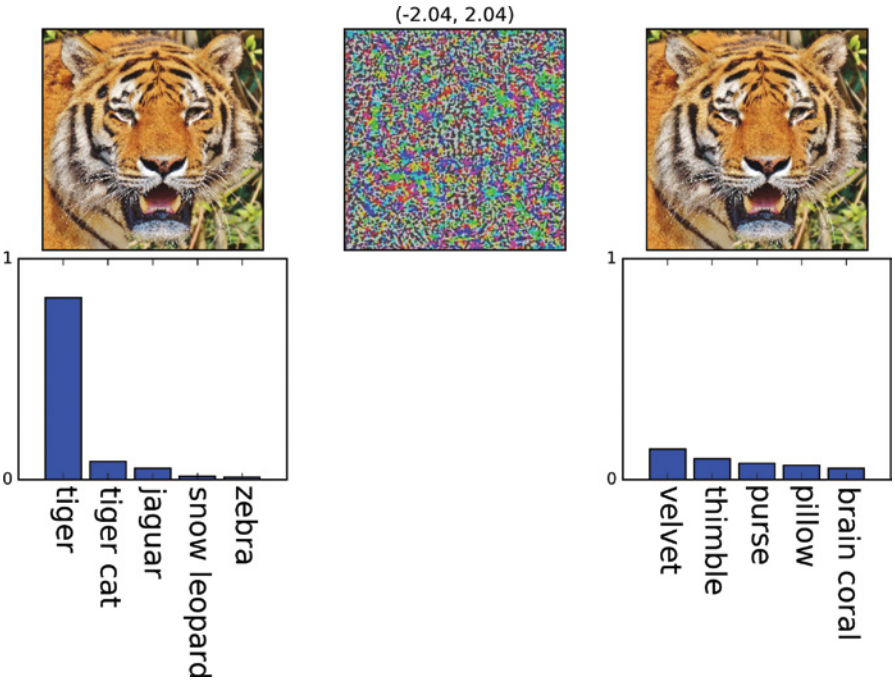


Figure 17-24: An adversarial attack on an image. Left: The input and VGG16's top five classes. Middle: The adversarial image, where the pixel values are in the range of about [–2, 2], but shown here scaled to the range [0, 255] so they can be seen. Right: The result of adding the image and the original (unscaled) adversary together, and the new top five classes.

In the middle of Figure 17-24 we show an image computed by an algorithm designed to find adversaries. All of the values in this image are about in the range [–2, 2], but for this figure, we scaled the values to the range [0, 255] so they'd be easier to see. In the top right of Figure 17-23 we show the result of adding the tiger and the adversary, so each of the original tiger's pixels is changed by a value within the range [–2, 2]. To our eyes, the tiger seems unchanged. Even the thin whiskers look the same. Below that image are VGG16's top five predictions for this new image. The system comes up with completely different predictions for the image, none of which come anywhere close to the correct class. Except for the low-probability class of brain coral, the system doesn't even think this image is an animal.

The perturbation image in Figure 17-24 may look random to our eyes, but it's not. This picture was specifically computed to throw off VGG16's prediction for the image of the tiger.

There are many different ways to compute adversarial images (Rauber, Brendel, and Bethge 2018). The range of values in the perturbations these

methods create for a given image can vary considerably, so to find the smallest perturbation, it's often worth trying a few different methods, also called *attacks*. We can compute adversaries to achieve different goals (Rauber and Brendel 2017b). For example, we can ask for a perturbation that simply causes the input to be misclassified. Another option asks for a perturbation that causes the input to be classified as a specific, desired class. To make Figure 17-24, we used an algorithm that is designed to make the classifier's top seven predictions much more unlikely. That is, it takes in the starting image and the top seven predictions from the classifier and produces an adversary. When we add the adversary to the input and hand that to the classifier, none of its new top seven predictions contain any of the previous top seven predictions.

We have to carefully construct adversarial perturbations, which suggests that they're exploiting something subtle in our convnets.

We may find a way to build convnets that resist these attacks, but convolutional networks may be inherently vulnerable to these subtle image manipulations (Gilmer et al. 2018). The existence of adversaries suggests that convnets still hold surprises for us, and they shouldn't be considered foolproof. There's more to be learned about what's going on inside of convolutional networks.

Summary

In this chapter we looked at a couple of real convnets: a small one for classifying handwritten MNIST digits and the larger VGG16 network for classifying photos. Though our MNIST network was quite small, it was able to classify digits with about 99 percent accuracy.

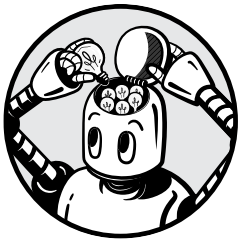
We looked at the structure of VGG16, and then two different types of visualizations of its filters. We saw that the filters in this network start by looking for simple structures like edges and build up to complex and beautiful organic patterns.

Finally, we saw that convolutional networks used as image classifiers are susceptible to being fooled by adjusting the pixel values by tiny amounts that are imperceptible to a human observer.

In the next chapter we'll look at how to build networks that figure out how to compress an input into a much smaller representation and then expand that again to produce something close to the original.

18

AUTOENCODERS



This chapter is about a particular kind of learning architecture called an *autoencoder*. One way to think about a standard autoencoder is that it's a mechanism for compressing input, so it takes up less disk space and can be communicated more quickly, much as an MP3 encoder compresses music, or a JPG encoder compresses an image. The autoencoder gets its name from the idea that it *automatically* learns, by virtue of training, how best to *encode*, or represent, the input data. In practice, we usually use autoencoders for two types of jobs: removing the noise from a dataset, and reducing the dimensionality of a dataset.

We begin this chapter by looking at how to compress data while preserving the information we care about. Armed with this information, we look at a tiny autoencoder. We'll use it to get our bearings and to discuss key ideas about how these systems work and how their version of representing the data lets us manipulate it in meaningful ways. Then we'll make a

bigger autoencoder and look more closely at its data representation. We'll see that the encoded data has a surprising amount of natural structure. This enables us to use the second half of the autoencoder as a standalone *generator*. We can feed the generator random inputs and get back new data that looks like the training data but is, in fact, a wholly new piece of data.

We then expand our network's usefulness by including convolution layers, which enable us to work directly with images and other 2D data. We will train a convolution-based autoencoder to *denoise* grainy images, giving us back a clean input. We wrap the chapter up with a look at *variational autoencoders*, which create a more nicely organized representation of the encoded data. This makes it even easier to use the second part as a generator, since we will have better control over what kind of data it will produce.

Introduction to Encoding

Compressing files is useful throughout computing. Many people listen to their music saved in the MP3 format, which can reduce an audio file by a huge amount while still sounding acceptably close to the original (Wikipedia 2020b). We often view images using the JPG format, which can compress image files down by as much as a factor of 20 while still looking acceptably close to the original image (Wikipedia 2020a). In both cases, the compressed file is only an approximation of the original. The more we compress the file (that is, the less information we save), the easier it is to detect the differences between the original and the compressed version.

We refer to the act of compressing data, or reducing the amount of memory required to store it, as *encoding*. Encoders are part of everyday computer use. We say that both MP3 and JPG take an *input* and *encode* it; then we *decode* or *decompress* that version to *recover* or *reconstruct* some version of the original. Generally speaking, the smaller the compressed file, the less well the recovered version matches the original

The MP3 and JPG encoders are entirely different, but they are both examples of *lossy encoding*. Let's see what this means.

Lossless and Lossy Encoding

In previous chapters we used the word *loss* as a synonym for error, so our network's error function was also called its loss function. In this section, we use the word with a slightly different meaning, referring to the degradation of a piece of data that has been compressed and then decompressed. The greater the mismatch between the original and the decompressed version, the greater the loss.

The idea of loss, or degradation of the input, is distinct from the idea of making the input smaller. For example, in Chapter 6, we saw how to use Morse code to carry information. The translation of letters to Morse code symbols carries no loss, because we can exactly reconstruct the original message from the Morse version. We say that converting, or encoding, our message into Morse code is a *lossless* transformation. We're just changing format, like changing a book's typeface or type color.

To see where loss can get involved, let's suppose that we're camping in the mountains. On a nearby mountain, our friend Sara is enjoying her birthday. We don't have radios or phones, but both groups have mirrors, and we've found we can communicate between the mountains by reflecting sunlight off our mirrors, sending Morse code back and forth. Suppose that we want to send the message, "HAPPY BIRTHDAY SARA BEST WISHES FROM DIANA" (for simplicity, we leave out punctuation). Counting spaces, that's 42 characters. That's a lot of mirror-wiggling. We decide to leave out the vowels, and send "HPP BRTHD SR BST WSHS FRM DN" instead. That's only 28 letters, so we can send this in about two-thirds the time of the full message.

Our new message has lost some information (the vowels) by being compressed in this way. We say that this is a *lossy* method of compression.

We can't make a blanket statement about whether it is or isn't okay to lose some information from any message. If there is loss, then the amount of loss we can tolerate depends on the message and all the context around it. For example, suppose that our friend Sara is camping with her friend Suri, and it just happens that they share a birthday. In this context, "HPP BRTHD SR" is ambiguous, because they can't tell who we're addressing.

An easy way to test if a transformation is lossy or lossless is to consider if it can be *inverted*, or run backward, to recover the original data. In the case of standard Morse code, we can turn our letters into dot-dash patterns and then back to letters again with nothing lost in the process. But when we deleted the vowels from our message, those letters were lost forever. We can usually guess at them, but we're only guessing and we can get it wrong. Removing the vowels creates a compressed version that is not invertible.

Both MP3 and JPG are lossy systems for compressing data. In fact, they're very lossy. But both of these compression standards were carefully designed to throw away just the "right" information so that in most everyday cases, we can't tell the compressed version from the original.

This was achieved by carefully studying the properties of each kind of data and how it was perceived. For example, the MP3 standard is based not just on the properties of sound in general, but on the properties of music and of the human auditory system. In the same way, the JPG algorithm is not only specialized toward the structure of data within images, but it also builds on science describing the human visual system.

In a perfect but impossible world, compressed files are tiny, and their decompressed versions match their corresponding originals perfectly. In the real world, we trade off the *fidelity*, or accuracy, of the decompressed image for the file size. Generally speaking, the bigger the file, the better the decompressed file matches the original. This makes sense in terms of information theory: a smaller file holds less information than a larger one. When the original file has redundancy, we can exploit that to make a lossless compression in a smaller file (for example, when we compress a text file using the ZIP format). But in general, compression usually implies some loss.

The designers of lossy compression algorithms work hard to selectively lose just the information that matters to us the least for that particular

type of file. Often this question of "what matters" to a person is an issue of debate, leading to a variety of different lossy encoders (such as FLAC and AAC for audio, and JPEG and JPEG 2000 for images).

Blending Representations

Later in this chapter, we will find numerical representations of multiple inputs and then *blend* those to create new data that has aspects of each input. There are two general approaches to blending data. We can describe the first as *content blending*. That's where we blend the content of two pieces of data with each other. For example, content blending the images of a cow and zebra gives us something like Figure 18-1.



Figure 18-1: Content blending images of a cow and zebra. Scaling each by 50 percent and adding the results together gives us a superposition of the two images, rather than a single animal that's half cow and half zebra.

The result is a combination of the two images, not an in-between animal that is half cow and half zebra. To get a hybrid animal, we would use a second approach, called *parametric blending*, or *representation blending*. Here we work with parameters that describe the thing we're interested in. By blending two sets of parameters, depending on the nature of the parameters and the algorithm we use to create the object, we can create results that blend the inherent qualities of the things themselves.

For example, suppose we have two circles, each described by a center, radius, and color, as in Figure 18-2.

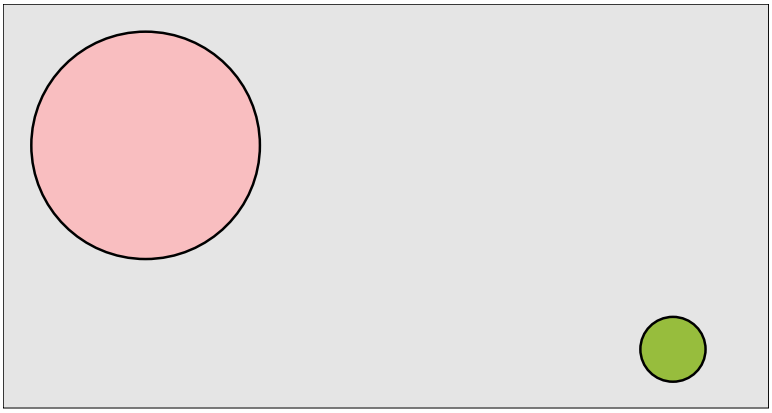


Figure 18-2: Two circles we'd like to blend

If we blend the parameters (that is, we blend the two values representing the x component of the circle's center with each other, and the two values for y, and similarly for radius and color) then we get an in-between circle, as in Figure 18-3.

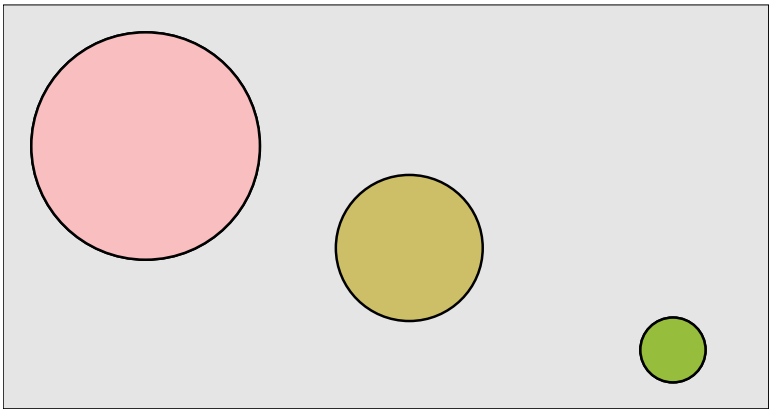


Figure 18-3: Parametric blending of the two circles means blending their parameters (center, radius, and color).

This works well for uncompressed objects. But if we try this with compressed objects, we rarely get reasonable in-between results. The problem is that the compressed form may have little in common with the internal structure that we need to meaningfully blend the objects. For example, let's take the sounds of the words *cherry* and *orange*. These sounds are our objects. We can blend these sounds together by having two people say the words at the same time, creating the audio version of our cow and zebra in Figure 18-1.

We can think of turning these sounds into written language as a form of compression. If it takes a half-second to say the word *cherry*, then if we use MP3 at a popular compression setting of 128 Kbps, we need about 8,000 bytes (AudioMountain 2020). If we use the Unicode UTF-32 standard (which requires 4 bytes per letter), the written form requires only 24 bytes, which is vastly smaller than 8,000. Since the letters are drawn from the alphabet, which has a given order, we can blend the representations by blending the letters through the alphabet. This isn't going to work for letters, but let's follow the process through because a version of this will work for us later.

The first letters of "cherry" and "orange" are C and O. In the alphabet, the region spanned by these letters is CDEFGHIJKLMNO. Right in the middle is the letter I, so that's the first letter of our blend. When the first letter appears later in the alphabet than the second, as in E to A, we count backward. When there's an even number of letters in the span, we chosen the earlier one. As shown in Figure 18-4 this blending gives us the sequence IMCPMO.

```
C ——— DEFGHIJKLMN ——— O
H ——— IJKLMNOPQ ——— R
E ——————— DCB ——————— A
R ——————— QPO ——————— N
R ——— QPONMLKJIH ——— G
YXWVUTSRQPONMLKJIHGFE
```

Figure 18-4: Blending the written words cherry *and* orange *by finding the midpoint of each letter in the alphabet*

What we wanted was something that, when uncompressed, sounded like a blend between the sound of *cherry* and the sound of *orange*. Saying the word *imcpmo* out loud definitely does not satisfy that goal. Beyond that, it's a meaningless string of letters that doesn't correspond to any fruit, or even any word in English.

In this case, blending the compressed representations doesn't give us anything like the blended objects. We will see that a remarkable feature of autoencoders, including the variational autoencoder we see at the end of the chapter, is that they do allow us to blend the compressed versions, and (to a point) recover blended versions of the original data.

The Simplest Autoencoder

We can build a deep learning system to figure out a compression scheme for any data we want. The key idea is to create a place in the network where the entire dataset has to be represented by fewer numbers than there are in the input. That, after all, is what compression is all about.

For instance, let's suppose that our input consists of grayscale images of animals, saved at a resolution of 100 by 100. Each image has $100 \times 100 = 10{,}000$ pixels, so our input layer has 10,000 numbers. Let's arbitrarily say we want to find the best way to represent those images using only 20 numbers.

One way to do this is to build a network as in Figure 18-5. It's just one layer!

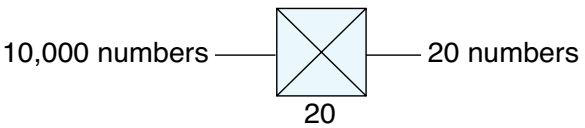


Figure 18-5: Our first encoder is a single dense, or fully connected, layer that turns 10,000 numbers into 20 numbers.

Our input is 10,000 elements, going into a fully connected layer of only 20 neurons. The output of those neurons for any given input is our compressed version of that image. In other words, with just one layer, we've built an encoder.

The real trick now would be to be able to recover the original 10,000 pixel values, or even anything close to them, starting from just these 20 numbers. To do that, we follow the encoder with a decoder, as in Figure 18-6. In this case, we just make a fully connected layer with 10,000 neurons, one for each output pixel.

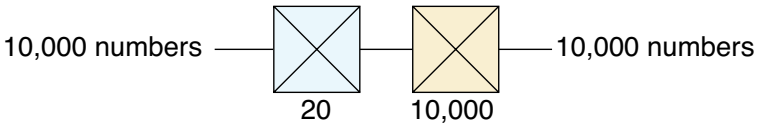


Figure 18-6: An encoder (in blue) turns our 10,000 inputs into 20 variables, then a decoder (in beige) turns those back into 10,000 values.

Because the amount of data is 10,000 elements at the start, 20 in the middle, and 10,000 again at the end, we say that we've created a *bottleneck*. Figure 18-7 shows the idea.

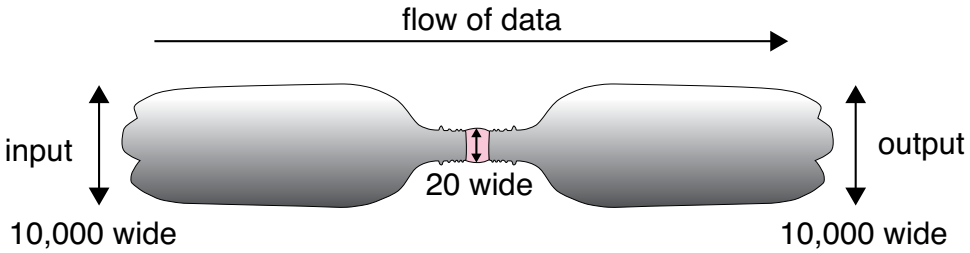


Figure 18-7: We say the middle of a network like the one shown in Figure 18-6 is a bottleneck because it's shaped like a bottle with a narrow top, or neck.

Now we can train our system. Each input image is also the output target. This tiny autoencoder tries to find the best way to crunch the input into just 20 numbers that can be uncrunched to match the target, which is the input itself. The compressed representation at the bottleneck is called the *code*, or the *latent variables* (*latent* suggests that these values are inherent in the input data, just waiting for us to discover them). Usually we make the bottleneck using a small layer in the middle of a deep network, as in Figure 18-6. Naturally enough, this layer is often called the *latent layer* or the *bottleneck layer*. The outputs of the neurons on this layer are the latent variables. The idea is that these values represent the image in some way.

This network has no category labels (as with a categorizer) or targets (as with a regression model). We don't have any other information for the system other than the input we want it to compress and then decompress. We say that an autoencoder is an example of *semi-supervised learning*. It sort-of is supervised learning because we give the system explicit goal data (the output should be the same as the input), and it sort-of isn't supervised learning because we don't have any manually determined labels or targets on the inputs.

Let's train our tiny autoencoder of Figure 18-6 on an image of a tiger and see how it does. We'll feed it the tiger image over and over, encouraging the system to output a full-size image of the tiger, despite the compression down to just 20 numbers at the bottleneck. The loss function compares the pixels of the original tiger with the pixels from the autoencoder's output and adds up the differences, so the more the pixels differ, the larger the loss. We trained it until it stopped improving. Figure 18-8 shows the result. Each error value shown on the far right is the original pixel value minus the corresponding output pixel value (the pixels were scaled to the range [0,1]).



Figure 18-8: Training our autoencoder of Figure 18-6 on a tiger. Left: The original, input tiger. Middle: The output. Right: The pixel-by-pixel differences between the original and output tiger (pixels are in the range [0,1]). The autoencoder seems to have done an amazing job since the bottleneck had only 20 numbers.

This is fantastic! Our system took a picture composed of 10,000 pixel values and crunched them down to 20 numbers, and now it appears to have recovered the entire picture again, right down to the thin, wispy whiskers. The biggest error in any pixel was about 1 part in 100. It looks like we've found a fantastic way to do compression!

But wait a second. This doesn't make sense. There's just no way to rebuild that tiger image from 20 numbers without doing something sneaky. In this case, the sneaky thing is that the network has utterly overfit and memorized the image. It simply set up all 10,000 output neurons to take those 20 input numbers and reconstruct the original 10,000 input values. Put more bluntly, the network merely memorized the tiger. We didn't really compress anything at all. Each of the 10,000 inputs went to each of the 20 neurons in the bottleneck layer, requiring $20 \times 10{,}000 = 200{,}000$ weights, and then the 20 bottleneck results all went to each of the 10,000 neurons in the output layer, requiring another 200,000 weights, which then produced the picture of the tiger. We basically found a way to store 10,000 numbers using only 400,000 numbers. Hooray?

In fact, most of those numbers are irrelevant. Remember that each neuron has a bias that's added alongside the incoming weighted inputs. The output neurons are relying on mostly their bias values and not too much on the inputs. To test this, Figure 18-9 shows the result of giving the autoencoder a picture of a flight of stairs. It doesn't do a poor job of compressing

and decompressing the stairs. Instead, it mostly ignores the stairs, and gives us back the memorized tiger. The output isn't exactly the input tiger, as shown by the rightmost image, but if we just look at the output, it's hard to see any hint of the stairs.



Figure 18-9: Left: We present our tiny autoencoder trained on just the tiger with an image of a stairway. Middle: The output is the tiger! Right: The difference between the output image and the original tiger.

The error bar on the right of Figure 18-9 shows that our errors are much larger than those of Figure 18-8, but the tiger still looks a lot like the original.

Let's make a real stress test of the idea that the network is mostly relying on the bias values. We can feed the autoencoder an input image that is zero everywhere. Then it has no input values to work with, and only the bias values contribute to the output. Figure 18-10 shows the result.



Figure 18-10: When we give our tiny autoencoder a field of pure black, it uses the bias values to give us back a low quality, but recognizable, version of the tiger. Left: The black input. Middle: The output. Right: The difference between the output and the original tiger. Note that the range of differences runs from 0 to almost 1, unlike Figure 18-9 where they ran from about −0.4 to 0.

No matter what input we give to this network, we will always get back some version of the tiger as output. The autoencoder has trained itself to produce the tiger every time.

A real test of this autoencoder would be to teach it a bunch of images and see how well it compresses them. Let's try again with a set of 25 photographs, shown in Figure 18-11.

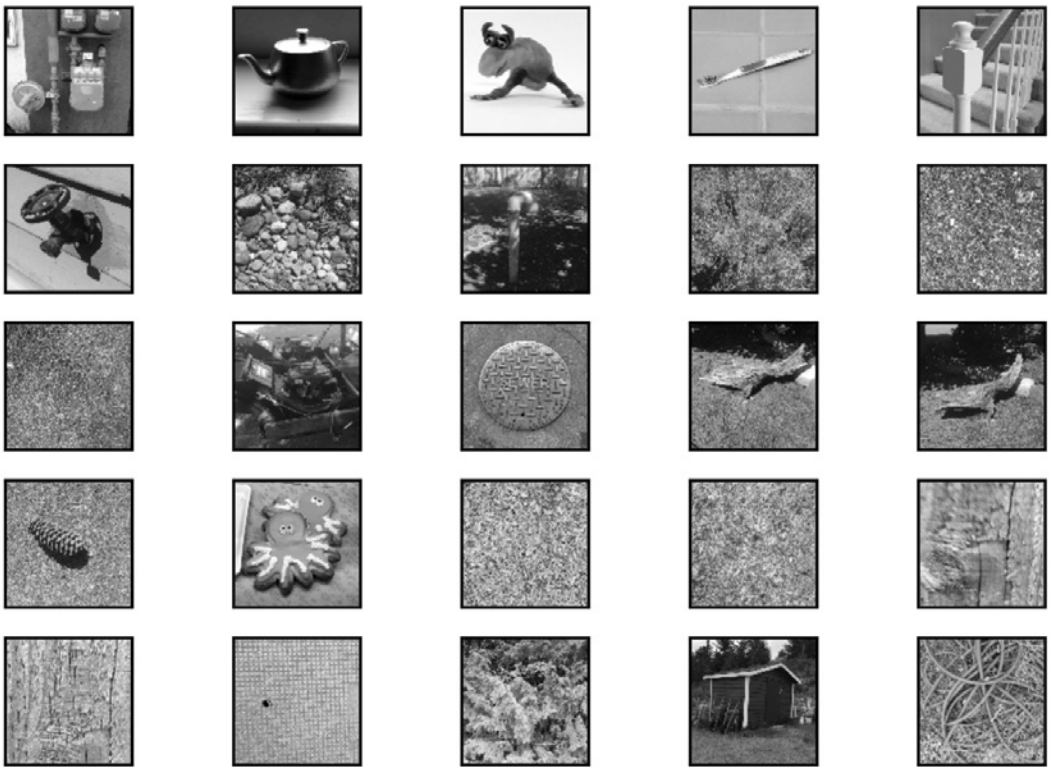


Figure 18-11: The 25 photographs that we used, in addition to the tiger, to train our tiny autoencoder. Each image was rotated by 90 degrees, 180 degrees, and 270 degrees during training.

We made the database larger by training not just on each image, but also on each image rotated by 90 degrees, 180 degrees, and 270 degrees. Our training set was the tiger (and its three rotations) and the 100 images of Figure 18-11 with rotations, for a total of 104 images.

Now that the system is trying to remember how to represent all 104 of these pictures with just 20 numbers, it should be no surprise that it can't do a very good job. Figure 18-12 shows what this autoencoder produces when we ask it to compress and decompress the tiger.

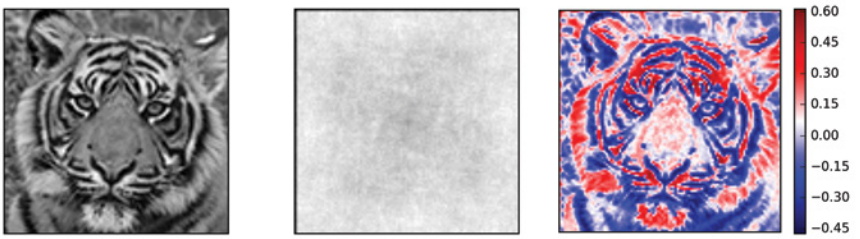


Figure 18-12: We trained our autoencoder of Figure 18-6 with the 100 images of Figure 18-11 (each image plus its rotated versions), along with the four rotations of the tiger. Using this training, we gave it the tiger on the left, and it produced the output in the middle.

Now that the system isn't allowed to cheat, the result doesn't look like a tiger at all, and everything makes sense again. We can see a little bit of four-way rotational symmetry in the result, owing to our training on the rotated versions of the input images. We could do better by increasing the number of neurons in the bottleneck, or latent, layer. But since we want to compress our inputs as much as possible, adding more values to the bottleneck should be a last resort. We'd rather do the best possible job we can with as few values as we can get away with.

Let's try to improve the performance by considering a more complex architecture than just the two dense layers we've been using so far.

A Better Autoencoder

In this section, we'll explore a variety of autoencoder architectures. To compare them, we'll use the MNIST database we saw in Chapter 17. To recap, this is a big, free database of hand-drawn, grayscale digits from 0 to 9, saved at a resolution of 28 by 28 pixels. Figure 18-13 shows some typical digit images from the MNIST dataset.

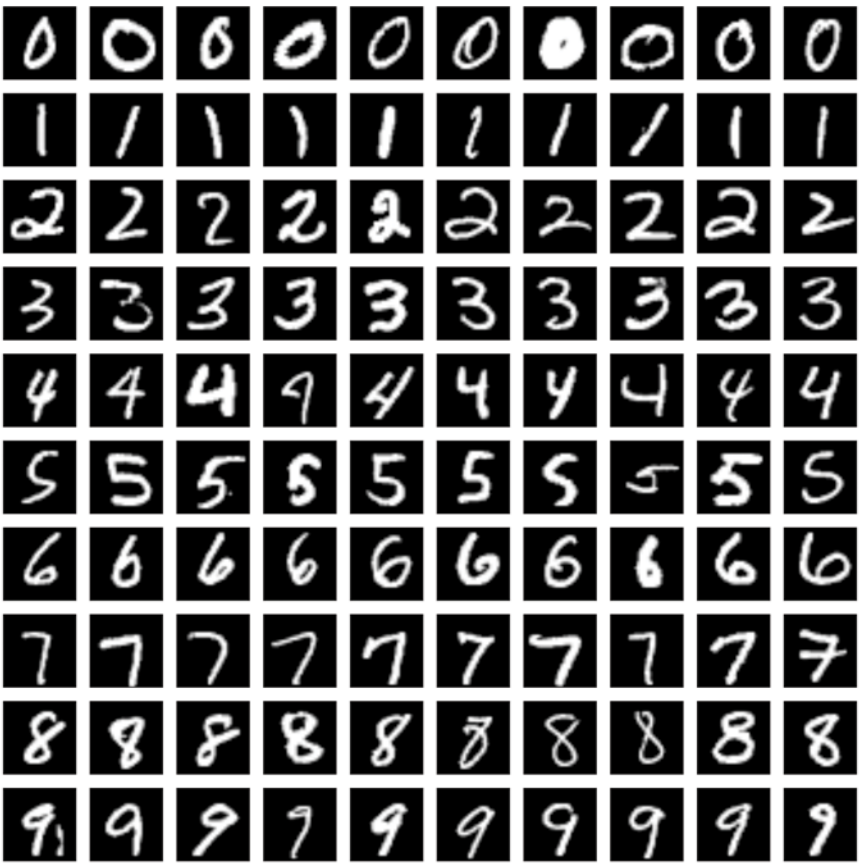


Figure 18-13: A sampling of the handwritten digits from the MNIST dataset

To run our simple autoencoder on this data, we need to change the size of the inputs and outputs of Figure 18-6 to fit the MNIST data. Each image has 28 × 28 = 784 pixels. Thus, our input and output layer now needs 784 elements instead of 10,000. Let's flatten the 2D image into a single big list before we feed it to the network and leave the bottleneck at 20. Figure 18-14 shows the new autoencoder. In this diagram, as well as those to come, we won't draw the flatten layer at the start, or the reshaping layer at the end that "undoes" the flattening and turns the list of 784 numbers back into a 28 by 28 grid.

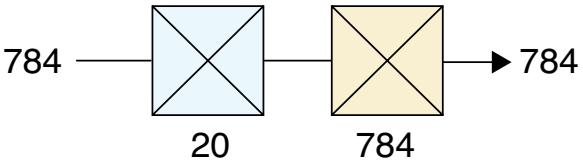


Figure 18-14: Our two-layer autoencoder for MNIST data

Let's train this for 50 epochs (that is, we run through all 60,000 training examples 50 times). Some results are shown in Figure 18-15.



Figure 18-15: Running five digits from the MNIST dataset through our trained autoencoder of Figure 18-14, which uses 20 latent variables. Top row: Five pieces of input data. Bottom row: The reconstructed images.

Figure 18-15 is pretty amazing. Our two-layer network learned how to take each input of 784 pixels, squash it down to just 20 numbers, and then blow it back up to 784 pixels. The resulting digits are blurry, but recognizable.

Let's try reducing the number of latent variables down to 10. We expect things are going to look a lot worse. Figure 18-16 shows that they are indeed worse.



Figure 18-16: Top row: The original MNIST images. Bottom row: The output of our autoencoder using 10 latent variables.

This is getting pretty bad. The 2 seems to be turning into a 3 with a bite taken out of it, and the 4 seems to be turning into a 9. But that's what we get

for crushing these images down to 10 numbers. That's just not enough to enable the system to do a good job of representing the input.

The lesson is that our autoencoder needs to have both enough computational power (that is, enough neurons and weights) to figure out how to encode the data, and enough latent variables to find a useful compressed representation of the input.

Let's see how deeper models perform. We can build the encoder and decoder with any types of layers we like. We can make deep autoencoders with lots of layers, or shallow ones with only a few, depending on our data. For now, let's continue using fully connected layers, but let's add some more of them to create a deeper autoencoder. We'll construct the encoder stage from several hidden layers of decreasing size until we reach the bottleneck, and then we'll build a decoder from several more hidden layers of increasing size until they reach the same size as the input.

Figure 18-17 shows this approach where now we have three layers of encoding and three of decoding.

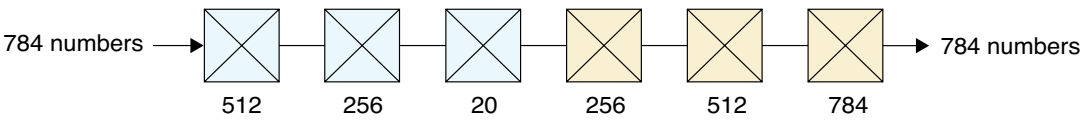


Figure 18-17: A deep autoencoder built out of fully connected (or dense) layers. Blue icons: A three-layer encoder. Beige icons: A three-layer decoder.

We often build these fully connected layers so that their numbers of neurons decrease (and then increase) by a multiple of two, as when we go between 512 and 256. That choice often works out well, but there's no rule enforcing it.

Let's train this autoencoder just like the others, for 50 epochs. Figure 18-18 shows the results.

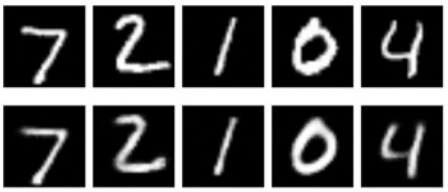


Figure 18-18: Predictions from our deep autoencoder of Figure 18-17. Top row: Images from the MNIST test set. Bottom row: Output from our trained autoencoder when presented with the test digits.

The results are just a little blurry, but they match the originals unambiguously. Compare these results to Figure 18-15, which also used 20 latent variables. These images are much clearer. By providing additional compute power to find those variables (in the encoder), and extra power in turning them back into images (in the decoder), we've gotten much better results out of our 20 latent variables.

Exploring the Autoencoder

Let's look more closely at the results produced by the autoencoder network in Figure 18-17.

A Closer Look at the Latent Variables

We've seen that the latent variables are a compressed form of the inputs, but we haven't looked at the latent variables themselves. Figure 18-19 shows graphs of the 20 latent variables produced by the network in Figure 18-17 in response to our five test images, and the images that the decoder constructs from them.

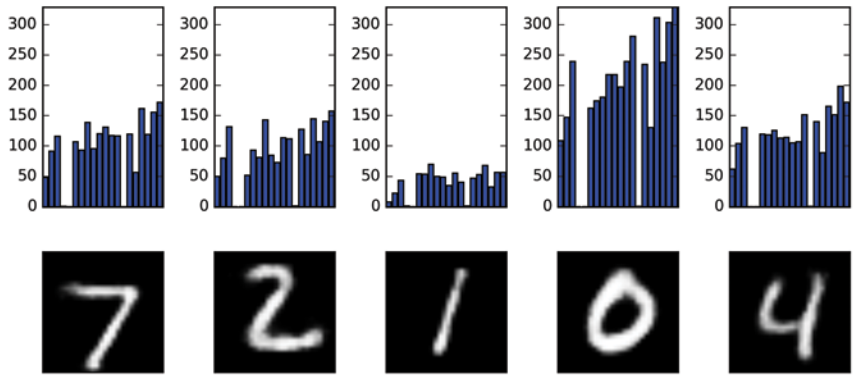


Figure 18-19: Top row: The 20 latent variables for each of our five images produced by the network in Figure 18-17. Bottom row: The images decompressed from the latent variables above them.

The latent variables shown in Figure 18-19 are typical, in the sense that latent variables rarely show any obvious connection to the input data from which they were produced. The network has found its own private, highly compressed form for representing its inputs, and that form often makes no sense to us. For example, we can see a couple of consistent holes in the graphs (in positions 4, 5, and 14), but there's no obvious reason from this one set of images why those values are 0 (or nearly 0) for these inputs. Looking at more data would surely help, but the problem of interpretation remains, in general.

The mysterious nature of the latent variables is fine, because we rarely care about directly interpreting these values. Later on, we'll play with the latent values, by blending and averaging them, but we won't care about what these numbers represent. They're just a private code that the network created during training that lets it compress and then decompress each input as well as possible.

The Parameter Space

Though we usually don't care about the numerical values in the latent variables, it is still useful to get a feeling for what latent variables are produced by similar and different inputs. For example, if we feed the system

two images of a seven that are almost the same, will the images be assigned almost the same latent variables? Or might they be wildly far apart?

To answer these questions, let's continue with the simple deep autoencoder of Figure 18-17. But instead of making the last stage of the encoder a fully connected layer of 20 neurons, let's drop that to merely two neurons, so we have just two latent variables. The point of this is that we can plot the two variables on the page as (x,y) pairs. Of course, if we generate images from just two latent variables, those images will come out extremely blurry, but it's worth the exercise so we can see the structure of these simple latent variables.

In Figure 18-20 we encoded 10,000 MNIST images, found each image's two latent variables, and then plotted them as a point. Each dot is color-coded for the label assigned to the image it came from. We say that an image like Figure 18-20 is a visualization of *latent variable space*, or more simply, *latent space*.

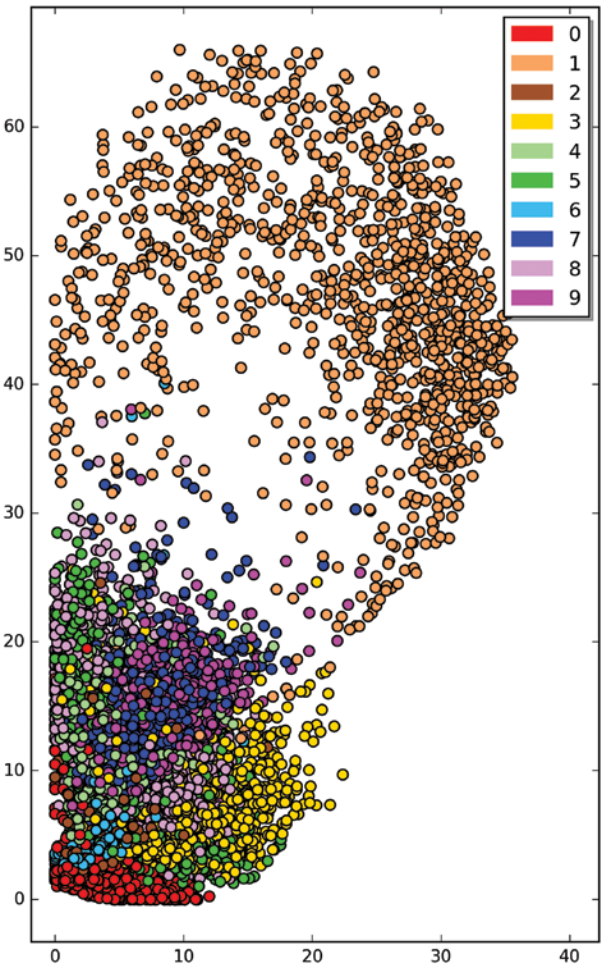


Figure 18-20: After training a deep autoencoder with only two latent variables, we show the latent variables assigned to each of 10,000 MNIST images.

There's a lot of structure here! The latent variables aren't being assigned numerical values totally at random. Instead, similar images are getting assigned similar latent variables. The 1's, 3s, and 0s seem to fall into their own zones. Many of the other digits seem to be scrambled in the lower-left of the plot, getting assigned similar values. Figure 18-21 shows a close-up view of that region.

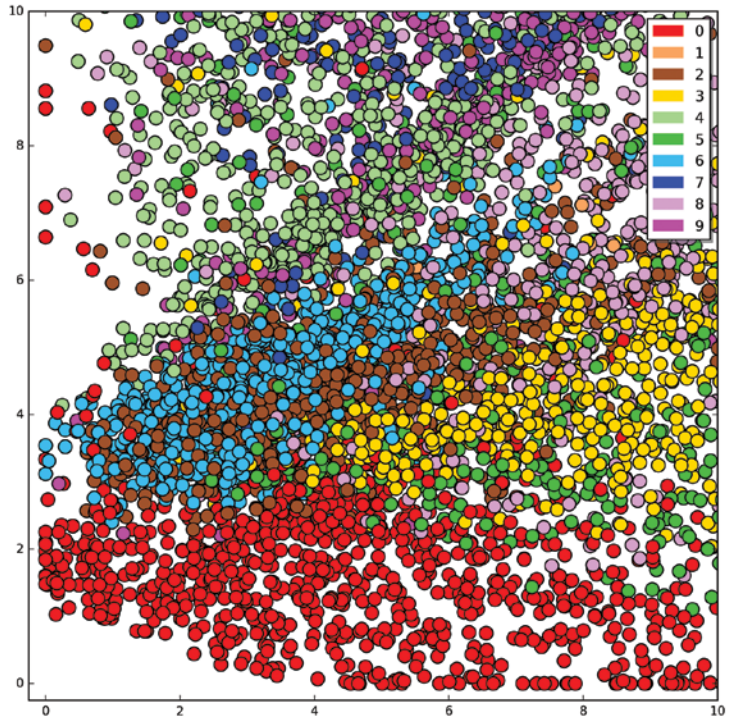


Figure 18-21: A close-up of the lower-left corner of Figure 18-20

It's not a total jumble. The 0's have their own band, and while the others are a bit mixed up, we can see they all seem to fall into well-defined zones.

Though we expect the images to be blurry, let's make pictures from these 2D latent values. We can see from Figure 18-20 that the first latent variable (which we're drawing on the X axis) takes on values from 0 to about 40, and the second latent variable (which we're drawing on the Y axis) takes on values from 0 to almost 70.

Let's make a square grid of decoded images, following the recipe in Figure 18-22. We can make a box that runs from 0 to 55 along each axis (that's a little too short in Y, but a little too long in X). We can pick (x,y) points inside this grid, and then feed those two numbers to the decoder, producing a picture. Then we can draw the picture at that (x,y) position in a corresponding grid. We found that 23 steps on each axis produced a nice image that's dense, but not overly so.

Figure 18-23 shows the result.

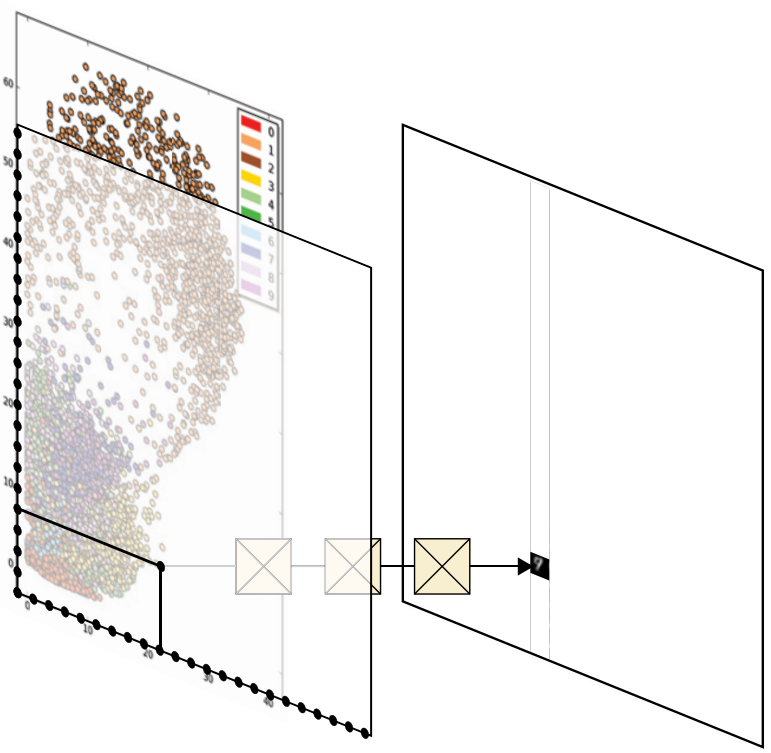


Figure 18-22: Making a grid of images by decoding (x,y) pairs from Figure 18-20 (and a little beyond). On the left, we select an (x,y) pair located at about (22,8). Then we pass these two numbers through the decoder, creating the tiny output image on the right.

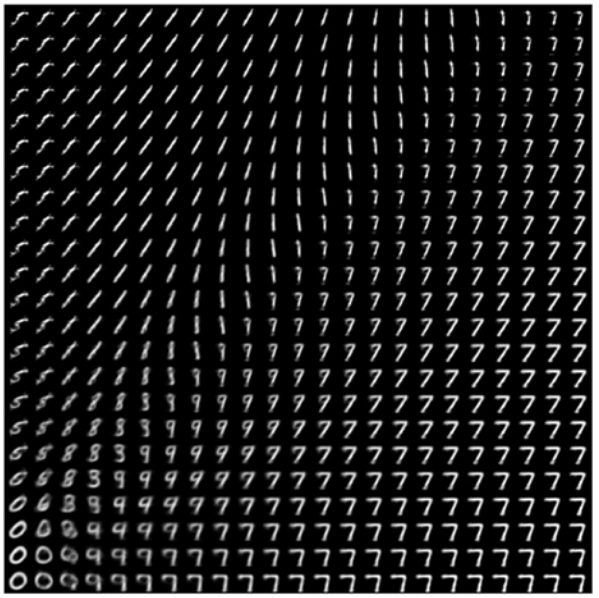


Figure 18-23: Images generated from latent variables in the range of Figure 18-22

The 1's spray along the top, as expected. Surprisingly, the 7's dominate the right side. As before, let's look at the images in a close-up of the lower-left corner, shown in Figure 18-24.

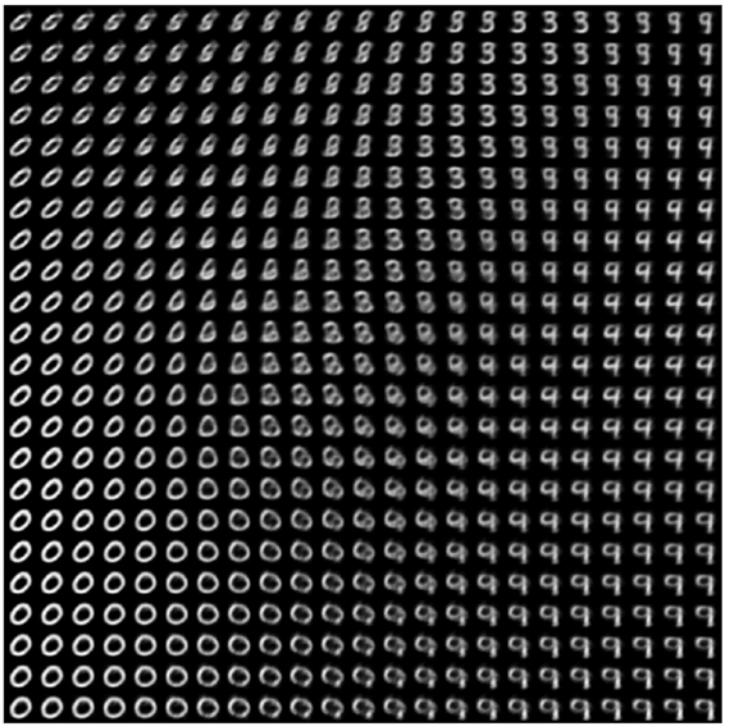


Figure 18-24: Images from the close-up range of latent variables in Figure 18-23

The digits are frequently fuzzy, and they don't fall into clear zones. This is not just because we're using a very simple encoder, but because we're encoding our inputs into just two latent variables. With more latent variables, things become more separated and distinct, but we can't draw simple pictures of those high-dimensional spaces. Nevertheless, this shows us that even with an extreme compression down to just two latent variables, the system assigned those values in ways that grouped similar digits together.

Let's look more closely at this space. Figure 18-25 shows the images produced by taking (x,y) values along four lines through the plot and feeding those to the decoder to produce images.

This confirms that the encoder assigned similar latent variables to similar images and seemed to build clusters of different images, with each variation of the image in its own region. That's a whole lot of structure. As we increase our number of latent variables from this ridiculously small value of two, the encoder continues to produce clustered regions, but they become more distinct and there is less overlapping.

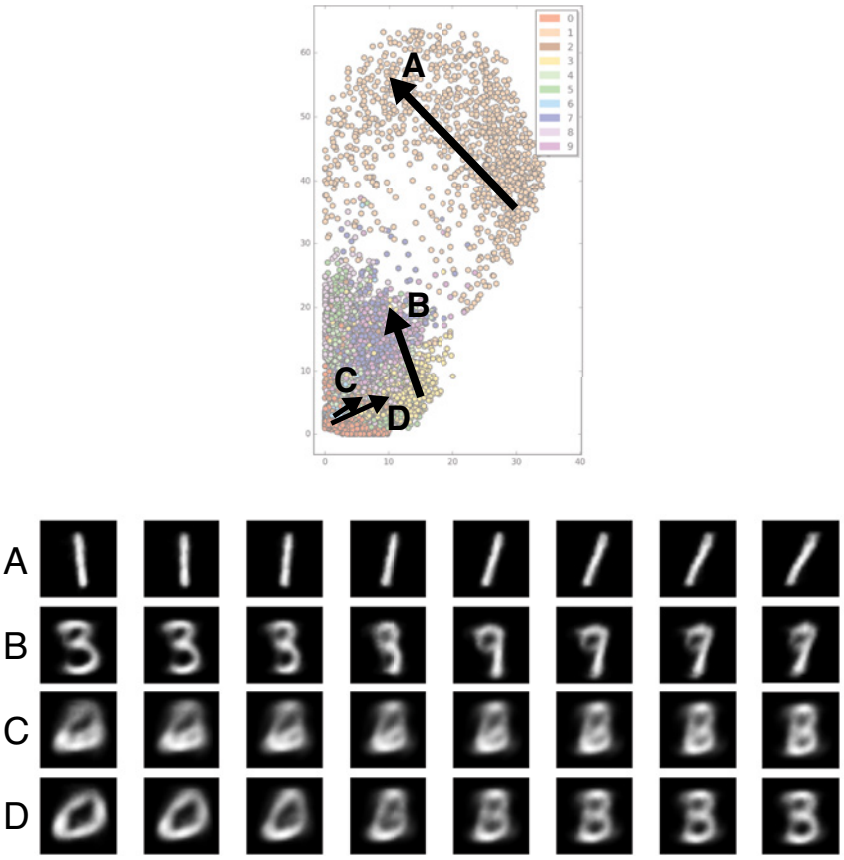


Figure 18-25: For each arrow, we took eight equally spaced steps from the start to the end, producing eight (x,y) pairs. The decoded images for these pairs are shown in the corresponding row.

Blending Latent Variables

Now that we've seen the structure inherent in the latent variables, we can put it to use. In particular, let's blend some pairs of latent variables together and see if we get an intermediate image. In other words, let's do parametric blending on the images, as we discussed earlier, where the latent variables are the parameters.

We actually did this in Figure 18-25 as we blended the two latent variables from one end of an arrow to the other. But there, we were using an autoencoder with just two latent variables, so it wasn't able to represent the images very well. The results were mostly blurry. Let's use some more latent variables so we can get a feeling for what this kind of blending, or *interpolation*, looks like in more complex models.

Let's return to the six-layer version of our deep autoencoder of Figure 18-17, which has 20 latent variables. We can pick out pairs of images, find the latent variables for each one, and then simply average each pair of latent variables. That is, we have a list of 20 numbers for the first image (its

latent variables) and a list of 20 numbers for the second image. We blend the first number in each list together, then the second number in each list, and so on, until we have a new list of 20 numbers. This is the new set of latent variables that we hand to the decoder, which produces an image.

Figure 18-26 shows five pairs of images blended this way.

As we expect, the system isn't simply blending the images with content blending (like we did for the cow and zebra in Figure 18-1). Instead, the autoencoder is producing intermediate images that have qualities of both inputs.



Figure 18-26: Examples of blending latent variables in our deep autoencoder. Top row: Five images from the MNIST dataset. Middle row: Five other images. Bottom row: The image resulting from averaging the latent variables of the two images directly above, and then decoding.

These results aren't absurd. For example, in the second column, the blend between a 2 and 4 looks like a partial 8. That makes sense. Figure 18-23 shows us that the 2s, 4s, and 8s are close together in the diagram with only 2 latent variables, so it's reasonable that they could still be near one another in a 20-dimensional diagram with 20 latent variables.

Let's look at this kind of blending of latent variables more closely. Figure 18-27 shows three new pairs of digits with six equally spaced steps of interpolation.



Figure 18-27: Blending the latent variables. For each row, we blend between the leftmost and rightmost sets of latent variables.

The far left and right of each row are images from the MNIST data. We found the 20 latent variables for each endpoint, created six equally spaced blends of those latent variables, and then ran those blended latents through the decoder. The system is trying to move from one image to another, but it's not producing very reasonable intermediate digits. Even when going from a 5 to a 5 in the middle row, the intermediate values almost break up into two separate pieces before rejoining. Some of the blends near the middle of the top and bottom rows don't look like any digits at all. Although the ends are recognizable, the blends fall apart very quickly. Blending latent parameters in this autoencoder smoothly changes the image from one digit to another, but the in-betweens are just weird shapes, rather than some kind of blended digits. We've seen that sometimes this is due to moving through dense regions where similar latent variables encode different digits. A bigger problem is conceptual. These examples may not even be wrong, since it's not clear what a digit that's partly 0 and partly 1 *should* look like, were we able to make one. Maybe the 0 should get thinner? Maybe the 1 should curl up into a circle? So although these blends don't look like digits, they're reasonable results.

Some of these interpolated latent values can land in regions of latent space where there's no nearby data. In other words, we're asking the decoder to reconstruct an image from values of latent variables that don't have any nearby neighbors in latent space. The decoder is producing *something*, and that output has some qualities of the nearby regions, but the decoder is essentially guessing.

Predicting from Novel Input

Let's try to use this deep autoencoder trained on MNIST data to compress and then decompress our tiger image. We will shrink the tiger to 28 by 28 pixels to match the network's input size, so it's going to look very blurry.

The tiger is like nothing the network has ever seen before, so it's completely ill-equipped to deal with this data. It tries to "see" a digit in the image and produces a corresponding output. Figure 18-28 shows the results.



Figure 18-28: Encoding and then decoding a 28 by 28 version of our tiger of Figure 18-8 with our deep autoencoder of 20 latent variables, trained on the MNIST handwritten digit dataset

It looks like the algorithm has tried to find a spot that combines several different digits. The splotch in the middle isn't much of a match to the tiger, but there's no reason it should be.

Using information learned from digits to compress and decompress a tiger is like trying to build a guitar using parts taken from pencil sharpeners. Even if we do our best, the result isn't likely to be a good guitar. An autoencoder can only meaningfully encode and decode the type of data it's been trained on because it created meaning for the latent variables only to represent that data. When we surprise it with something completely different, it does its best, but it's not going to be very good.

There are several variations on the basic autoencoder concept. Since we're working with images, and convolution is a natural approach for that kind of data, let's build an autoencoder using convolution layers.

Convolutional Autoencoders

We said earlier that our encoding and decoding stages could contain any kind of layers we wanted. Since our running example uses image data, let's use convolutional layers. In other words, let's build a *convolutional autoencoder*.

We will design an encoder to use several layers of convolution to scale down the original 28 by 28 MNIST image in stages until it's just 7 by 7. All of our convolutions will use 3 by 3 filters, and zero-padding. As shown in Figure 18-29, we start with a convolution layer with 16 filters and follow it by a maximum pooling layer with a 2 by 2 cell, giving us a tensor that is 14 by 14 by 16 (we could have used striding during convolution, but we've separated the steps here for clarity). Then we apply another convolution, this time with 8 filters, and follow that with pooling, producing a tensor that's 7 by 7 by 8. The final encoder layer uses three filters, producing a tensor that's 7 by 7 by 3 at the bottleneck. Thus, our bottleneck represents the 768 inputs with $7 \times 7 \times 3 = 147$ latent variables.

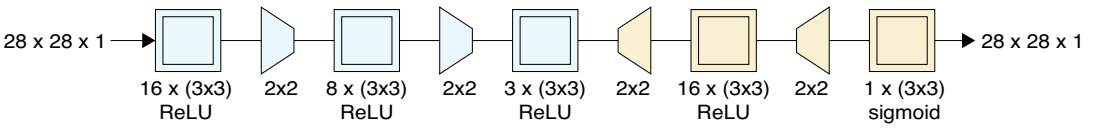


Figure 18-29: The architecture of our convolutional autoencoder. In the encoding stage (blue), we have three convolutional layers. The first two layers are each followed by a pooling layer, so by the end of the third convolutional layer, we have an intermediate tensor of shape 7 by 7 by 3. The decoder (beige) uses convolution and upsampling to grow the bottleneck tensor back into a 28 by 28 by 1 output.

Our decoder runs the process in reverse. The first upsampling layer produces a tensor that's 14 by 14 by 3. The following convolution and upsampling gives us a tensor that's 28 by 28 by 16, and the final convolution produces a tensor of shape 28 by 28 by 1. As before, we're leaving out the flattening step at the start and the reshaping step at the end.

Since we've got 147 latent variables, along with the power of the convolutional layers, we should expect better results than with our previous

autoencoder of just 20 latent variables. We trained this network for 50 epochs, just as before. The model was still improving at that point, but we stopped at 50 epochs for the sake of comparison with the previous models.

Figure 18-30 shows five examples from the test set and their decompressed versions after running through our convolutional autoencoder.

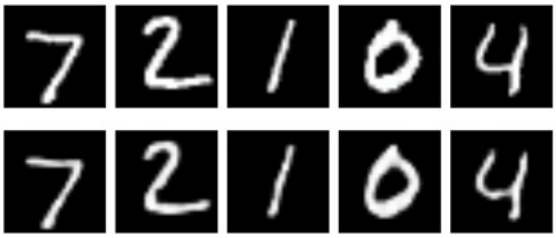


Figure 18-30: Top row: Five elements from the MNIST test set. Bottom row: The images produced by our convolutional autoencoder given the image above it as input.

These results are pretty great. The images aren't identical, but they're very close.

Just for fun, let's try giving the decoder step nothing but noise. Since our latent variables are a tensor of size 7 by 7 by 3, our noise values need to be a 3D volume of the same shape. Rather than try to draw such a block of numbers, we will just show the topmost 7 by 7 slice of the block. Figure 18-31 shows the results.

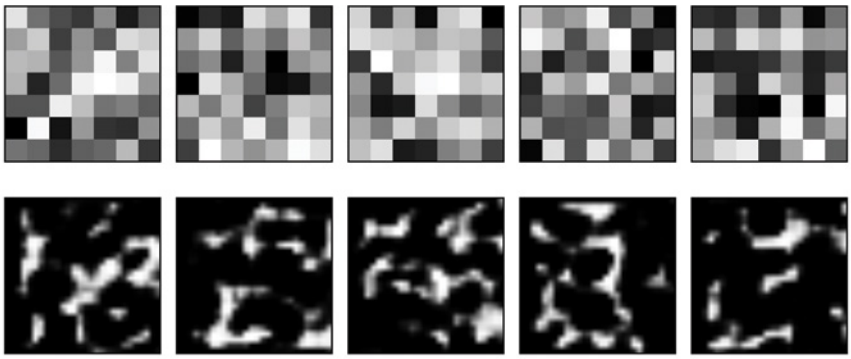


Figure 18-31: Images produced by handing an input tensor of random values to the decoder stage of our convolutional neural network

This just produces randomly splotchy images, which seems a fair output for a random input.

Blending Latent Variables

Let's blend the latent variables in our convolutional autoencoder and see how it goes. In Figure 18-32 we show our grid using the same images as in

Figure 18-26. We find the latent variables for each image in the top two rows, blend them equally, and then decode the interpolated variables to create the bottom row.

The results are pretty gloppy, though some have a feeling of being a mix of the images from the rows above. Again, we shouldn't be too surprised, since it's not clear what a digit halfway between, say, 7 and 3 ought to look like.



Figure 18-32: Blending latent variables in the convolutional autoencoder. Top two rows: Samples from the MNIST dataset. Bottom row: The result of an equal blend of the latent variables from each of the above images.

Let's look at multiple steps along the way in the same three blends that we used before in Figure 18-27. The results are shown in Figure 18-33.



Figure 18-33: Blending the latent variables of two MNIST test images and then decoding

The left and right ends of each row are images created by encoding and decoding an MNIST image. In between are the results of blending their latent variables and then decoding. This isn't looking a whole lot better than our simpler autoencoder. So just because we have more latent variables, we still run into trouble when we try to reconstruct using inputs that are too unlike the samples that the system was trained on. For example, in the top row we didn't train on any input images that were in some way "between" a 4 and 3, so the system didn't have any good information on how to produce images from latent values representing such a thing.

Predicting from Novel Input

Let's repeat our completely unfair test by giving the low-resolution tiger to our convolutional neural net. The results are shown in Figure 18-34.

If we squint, it looks like the major dark regions around the eyes, the sides of the mouth, and the nose, have been preserved. Maybe. Or maybe that's just imagination.

As with our earlier autoencoder built from fully connected layers, our convolutional autoencoder is trying to find a tiger somewhere in the latent space of digits. We shouldn't expect it to do well.



Figure 18-34: The low-resolution tiger we applied to our convolutional autoencoder, and the result. It's not very tiger-like.

Denoising

A popular use of autoencoders is to remove noise from samples. A particularly interesting application is to remove the speckling that sometimes appears in computer-generated images (Bako et al. 2017; Chaitanya 2017). These bright and dark points, which can look like static, or snow, can be produced when we generate an image quickly, without refining all the results.

Let's see how to use an autoencoder to remove bright and dark dots in an image. We will use the MNIST dataset again, but this time, we'll add some random noise to our images. At every pixel, we pick a value from a Gaussian distribution with a mean of 0, so we get positive and negative values, add them in, and then clip the resulting values to the range 0 to 1. Figure 18-35 shows some MNIST training images with this random noise applied.

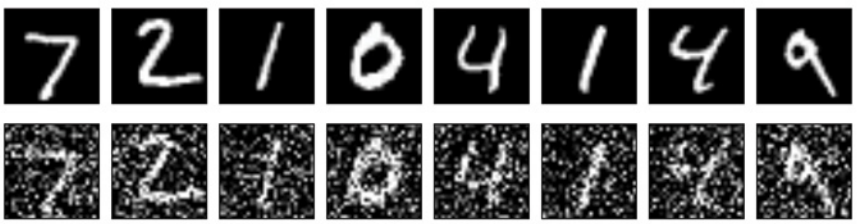


Figure 18-35: Top: MNIST training digits. Bottom: The same digits but with random noise.

Our goal is to give our trained autoencoder the noisy versions of the digits in the bottom row of Figure 18-35 and have it return cleaned-up versions like the top row of in Figure 18-35. Our hope is that the latent variables won't encode the noise, so we'll get back just the digits.

We'll use an autoencoder with the same general structure as Figure 18-29, though with different numbers of filters (Chollet 2017). Figure 18-36 shows the architecture.

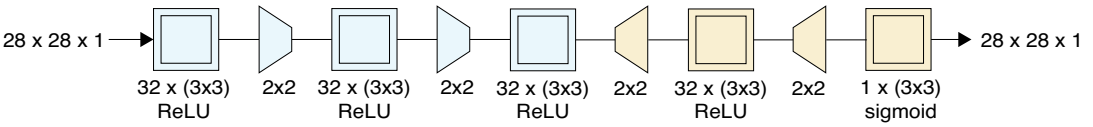


Figure 18-36: A denoising autoencoder

To train our autoencoder, we'll give it noisy image inputs and their corresponding clean, noise-free versions as the targets we want it to produce. We'll train with all 60,000 images for 100 epochs.

The tensor at the end of the decoding step in Figure 18-35 (that is, after the third convolution) has size 7 by 7 by 32, for a total of 1,568 numbers. So our "bottleneck" in this model is twice the size of the input. That would be bad if our goal was compression, but here we're trying to remove noise, so minimizing the number of latent variables isn't as much of a concern.

How well does it perform? Figure 18-37 shows some of the noisy inputs and the autoencoder's outputs. It cleaned up the pixels very well, giving us great-looking results.



Figure 18-37: Top row: Digits with noise added. Bottom row: The same digits denoised by our model of Figure 18-36.

In Chapter 16, we discussed that explicit upsampling and downsampling layers are falling out of favor, replaced by striding and transposed convolution. Let's follow that trend to simplify our model of Figure 18-36 to make Figure 18-38, which is now made up of nothing but a sequence of five convolution layers. The first two convolutions use striding to replace explicit downsampling layers, and the last two layers use repetition instead of explicit upsampling layers. Recall that we're assuming zero-padding in each convolution layer.

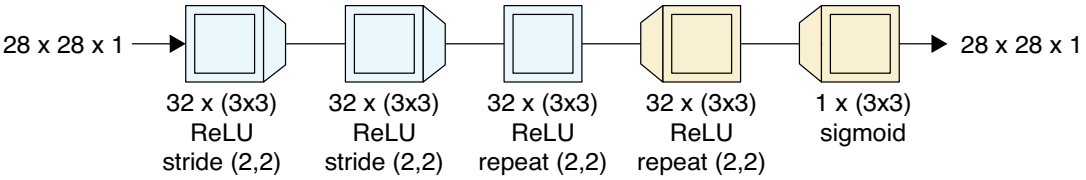


Figure 18-38: The autoencoder of Figure 18-36 but using downsampling and upsampling inside the convolution layers, as shown by the wedges attached to the convolution icons

Figure 18-39 shows the results.



Figure 18-39: The results of our denoising model of Figure 18-38

The outputs are quite close, though there are small differences (for example, look at the bottom-left of the 0). The first model, Figure 18-36, with explicit layers for upsampling and downsampling, took roughly 300 seconds per epoch on a late 2014 iMac with no GPU support. The simpler model of Figure 18-38 took only about 200 seconds per epoch so it shaved off about one-third of the training time.

It would require a more careful problem statement, testing, and review of the results to decide if either of these models produces better results than the other for this task.

Variational Autoencoders

The autoencoders we've seen so far have tried to find the most efficient way to compress an input so that it can later be re-created. A *variational autoencoder (VAE)* shares the same general architecture as those networks but does an even better job of clumping the latent variables and filling up the latent space.

VAEs also differ from our previous autoencoders because they have some unpredictability. Our previous autoencoders were deterministic. That is, given the same input, they always produce the same latent variables, and those latent variables always then produce the same output. But a VAE uses probabilistic ideas (that is, random numbers) in the encoding stage; if we run the same input through the system multiple times, we get a slightly different output each time. We say that a VAE is *nondeterministic*.

As we look at the VAE, let's continue to phrase our discussion in terms of images (and pixels) for concreteness, but like all of our other machine learning algorithms, a VAE can be applied to any kind of data: sound, weather, movie preferences, or anything else we can represent numerically.

Distribution of Latent Variables

In our previous autoencoders we didn't impose any conditions on the structure of the latent variables. In Figure 18-20, we saw that a fully connected encoder seemed to naturally group the latent variables into blobs radiating to the right and upward from a common starting point at (0,0). That structure wasn't a design goal. It just came out that way as a result of the nature of the network we built. The convolutional network in Figure 18-38 produces similar results when we reduce the bottleneck to two latent variables, shown here in Figure 18-40.

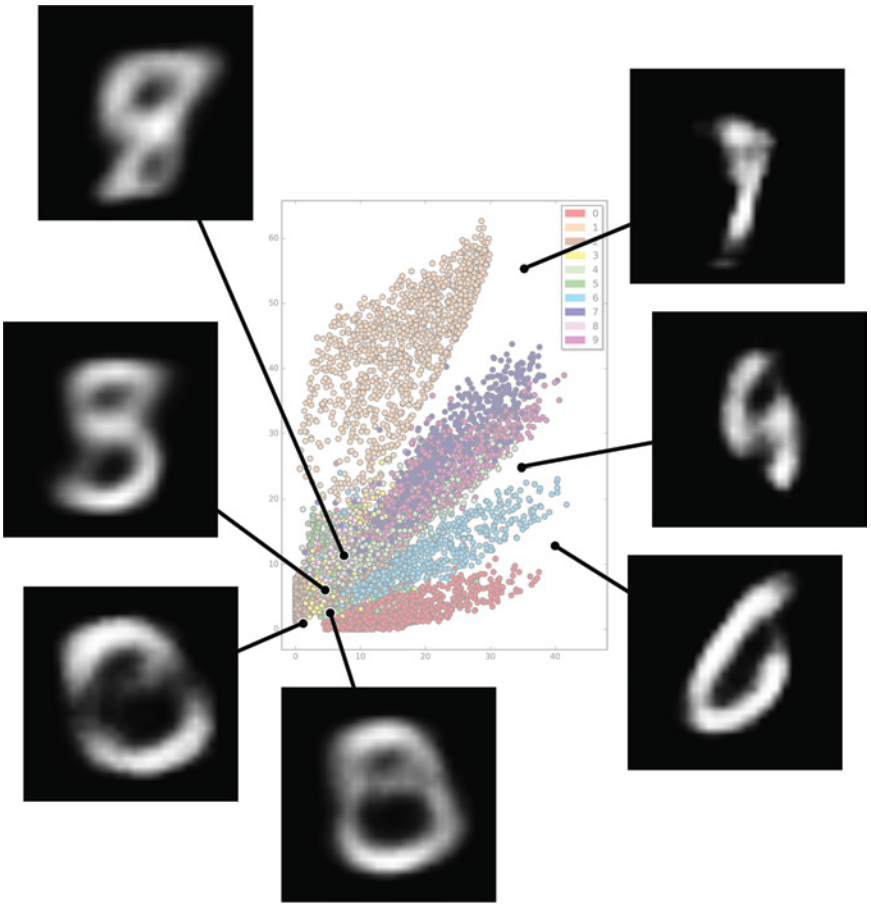


Figure 18-40: The latent variables produced by the convolutional autoencoder of Figure 18-38 with a bottleneck of two latent variables. These samples are drawn from both densely mixed and sparse regions.

Figure 18-40 shows decoded images generated from latent variables chosen from both dense and sparse regions.

In Figure 18-22 we saw that we can pick any pair of latent variables and run those values through a decoder to make an image. Figure 18-40 shows that if we pick these points in the dense zones, or the unoccupied zones, we often get back images that don't look like digits. It would be great if we could find a way to set things up so that any pair of inputs always (or almost always) produces a good-looking digit.

Ideally, it would be great if each digit had its own zone, the zones didn't overlap, and we didn't have any big, empty spaces. There's not much we can do about filling in empty zones, since those are places where we just don't have input data. But we can try to break apart the mixed zones so that each digit occupies its own region of the latent space.

Let's see how a variational autoencoder does a good job of meeting these goals.

Variational Autoencoder Structure

As so often happens with good ideas, the VAE was invented simultaneously but independently by at least two different groups (Kingma and Welling 2014; Rezende, Mohamed, and Wierstra 2014). Understanding the technique in detail requires working through some math (Dürr 2016), so instead, let's take an approximate and conceptual approach. Because our intent is to capture the gist of the method rather than its precise mechanics, we will skip some details and gloss over others.

Our goal is to create a generator that can take in random latent variables and produce new outputs that are reasonably like inputs that had similar latent values. Recall that the distribution of the latent variables is created together by the encoder and decoder during training. During that process, in addition to making sure the latent variables let us reconstruct the inputs, we also desire that the latent variables obey three properties.

First, all of the latent variables should be gathered into one region of latent space so we know what the ranges should be for our random values. Second, latent variables produced by similar inputs (that is, images that show the same digit) should be clumped together. Third, we want to minimize empty regions in the latent space.

To satisfy these criteria, we can use a more complicated error term that punishes the system when it makes latent samples that don't follow the rules. Since the whole point of learning is to minimize the error, the system will learn how to create latent values that are structured the way we want. The architecture and the error term are designed to work together. Let's see what that error term looks like.

Clustering the Latent Variables

Let's first tackle the idea of keeping all latent variables together in one place. We can do that by imposing a rule, or constraint, which we build into the error term.

Our constraint is that the values for each latent variable, when plotted, come close to forming a unit Gaussian distribution. Recall from Chapter 2 that a Gaussian is the famous bell curve, illustrated in Figure 18-41.

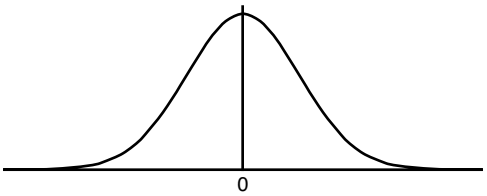


Figure 18-41: A Gaussian curve

When we place two Gaussians at right angles to one another, we get a bump above the plane, as in Figure 18-42.

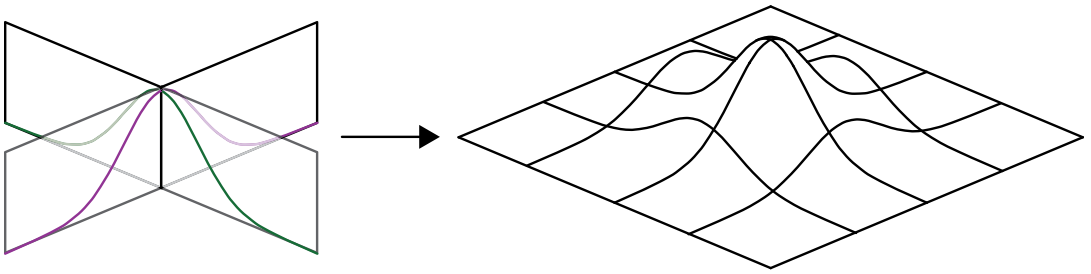


Figure 18-42: In 3D, we can place two Gaussians at right angles. Together, they form a bump over the plane.

Figure 18-42 shows a 3D visualization of a 2D distribution. We can create an actual 3D distribution by including another Gaussian on the Z axis. If we think of the resulting bump as a density, then this 3D Gaussian is like a dandelion puff, which is dense in the center but becomes sparser as we move outward.

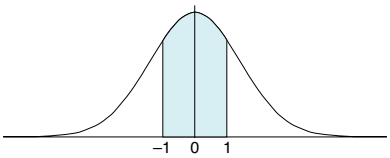


Figure 18-43: A Gaussian is described by its mean (the location of its center), and its standard deviation (the symmetrical distance that contains about 68 percent of its area). Here we have a center of 0, and a standard deviation of 1.

By analogy, we can imagine a Gaussian of any number of dimensions, just by saying that each dimension's density follows a Gaussian curve on its axis. And that's what we do here. We tell the VAE to learn values for the latent variables so that, when we look at the latent variables for lots of training samples and we count up how many times each value occurs, every variable's counts form a distribution like a Gaussian that has its mean

(or center) at 0, and a standard deviation (that is, its spread) of 1, as in Figure 18-43. Recall from Chapter 2 that this means that about 68 percent of the values we produce for this latent variable fall between –1 and 1.

When we're done training, we know that our latent variables will be distributed according to this pattern. If we pick new values to feed to the decoder, and we select them from this distribution (where we're more likely to pick each value near its center and within the bulk of its bump rather than off to the edges), we are likely to generate sets of latent values that are near values we learned from the training set, and thus we can create an output that is also like the training set. This naturally also keeps the samples together in the same area, since they're all trying to match a Gaussian distribution with a center of 0.

Getting the latent variables to fall within unit Gaussians, as shown in Figure 18-43, is an ideal we rarely achieve. There's a tradeoff between how well the variables match Gaussians and how accurate the system can be in re-creating inputs (Frans 2016). The system automatically learns that tradeoff during the training session, striving to keep the latents Gaussian-like while also reconstructing the inputs well.

Clumping Digits Together

Our next goal is getting the latent values of all images with the same digits to clump together. To do this, let's use a clever trick that involves some randomness. It's a bit subtle.

Let's start by assuming that we've *already achieved this goal*. We will see what this implies from a particular point of view, and that will tell us how to actually bring it about. For example, we're assuming that every set of latent variables for an image of, say, the digit 2 is near every other set of latent variables for images of the digit 2. We can do even better, though. Some 2s have a loop in the lower-left corner. So, in addition to having all the 2s clumped together, we can keep all the 2s with loops together and all the 2s without loops together, and the region between those clumps is filled with the latent variables of 2s that sort-of have a loop, as in Figure 18-44.

Now let's carry this idea to its limit. Whatever the shape and style and line thickness and tilt and so on of every image that's labeled a 2, we'll assign that image latent variables that are near other images labeled 2 that show about the same shape and style. We can gather together all the 2s with a loop and all those without, all those drawn with straight lines and all those drawn with curves, all those with a thick stroke and all those with a thin one, all the 2s that are tall, and so on. That's the major value of using lots of latent variables: they let us clump together all of the different combinations of these features, which wouldn't be possible in just two dimensions. In one place we have thin straight no-loop 2s, another region has thick curved no-loop 2s, and so on, for every combination.

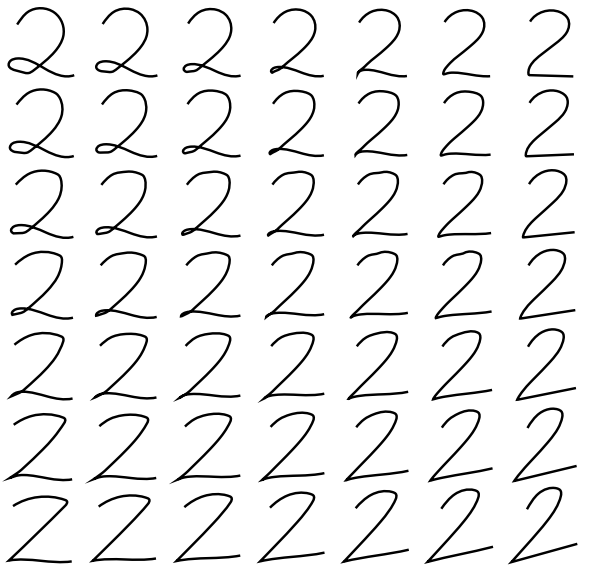


Figure 18-44: A grid of 2s organized so that neighbors are all like one another. We want the latent variables for these 2s to follow roughly this kind of structure.

If we had to identify all of these features ourselves, this scheme wouldn't be very practical. But a VAE not only learns the different features, it automatically creates all the different groupings for us as it learns. As usual, we just feed in images and the system does all the rest of the work.

This "nearness" criterion is measured in latent space, where there's one dimension for each latent variable. In two dimensions, each set of latent variables creates a point on the plane, and their distance (or "nearness") is the length of the line between them. We can generalize this idea to any number of dimensions, so we can always find the distance between two sets of latent variables, even if each one has 30 or even 300 values: we just measure the length of the line that joins them.

We want the system to clump together similar-looking inputs. But recall that we also want each latent variable to form a Gaussian distribution. These two criteria can come into conflict. By introducing some randomness, we can tell the system to "usually" clump the latent variables for similar input, and "usually" also distribute those variables along a Gaussian curve. Let's see how randomness lets us make that happen.

Introducing Randomness

Suppose that our system is given an input of an image of the digit 2, and as usual, the encoder finds the latent variables for it. Before we hand these to the decoder to produce an output image, let's add a little randomness to each of the latent variables and pass those modified values to the decoder, as in Figure 18-45.

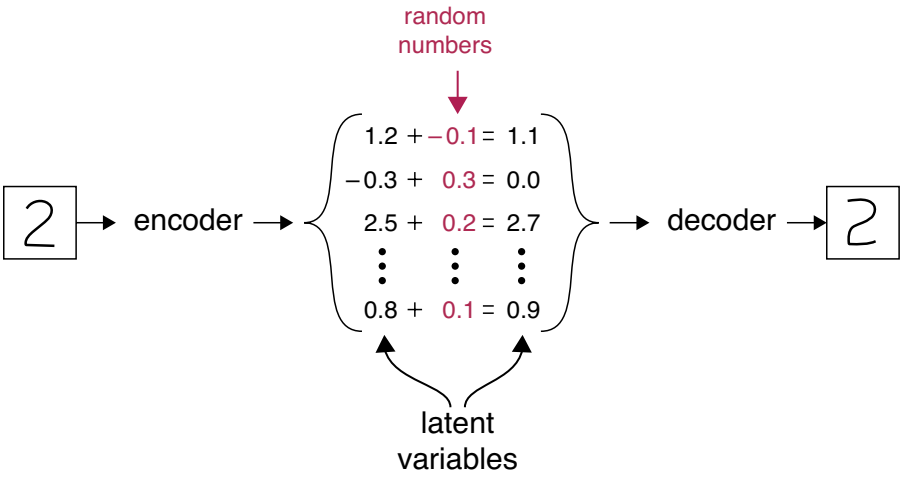


Figure 18-45: One way to add randomness to the output of the encoder is to add a random value to each latent variable before passing them to the decoder.

Because we're assuming that all of the examples of the same style are clumped together, the output image we generate from the perturbed latent variables will be similar to (but different from) our input, and thus the error that measures the difference between the images will also be low. Then we can make lots of new 2's that are like the input 2, just by adding different small random numbers to the same set of latent values.

That's how it works once the clumping has already been done. To get the clumping done in the first place, all we have to do is give the network a big error score during training when this perturbed output doesn't come very close to matching the input. Because the system wants to minimize the error, it learns that latent values that are close to the input's original latent values should produce images that are close to the input image. As a result, the latent values for similar inputs get clumped together, just as we desired.

But we took a shortcut just now that we can't follow in practice. If we just add random numbers as in Figure 18-45, we won't be able to use the backpropagation algorithm we saw in Chapter 14 to train the model. The problem comes about because backpropagation needs to compute the gradients flowing through the network, but the mathematics of an operation like Figure 18-45 don't let us calculate the gradients the way we need to. And without backpropagation, our whole learning process disappears in a puff of smoke.

VAEs use a clever idea to get around this problem, replacing the process of adding random values with a similar idea that does about the same job, but which lets us compute the gradient. It's a little bit of mathematical substitution that lets backpropagation work again. This is called the *reparameterization trick*. (As we've seen a few times, mathematicians sometimes use the word *trick* as a compliment when referring to a clever idea.)

It's worth knowing about this trick because it often comes up when we're reading about VAEs (there are other mathematical tricks involved, but we won't go into them). The trick is this: instead of just picking a

random number from thin air for each latent variable and adding it in, as in Figure 18-45, we draw a random variable from a probability distribution. That value now becomes our latent variable (Doersch 2016). In other words, rather than start with a latent value and then add a random offset to it to create a new latent value, we use the latent value to control a random number generation process, and the result of that process becomes the new latent value.

Recall from Chapter 2 that a probability distribution can give us random numbers, where some are more likely than others. In this case, we use a Gaussian distribution. This means that when we ask for a random value, we're most likely to get a number near where the bump is high, and we're less and less likely to get numbers that are farther away from the center of the bump.

Since each Gaussian requires a center (the mean) and a spread (the standard deviation), the encoder produces this pair of numbers for each latent variable. If our system has eight latent variables, then the encoder produces eight pairs of numbers: the center and spread for a Gaussian distribution for each one. Once we have them, then for each pair of values, we pick a random number from the distribution they define, and that's the value of the latent variable that we then give to the decoder. In other words, we create a new value for each latent variable that's pretty close to where it was, but has some randomness built in. The restructuring of the perturbation process lets us apply backpropagation to the network.

Figure 18-46 shows the idea.

The structure of our autoencoder, as shown in Figure 18-46, requires the network to *split* after the computation of the latent value. Splitting is a new technique for our repertoire of deep learning architectures: it just takes a tensor and duplicates it, sending the two copies to two different subsequent layers. After the split, we use one layer to compute the center of the Gaussian and one to compute the spread. We sample this Gaussian and that gives us our new latent value.

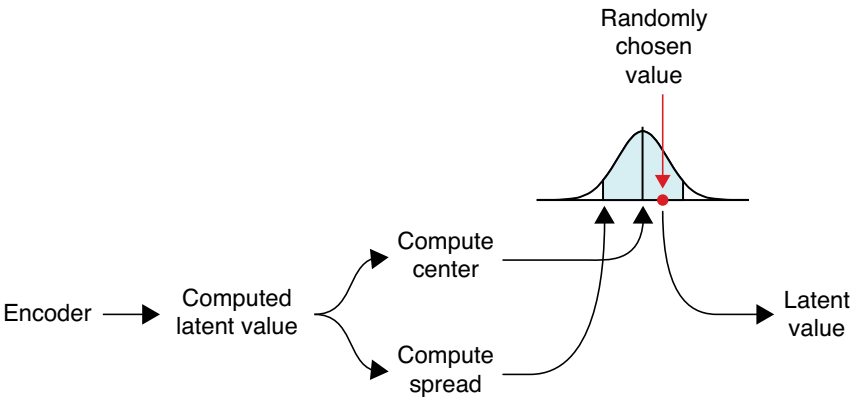


Figure 18-46: We use the computed latent value to get the center and spread of a Gaussian bump. We pick a number from that bump, and that becomes our new latent value.

To apply our sampling idea of Figure 18-46, we create a Gaussian for each latent variable and sample it. Then we feed all of the new latent values into a *merge* or *combination* layer, which simply places all of its inputs one after the other to form a single list (in practice, we often combine the sampling and merging steps together into one layer). Figure 18-47 shows how we'd process a latent vector with three values.

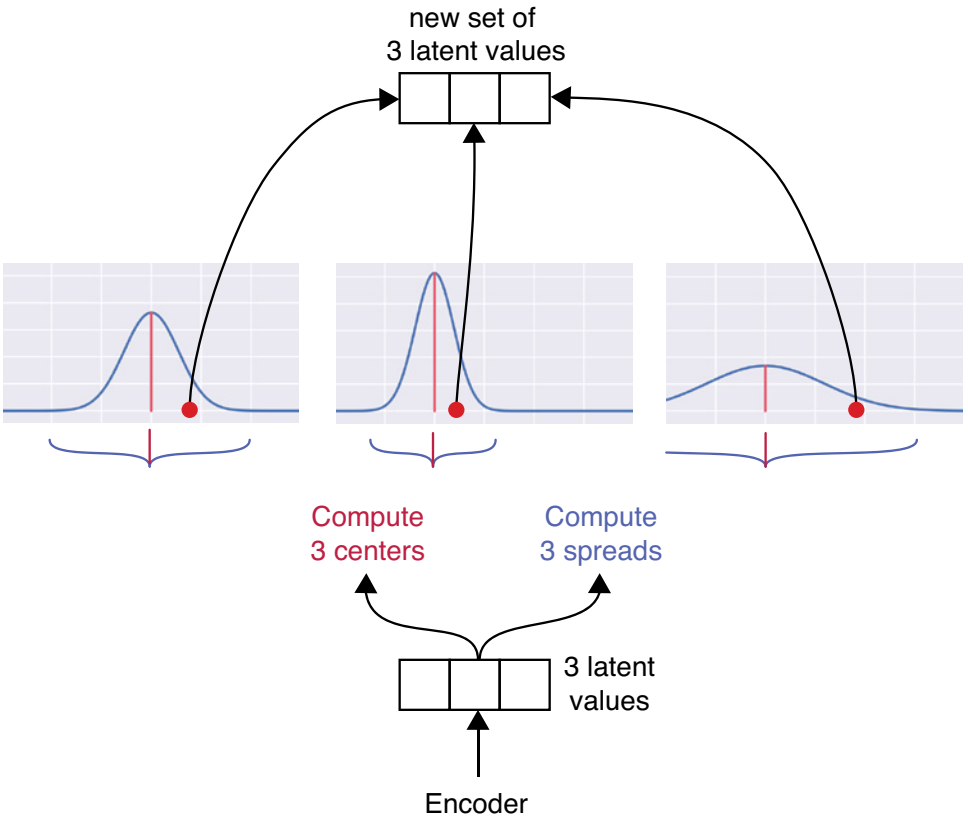


Figure 18-47: Picturing the split-and-combine sampling step of a VAE for three latent variables

In Figure 18-47, the encoder ends with three latent variables, and for each one, we compute a center and spread. Those three different Gaussian bumps are then randomly sampled, and those selected values are merged, or combined, to form the final latent variables computed for that input. These variables are the output of the encoder section.

During the learning process, the network learns what the centers and spreads should be for each Gaussian.

This operation is why we said earlier that each time we send a sample into a trained VAE (that is, after learning is done), we get back a slightly different result. The encoder is deterministic up to and including the split. But then the system picks a random value for each latent variable from its Gaussian, and those are different each time.

Exploring the VAE

Figure 18-48 shows the architecture of a fully connected VAE. It's just like our deep autoencoder built from fully connected layers of Figure 18-17, but with two changes (we chose fully connected layers rather than convolution layers for simplicity).

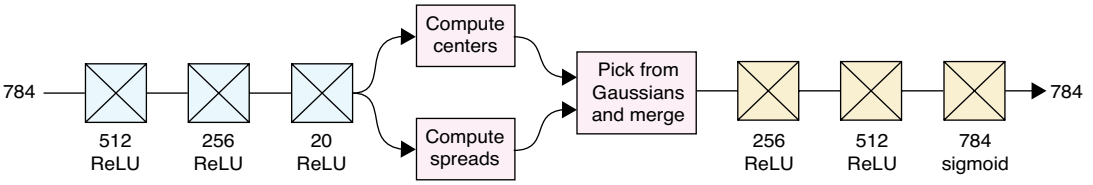


Figure 18-48: The architecture of our VAE for MNIST data. There are 20 latent values.

The first change is that we now we have the split-select-merge process at the end of the encoder. The second change is that we use our new loss, or error, function.

Another job we'll assign to our new loss function is to measure the similarity between the fully connected layers of the encoding and decoding stages. After all, whatever the encoding stage is doing, we want the decoding stage to undo it.

The perfect way to measure this is with the Kullback–Leibler (or KL) divergence that we saw in Chapter 6. Recall that this measures the error we get from compressing information using an encoding that is different from the optimal one. In this case, we're asserting that the optimal encoder is the opposite of the decoder, and vice versa. The big picture is that as the network tries to decrease the error, it is therefore decreasing the differences between the encoder and decoder, bringing them closer to mirroring each other (Altosaar 2020).

Working with the MNIST Samples

Let's see what comes out of this VAE for some of our MNIST samples. Figure 18-49 shows the result.

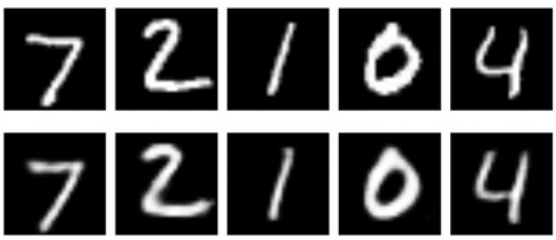


Figure 18-49: Predictions from our VAE of Figure 18-48. Top row: Input MNIST data. Bottom row: Output of the variational autoencoder.

It's no surprise that these are pretty good matches. Our network is using a lot of compute power to make these images! But as we have seen from its architecture, the VAE produces different outputs each time it sees the same image. Let's take the image of the two from this test set and run it through the VAE eight times. The results are in Figure 18-50.

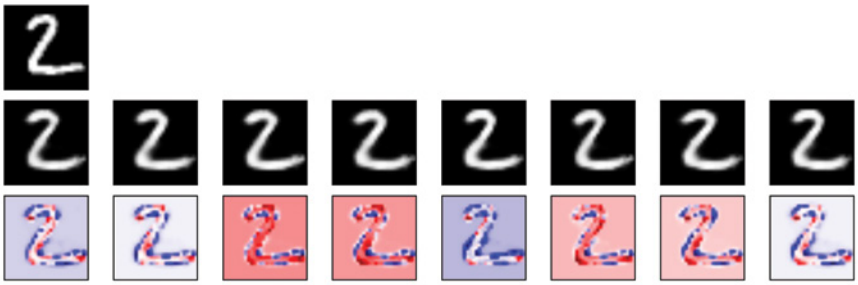


Figure 18-50: The VAE produces a different result each time it sees the same input. Top row: The input image. Middle row: The output from the VAE after processing the input eight times. Bottom row: The pixel by pixel differences between the input and each output. Increasing red means larger positive differences, increasing blue means larger negative differences.

These eight results from the VAE are similar to each other, but we can see obvious differences.

Let's go back to our eight images from Figure 18-49 but add additional noise to the latent variables that come out of the encoder. That is, just before the decoder stage, we add some noise to the latent variables. This gives us a good test of how clumped-together the training images are in latent space.

Let's try adding a random value that's up to 10 percent of each latent variable's amount. Figure 18-51 shows the result of adding this moderate amount of noise to the latent variables.

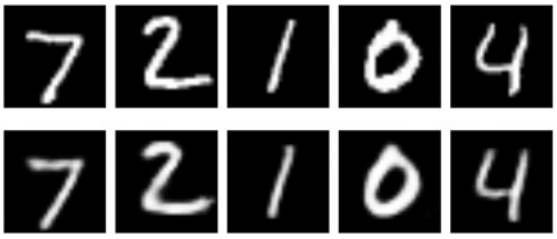


Figure 18-51: Adding 10 percent noise to the latent variables coming out of the VAE encoder. Top row: Input images from MNIST. Bottom row: The decoder output after adding noise to the latent variables produced by the encoder.

Adding noise doesn't seem to change the images much at all. That's great, because it's telling us that these noisy values are still "near" the original inputs. Let's crank up the noise, adding in a random number as much as 30 percent of the latent variable's value. Figure 18-52 shows the result.

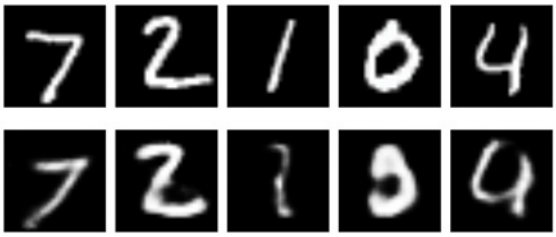


Figure 18-52: Perturbing the latent variables by up to 30 percent. Top row: The MNIST input images. Bottom row: The results from the VAE decoder.

Even with a lot of noise, the images still look like digits. For example, the 7 changes significantly, but it changes into a bent 7, not a random splotch.

Let's try blending the parameters for our digits and see how that looks. Figure 18-53 shows the equal blends for the five pairs of digits we've seen before.



Figure 18-53: Blending latent variables in the VAE. Top and middle row: MNIST input images. Bottom row: An equal blend of the latent variables for each image, decoded.

The interesting thing here is that these are all looking roughly like digits (the leftmost image is the worst in terms of being a digit, but it's still a coherent shape). That's because there's less unoccupied territory in latent space, so the intermediate values are less likely to land in a zone far from other data (and thus produce a strange, nondigit image).

Let's look at some linear blends. Figure 18-54 shows the intermediate steps for the three pairs of digits we've seen before.



Figure 18-54: Linear interpolation of the latent variables in a VAE. The leftmost and rightmost image in each row are the output of the VAE for an MNIST sample. The images in between are decoded versions of the blended latent variables.

The 5 is looking great, moving through a space of 5s from one version to another. The top and bottom rows have plenty of images that aren't digits. They might be passing through empty zones in latent space, but as we mentioned before, it's not clear that these are wrong in any sense. After all, what should an image partly between a four and a three look like?

Let's run our tiger through the system just for fun. Remember, this is a completely unfair thing to do, and we shouldn't expect anything meaningful to come out. Figure 18-55 shows the result.

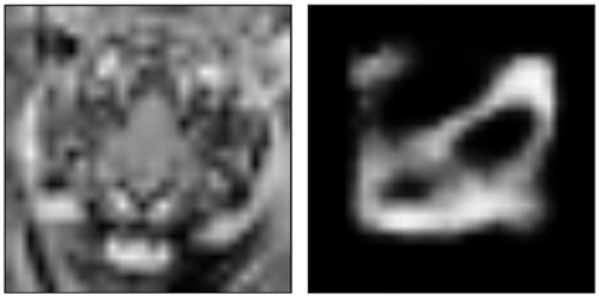


Figure 18-55: Running our low-resolution tiger through the VAE

The VAE created something with a coherent structure, but it's not much like a digit.

Working with Two Latent Variables

For comparison to our other autoencoders, we trained our VAE with just 2 latent variables (rather than the 20 we've been using), and plotted them in Figure 18-56.

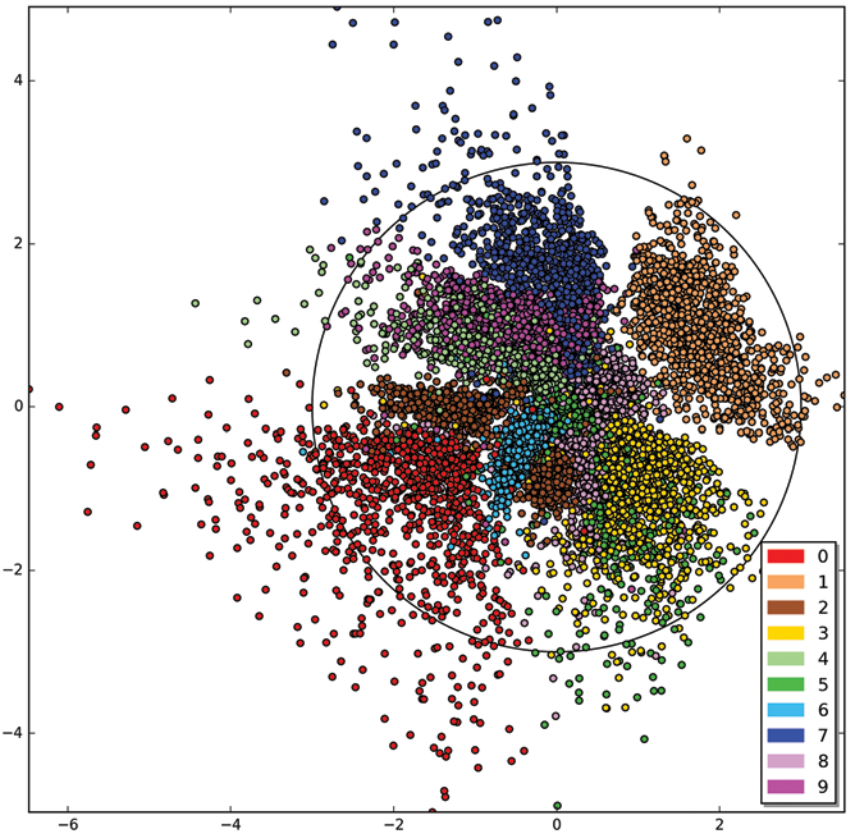


Figure 18-56: The placement of latent variables for 10,000 MNIST images from our VAE trained with two latent variables

This is a great result. The standard deviation of the Gaussian bump the latent variables are trying to stay within is represented here by a black circle, and it seems pretty well populated. The various digits are generally well clumped. There's some confusion in the middle, but remember that this image uses just two latent variables. Curiously, the 2s seem to form two clusters. To see what's going on, let's make a grid of images that correspond to our two latent variables using the recipe shown in Figure 18-22, but using the latent variables of Figure 18-56. Let's take the x and y values of each point on the grid and feed them to the decoder as though they were latent variables. Our range is –3 to 3 on each axis, like the circle in Figure 18-56. The result is Figure 18-57.

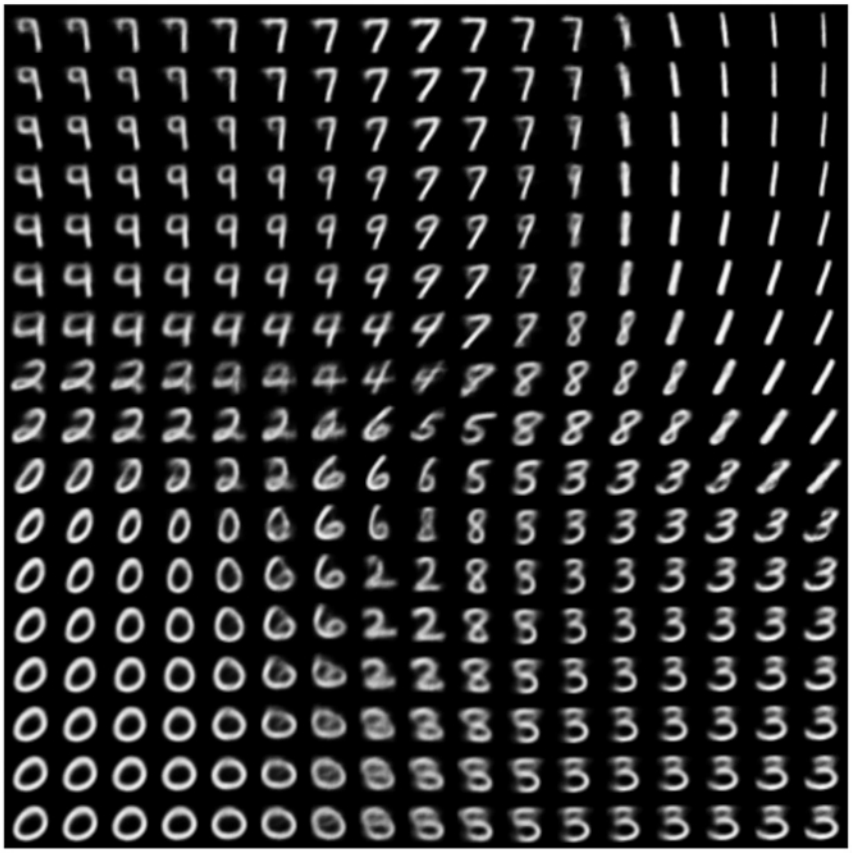


Figure 18-57: The output of the VAE treating the x and y coordinates as the two latent variables. Each axis runs from –3 to 3.

The 2s without a loop are grouped together near the lower middle, and the 2s with a loop are grouped in the middle left. The system decided that these were so different that they didn't need to be near each other.

Looking over this figure, we can see how nicely the digits have been clumped together. This is a far more organized and uniform structure than we saw in Figure 18-23. In a few places the images get fuzzy, but even with just two latent variables, most of the images are digit-like.

Producing New Input

In Figure 18-58 we've isolated the decoder part of the VAE to use as a generator. For the moment, we'll continue to use a version where we've reduced the 20 latent values to just 2.

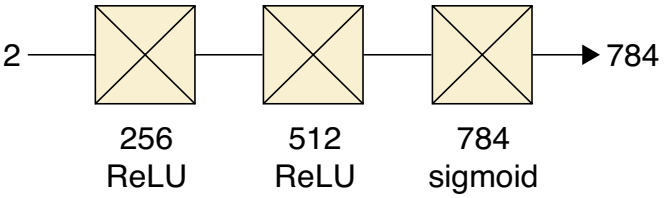


Figure 18-58: The decoder stage of our VAE

Since we have only two latent variables, we randomly picked 80 random (x,y) pairs from a circle centered at (0,0) with radius 3, fed them into the decoder, and gathered the resulting images together into Figure 18-59.

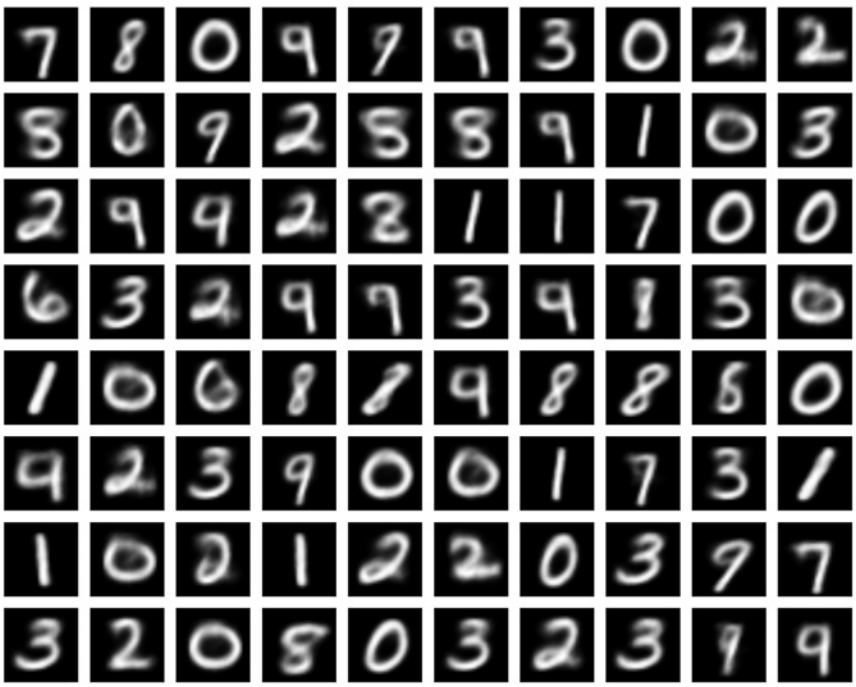


Figure 18-59: Images produced by the VAE decoder when presented with two random latent variables

These are looking pretty great for the most part. Some aren't quite legible, but overall, most of the images are recognizable digits. Many of the mushiest shapes seem to have come from the boundary between the 8s and the 1s, leading to narrow and thin 8s.

Most of these digits are fuzzy, because we're using only two latent variables. Let's sharpen things up by training and then using a deeper VAE with more latent variables. Figure 18-60 shows our new VAE. This architecture is based on the MLP autoencoder that's part of the *Caffe* machine-learning library (Jia and Shelhamer 2020; Donahue 2015). (Recall that MLP stands for multilayer perceptron, or a network built only out of fully connected layers.)

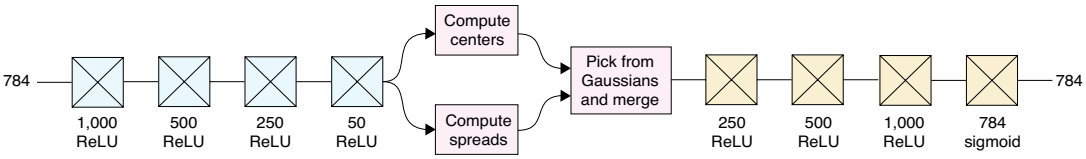


Figure 18-60: The architecture of a deeper VAE

We trained this system with 50 latent variables for 25 epochs and then generated another grid of random images. As before, we used just the decoder stage, shown in Figure 18-61.

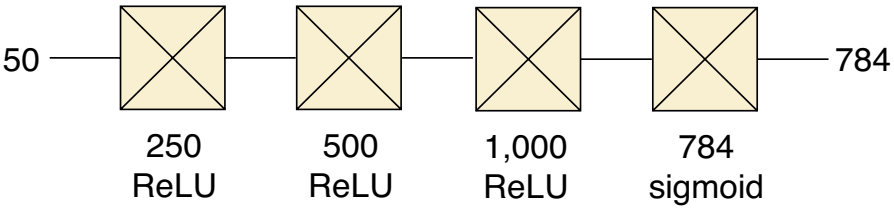


Figure 18-61: We generate new output using just the decoder stage of our deeper VAE, feeding in 50 random numbers to produce images.

The results are in Figure 18-62.

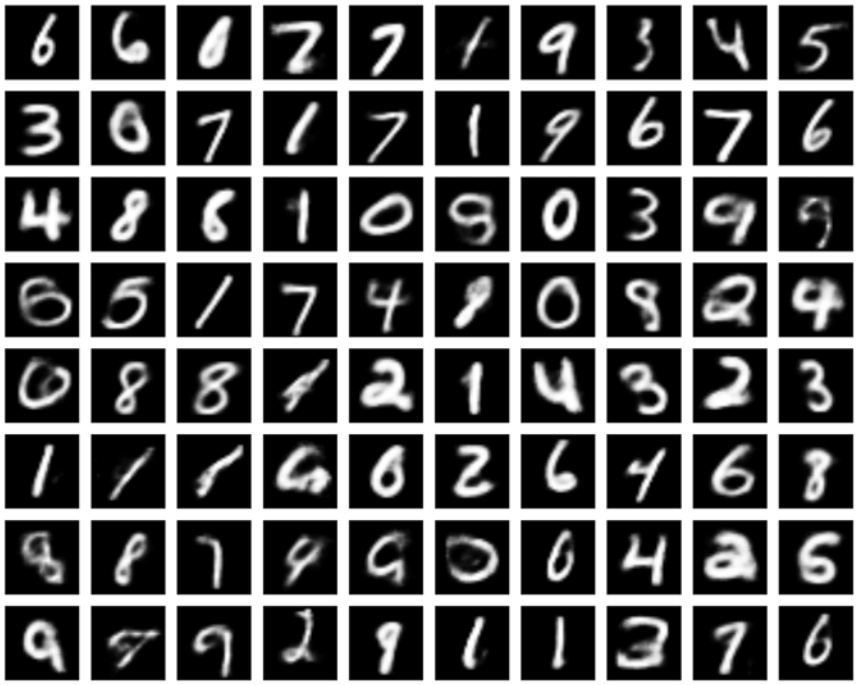


Figure 18-62: Images produced by our bigger VAE when provided with random latent variables

These images have significantly crisper edges than the images in Figure 18-59. For the most part, we've generated entirely recognizable and plausible digits from purely random latent variables, though, as usual, some weird images that aren't much like digits show up. These are probably coming from the empty zones between digits, or zones where different digits are near one another, causing oddball blends of the shapes.

Once our VAE has been trained, if we want to make more digit-like data, we can ignore the encoder and save the decoder. This is now a generator that we can use to create as many new digit-like images as we like. If we were to train the VAE on images of tractors, songbirds, or rivers, we could generate more of those types of images, too.

Summary

In this chapter we saw how autoencoders learn to represent a set of inputs with latent variables. Usually there are fewer of these latent variables than there are values in the input, so we say that the autoencoder compresses the input by forcing it through a bottleneck. Because some information is lost along the way, it's a form of lossy compression.

By feeding latent values of our own choice directly into the second half of a trained autoencoder, we can view that set of layers as a generator, capable of producing new output data that is like the input, but wholly novel.

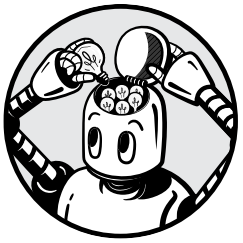
Autoencoders may be built using many kinds of layers. In this chapter, we saw examples of networks built from fully connected layers and convolution layers. We looked at the structure of 2D latent variables generated by a trained autoencoder built of fully connected layers, and found that it had a surprising amount of organization and structure. Picking new pairs of latents from a populated region of these latents and handing them to a generator usually produced an output that was blurry (because we had only two latent values), but plausibly like the input. We then looked at convolutional autoencoders, built primarily (or exclusively) with convolution layers.

We saw that we could blend latent variables, in essence creating a series of in-between latent variables between the endpoints. The more latent variables we used in our bottleneck, the better these interpolated outputs appeared. We then saw that an autoencoder can be trained to denoise the input, simply by telling it to generate the clean value of a noisy input. Finally, we looked at variational autoencoders, which do a better job of clumping similar inputs and filling up a region of the latent space, at the cost of introducing some randomization into the process.

Autoencoders are often used for denoising and simplifying datasets, but people have found creative ways to use them for many kinds of tasks, such as creating music and modifying input data to help networks learn better and faster (Raffel 2019).

19

RECURRENT NEURAL NETWORKS



In most of this book we've considered every sample as an isolated entity, unrelated to any other samples. This makes sense for things like photographs. If we're classifying an image and decide that we're looking at a cat, it doesn't matter if the image before or after this one is a dog, a squirrel, or an airplane. The images are independent of each other. But if an image is a frame of a movie, then it can be helpful to look at it in the context of the other images around it. For example, we can track objects that might be temporarily obscured.

When we work with multiple samples whose order matters, we call that a *sequence*. The flow of words in any human language are an important type of sequence and will be our focus in this chapter.

Algorithms that understand and process sequences have another bonus: they are frequently capable of *generating*, or creating, new sequences. Trained systems can generate stories (Deutsch 2016a) or TV scripts (Deutsch 2016b), Irish jigs (Sturm 2015b), polyphonic melodies (LISA Lab 2018), and complex songs (Johnson 2015; O'Brien and Román 2017). We can create lyrics (Krishan 2016) for pop music (Chu, Urtasun, and Fidler 2016), folk music (Sturm 2015a), rap (Barrat 2018), or country (Moocarme 2020). We can turn

speech into text (Geitgey 2016; Graves, Mohamed, and Hinton 2013) and write captions for images and video (Karpathy and Li 2013; Mao et al. 2015).

In this chapter, we look at a method for handling sequences based on remembering something about each element as it comes by. The models we build are called recurrent neural networks (RNNs).

When we work with sequences, each element of the input is called a *token*. A token represents a word, or a fragment of a word, a measurement, or anything else we can represent numerically. In this chapter, we use language as our most frequent source of data, and we focus on whole words, so we use *word* and *token* interchangeably.

Working with Language

The general field that studies natural language is called *natural language understanding*, or *NLU*. Most of today's algorithms are unconcerned with any kind of actual understanding of the language they process. Instead, they extract statistics from the data and use those statistics as the basis for tasks like answering questions or generating text. These techniques are generally called *natural language processing*, or *NLP*.

We saw in Chapters 16 and 17 that convolutional neural networks, or CNNs, can recognize objects in photos without having any actual understanding of the photo. They just process the statistics of the pixels. In the same way, NLP systems don't understand the language they manipulate. Instead, they assign numbers to words and find useful statistical relationships between those numbers.

In a fundamental sense, these systems have no knowledge that there is even a thing such as language, or that the objects they manipulate have semantic meanings. As always, the system is using statistics to generate outputs that we declare to be acceptable in a given situation, without even a glimmer of comprehension of what it's doing or what the outputs might mean to a person.

Common Natural Language Processing Tasks

The applications of natural language algorithms are commonly called *tasks*. Here are some popular tasks:

Sentiment Analysis: Given opinionated text like a movie review, determine whether the overall sense is positive or negative.

Translation: Turn text into another language.

Answer Questions: Answer questions about the text, like who is the hero, or what actions occurred.

Summarize or Paraphrase: Provide a short overview of the text, emphasizing the main points.

Generate New Text: Given some starting text, write more text that seems to follow from it.

Logical Flow: If a sentence first asserts a premise and the following sentence asserts a conclusion based on that premise, determine whether the conclusion logically follows from the premise.

In this chapter and the next, we focus mainly on two tasks: translation and text generation. The other tasks have much in common with these (Rajpurkar, Jia, and Liang 2018; Roberts, Raffel, and Shazeer, 2020). In particular, logical flow is extra difficult and benefits from human-computer partnerships (Full Fact 2020).

Translation requires, at a minimum, the text we want to translate, and the source and target languages. We might also want to know some context to help us understand idioms and other language features that change from one place to another or over time.

Text generation typically starts with a *seed* or *prompt*. The algorithm takes that as the start of the text and then builds from there. Typically, it does this one word at a time. Given a prompt, it predicts the next word. That word is added to the end of the prompt, and the system uses that new, longer prompt to predict the next word after it. We can repeat this process endlessly to produce a sentence, essay, or book. We call this technique *autoregression* because we're predicting, or regressing, the next word in the sequence by automatically appending previous outputs together and using them as the input. Autoregressive systems are called *autoregressors*. More generally, creating text algorithmically is called *natural language generation*, or *NLG*.

Both translation and text generation make use of a concept called a *language model*. This is any kind of computation that takes a sequence of words as an input and tells us how likely it is that the sequence is a well-formed sentence. Note that it doesn't tell us if it's a particularly well-written sentence, or even if it's meaningful or true. It's often convenient to refer to a trained neural network as itself being a language model (Jurafsky 2020).

Transforming Text into Numbers

To build systems that can help us with translation and text generation, we have to first transform our text into a form that's useful to the computer. As usual, we'll turn everything into numbers. There are two popular ways to do this.

The first is *character based*, where we number all the symbols that can appear in our text. The most extensive tabulation of written characters in human language is called Unicode. The most recent version, Unicode 13.0.0, encompasses 154 written human languages and identifies 143,859 distinct characters (Unicode Consortium 2020). We can assign every symbol from any of these writing systems a unique number from 0 to about 144,000. In this chapter, we keep things simple and show a few examples of text generation using the 89 characters most common in English text.

The second approach is *word based*, where we number all the words that can appear in our text. Counting all the words in all the languages of the world would be a daunting task. In this book, we stick to English, but even there, we have no definitive count of the number of words. Most modern English dictionaries have about 300,000 entries (Dictionary.com 2020). Imagine working through the dictionary and assigning each entry a unique number starting at 0. These words and their corresponding numbers would then make up our *vocabulary*. Most of the examples in this chapter take a word-based approach.

Now we can create a computer-friendly, numerical representation of any sentence. We can generate more text by handing this list of numbers to a trained autoregressive network. The network predicts the number of the next word, that word gets appended to the words used as its input, the network then predicts the next word, which again gets appended to the words used as its input, and so on. For us to see what text this corresponds to, we can turn each number back into its corresponding word. For many of our discussions in the following pages, we take these transformations into numbers as a given and illustrate our inputs and outputs as words, not numbers. We'll see later that while a single number is workable, there's a much richer way to represent words that includes their context and how they're used in a sentence.

Fine-Tuning and Downstream Networks

It's often useful to train a system on a generic database and then specialize it. For example, we might enhance a general-purpose image classifier into one that can recognize leaf shapes and tell us what kind of tree they came from. The process is called *transfer learning*. When used with a classifier, it often involves freezing the existing network, adding a few new layers at the end of the classification section, and training those. That way, the new layers can make use of all the information that the existing network has learned to extract from each image.

In NLP, we say that a system that has learned from a general database is *pretrained*. Then when we want to learn a new type of specialized language, like the language used in law, poetry, or engineering, we *fine-tune* the network with the new data. Unlike transfer learning, we typically modify all the weights in the system when we fine-tune.

If we don't want to retrain the system, we can create a second model to take the language system's output and turn it into something more useful to us, which is close in spirit to transfer learning. Here the language model is frozen, and its output is fed to a new model. We call this second model a *downstream network*, which carries out a *downstream task*. Some language models are designed to create rich, dense summaries of their input text so they can be used to drive a wide variety of downstream tasks.

These two approaches of fine-tuning and downstream training are useful conceptual distinctions, but in practice, many systems blend together some of both techniques.

Fully Connected Prediction

As we've discussed, we're going to treat language as sequences of numbers. To get a feeling for working with such sequences in general, let's set aside language for a moment and focus just on the numbers. We'll build a tiny network that learns to take in a few numbers from a sequence, and produce the next number. We'll do it in perhaps the simplest possible way, with just two layers: a fully connected layer of a mere five neurons, followed by a fully connected layer with a single neuron, as in Figure 19-1. We'll use a leaky

ReLU activation function with slope 0.1 on the first layer and no activation function on the output layer.

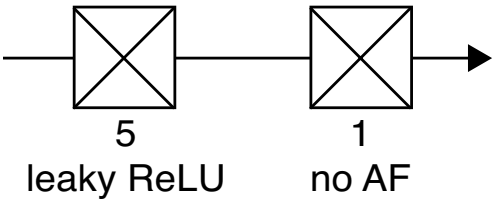


Figure 19-1: A tiny network for sequence prediction

Testing Our Network

To try out this tiny network, let's use a synthetic dataset created by adding a bunch of sine waves together. The first 500 samples are shown in Figure 19-2.

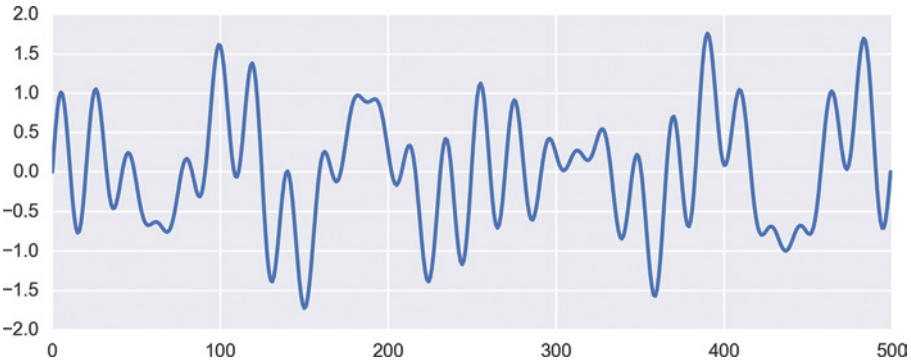


Figure 19-2: Synthetic training data

To train our system, we'll take the first five values from our dataset and ask our little network to produce the sixth value. Then we'll take values 2 through 6 of the dataset and ask it to predict the seventh value. We say that we're using a *sliding window* to choose each set of inputs, as in Figure 19-3.

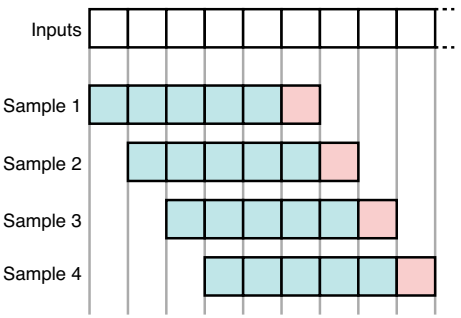


Figure 19-3: Using a sliding window to create training samples, shown in blue, from 5-element sequences of the training data. The value we want to predict for each sample is in red.

From our starting 500 values, we can make 495 samples in this way. We trained our little network on these samples for 50 epochs. When we run the training data through again and ask for predictions, we get the results on the left of Figure 19-4, showing the original training data in blue, and the predictions in orange. Not bad!

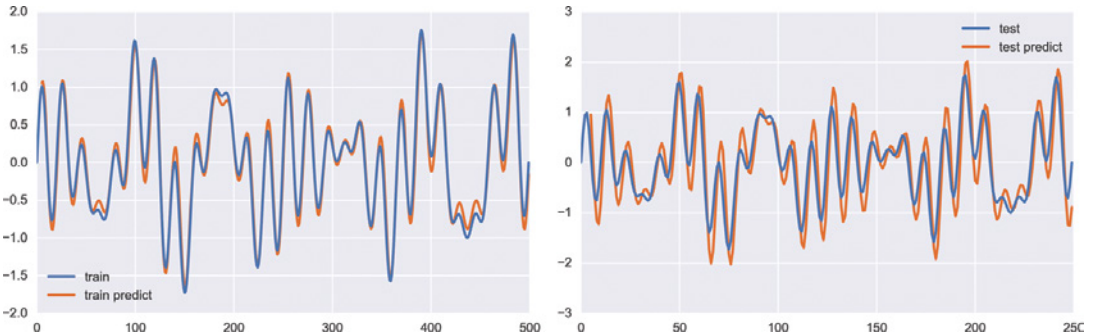


Figure 19-4: Left: Training data and predictions. Right: Test data and predictions.

Let's now run this on 250 points of test data from later in the curves. The data and predictions are shown on the right of Figure 19-4. The predictions aren't perfect, but they are pretty great, considering how small our network is.

This was easy data, though, since it was so smooth. Let's try a more realistic dataset composed of the average number of sunspots recorded monthly from 1749 to 2018 (Kaggle 2020). Figure 19-5 shows the inputs and outputs using the same arrangement as in Figure 19-4. The peaks and valleys correspond to the roughly 11-year solar cycle. Though it doesn't quite reach the extremes of the data, our tiny regressor seems to follow the general ups and downs of the data quite well.

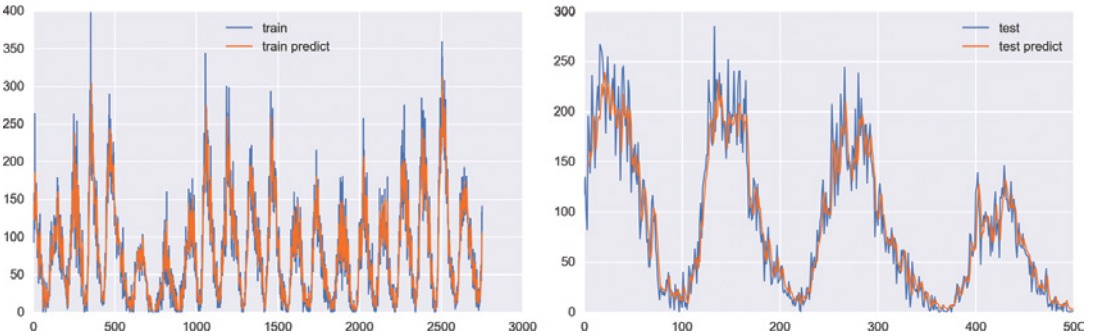


Figure 19-5: Left: Training sunspot data and predictions. Right: Test data and predictions.

Unfortunately, this little network is not going to be able to generate enjoyable novels. To see why, let's change our data to numbered words. For our text, we'll use the first six chapters of Charles Dickens' novel *A Tale of*

Two Cities (Dickens 1859). To make processing easier, we stripped out all the punctuation and turned everything into lowercase.

Since we're going to work at word level, we need to assign a number to every word we'll use. Numbering an entire dictionary would be overkill, and we'd miss all the people and place names in the text. Instead, let's build our vocabulary from the book itself. Let's assign the value 0 to the first word in the book and then work our way forward one word at a time. Each time we see a word we haven't seen before, we assign it the next available number. This opening chunk of the novel contains 17,267 words total but has a vocabulary of only 3,458 unique words, so our words have values from 0 to 3,457.

Now that every word in this part of the novel has a number, we split the database into training and test sets. At the end of the training data, we have only seen about 3,000 unique words. So that we don't ask the network to predict word numbers it hasn't been trained with, we removed all sequences in the test set where any word numbers (or the target) are above this value. That is, the test data consists of sequences that only use words that are present in the training data.

We repeated the previous experiment and fed windows of five consecutive word numbers to the little network of Figure 19-1, collecting from the output its prediction of the next word. We told it to train for 50 epochs, but the error quickly stopped improving and early stopping brought training to a close after 8 epochs, giving us the results in Figure 19-6. As we can see in the training data on the left, the word numbers gradually increase as we get further into the book. The orange lines are the word numbers predicted by the system in response to each set of five inputs.

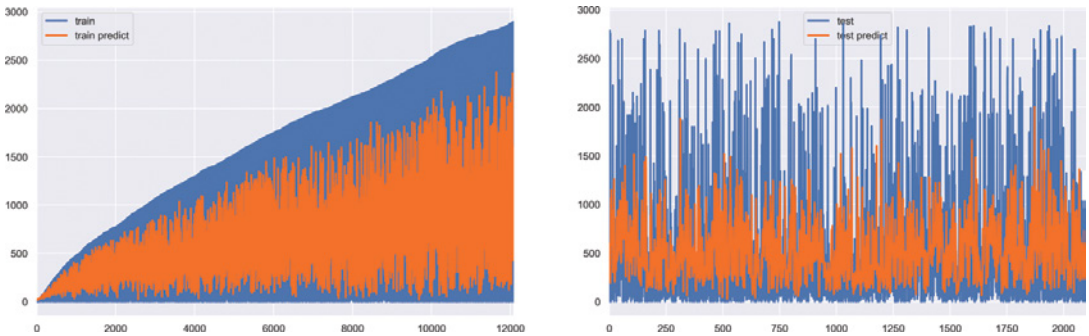


Figure 19-6: Left: Training and predictions for the first roughly 12,000 words from the first six chapters of A Tale of Two Cities. *Right: Test data and predictions for roughly 2,000 more words.*

That's not good at all. The predictions definitely aren't matching either the training or test data. The structure of the test data and predictions is easier to see in the close-up shown in Figure 19-7.

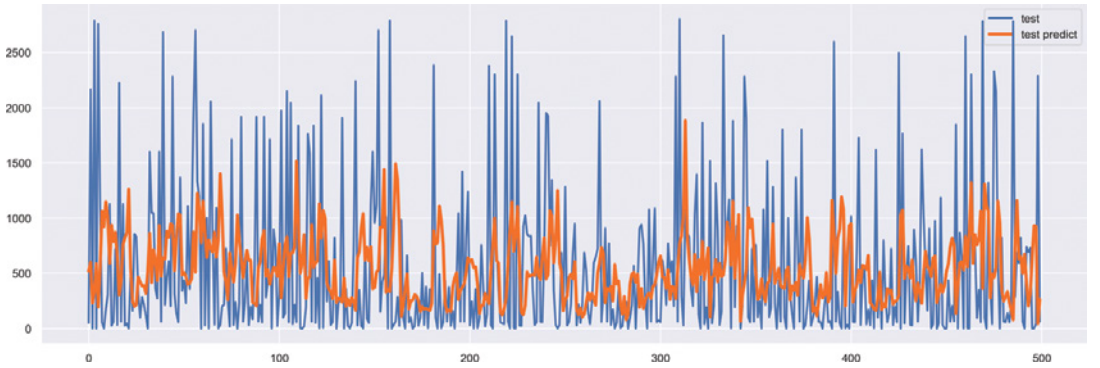


Figure 19-7: Close-up of 500 pieces of test data and predictions from Figure 19-6

The predictions appear to vaguely follow the targets, but they're way off.

Why Our Network Failed

Let's turn the numbers of Figure 19-7 back into words. Here's a typical extract:

> pricked hollows mud crosses argument ripples loud want joints upon harness followed side three intensely atop fired wrote pretence

That's not great literature, even if we put some punctuation back in. A number of things went wrong here. First, this one little network clearly doesn't have anywhere near enough power for this job. We'd need many more neurons, maybe on many layers, to get anywhere near readable text.

Even much larger fully connected networks will struggle with this task, though, because they have no way of capturing the structure of the text, also called its *semantics*. The structure of language is fundamentally different than that of the curves and sunspot data we saw before. Consider the five-word string Just yesterday, I saw a. This fragment can be completed by any noun. By one estimate, the number of nouns in English runs to at least tens of thousands (McCrae 2018). How could any network possibly guess the one we want? One answer is to make the window bigger, so the network has more preceding words and may be able to make a more informed choice. For example, given the input, I've been spending my time watching tigers very closely. Just yesterday, I saw a, most English nouns can now be reasonably ruled out as unlikely.

Let's try this out. We enlarged our little network in Figure 19-1 to have 20 neurons on the first layer. We gave it 20 elements at a time and asked it to predict the 21st. The results for the curve data are shown in Figure 19-8.

Though the training data is still pretty okay, the test results are much worse. To handle all the information coming from this bigger window, we need a far bigger network. Making the window bigger means we need a bigger network, which means it needs more training data, more memory, more compute power, more electricity, and more training time.

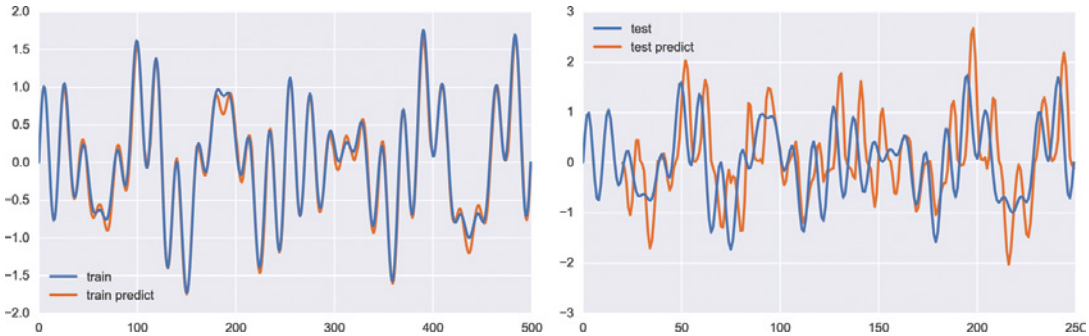


Figure 19-8: An enlarged network predicting sine wave data using a window of 20 elements

But there's an even bigger problem that won't improve just by using a bigger network. The issue is that even a tiny error in the prediction leads to incomprehensible text. To see this, let's arbitrarily look at the words that were assigned values 1,003 and 1,004. These numbers correspond to the words keep and flint. The words seem entirely unrelated, but searching the text turns up this passage near the start of the book: he had only to shut himself up inside, keep the flint and steel sparks well off the straw. The word the has already appeared as the third word of the book, so since neither keep nor flint had appeared earlier, when we numbered the book's words, keep and flint were assigned successive numbers.

Suppose that in response to some input, our network predicts the next word to be 1,003.49. We need to turn this into an integer to look up the corresponding word. The nearest integer is 1,003, giving us keep. But if the system predicts the slightly larger value 1,003.51, the nearest integer is 1,004, giving us flint. These two words are entirely unrelated. This demonstrates that even a tiny numerical difference in the prediction can create nonsensical output.

Looking back on our predictions in the graphs for this network, we can see lots of errors that didn't seem too terrible for the curve and sunspot data, but would wreak havoc on language data. Throwing more compute power at this problem will reduce it, but our need for pinpoint accuracy won't go away.

Our little network of Figure 19-1 is hiding another flaw: it doesn't track the locations of the words in its input. Suppose we are given the sentence, Bob told John that he was hungry, and we want to know who the pronoun he refers to. The answer is Bob. But word order matters, because if we instead were given the sentence, John told Bob that he was hungry, then he would refer to John. The need for accuracy would encourage us to extend the network with more fully connected layers, and we'd lose the implicit ordering of the words when they arrived at the first layer. Later layers wouldn't have any chance at working out which word corresponds to he.

To address these issues, and many others, we want something more sophisticated than fully connected layers and words represented by single numbers. We might try using a CNN, and there has been some work on

using CNNs to handle sequence data (Chen and Wu 2017; van den Oord et al. 2016), but those tools are still developing. Instead, let's look at something explicitly designed to handle sequences.

Recurrent Neural Networks

A better way to handle language is to build a network that is explicitly designed to manage words as an ordered sequence. One such type of network, and the focus of this chapter, is the recurrent neural network, or RNN. Such networks build on a few concepts we haven't looked at before, so let's consider them now and then use them to build an RNN.

Introducing State

RNNs make use of an idea called *state*. This is just a description of a system (such as a neural network) at any given time. For example, imagine preheating an oven. In this process, the oven takes on three unique states: off; preheating; and at the desired temperature. The state can also contain additional information. For example, as the oven warms up, we can pack three pieces of information into the oven's state: its current status (such as preheating); the temperature it's currently at; and the temperature it's aiming for. So, a state can represent the current condition of a system, plus any other information it's convenient to remember.

Because state is so important, let's see some of its subtleties with another example.

Suppose that you're working at an ice cream shop and you're learning how to make a simple fudge sundae. In this story, you play the role of the system, and the recipe you're building up in your head is your state.

Before getting any instructions, your *starting state* or *initial state* would be "An empty cup." So, let's say you have an empty cup. Your starting state is shown at the far left of Figure 19-9.

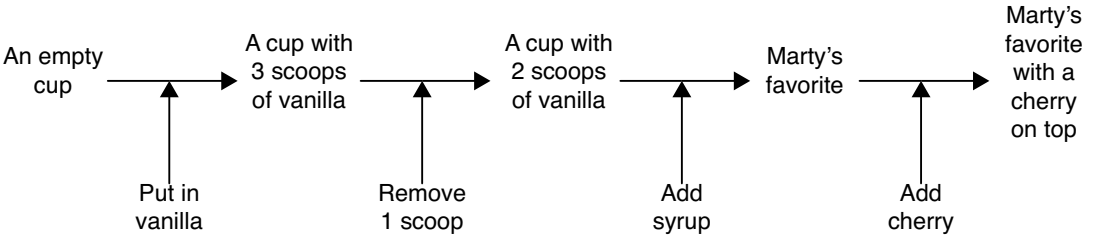


Figure 19-9: Your evolving state, or recipe, as you learn to make a dessert

Your manager says that the first step is to put in some vanilla ice cream. So, you update your internal recipe, or state, to "An empty cup with three scoops of vanilla ice cream." You put three scoops of ice cream into the cup.

Your manager says that's too much, and you should remove one scoop. You do so, and you update your state to "An empty cup with two scoops of vanilla ice cream."

Now your manager says to pour on enough chocolate syrup to cover the ice cream. You do this, and update your state to "an empty cup with two scoops of vanilla ice cream covered in chocolate syrup." But this reminds you of your friend Marty, because this is his favorite dessert. So, you simplify your state by throwing out what you had, now remembering only "Marty's favorite."

Finally, your manager says you should place a cherry on the top. So, you update your state to "Marty's favorite with a cherry on top." Congratulations, your sundae is complete!

There are a few key things to take away from this story and the concept of state.

First, your state is not simply a snapshot of the current situation or a list of the information you were given. It captures both of those ideas, perhaps in a compressed or modified form. For example, instead of remembering to put in three scoops of ice cream and then removing one, you remembered instead to put in two scoops.

Second, after receiving new information at each step, you updated your state and produced an output. The output depends on the input you received and your internal state, but an outside observer can't see your state, and so they might not understand how your output resulted from the input you just received. In fact, outside observers usually don't get to see a system's internal state. We emphasize this by sometimes referring to a system's state as its *hidden state*.

Finally, the order of the inputs matters. This is the essential aspect of this example that makes it about a sequence, rather than just a bunch of inputs, and thus distinguishes it from our simple, fully connected layer at the start of the chapter. If you'd put the chocolate in the cup first, you'd have made quite a different dessert, and you probably wouldn't have created a reference to your friend Marty in your state.

We call each input a *time step*. This makes sense when the inputs represent events in time, as they were here. Other sequences might not have a time component, like a sequence describing the depth of a river at successive points along its length from its source to its terminus. In particular, words in a sentence have a time component when they're spoken aloud, but that idea doesn't really apply when they're printed. Nevertheless, the term *time step* is widely used to refer to each successive element of a sequence.

Rolling Up Our Diagram

If we had a long sequence of inputs to process, a drawing like Figure 19-9 can consume a lot of space on the page. So, we usually draw something like this in a more compact form, as in Figure 19-10. We've put hyphens between the words here to suggest that each little phrase is to be understood as a single chunk of information.

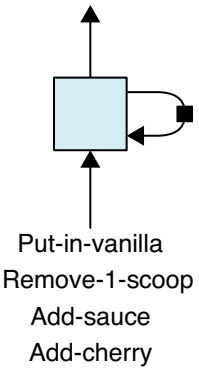


Figure 19-10: The rolled-up version of Figure 19-9

The loop on the right represents the state between one input and the next. After each input, the system (represented by the big, light blue box) creates a new state, which goes into the black square. This square is called the *delay*, and we can think of it as a little piece of memory. When the next input arrives, the system pulls the state out of the delay and computes an output and a new state. That new state again emerges from the system and sits in the delay until the next input arrives. The purpose of the delay is to make it clear that the state produced during each time step is not immediately used again in some way, but is held until it's needed to process the next input.

We say that the diagram in Figure 19-9 is the *unrolled* version of the process. The more compact version in Figure 19-10 is called the *rolled-up* or *rolled* version.

In deep learning, we implement the process of managing state and presenting output by packaging everything up into a *recurrent cell*, as shown in Figure 19-11. The word *recurrent* refers to the fact that we use the state memory over and over, even though its contents usually change from one input to the next (note that this is not the word *recursive*, which sounds similar but means something quite different). The workings of the cell are usually managed by multiple neural networks. As usual, these networks learn how to do their job when we train the complete network, which contains Figure 19-11 as a layer.

We'll see that even though a cell's internal state is usually private, some networks can make good use of this information, so here we show the exported state as a dashed line, suggesting that it's available, but can be ignored if not needed.

We often place a recurrent cell on a layer of its own and call that a *recurrent layer*. A network that is dominated by recurrent layers is called a *recurrent neural network*, or *RNN*. The same term is frequently applied to the recurrent layers themselves, and sometimes even the recurrent cells, since they have neural networks inside them. The correct interpretation is usually clear from context.

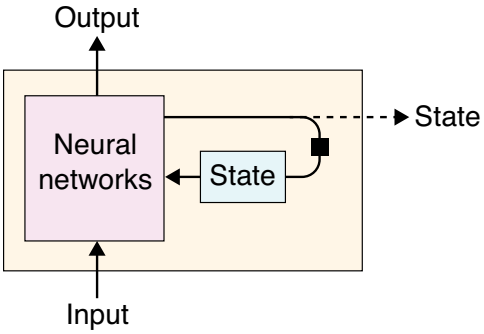


Figure 19-11: A recurrent neural cell. The hidden state can be exported outside of the cell if needed.

The internal state of a recurrent cell is saved as a tensor. Because this tensor is frequently just a one-dimensional list of numbers, we sometimes speak of the *width* or *size* of a recurrent cell, referring to the number of memory elements in the state. If all cells in a network have the same width, we sometimes refer to it as the network's width.

The left side of Figure 19-12 shows our icon for a recurrent cell, which we usually use in unrolled diagrams. The right side shows the icon when we place the cell in a layer, where we roll it up for convenience. In the layer version, we don't draw the cell's internal state.

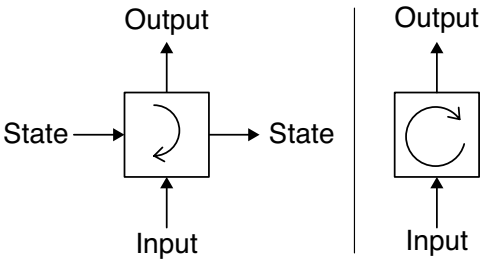


Figure 19-12: Left: Our icon for a recurrent cell. Right: Our icon for a recurrent layer.

We could use the bare-bones recurrent cell in Figure 19-11 to build up a language model. Suppose that the box marked "neural networks" holds a small neural network, built from any layers we like. We could feed the cell sequences of words (in numerical form). After each word, the cell would produce an output predicting the next word to come, and update its internal state to remember the words that have come so far. To replicate our experiment from the start of this chapter, we could feed the cell five words in a row, ignoring the cell's outputs for the first four. Its output after the fifth input would be its prediction for the sixth word. If we're training and the prediction isn't correct, then as usual, we use backpropagation and optimization to improve the values of the weights in the neural networks inside the cell and continue training. The goal is that eventually the networks will become so good at interpreting input and controlling the state that they will be able to make good predictions.

Recurrent Cells in Action

Let's see how a recurrent cell might predict the next word of a five-word sequence. We can see the inputs and possible outputs with an unrolled diagram, shown in Figure 19-13.

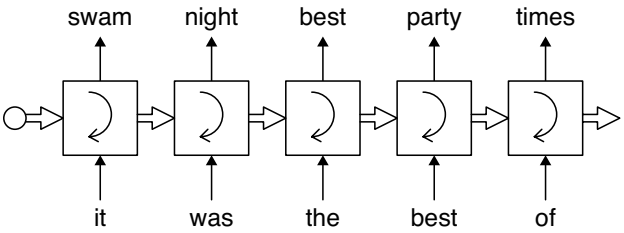


Figure 19-13: A recurrent cell predicting words. The diagram is in unrolled form. Predictions come out of the top of the cell, whereas state is indicated by the open horizontal arrow.

We begin with a cell whose hidden state has been initialized to something generic like all zeros, representing that nothing has been learned yet. That's the open circle at the far left. The first word, it, arrives. The cell considers the input and its hidden state, and predicts the next word, swam. The cell is telling us that the sentence that begins with it is most likely to continue with the word swam, but we ignore this because we only care about the prediction after the fifth word.

Now comes the interesting part. Using the information it learned during training, the RNN updates its hidden state to contain some representation of the fact that it received the word it as input, and produced swam as output.

Now comes the second word from the text, was. Again, the cell consults its hidden state and the input, and produces a new output prediction. Here it's night, completing the phrase it was night. The cell updates its hidden state to remember receiving it and then was and then predicting night. Again, we ignore the prediction of night.

This goes on until we provide the fifth word, of. If we're near the start of training, the system might produce something like jellyfish, completing the sentence it was the best of jellyfish. But after enough training on the original text, the networks inside the recurrent cell will have learned how to represent the consecutive words of the phrase it was the best of inside the hidden state in such a way that the word times has a high probability.

Training a Recurrent Neural Network

Suppose that we're at the start of training the recurrent cell in Figure 19-13. We give it the five words of input, and then compute an error by comparing the cell's final prediction with the next word from the text. If the prediction doesn't match the text, we run backprop and then optimization as usual. Looking at the diagram, we start by finding the gradients in the rightmost cell in the diagram, then we propagate the gradients to the preceding cell

to its left, then propagate the gradients again to the cell preceding that, and so on. It's important to apply backprop in sequence because these are sequential steps of processing.

But we can't really apply optimization to each box in Figure 19-13 because these are all the same cell! To the system, it looks like just one instance of Figure 19-11 sitting on a layer of its own, rather than some unrolled list of repeated uses of the same cell. Somehow we have to apply backprop to the same layer repeatedly, which can create a confusing mess of bookkeeping. To handle this, we use a special variant of backpropagation, called *backpropagation through time*, or *BPTT*. It handles these details so that we can interpret Figure 19-13 literally for the purposes of training.

BPTT allows us to train a recurrent cell efficiently, but it doesn't solve the training problem completely. Suppose that while using BPTT, we compute a gradient for a particular weight in the rightmost cell in Figure 19-13. Then as we propagate the gradient left, we find that the gradient for that same weight in the previous cell is smaller. This means that as we push the gradient to the left, through the same cell over and over, the same process will repeat and the gradient will get smaller and smaller. If the gradient gets 60 percent smaller each time, then after just eight cells, it is down to less than a thousandth of its original size. All it takes for this process to get started is for a gradient to become smaller as we move backward, which is common. Then it inevitably gets smaller by the same percentage on every step backward.

This is very bad news. Recall that when a gradient becomes very small, learning slows down, and if a gradient becomes zero, learning stops entirely. This is not only bad for the recurrent cell, which stops learning, but for neurons on the layers that precede it, because they lose the opportunity to improve, too. The whole learning process can grind to a halt long before we've reached the network's smallest possible error.

This phenomenon is called the *vanishing gradient* problem (Hochreiter et al. 2001; Pascanu, Mikolov, and Bengio 2013). A similar problem comes up if the gradient gets larger every time we step backward through the unrolled diagram. After the same eight steps, a gradient that grows by 60 percent on each step is almost 43 times larger by the time it reaches the first cell. This is called the *exploding gradient* problem (R2RT 2016). These are serious problems that can prevent a network from learning.

Long Short-Term Memory and Gated Recurrent Networks

We can avoid both vanishing and exploding gradients with a fancier recurrent cell, called a *long short-term memory*, or *LSTM*. The name can be confusing, but it refers to the fact that the internal state changes frequently, so it can be considered a short-term memory. But sometimes we can choose to keep some information in the state for a long time. It might make more sense to think of this as a *selectively persistent short-term memory*. A block diagram of an LSTM is shown in Figure 19-14.

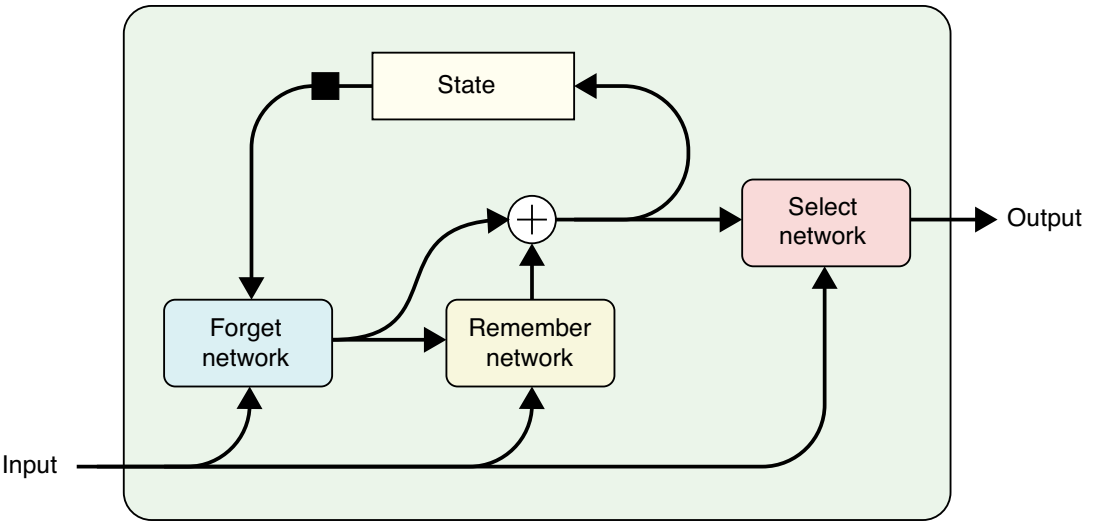


Figure 19-14: A block diagram of a long short-term memory, or LSTM

The LSTM uses three internal neural networks. The first is used to remove (or forget) information from the state that is no longer needed. The second inserts new information the cell wants to remember. The third network presents a version of the internal state as the cell's output.

The convention is that "forgetting" a number simply means moving it toward zero, and remembering a new number means adding it in to the appropriate location in the state memory.

The LSTM doesn't require repeated copies of itself, like the basic recurrent cell of Figure 19-11, so it avoids the problems of vanishing and exploding gradients. We can place this LSTM cell on a layer and train the neural networks inside it using normal backprop and optimization. A practical implementation has many details that we've skipped over here, but they follow this general flow (Hochreiter et al. 2001; Olah 2015).

The LSTM has proven to be such a good way to implement a recurrent cell that when people speak of "an RNN" they often mean a network that uses the LSTM in particular. A popular variation of the LSTM is the *gated recurrent unit*, or *GRU*. It's not uncommon to try out both the LSTM and GRU in a network to see which performs better on a specific task.

Using Recurrent Neural Networks

It's easy to build a network with a recurrent cell (whether it's an LSTM, a GRU, or something else). We just place a recurrent layer in our network and train as usual.

Working with Sunspot Data

Let's demonstrate this with our sunspot data. We'll train a network with a single recurrent layer holding a tiny LSTM with just three values in its

hidden state, as shown in Figure 19-15 (our convention in this book is that a recurrent cell is an LSTM unless stated otherwise). Let's compare this to the output of our old fully connected network of five neurons in Figure 19-1. We have to be careful about comparing apples and oranges, because these approaches are so different, but both networks are about as small as they can be and still do something useful.

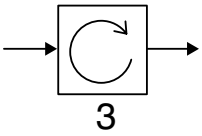


Figure 19-15: A tiny RNN consisting of a single LSTM with three values in its hidden state

Like before, let's train using five sequential values taken from the training data. In contrast to the fully connected layer, which received all five values at once, the RNN gets the values one at a time in five successive steps. The results are shown in Figure 19-16. Keeping in mind our warning about apples and oranges, the results for this little RNN look very much like the results from our fully connected network, shown in Figure 19-5 (the loss values and overall error measured during training were also roughly the same).

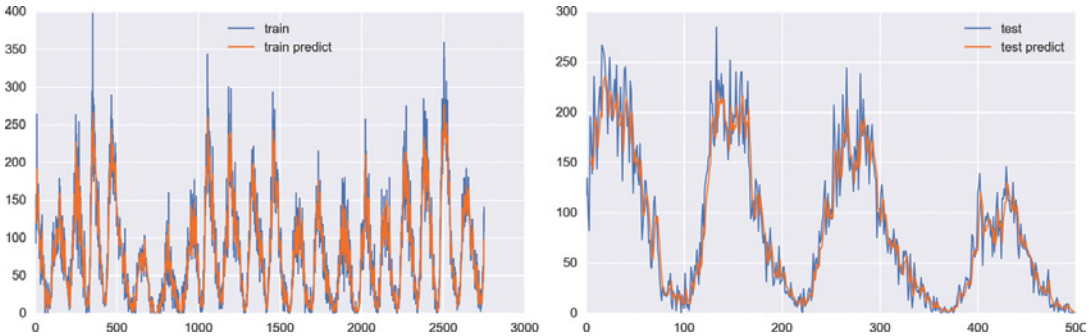


Figure 19-16: Predicting sunspot data with the tiny RNN of Figure 19-15

Generating Text

The last results were encouraging, so let's try the next challenge and use an RNN to generate text. Rather than predict the next word, as we did earlier, for this example let's give our system a sequence of letters and ask it to predict the next letter. As we saw earlier, this is a much easier task, because there are far fewer letters than words. We'll use 89 symbols from the standard English keyboard as our character set. With luck, using characters will let us get away with a smaller network than a word-based approach would require.

Let's train our RNN on sequences of characters taken from the collected short stories of Sherlock Holmes, and ask it to predict the next character (Doyle 1892).

Training an RNN requires tradeoffs. We can use more cells, or more state in each cell, but these all cost time or memory. Larger networks let us work with longer windows, which will probably lead to better predictions.

On the other hand, using fewer, smaller units and smaller windows makes the system faster, so we can run through more training samples in any given span of time. As usual, the best choice for any given system and data requires some experimentation.

After some trial and error, we settled on the network of Figure 19-17. This can surely be improved, but it's small and works well enough for this discussion. Our input window is 40 characters long. Each LSTM cell contains 128 elements of state memory. The final fully connected layer has 89 outputs, one for each possible symbol. The small box after the last fully connected layer is our shorthand in this chapter (and Chapter 20) for a softmax activation function. Thus, the output of this network is a list of 89 probabilities, one for each possible character. We'll choose the most probable character every time.

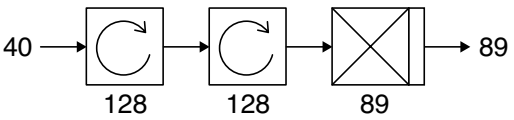


Figure 19-17: A small RNN for processing text one character at a time

To create the training set, we chopped up the original source material into about a half-million overlapping strings of 40 characters, starting every third character.

Once training is done, we can generate new text by autoregression, creating each new 40-character input by adding the last output to the end of the previous input and dropping that previous input's first entry (Chen et al. 2017). We can repeat this as many times as we desire. Figure 19-18 illustrates autoregression for a window of four characters.

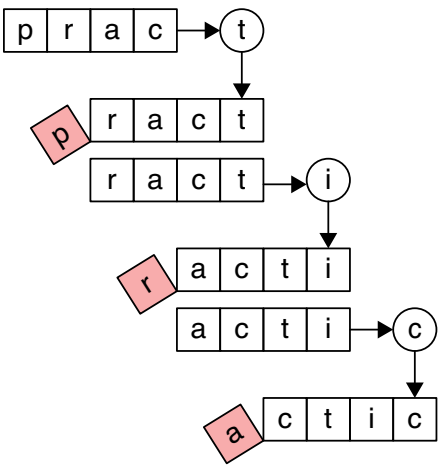


Figure 19-18: Generating text one character at a time with autoregression

To watch the progress of the network, after each epoch of training, we generated some text using the network so far. We started with a seed of 40

sequential characters starting from a random location in the source material. A nice thing about autoregression is that we can run it as long as we like and generated unlimited amounts of output. Here is the beginning of one run after the first epoch of training (the seed is shown in red):

> er price." "If he waits a little longer wew fet ius ofuthe henss lollinod fo snof thasle, anwt wh alm mo gparg lests and and metd tingen, at uf tor alkibto-Panurs the titningly ad saind soot on ourne" Fy til, Min, bals' thid the

In a sense, that's remarkably good. The "words" are about English-sized, and although they're not real words, they could be. That is, they're not strings of random characters. Many of them can even be easily pronounced. And this was after just a single epoch. After 50 epochs, things improved a lot. Here's some output in response to a new random seed.

> nt blood to the face, and no man could hardly question off his pockets of trainer, that name to say, yisligman, and to say I am two out of them, with a second. "I conturred these cause they not you means to know hurried at your little platter.' "'Why shoubing, you shout it of them," Treating, I found this step-was another write so put." "Excellent!" Holmes to be so lad, reached.

Wow. Things are much better. Most of these words are real. The punctuation is great. And even some of the words that aren't in the dictionary, like conturred and shoubing, seem like they could be.

Remember that the system has no knowledge of words at all. It only knows the probabilities of letters following sequences of other letters. For such a simple network, this is remarkable. By letting this run, we can generate as much of this text as we like. It doesn't get any more coherent, but it doesn't get any more incoherent, either.

A larger model with bigger LSTMs, more of them, or both, will give us increasingly credible results at the cost of more training time (Karpathy 2015).

Different Architectures

We can incorporate recurrent cells into other types of networks, extending the capabilities of some types of networks we've already seen. We can also combine multiple recurrent cells to perform sequence operations beyond what any one cell can do. Let's look at a few examples.

CNN-LSTM Networks

We can mix our LSTM cells with a CNN to create a hybrid called a *CNN-LSTM network*. This is great for jobs like classifying video frames. The convolutional layers are responsible for finding and identifying objects, while the recurrent layers that come after are responsible for tracking how the objects move from one frame to the next.

Deep RNNs

Another way to use recurrent cells is to stack up many of them in a row. We call the result a *deep RNN*. We just take the outputs from the cells on one layer and use them as the inputs to the cells on the next

layer. Figure 19-19 shows one way to connect things up for three layers, drawn in both rolled-up and unrolled forms. As usual, the RNN units on each layer have their own internal weights and hidden state.

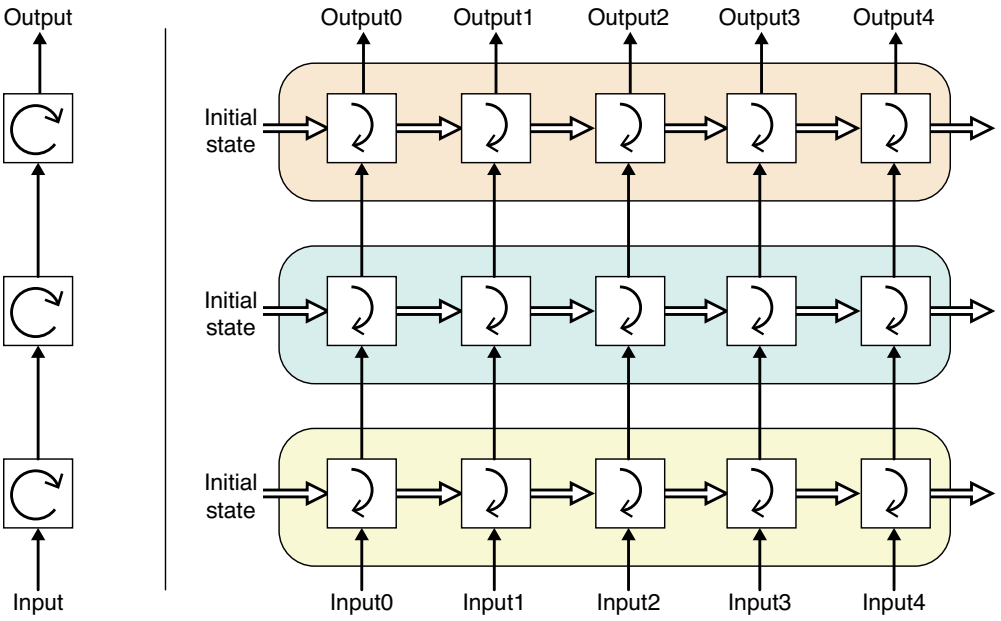


Figure 19-19: A deep RNN. Left: The network using our icons. Right: The layers in unrolled form.

The appeal of this architecture is that each RNN can be specialized for a particular task. For example, in Figure 19-19 the first layer might translate an input sentence into an abstract, common language, the second might rephrase it to change the mood, and then the third could translate that into a different target language. By training each LSTM individually, we gain the advantages of specialization, such as the freedom to update or improve each layer independently of the others. If we replace one LSTM layer with another, we will need to do some extra training on the whole network to make sure the layers work together smoothly.

Bidirectional RNNs

Let's return to translation and consider just how hard the problem is. Take the sentence, "I saw the hot dog train." We can find at least six different ways to interpret this (witnessing an exercise routine by a warm dog, an attractive dog, or a frankfurter, and witnessing a locomotive pulling a chain of each of these three kinds of things). Some interpretations are goofier than the others, but they're all valid. Which one do we choose when we translate?

Another famous sentence is, "I saw the man on the hill in Texas with the telescope at noon on Monday," which has 132 interpretations (Mooney 2019). Aside from the words themselves, delivery also makes a huge difference in meaning. By stressing each word of, "I didn't say he stole the money," we can produce seven completely distinct meanings (Bryant 2019).

Linguistic ambiguity is at the heart of a classic line from Groucho Marx in the film *Animal Crackers*: "One morning I shot an elephant in my pajamas. How he got into my pajamas, I'll never know" (Heerman 1930).

One way to get a handle on all this complexity is to consider multiple words in a sentence as we translate it, rather than each word one at a time. For example, consider these sentences: I cast my fate to the wind, The cast on my arm is heavy, and The cast of the play is all here. These sentences illustrate that the English word cast is a homonym, or a word that can have different meanings. Linguists call this *polysemy*, and it's a feature in many languages (Vicente and Falkum 2017). Our three sentences involving cast translate into Portuguese as, respectively, Eu lancei meu destino ao vento, O gesso no meu braço é pesado, and O elenco da peça está todo aqui. The word cast translates, respectively, to lancei, gesso, and elenco (Google 2020). In these examples, the only way to choose the proper word in Portuguese is to know the words that follow cast in the original sentence, in addition to those that come before.

If we're translating in real time, then we may not know which translation to use based only on the words we've heard so far. In such situations, all we can do is guess, or wait for more words to arrive, and then try to catch up. But if we're working with the whole sentence, such as when we're translating a written book or story, we have all the words already available.

One way to use the later words in the sentence is to feed the words into our RNN backward, such as wind the to fate my cast I. But this doesn't solve the problem in general because sometimes we might need the earlier words, too. What we really want is to have both the preceding and following words available.

We can do this with our existing tools and a bit of cleverness, by creating two independent RNNs. The first gets the words in their forward, or natural order. The second gets the words in their backward order, as shown in Figure 19-20. We call this a *bidirectional RNN*, or a *bi-RNN* (Schuster and Paliwal 1997).

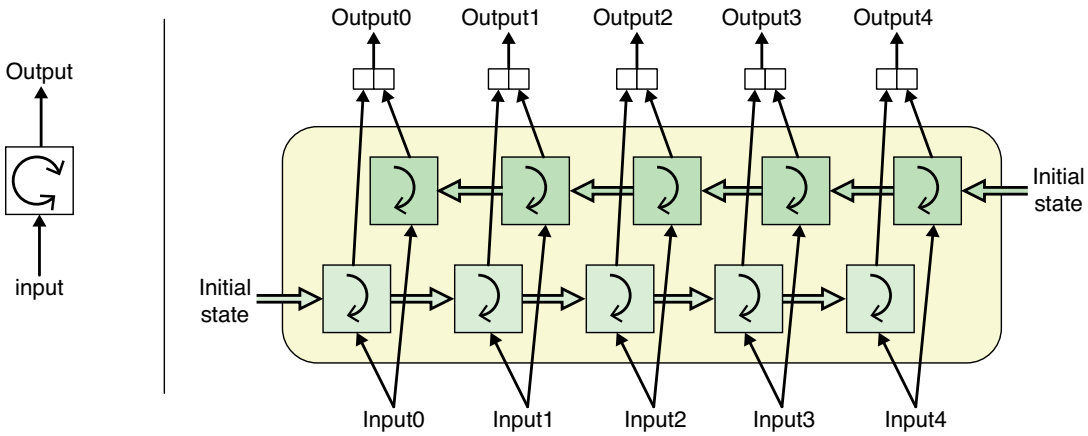


Figure 19-20: A bidirectional RNN, or bi-RNN. Left: Our icon for this layer. Right: An unrolled bi-RNN diagram.

In Figure 19-20 we feed the sentence simultaneously to the lower recurrent cell in forward order and the upper recurrent cell in backward order. That is, we give input 0 to the lower cell at the same time we give input 4 to the upper cell. Then we give input 1 to the lower cell while we give input 3 to the upper cell, and so on. Once all the words have been processed, each recurrent cell will have produced an output for each word. We simply concatenate those outputs and that's the output of the bi-RNN.

We can stack up lots of bi-RNNs to make a *deep bi-RNN*. Figure 19-21 shows such a network with three bi-RNN layers. On the left is our schematic for this layer, and on the right, we draw each layer in its unrolled form. In this diagram, we have three layers, each containing two independent recurrent cells.

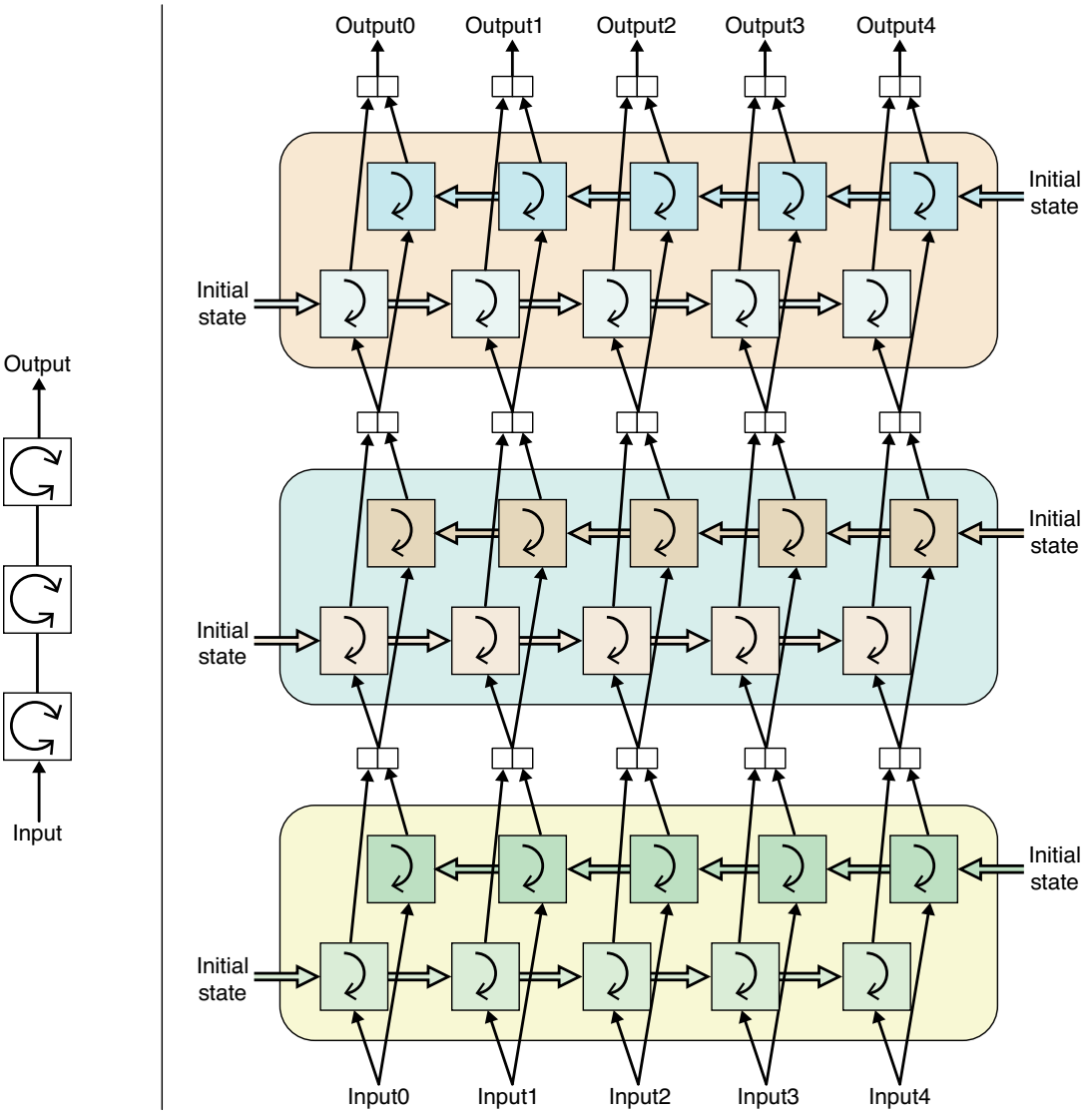


Figure 19-21: A deep bi-RNN. Left: A block diagram using our captions. Right: An unrolled deep bi-RNN.

As before, part of the value here is that each bi-RNN can be independently trained for a different task, and a new bi-RNN can be swapped in if we find (or train) another one that performs better.

Seq2Seq

A challenge for any translation system is that different languages use different word orders. A classic version of this is that in English, adjectives usually precede nouns, while it's not so simple in French. For example, I love my big friendly dog translates to J'adore mon gros chien amical, where chien corresponds to dog, but the adjectives gros and amical, corresponding to big and friendly, surround the noun.

This suggests that instead of translating one word at a time, we should translate entire sentences. This makes even more sense when the input and output sentences have different lengths. Take the five-word English sentence My dog is eating dinner. In Portuguese, this takes only four words: Meu cachorro está jantando, while in Scottish Gaelic it takes six: Tha mo chù ag ithe dinnear (Google 2020).

So rather than work word by word, let's turn a complete sequence into another complete sequence, possibly of a different length. A popular algorithm for converting one entire sequence into another sequence is called *seq2seq* (for "sequence to sequence") (Sutskever, Vinyals, and Le 2014).

The key idea of seq2seq is to use two RNNs, which we treat as an *encoder* and a *decoder*. Let's see how the system works after training is done. We feed our input to the encoder, one word at a time, as usual, but we ignore its outputs. When the whole input has been processed, we take the encoder's final hidden state and hand that to the decoder. The decoder uses the encoder's final hidden state as its own initial hidden state and produces the output sequence using autoregression. Figure 19-22 shows the idea.

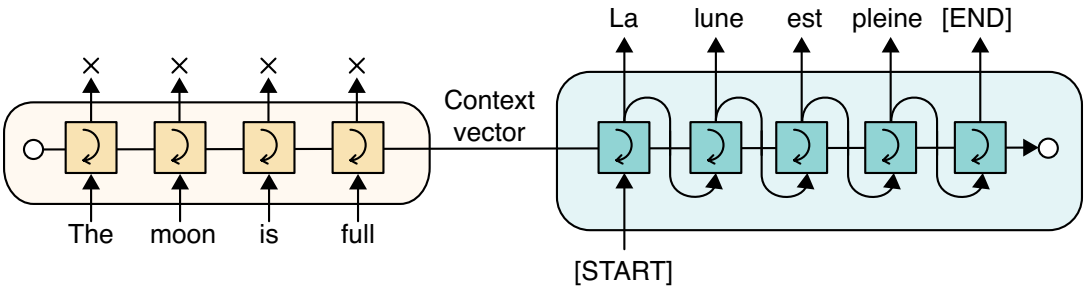


Figure 19-22: The architecture of seq2seq. The encoder, left, processes the input and sends its hidden state to the decoder, right, which produces the output.

In Figure 19-22 we're explicitly showing the autoregression step by feeding the output of each decoder step to the input of the next. If the encoder-decoder architecture looks familiar, it's because it's the same basic structure

as the autoencoders we saw in Chapter 18. In this use, what we previously called the latent vector is now called the *context vector*.

Let's look a little more closely at each of these two RNNs and how they translate a sentence.

The encoder starts with its hidden state set to some initial value, such as all zeros. It consumes the first word, updates its hidden state, and computes an output value. We simply ignore the output value. The only thing we care about is the evolving hidden state inside the encoder.

When the last word has been processed, the hidden state of the encoder is used to initialize the hidden state of the decoder.

Like any RNN, the decoder needs an input. By convention, we give the decoder a special start token. This can be written any way we like so long as it's obviously special and not part of the normal vocabulary of our inputs or outputs. A common convention writes it in all capitals between square or angle brackets, such as `[START]`. Like all the words in our vocabulary, this special token gets its own unique number.

Now that the decoder has an input, it updates its hidden state (initially, the final hidden state from the encoder) and produces an output value. We do pay attention to this output, because it's the first word of our translation.

Now we use autoregression to make the rest of the translation. The decoder takes in the previous output word as input, updates its hidden state, and produces a new output. This continues until the decoder decides that there are no more words to produce. It marks this by producing another special token, such as `[END]`, and stops.

We trained a seq2seq model to translate from English to Dutch (Hughes 2020). Both RNNs had 1,024 elements in their state. The training data consisted of about 50,000 sentences in Dutch, along with their English translations (Kelly 2020). We used about 40,000 sentences for training, and the rest for testing. We trained for ten epochs. In the following two examples, we provide an English sentence, the Dutch translation provided by seq2seq, and the translation of the Dutch back to English provided by Google Translate.

do you know what time it is

weet u hoe laat het is

Do you know what time it is

i like playing the piano

ik speel graag piano

i like to play the piano

Those are pretty great results for such a small network and training set! On the other hand, our small model doesn't degrade too gracefully when the inputs get more complex, as this set of inputs and outputs shows:

John told Sam that his bosses said that if he worked late, they would give him a bonus

hij nodig had hij een nieuw hij te helpen

he needed a new he help

The seq2seq method has much to recommend it. It's conceptually simple, it works well in many situations, and it's easy to implement in modern libraries (Chollet 2017; Robertson 2017). But seq2seq has a built-in limitation in the form of the context vector. This is just the hidden state of the encoder after the last word, so it's of a fixed, finite size. This one vector has to hold everything about the sentence, since it's the only information that the decoder gets.

If we give the encoder a sentence that begins The table has four sturdy, then we can imagine a reasonable amount of memory could retain enough information about each word in the sequence that it could remember we're talking about a table, and the next word should be legs. But no matter how much memory we give to our encoder's hidden state, we can always make a sentence longer than it can remember. For example, suppose our sentence was The table, despite all the long-distance moves, the books dropped onto it, the kids running full-speed into it, serving variously as a fort, a stepladder, and a doorstop, still had four sturdy. The next word should still be legs, but our hidden state would have to become a lot bigger to remember enough information to work that out.

No matter how big our hidden state is, a bigger sentence can always come along and require more memory than we have. This is called the *long-term dependency problem* (Hochreiter et al. 2001; Olah 2015). Figure 19-23 shows an unrolled seq2seq diagram where the input has many words (Karim 2019). A context vector that could remember all of that information would need to be large, with correspondingly large neural networks inside each RNN to manage and control it.

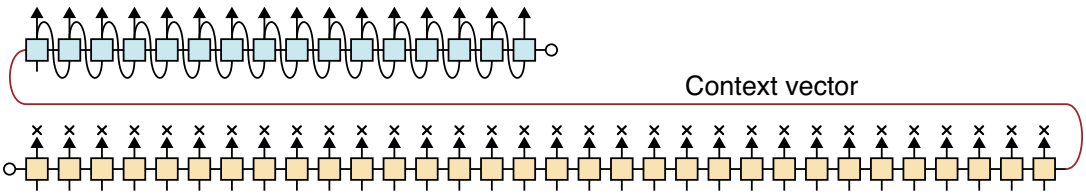


Figure 19-23: Encoding a very long input sentence before sending it a decoder

Maybe depending on a single context vector to represent every useful piece of information in the input isn't the best way to do things. The seq2seq architecture ignores all of the encoder's hidden states except the last. For a long input, those intermediate hidden states can hold information that gets forgotten by the time we reached the end of the sentence.

The dependence on a single context vector plus the need to train one word at a time are big problems for RNN architectures. Though they're useful in many applications, these are serious drawbacks.

Despite these problems, RNNs are a popular way to handle sequences, particularly if they're not too large.

Summary

We've covered a lot in this chapter about processing language and sequences. We saw that we can predict the next element of a sequence with fully connected layers, but they have problems because there's no memory of the inputs. We saw how to use recurrent cells with local, or hidden, memory to maintain a record of everything they've seen in a single context vector that is modified with each input.

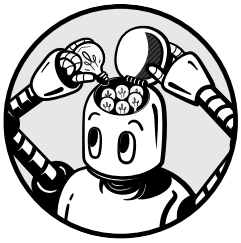
We saw some examples of using RNNs, and then how to use two RNNs to build a translator called seq2seq. Though seq2seq is simple and can do a good job, it has two drawbacks that are common to most RNN systems. For example, the system relies on one context vector to carry all the information about the sentence. Second, the network needs to be trained one word at a time.

Despite these issues, RNNs are a popular and powerful tool for processing sequential data of any kind, from language to seismic data, song lyrics, and medical histories.

In the next chapter, we'll look at another way to handle sequences that avoids the limitations of RNNs.

20

ATTENTION AND TRANSFORMERS



In Chapter 19 we looked at how to use RNNs to handle sequential data. Though powerful, RNNs have a few drawbacks. Because all of the information about an input is represented in a single piece of state memory, or context vector, the networks inside each recurrent cell need to work hard to compress everything that's needed into the available space. And no matter how large we make the state memory, we can always get an input that exceeds what the memory can hold, so something necessarily gets lost.

Another problem is that an RNN must be trained and used one word at a time. This can be a slow way to work, particularly with large databases.

An alternative approach is based on a small network called an *attention network*, which doesn't have a state memory and can be trained and used in parallel. Attention networks can be combined into larger structures called *transformers*, which are capable of serving as language models that can perform tasks like translation. The building blocks of transformers can be used in other architectures that provide even more powerful language models, including generators.

In this chapter, we start with a more powerful way to represent words rather than as single numbers, and then build our way up to attention and modern architectures that use transformer blocks to perform many NLP tasks.

Embedding

In Chapter 19 we promised to improve our word descriptions beyond a single number. The value of this change is that it allows us to manipulate the representations of words in meaningful ways. For example, we can find a word that is like another word, or we can blend two words to find one that's in between them. This concept is key to developing attention, and then transformers.

The technique is called *word embedding* (or *token embedding* when we use it on the more general idea of a token). It's a bit abstract, so let's see the ideas first with a concrete example.

Suppose that you work as an animal wrangler on a movie with a tempestuous director. Today you're filming a sequence where the human heroes are chased by some animals. The director asks you for a list of animals you can provide in sufficient numbers to produce a scary chase. You call your office, they prepare the list, and they even arrange those animals into a chart, where the horizontal axis represents each adult animal's average top speed and the vertical axis represents its average weight, as in Figure 20-1.

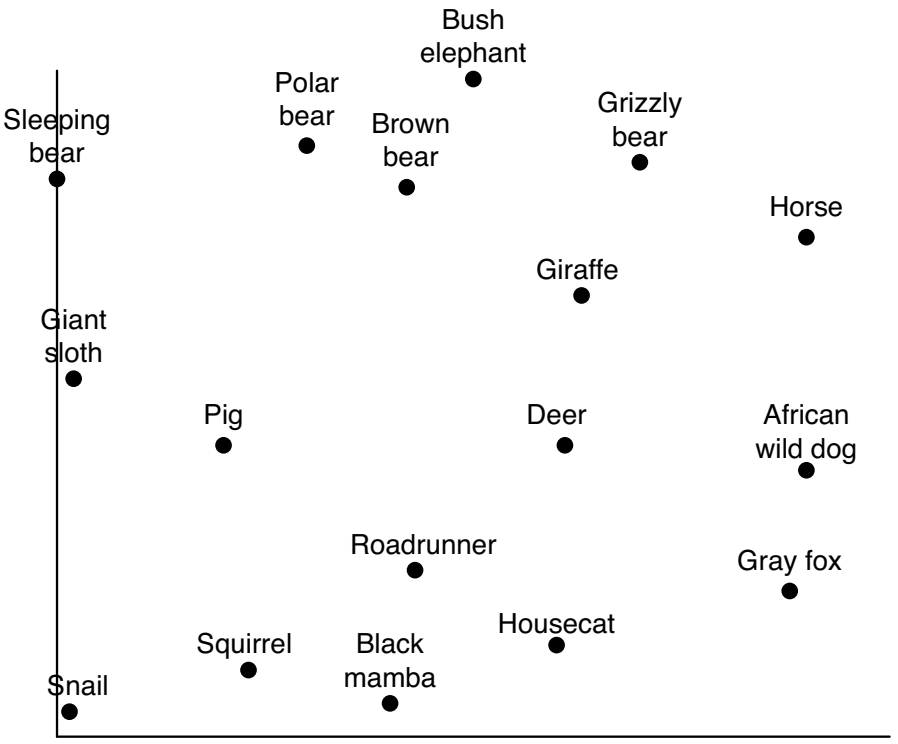


Figure 20-1: A collection of animals, organized roughly by land speed horizontally and adult weight vertically, though those axis labels aren't shown (data from Reisner 2020)

But due to a printer error, the chart your office sent you is missing the labels on the axes, so you have the chart with the animals laid out in 2D, but you don't know what the axes mean.

The director doesn't even look at the chart. "Horses," she says, "I want horses. They're exactly what I want and will be perfect and nothing else will do." So you bring in the horses, and they rehearse the scene.

Unfortunately, the director is unhappy. "No, no, no!" she says. "The horses are too twitchy and quick. They're like foxes. Give me horses that are less fox-like."

How on Earth can you satisfy this request? What does it even mean? Happily, you can do just as she asks with the chart, just by combining arrows.

You only need to do two things with arrows: add them and subtract them. To add arrow B to arrow A, place the tail of B onto the head of A. The new arrow A + B starts at the tail of A, and ends at the head of B, as in the middle of Figure 20-2.

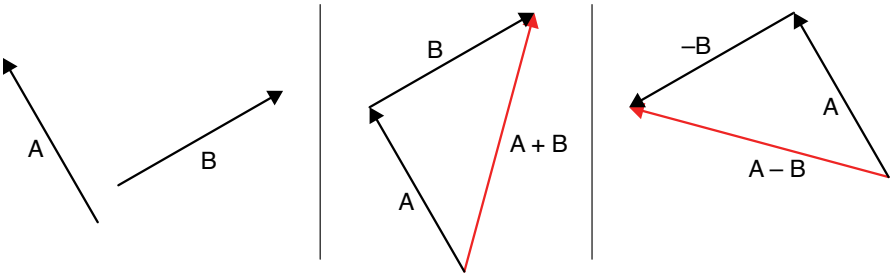


Figure 20-2: Arrow arithmetic. Left: Two arrows. Middle: The sum A + B. Right: The difference A – B.

To subtract B from A, just flip B around by 180 degrees to make –B, and add together A and –B. The result, A – B, starts at the tail of A and ends at the head of –B, as in the right of Figure 20-2.

Now you can satisfy the director's desire to remove the fox qualities from the horses. Start by drawing an arrow from the bottom left of the chart to the horse, and another to the fox, as in the left of Figure 20-3.

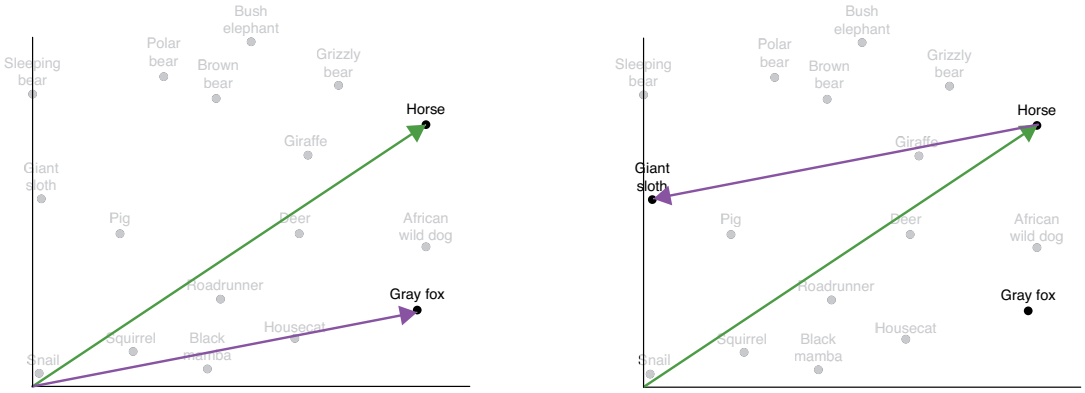


Figure 20-3: Left: Arrows from the bottom left to the horse and fox. Right: Subtracting fox from horse gives us a giant sloth.

Now subtract foxes from horses, as requested, by subtracting the fox arrow from the horse arrow. Following the rules of Figure 20-2, that means flipping the fox arrow around and placing its tail at the head of the horse arrow. We get the right side of Figure 20-3.

A giant sloth. Well, okay, it's what the director wanted. We can even write this like a little bit of arithmetic: horse – fox = giant sloth (at least, according to our diagram).

The director throws her latte on the ground. "No no no! Sure, sloths would look great, but they hardly move! Make them fast! Give me sloths that are like roadrunners!"

Now we know just how to satisfy this ridiculous demand: find the arrow from the bottom left to the roadrunner, as shown in the left of Figure 20-4, and add that to the head of the arrow pointing to the sloth, giving us a brown bear. That is, horse – fox + roadrunner = brown bear, as in the right of Figure 20-4.

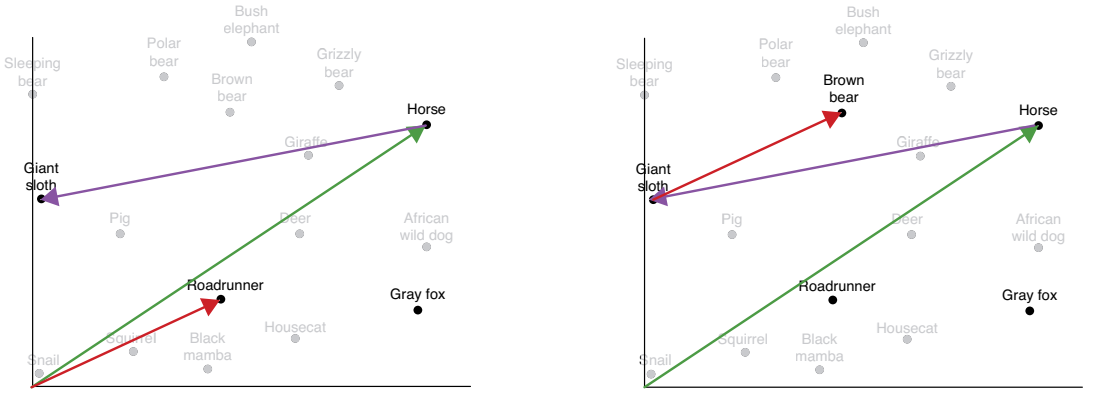


Figure 20-4: Left: We can draw an arrow to the roadrunner. Right: Giant sloth + roadrunner = brown bear.

You offer the director a group of brown bears (called a *sleuth* of bears). The director rolls her eyes dramatically. "Finally. Something that's fast like horses, but not twitchy like foxes, and quick like roadrunners. It's only what I asked for in the first place." They shoot the chase scene with bears, and the movie later comes out to great acclaim.

There are two key elements to this story. The first is that the animals in our chart were laid out in a useful way, even though we didn't know what that way was, or what the axes represented about the data.

The second key point is that we didn't need the axis labels after all. We were able to navigate the chart just by adding and subtracting arrows pointing to elements on the chart itself. That is, we didn't try to find a "slower horse." Rather, we worked strictly with the animals themselves, and their various attributes came along implicitly. Removing the speediness of a fox from a big animal like a horse gave us a big, slow animal.

What does this have to do with processing language?

Embedding Words

To apply what we've just seen to words, we'll replace the animals with words. And instead of using only two axes, we'll place our words in a space of hundreds of dimensions.

We do this with an algorithm that works out what each axis in this space should mean as it places every word at the appropriate point. Instead of assigning each word a single number, the algorithm assigns the word a whole list of numbers, representing its coordinates in a huge space.

This algorithm is called an *embedder*, and we say that this process is one of *embedding* the words in the *embedding space*, thereby creating *word embeddings*.

The embedder works out for itself how to construct the space and find the coordinates of each word so that it's near similar words. For example, if it sees a lot of sentences that begin with I just drank some, then whatever noun comes next is interpreted as some kind of drink, and it is placed near other kinds of drinks. If it sees I just ate a red, then whatever comes next is interpreted as something that's red and edible, and it is placed near other things that are red and near other things that are edible. The same thing is true of dozens or even hundreds of other relationships, both obvious and subtle. Because the space has so many dimensions and the axes can have arbitrarily complex meanings, words can belong simultaneously to many clusters based on seemingly unrelated characteristics.

This idea is both abstract and powerful, so let's illustrate it with some actual examples. We tried a few "word arithmetic" expressions using a pre-trained embedding of 684,754 words saved in a space of 300 dimensions (spaCy authors 2020). Our first test was a famous one: king – man + woman (El Boukkouri 2018). The system returned queen as the most likely result, which makes sense: we can imagine that the embedder worked out some sense of nobility on one axis and gender on another. Other tests were close but not perfect. For example, lemon – yellow + green came back with ginger as the best match, but the expected lime wasn't far back as the fifth-closest word. Similarly, trumpet – valves + slide returned saxophone as the most likely result, but the expected trombone was the first runner-up.

The beauty of training an embedder in a space with hundreds (or even thousands) of dimensions is that it can use the space much more efficiently than any person probably would, enabling it to simultaneously represent an enormous number of relationships.

The word arithmetic we just saw is a fun demonstration of embedding spaces, but it also enables us to meaningfully perform operations on words like comparing them, scaling them, and adding them, all of which are important to the algorithms in this chapter.

Once we have word embeddings, it's easy to incorporate them into almost any network. Instead of assigning a single integer to each word, we assign the word embedding, which is a list of numbers. So instead of processing zero-dimensional tensors (single numbers), the system processes one-dimensional tensors (lists of numbers).

This neatly addresses the problem we saw in Chapter 19 where predictions that were close to the target but not exactly right gave us nonsense. Now we can tolerate a bit of imprecision, because similar words are embedded near one another. For example, we might give our language model the phrase The dragon approached and let out a mighty, expecting the next word to be roar. The algorithm might predict a tensor that's near roar but not exactly on it, giving us bellow or blast instead. We probably wouldn't get back something unrelated, like daffodil.

Figure 20-5 shows six sets of four related words that we gave to a standard word embedder. The more the embeddings of any two words are like one another, the higher that pair of words scored, so the darker their intersection appears. The graph is symmetric around the diagonal from the upper left to the lower right, since the order in which we compare the words doesn't matter.

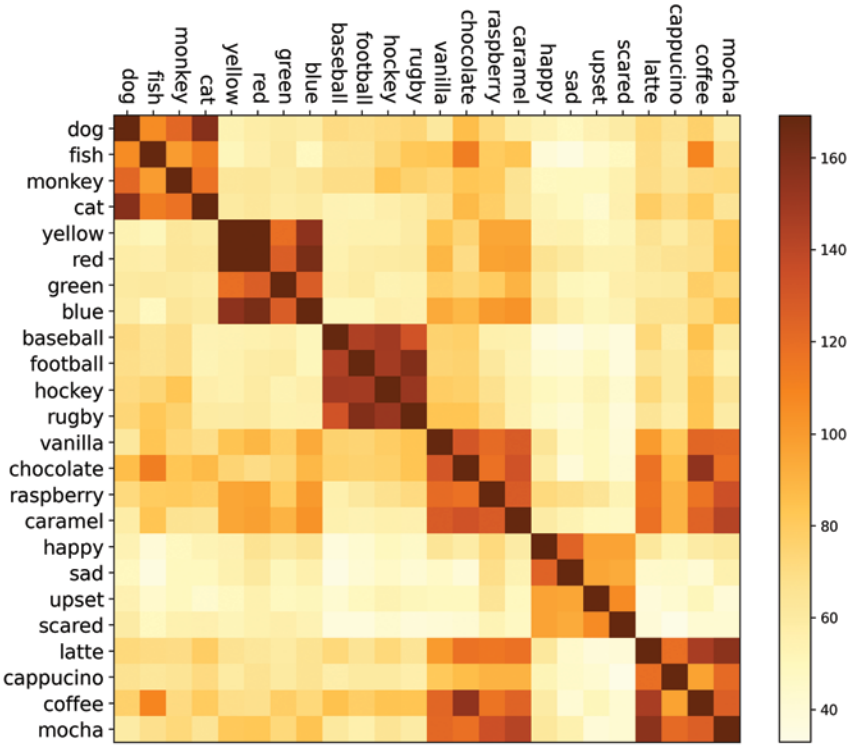


Figure 20-5: Comparing pairs of words by comparing the similarity of their embeddings

We can see from Figure 20-5 that each word matches itself most strongly and also matches related words more strongly than unrelated words. Because we placed related words side by side, the graph shows their similarities as small blocks. There are a few curiosities, however. For example, why does fish match better than average with chocolate and coffee, and why does blue score well with caramel? These might be artifacts from the particular training data used for this embedder.

The coffee drinks and the flavors score well with one another, perhaps because people order coffee drinks with those flavored syrups. There's also a hint of a relationship between the colors and the flavors.

Many pretrained word embedders are widely available for free, and easily downloaded into almost any library. We can simply import them and immediately get the vector for any word. The GLoVe (Mikolov et al. 2013a; Mikolov et al. 2013b) and word2vec (Pennington, Socher, and Manning 2014) embeddings have been used in many projects. The more recent fastText (Facebook Open Source 2020) project offers embeddings in 157 languages.

We can also embed entire sentences, so that we can compare them as a whole, rather than word by word (Cer et al. 2018). Figure 20-6 shows comparisons between embeddings for a dozen sentences (TensorFlow 2018). In this book, we focus on word embeddings rather than sentences.

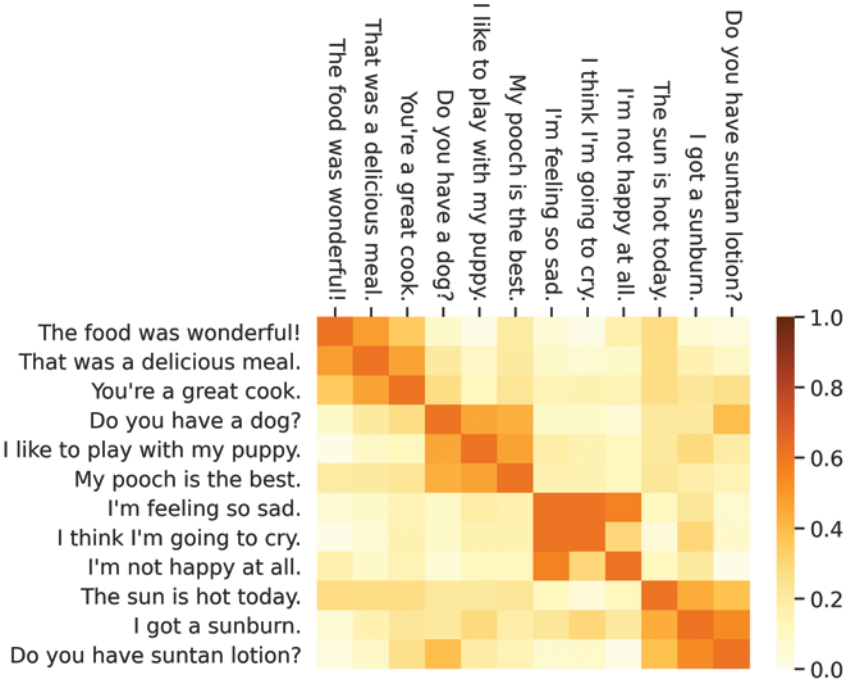


Figure 20-6: Comparing sentence embeddings. The larger the score, the more the sentences are considered like one another.

ELMo

Word embeddings are a huge advance over assigning single integers to words. But even though word embeddings are powerful, the approach we described earlier to create them has a problem: nuance.

As we saw in Chapter 19, many languages have words with different meanings but are written and pronounced the same way. If we want to make sense of words, we need to distinguish these meanings. One way to do that is to give every meaning of a word its own embedding. So cupcake, which has one

meaning, has one embedding. But train has two embeddings, one for when it's a noun (as in, "I rode on a train"), and one for when it's a verb (as in, "I like to train dogs"). These two meanings of train really are entirely different ideas that just happen to use the same sequence of letters.

Such words present two challenges. First, we have to create unique embeddings for each meaning. Second, we have to select the correct embedding when such words are used as input. Solving these challenges requires that we take into account the context of every word. The first algorithm to do this in a big way was called *Embedding from Language Models*, but it's better known by its friendly acronym *ELMo* (Peters et al. 2018), which is the name of a Muppet on the children's television show *Sesame Street*. We say that ELMo produces *contextualized word embeddings*.

ELMo's architecture is similar to that of a pair of bi-RNNs, which we saw in Figure 19-20, but the pieces are organized differently. In a standard bi-RNN, we couple two RNNs running in opposite directions.

ELMo changes this around. Although it uses two RNN networks that run forward and two that run backward, they are grouped by direction. Each of these groups is a two-layer-deep RNN, like the one we saw in Figure 19-21. ELMo's architecture is shown in Figure 20-7. It's traditional to draw ELMo diagrams with a red color scheme, since Elmo on *Sesame Street* is a bright red character.

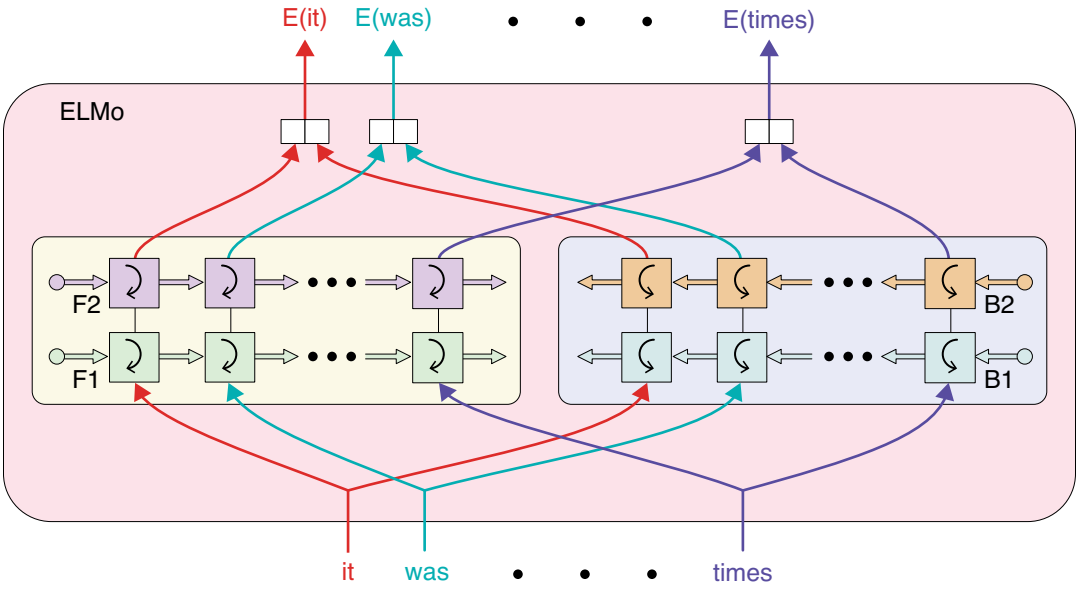


Figure 20-7: The structure of ELMo in unrolled form. The input text is at the bottom. The embedding of each input element is at the top.

This architecture means each input word is turned into two new tensors, one from the forward networks (labeled F1 and F2), that take into account the preceding words, and one from the backward networks (labeled B1 and B2) that consider the following words. By concatenating

these results together, we get contextualized word embeddings informed by all the other words in the sentence.

Trained versions of ELMo are widely available for free downloads in a variety of sizes (Gluon 2020). Once we have a pretrained ELMo, it's easy to use in any language model. We give our entire sentence to ELMo, and we get back a contextualized word embedding for each word, given its context.

Figure 20-8 shows four sentences that use the homonym train as a verb, and four that use train as a noun. We gave these to a standard ELMo model trained on a database of 1 billion words that places each word into a space of 1,024 dimensions (TensorFlow 2020a). We extracted ELMo's embedding of the word train in each sentence, and compared its embedding to that of the word train in all the other sentences. Although the word is written in the identical way in each sentence, ELMo is able to identify the correct embedding based on the word's context.

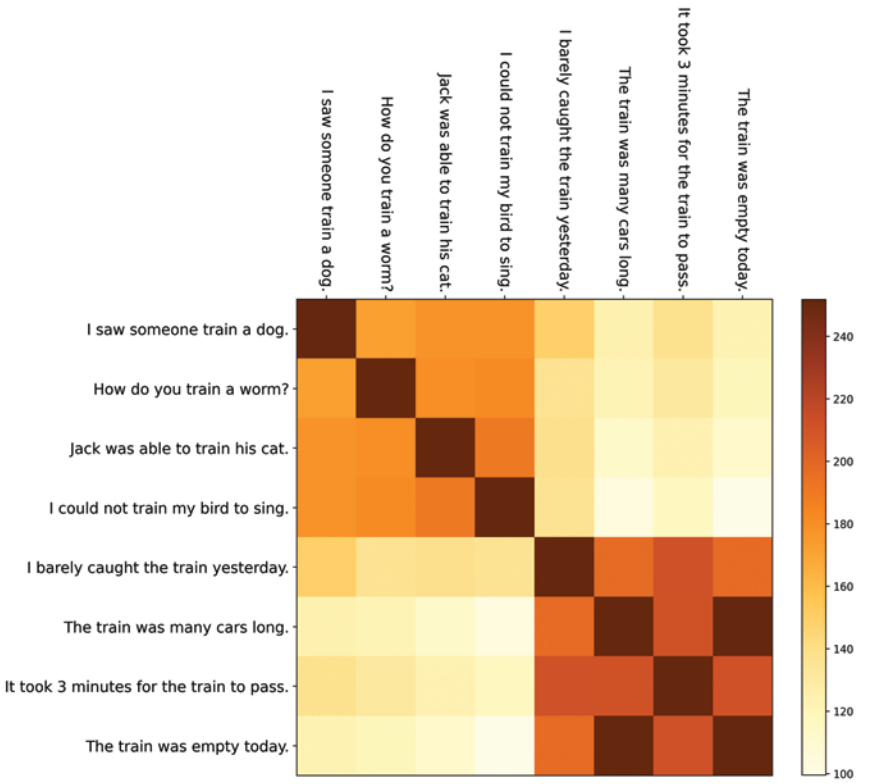


Figure 20-8: Comparing ELMo's embeddings of train *resulting from its use in different sentences. Darker colors mean more similar embeddings.*

We usually place embedding algorithms like ELMo on their own layer in a deep learning system. This is often the very first layer in a language processing network. Our icon for an embedding algorithm, shown in Figure 20-9, is meant to suggest taking the space of words and placing it inside the larger embedding space.

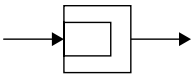


Figure 20-9: Our icon for an embedding layer

ELMo and other algorithms like it, such as the *Universal Language Model Fine-Tuning*, or *ULMFiT* (Howard and Ruder 2018), are typically trained on general-purpose databases, such as books and documents from the web. When we need them for some specific downstream task, such as medical or legal applications, we usually fine-tune them with additional examples from those domains. The result is a set of embeddings that include the specialized language of those fields, clustered by their special meanings in that jargon.

We'll use embeddings in the systems we will build later in this chapter. Those networks will rely on the mechanism of attention, so let's look at that now.

Attention

In Chapter 19 we saw how to improve translation by taking into account all of the words in a sentence. But when we're translating a particular word, not every word in the sentence is equally important, or even relevant.

For example, suppose we're translating the sentence I saw a big dog eat his dinner. When we're translating dog, we probably don't care about the word saw, but to translate the pronoun his correctly may require us to connect that to the two words big dog.

If we can work out, for each word in the input, which other words can influence our translation, then we can focus just on those words and ignore the others. This would be a big savings in both memory and computation time. And if we can work this out in a way that doesn't depend on processing the words serially, we can even do it in parallel.

The algorithm that does this job is called *attention*, or *self-attention* (Bahdanau, Cho, and Bengio 2016; Sutskever, Vinyals, and Le 2014; Cho et al. 2014). Attention lets us focus our resources on only the parts of the input that matter.

Modern versions of attention are often based on a technique called *query, key, value*, or simply *QKV*. These terms come from the field of databases and can seem somewhat obscure in this context. So we'll describe the concepts using a different set of terms and then connect them back to query, key, and value at the end.

A Motivating Analogy

Let's begin with an analogy. Suppose that you need to buy some paint, but all you've been told is that the color should be "light yellow with a bit of dark orange."

At the only paint store in town, the only clerk on duty is new to the paint department and isn't personally familiar with the colors. You both

presume you'll need to mix together a few of their standard paints to get the color you want, but you don't know which paints to choose or how much of each to use.

The clerk suggests that you compare your desired color description with the color names on each can of paint they carry. Some names will probably match better than others. The clerk puts a funnel on top of an empty can and suggests that you pour in some of each can of paint on the shelves, guided by how well that can's name matches your description. That is, you'll compare your desired description "light yellow with a bit of dark orange" with what's printed on the label of each can, and the better the match, the more of that paint you'll pour into the funnel.

Figure 20-10 shows the idea visually for six cans of paint. It shows their names and the quality of each name's match with your desired color's description. We got good matches on "Sunny Yellow" and "Orange Crush," though a little bit of "Lunch with Teal" snuck in thanks to the match with the word "with."

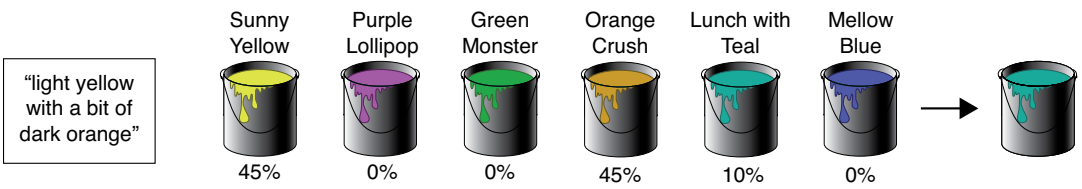


Figure 20-10: Given a color description (left), we combine some of each can based on how well its name matches the description (middle), to get a final result (right).

There are three things to focus on in this story. First, there's your *request*: "light yellow with a bit of dark orange." Second, there's the *description* on each can of paint, like "Sunny Yellow" or "Mellow Blue." Third, there's the *content* of the paint that's actually inside each can. In the story, you compared your request with each can's description to find out how well they match. The better the match, the more of that can's content you used in the final mixture.

That's attention in a nutshell. Given a request, compare it to the description of each possible item and include some of the content of each item based on how well its description matches the request.

The authors of the first paper on attention compared this process to a common type of transaction used with a database. In database language, we look something up by sending a *query* to a database. In such a process, every object in the database has a descriptive *key*, which can be different than the actual *value* of the object. Note that here the word *value* refers to the contents of the object, whether it's a single number or something more complicated, such as a string or tensor.

The database system compares the query (or request) with each key (or description) and uses that score to determine how much of the object's value (or content) to include in the final result. So our terms of request, description, and content correspond to query, key, and value, or, more commonly, QKV.

Self-Attention

Figure 20-11 shows the fundamental operation of attention in abstracted form. Here we have five words of input. Each of the three colored boxes represents a small neural network that takes the numerical representation of a word and transforms it into something new (often, these networks are each just a single fully connected layer). In this example, the word dog is the one we want to translate. So a neural network (in red) transforms the tensor for dog and turns it into a new tensor representing the query, Q. As the figure shows, two more small neural networks translate the tensor for dinner into new tensors, corresponding to its key, K (from the blue network) and its value, V (from the green network).

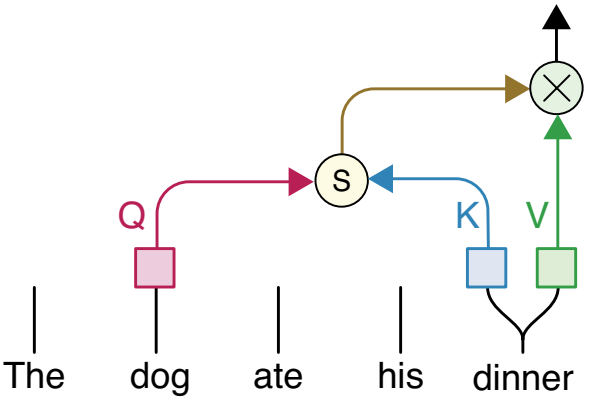


Figure 20-11: The core step of attention using dog *for the query to determine the relevance of the word* dinner. *Each box represents a small neural network that transforms its input into a query, key, or value.*

In practice, we compare the query for dog against the key of every word in the sentence, including dog itself. For this illustration, we limit our focus to the comparison with the word dinner.

We compare the query and the key to determine how alike they are. We do this with a little scoring function that we're indicating with the letter *S* in a circle. Without getting into the math, this function compares two tensors and produces a single number. The more that the two tensors are like one another, the larger that number. The scoring function is usually designed to produce a number between 0 and 1, with larger values indicating a better match.

We use the output from the scoring function to scale the tensor representing the value for dinner. The more the query and the key match, the larger the output of the scaling step, and the more the value of dinner will make it into the output.

Let's see what it looks like when we apply this fundamental step to all the words in the input simultaneously. We'll continue to look at translating the word dog. The overall result is the sum of the individual scaled values of all the input words. Figure 20-12 shows how this looks.

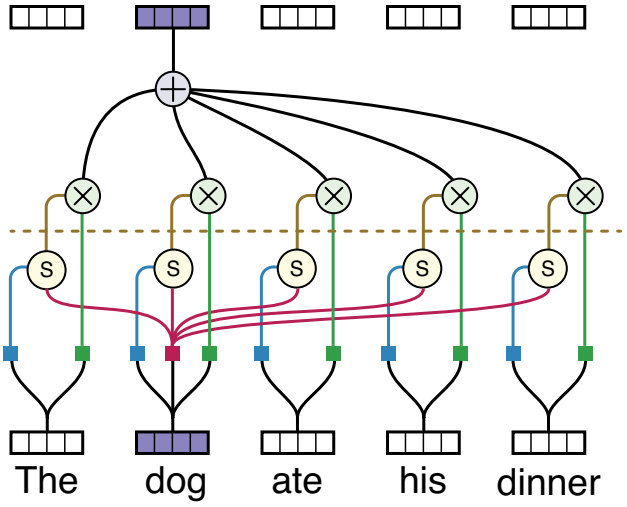


Figure 20-12: Using attention to simultaneously determine the contribution of all five words in the sentence to the word dog. The QKV spatial and color coding matches Figure 20-11. All data flows upward in the figure.

There are a few things to note in Figure 20-12. First, only three neural networks are involved—one each to compute the query, key, and value tensors. We use the same "input to query" network (in red in the figure) to turn each input into its query, the same "input to key" network (in blue in the figure) to turn each input into its key, and the same "input to value" network (in green in the figure) to turn each input into its value. We only need to apply these transformations once to each word.

Second, there's a dashed line after the scores and before the scaling of the values. This represents a softmax step applied to the scores, followed by a division. These two operations keep the numbers coming out of the scores from getting too big or small. The softmax also exaggerates the influence of close matches.

Third, we sum up all the scaled values to get a new tensor for dog, including that from the value of dog itself. We often find that each word scores most highly with itself. This isn't a bad thing, as in this case, the most important word for translating dog is indeed dog itself. But there are times when other words will matter more. Some examples include when word order changes, when a word has no direct translation and must rely on other words, or when we're trying to resolve a pronoun.

The fourth important point is that we apply the processing of Figure 20-12 to all the words in the input sentence simultaneously. That is, each word is considered the query, and the whole process executes independently for that word, as shown in Figure 20-13.

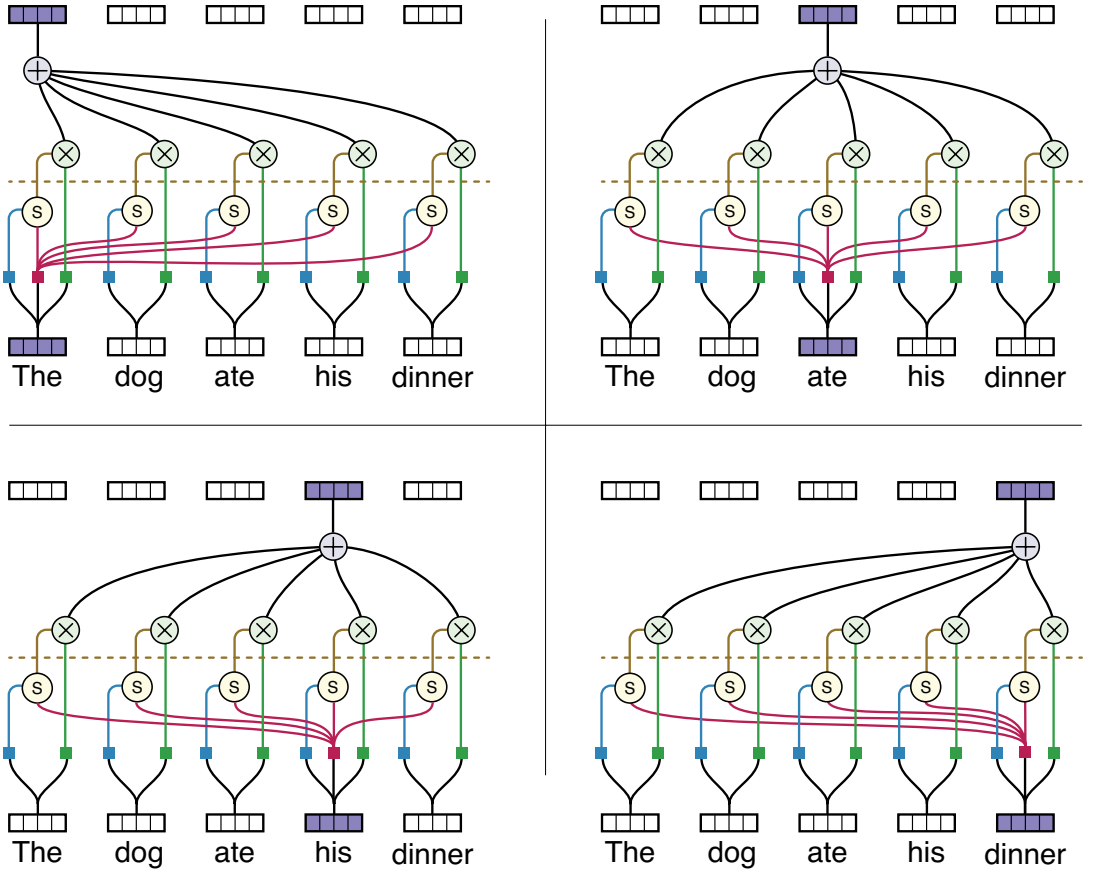


Figure 20-13: Applying attention to the other four words in our sentence

Our fifth and last point is just an explicit recap of something we've been noting all along: all of this processing in Figure 20-12 and Figure 20-13 together can be done in parallel in just four steps, regardless of the length of the sentence. Step 1 transforms the inputs into query, key, and value tensors. Step 2 scores all the queries and keys against one another. Step 3 uses the scores to scale the values, and step 4 adds up the scaled values to produce a new output for each input.

None of these steps depend on how long the input is, so we can process long sentences in the same amount of time required by short ones, as long as we have the memory and computing power needed.

We call the process of Figure 20-12 and Figure 20-13 *self-attention*, because the attention mechanism is using the same set of inputs for computing everything: the queries, keys, and values. That is, we're finding how much the input should be paying attention to itself.

When we place self-attention in a deep network, we put it onto its own *self-attention layer*, often simply called an *attention layer*. The input is a list of words in numerical form, and the output is the same.

The engines that power attention are the scoring function and the neural networks that transform the inputs into queries, keys, and values. Let's consider them briefly.

The scoring function compares a query to a key, returning a value from 0 to 1, where the more the two values are similar, the higher their score. So somehow, the inputs that we think of as being similar need to have similar values going into the scoring function. Now we can see the practical value of embeddings. Recall our discussion of *A Tale of Two Cities*, in Chapter 19 where we assigned each word a number given by its order in the text. That gave the words keep and flint numbers 1,003 and 1,004 respectively. If we just compared these numbers, they would get a high similarity score. For most sentences, this is not what we want. If we're using the query value for the verb keep, we usually want it to be similar to the keys for synonyms like retain, hold, and reserve, and not at all like the keys for unrelated words like flint, preposterous, or dinosaur. Embeddings are the means by which similar words (or words used in similar ways) are given similar representations.

Doing any necessary fine-tuning to the embeddings is the job of the neural networks, which transform the input words into representations where they can be meaningfully compared in the context of the sentence they're used in. The only reason we have any chance of that is that the words are already embedded in a space where similar words are near one another.

Similarly, it's the job of the network that turns inputs into values to represent those values in a way that allow them to be usefully scaled and combined. Mixing two embedded words gives us a word that's somewhere between them.

Q/KV Attention

In the self-attention network of Figure 20-12, the queries, keys, and values are all derived from the same inputs, which led to the name self-attention.

A popular variation uses one source for the queries and another for the keys and values. This more closely matches our paint store analogy, where we came in with the query, and the store had the keys and values. We call this variation a *Q/KV* network, where the slash indicates that the queries come from one source, and the keys and values from another. This version is sometimes used when we add attention to a network like seq2seq, where the queries come from the encoder, and the keys and values from the decoder, so it's sometimes also called an *encoder-decoder attention* layer. The structure is shown in Figure 20-14.

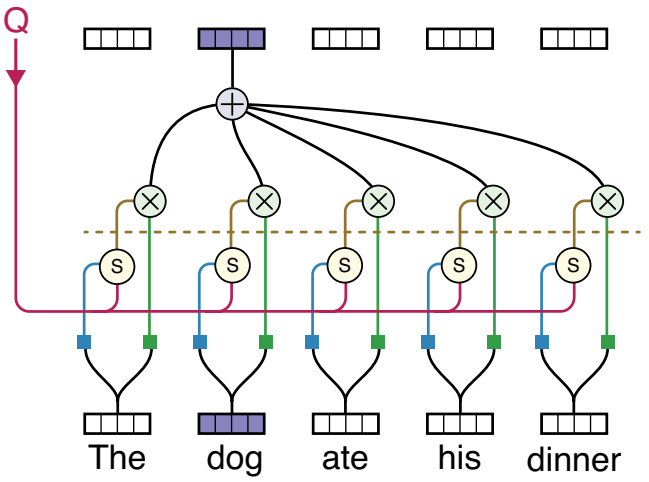


Figure 20-14: A Q/KV layer is like self-attention as shown in Figure 20-12, except that the queries don't come from the inputs.

Multi-Head Attention

The idea of attention is to identify words that are alike and create a useful mix of them. But words can be considered alike based on many different metrics. We might consider nouns to be alike, or colors, or spatial ideas like up and down, or temporal ideas like yesterday and tomorrow. Which of these is the best choice?

Of course, there is no one best answer. In fact, we often want to compare words using multiple criteria at once. For instance, when writing song lyrics, we may want to assign high scores to pairs of words that have similar meanings, similar sounds in their last syllable, the same number of syllables, and the same stress pattern in the syllables. When writing about sports, we might instead want to say that players on the same teams and with the same roles are like one another.

We can score words along multiple criteria by simply running multiple independent attention networks simultaneously. Each network is called a *head*. By initializing each head independently, we hope that during training, each head will learn to compare the inputs according to criteria that are simultaneously useful and different from those used by the other layers. If we want, we can add additional processing to explicitly encourage different heads to attend to different aspects of the inputs. The idea is called *multi-head attention*, and we can apply it to both self-attention networks like Figure 20-12 and Q/KV networks like Figure 20-14.

Each head is a distinct attention network. The more heads we have, the more different aspects of the input they can focus on.

A diagram for a multi-head attention layer is shown in Figure 20-15. As the figure shows, we usually combine the outputs of the heads into a list and

run that through a single fully connected layer. This allows the entire multi-head network's output to have the same shape as its input. This approach makes it easy to place multiple multi-head networks one after another.

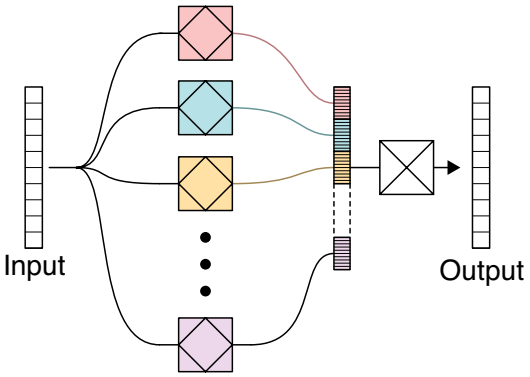


Figure 20-15: A multi-head attention layer. A box with a diamond inside is our icon for an attention layer.

Attention is a general concept that we can apply in different forms to any kind of deep network. For example, in a CNN we can scale a filter's outputs to emphasize the values produced in response to the most relevant locations in the input (Liu et al. 2018; H. Zhang et al. 2019).

Layer Icons

Figure 20-16 shows our icons for the different types of attention layers. Multi-head attention is drawn as a little 3D box, suggesting a stack of attention networks. For Q/KV attention, we place a short line inside the diamond to identify the Q inputs and bring in the K and V inputs on an adjacent side.

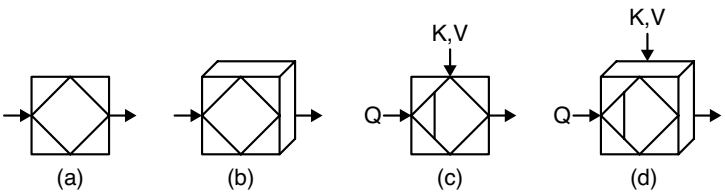


Figure 20-16: Attention layer icons. (a) Self-attention. (b) Multi-head self-attention. (c) Q/KV attention. (d) Multi-head Q/KV attention.

Transformers

Now that we have embedding and attention, we're ready to make good on our earlier promise to improve on RNNs.

Our goal is to build a translator based not on RNNs, but on attention networks. The key idea is that the attention layers will learn how to transform our inputs into their translations, based on the relationships between words.

This approach first appeared in a paper with the great title, "Attention Is All You Need" (Vaswani et al. 2017). The authors called their attention-based model a *transformer* (an unfortunately ambiguous name, but it's now firmly stuck in the language of the field). The transformer model works so well that we now have a new class of language models that not only can be trained in parallel, but also can outperform RNNs in a wide variety of tasks.

Transformers use three more ideas we haven't discussed yet. Let's cover them now, so when we get to the actual transformer architecture, it will be smooth sailing.

Skip Connections

The first new idea we cover is called a *residual connection* or *skip connection* (He et al. 2015). The inspiration is to reduce the amount of work that's required of a deep network layer.

Let's start with an analogy. Suppose you're painting a real, physical portrait using acrylic paints on canvas. After weeks of sittings, the portrait is done, and you send it to your subject for their approval. They say that they like it, but they regret having worn a particular ring on one finger, and wish they'd worn a different one that they like more. Can you change that?

One way to proceed would be to invite your subject back to the studio and paint a whole new portrait from scratch on a blank canvas, only this time with the new ring on their finger. That would require a lot of time and effort. If they'd allow it, a more expeditious approach would be to take the portrait you have, and unobtrusively paint the new ring over the old one.

Now consider a layer in a deep network. A tensor comes in, and the layer does some processing to change that tensor. If the layer only needs to change the input by small amounts, or only in some places, then it would be wasteful to expend resources processing the parts of the tensor that don't need to change. Just as with the painting, it would be much more efficient for the layer to compute only the changes it wants to make. Then it can combine those changes with the original input to produce its output.

This idea works beautifully in deep learning networks. It lets us make layers that are smaller and faster, and it even improves the flow of gradients in backpropagation, which lets us efficiently train networks of dozens or even hundreds of layers.

The mechanism is shown on the left side of Figure 20-17. We feed an input tensor to some layer as usual, let it compute the changes, and then we add the layer's output to its input tensor.

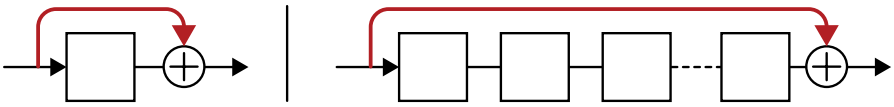


Figure 20-17: Left: A skip connection, shown in red. Right: We can place a skip connection around multiple layers.

The extra line in the drawing that carries the input to the addition node is called a *skip connection*, or a *residual connection* because of its mathematical interpretation.

We can place a skip connection around multiple layers in sequence, if we like, as on the right of Figure 20-17.

The skip connection works because each layer is trying to reduce its own contribution to the final error, while participating in the network made up of all the other layers. The skip connection is part of the network, so the layer learns it doesn't need to process the parts of the tensor that don't change. This makes the layer's job simpler, enabling it to be smaller and faster.

We'll see later that transformers use skip connections not just for efficiency and speed, but also because they allow the transformer to cleverly keep track of the location of each element in its input.

Norm-Add

The second idea on our road to transformers is really more of a conceptual and notational shorthand. In transformers, we usually apply a regularization step called *layer normalization*, or *layer norm*, to the outputs of a layer, as shown on the left in Figure 20-18 (Vaswani et al. 2017). Layer norm belongs to the class of regularization techniques that we saw in Chapter 15, such as dropout and batchnorm, which help control overfitting by keeping the values flowing through the network from getting too big or too small. The layer norm step learns to adjust the values coming out of a layer so that they approximate the shape of a Gaussian bump with a mean of 0 and standard deviation of 1.

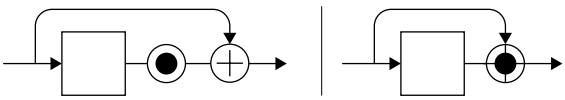


Figure 20-18: Left: A layer normalization followed by the addition step of a skip connection. Right: A combined icon for norm-add. This is just a visual and conceptual shorthand for the network on the left.

Performing layer norms is important in getting a transformer to work well, but there's some flexibility about exactly where this step can be located. A popular approach places the layer norm just before the addition step of a skip connection, as in the left side of Figure 20-18. Since these two operations always come in pairs, it's convenient to combine them into a single operation that we call *norm-add*. Our icon for norm-add is a combination of the layer norm and summation icons, and is shown on the right in Figure 20-18. This is just a visual shorthand for the two separate steps of layer norm followed by skip connection addition.

People have experimented with other locations for the layer norm operation, such as before the layer (Vaswani et al. 2017), or after the addition node (TensorFlow 2020b). These approaches differ in their details, but in

practice, it seems that all of these choices are comparable. We'll stick with the version in Figure 20-18 here.

Positional Encoding

The third idea to cover before we get to transformers was designed to solve a problem that comes up as soon as we take RNNs out of our system: we lose track of where each word is located in the input sentence. This important information is inherent in the RNN structure, because the words come in one at a time, allowing the hidden state inside a recurrent cell to remember the order in which the words arrived.

But as we've seen, attention mixes together the representations of multiple words. How can later stages know where each word belongs in the sentence?

The answer is to insert each word's position, or index, into the representation for the word itself. That way, as the word's representations get processed, the position information naturally comes along for the ride. The generic name for this process is *positional encoding*.

A simple approach to positional encoding is to append a few bits to the end of each word to hold its location, as shown on the left of Figure 20-19. But at some point, we might get a sentence that requires more bits than we've made available, and then we'd be in trouble because we wouldn't be able to assign each word a unique number for its location. And if we make the storage too big, it's just wasted and slows everything down. This approach is also awkward to implement, since we then need to introduce some special mechanism for handling those bits (Thiruvengadam 2018).

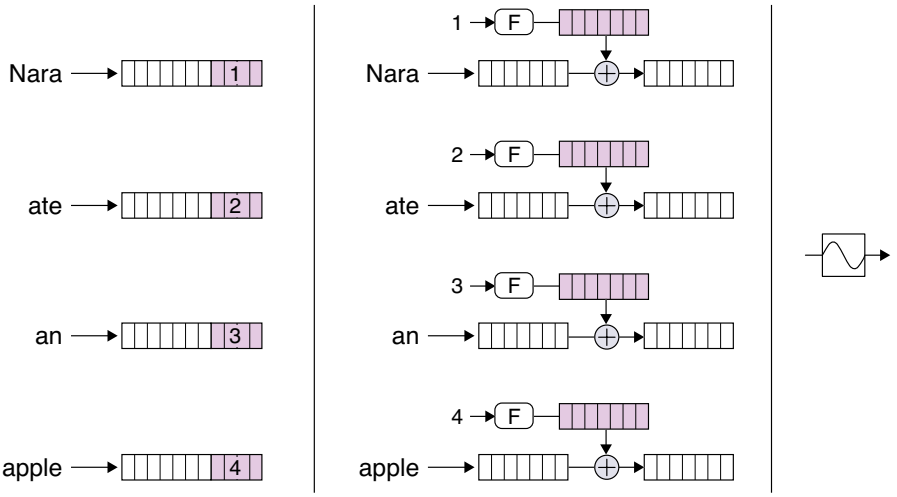


Figure 20-19: Tracking the location of each word in a sentence. Left: Appending an index to each word. Middle: Using a function F to turn each index into a vector, then adding it to the word's representation. Right: Our icon for a positional embedding layer.

A better answer is to use a mathematical function that creates a unique vector for each position in a sequence. Suppose that our word embeddings are 128 elements long. Then we give this function the index of each word

(which can be as large as it needs to be), and the function gives us back a new 128-element vector that somehow describes that location. Basically it turns the index into a unique list of values. Our expectation is that the network will learn to associate each of these lists with the word's position in the input.

Rather than appending this vector to the word's representation, we add the two vectors together, as in the middle of Figure 20-19. Here we literally add the number in each element in the encoding to the corresponding number in the word's embedding. The appeal of this approach is that we don't need any extra bits or special processing. This form of positional encoding is called *positional embedding*, because of its similarity to the *word embedding* we saw earlier in algorithms like ELMo. The right side of the figure shows our icon for this process, which is drawn with a little sine wave because a popular choice for the embedding function is based on sine waves (derived from Vaswani et al. 2017).

It may seem a bit weird to add position information to each word, rather than append it, as it changes the word's representation. It also seems that the position information is liable to get lost as the attention network processes the values.

It turns out that the specific function that is frequently used to compute the positional embedding vector usually affects only a few bits at one end of the word's vector (Vaswani et al. 2017; Kazemnejad 2019). Furthermore, it appears that transformers learn how to distinguish each word's representation and position information during processing so they're interpreted separately (TensorFlow 2019a).

But why doesn't the position embedding get lost altogether during processing? After all, attention changes its inputs using neural networks by turning them into QKV values and then mixing those values. Surely the positional information would be hopelessly scrambled and lost.

The clever solution to this problem is built into the architecture of the transformer itself. As we'll see, the transformer network wraps up each operation (except the very last) in a skip connection. The embedding information never gets lost, because it gets added back in after every stage of processing. Figure 20-20 illustrates how positional embedding and norm-add skip connections are structurally similar. In short, each layer can change its input vector in any way it wants, and then the positional embedding gets added back in so that it's available to the next layer.

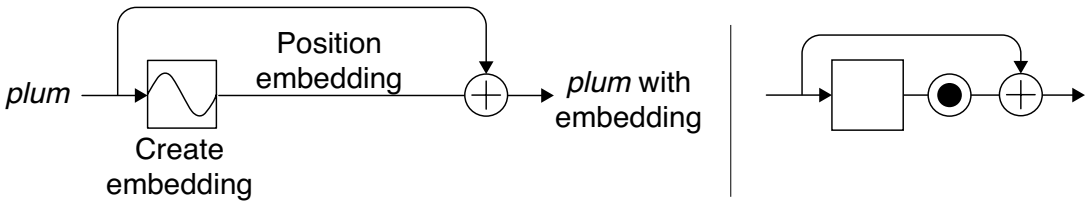


Figure 20-20: Left: Creating a position embedding and adding it to a word. Right: A norm-add operation implicitly adds a word's embedding information back in after processing.

Assembling a Transformer

We now have all the pieces in place to build a transformer. We'll continue to use word-level translation as our running example.

It's important to note that the name *transformer* refers to a wide variety of networks inspired by the architecture in the original transformer paper (Vaswani et al. 2017). In this discussion, we'll stick to a generic version.

Our block diagram of a transformer is shown in Figure 20-21. The blocks marked *E* and *D* are repeated sequences of layers, or *blocks*, built around attention layers. We'll look at both types of block in detail in a moment. The big picture is that an encoder stage (built from *encoder blocks*, marked with an *E*) accepts a sentence, and a decoder (build from *decoder blocks*, marked with a *D*) accepts information from the encoder and produces new output (the structure of this diagram is reminiscent in some ways of an unrolled seq2seq diagram, but there are no recurrent cells here).

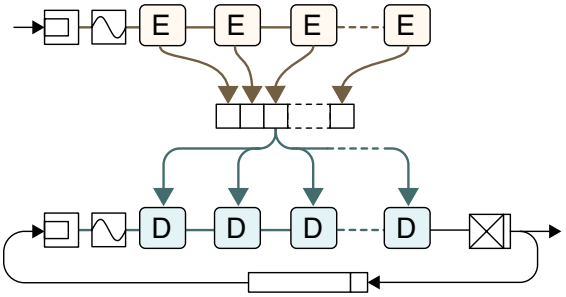


Figure 20-21: A block diagram of a transformer. An input is encoded and then decoded. The decoder's output is fed back to its input autoregressively. The dashed lines stand for repeated elements.

Both the encoder and decoder begin with word embedding followed by positional embedding. The decoder has the usual fully connected layer and softmax at the end for predicting the next word. The decoder is autoregressive, so it appends each output word to the list of its outputs (shown by the box at the bottom of the figure), and that list becomes the decoder's input for generating the next word. The decoder contains multi-head Q/KV attention networks, as in Figure 20-14, which receive their keys and values from the outputs of the encoder blocks, shown in the middle of Figure 20-21, where the encoder outputs are delivered to the decoder blocks. This illustrates why Q/KV attention is also called encoder-decoder attention.

Let's look more closely at the blocks in Figure 20-21 starting with the encoder block, shown in Figure 20-22.

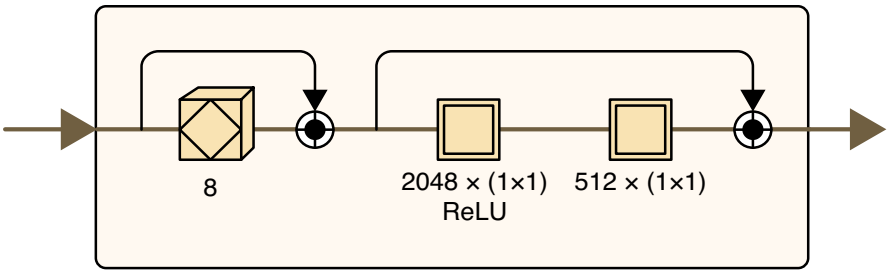


Figure 20-22: The transformer's encoder block. The first layer is self-attention.

The encoder block begins with a layer of multi-head self-attention, shown here with eight heads. Because this layer applies self-attention, the queries, keys, and values are all derived from the single set of inputs that arrive at the block. This multi-head attention is surrounded by a norm-add skip connection to help keep the numbers looking like a Gaussian and to retain the positional embeddings.

This is followed by two layers that are usually referred to collectively as a *pointwise feed-forward layer* (another unfortunately vague name). Though the original transformers paper described these as a pair of modified fully connected layers (Vaswani et al. 2017), we can more conveniently think of them as two layers of 1×1 convolution (Chromiak 2017; Singhal 2020; A. Zhang et al. 2020). They learn how to adjust the output of the multi-head attention layer to remove redundancy and focus on just the information that will be of the most value to whatever processing comes next. The first convolution uses a ReLU activation function, while the second has no activation function. As usual, these two steps are wrapped in a norm-add skip connection.

Now let's look at the decoder block, shown in Figure 20-23.

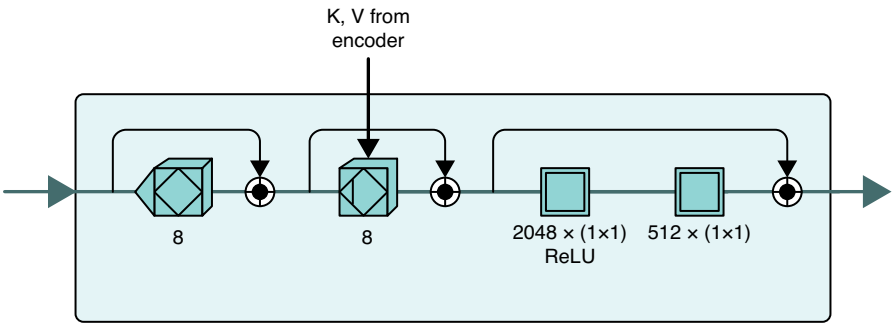


Figure 20-23: The transformer's decoder block. Note that the first attention layer is self-attention, whereas the second is Q/KV attention. The triangle on the left of the self-attention layer indicates that the layer uses masking.

At a high level, it looks a lot like the encoder block, with an extra step of attention. Let's walk through the layers.

We begin with a multi-head self-attention layer, just like the encoder block. The input to the layer is the words output so far by the transformer. If we're just beginning, this sentence contains only the `[START]` token. Like any self-attention layer, the purpose here is to look at all of the input words and work out which ones are most strongly related to which others. As usual, this is wrapped in a skip connection with a norm-add node at the end. During training, we add an extra detail called *masking* to this self-attention step (indicated with a small triangle in Figure 20-23), which we'll come back to shortly.

The self-attention layer is followed by a multi-head Q/KV attention layer. The query, or Q, vectors come from the output of the previous self-attention layer. The keys and values come from the concatenated outputs of all the encoder blocks. This layer also is wrapped in a skip connection with a norm-add node at the end. This stage uses the outputs of the previous attention network to choose among the keys coming from the encoder and then mix the values corresponding to those keys. Finally, we have a pair of 1×1 convolutions, following the same pattern as in the encoder block.

We can now put the pieces together. Figure 20-24 shows the structure of a transformer model.

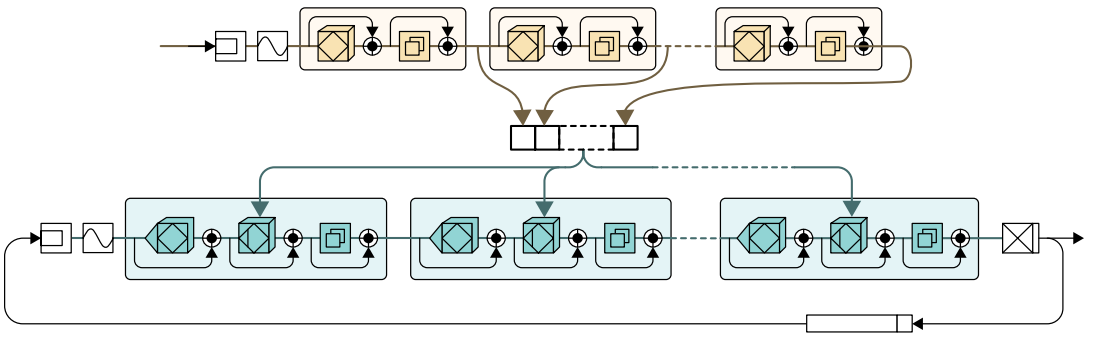


Figure 20-24: A complete transformer. The icons showing two stacked boxes represent two consecutive 1×1 convolutions. The dashed lines stand for repeated elements that are not drawn.

We promised to return to a detail regarding the first attention layer in each decoder block. As we mentioned, one of the great values of the attention mechanism at the heart of the transformer is that it allows for a lot of parallelism. Whether an attention block is given five words or five hundred, it runs in the same amount of time.

Suppose we're training the system to predict the next word in a sentence. We can provide it with the entire sentence and ask it to predict the first word, the second word, the third word, and so on, all in parallel.

But there's a problem here. Suppose the sentence is My dog loves taking long walks. We could give the system My dog loves taking long, and ask it to predict the sixth word, walks. But because we're training in parallel, we want it to use this same input to predict each of the previous words, at the same time. That is, we also want it to predict the fifth word, long, from the input My dog loves taking long.

That's too easy: the word long is right there! The system would find that all it has to do is return the fifth word, which is definitely not the same as learning how to predict it. We want to give the system My dog loves taking long as input, but for predicting the fifth word, it should only see My dog loves taking. We want to hide, or mask, the word long when we're trying to predict it. Similarly, to predict the fourth word, it should only see My dog loves, to predict the third word it should only see My dog, and so on.

In short, our transformer will run five parallel computations, each predicting a different word, but each computation should only be given the words that came before the one it's supposed to predict.

The mechanism to pull this off is called *masking*. We add an extra step to the first self-attention layer in the decoder block that masks, or hides, the words that each prediction step isn't supposed to see. Thus the computation predicting the first word sees no input words, the computation predicting the second word only sees My, the one predicting the third word only sees My dog, and so on. Because of this extra step, the first attention layer in the decoder block is sometimes called a *masked multi-head self-attention* layer, which is a mouthful, so we often just refer to it as a *masked attention* layer.

Transformers in Action

Let's see a transformer in action performing a translation. We trained a transformer following roughly the architecture of Figure 20-24 to translate from Portuguese to English (TensorFlow 2019b). We used a dataset of 50,000 training examples, which is small by today's standards but good enough to demonstrate the ideas while also of a practical size to train from on a home computer (Kelly 2020).

We gave our trained transformer the Portuguese question, você se sente da mesma maneira que eu? which Google Translate renders into English as do you feel the same that way I do? Our system produced the translation, do you see , do you get the same way i do ? This isn't perfect, but given the small training database, it does a great job of capturing the spirit of the question. As always, more training data and training time would surely improve the results.

Heatmaps showing the attention paid to each input word by each output word, for each of the eight heads in the final Q/KV attention layer of the decoder, are shown in Figure 20-25. The brighter the cell, the more attention was paid. Note that some input words were broken up into multiple tokens by a preprocessor.

Transformers trained on larger datasets than this example, and for longer periods, can produce results that are as good or better than RNNs, and they can be trained in parallel. They don't need recurrent cells with finite internal states that can run out of memory, nor do they need multiple neural networks to learn how to control those states. These are big advantages and explain why transformers have replaced RNNs in many applications.

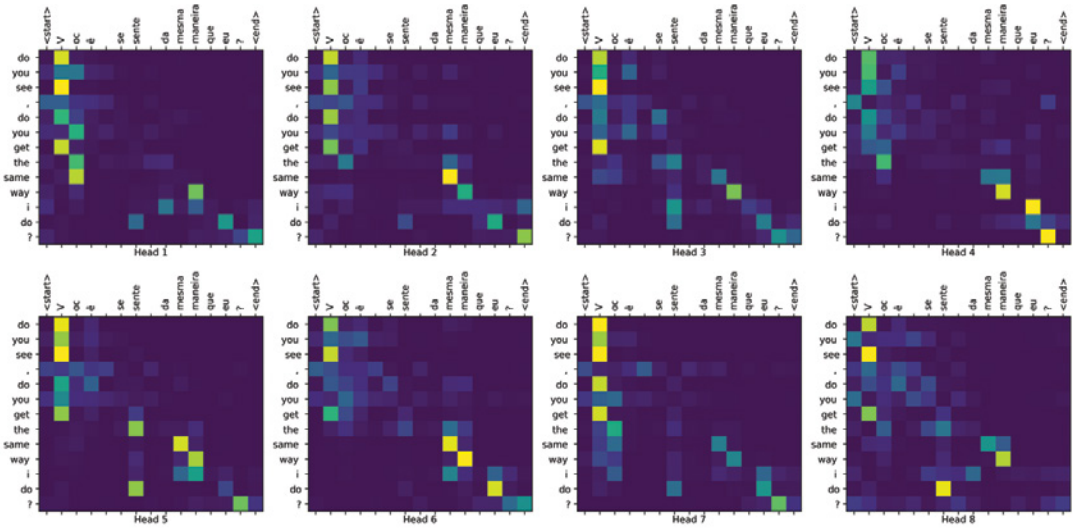


*Figure 20-25: Heatmaps for each of the eight heads in the final Q/KV attention layer of the decoder during a translation of "*Você se sente da mesma maneira que eu?*" from Portuguese to English*

One downside of transformers is that the memory required by the attention layers grows dramatically with the size of the input. There are ways to adjust the attention mechanism, and the transformer in general, to reduce these costs in different situations (Tay et al. 2020).

BERT and GPT-2

The full transformer model of Figure 20-24 consists of an encoder, which is designed to analyze the input text and create a series of context vectors that describe it, and a decoder, which uses that information to autoregressively generate a translation of the input.

The blocks making up the encoder and decoder are not specific to translation. Each is just one or more attention layers, followed by a pair of 1×1 convolutions. These blocks can be used as general-purpose processors for working out the relationship between elements of a sequence, and language in particular. Let's look at two recent architectures that have used transformer blocks in ways that go way beyond translation.

BERT

Let's use transformer blocks to create a general-purpose language model. It can be used for any of the tasks we listed at the start of Chapter 19.

The system is called *Bidirectional Encoder Representations from Transformers*, but it's more commonly known by its acronym, *BERT* (Devlin et al. 2019) (another Muppet from *Sesame Street*, and a nodding reference to the ELMo system we saw earlier). The structure of BERT begins with a word embedder and a position embedder, followed by multiple transformer encoder blocks.

The basic architecture is shown in in Figure 20-26 (in practice, other details help with training and performance, such as dropout layers). In this diagram, we're showing the many inputs and outputs so that it's clear that BERT is processing an entire sentence. For consistency and clarity, we're only using a single line inside the blocks, but the parallel operations are still being carried out. It's traditional to draw BERT diagrams with a yellow color scheme, since Bert on *Sesame Street* is a yellow character.

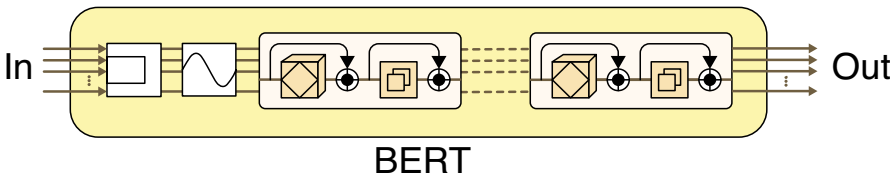


Figure 20-26: The basic structure of BERT. The dashed lines stand for more encoder blocks that are not drawn.

The original "large" version of BERT deserved its name, with 340 million weights, or parameters. The system was trained on Wikipedia and over 10,000 books (Zhu et al. 2015). Currently, 24 trained versions of the original BERT system are available freely online (Devlin et al. 2020), as well as a growing number of variations and improvements on the basic approach (Rajasekharan 2019).

BERT was trained on two tasks. The first is called *next sentence prediction*, or *NSP*. In this technique, we give BERT two sentences at once (with a special token to separate them), and we ask it to determine if the second sentence reasonably follows the first. The second task presents the system with sentences where some of the words have been removed, and we ask it to fill in the blanks (language educators call this the *cloze task;* Taylor 1953). It's the linguistic analog of the visual process called *closure*, describing the human tendency to fill in the blanks in images. Closure is illustrated in Figure 20-27.

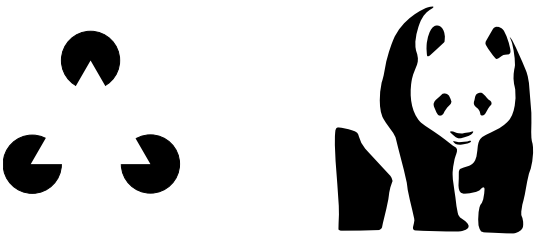


Figure 20-27: Demonstrating the principle of closure. Incomplete shapes like these are usually filled in by the human visual system to create objects.

BERT is able to do well on these tasks because, compared to the RNN-based methods we saw before, BERT's attention layers extract much more information from their inputs. Our first RNN models were *unidirectional*, reading inputs left to right. Then they became *bidirectional*, culminating in ELMo, which can be said to be *shallowly bidirectional*, where *shallow* refers to

the architecture's use of only two layers in each direction. Thanks to attention, BERT is able to determine the influence of every word on every other word, and by repeating the encoder block, it can do this many times in a row. BERT is sometimes called *deeply bidirectional*, but it might be more useful to think of it as *deeply dense*, since it considers every word simultaneously. The notion of direction really doesn't apply when we're using attention.

Let's take BERT out for a spin. We'll start with a pretrained model of 12 encoder blocks (McCormick and Ryan 2020). We'll fine-tune it to determine if an input sentence is grammatical or not (Warstadt, Singh, and Bowman 2018; Warstadt, Singh, and Bowman 2019). This is basically a classification problem, producing a yes/no answer. Therefore, our downstream model should be a classifier of some kind. Let's use a simple classifier consisting of a single fully connected layer. Our combined pair of models is shown in Figure 20-28.

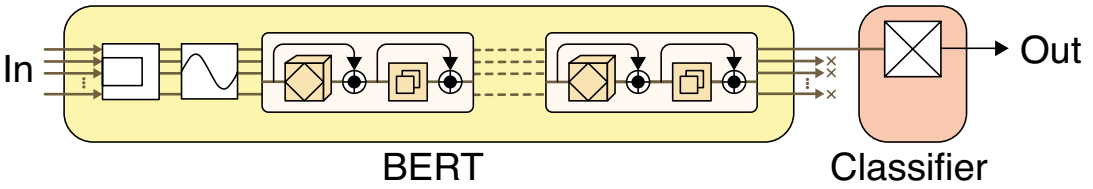


Figure 20-28: BERT with a small downstream classifier at the end. The dashed lines stand for the 10 additional, identical encoder blocks that are present, but not drawn.

After four epochs of training, here are six results from the testing data. The first three are grammatical, and the second three are not. BERT produced the correct answer on all six.

- Chris walks, Pat eats broccoli, and Sandy plays squash.
- There was some particular dog who saved every family.
- Susan frightens her.
- The person confessed responsible.
- The cat slept soundly and furry.
- The soundly and furry cat slept.

On the test set of about 1,000 sentences, this little version of BERT got about 82 percent of the examples correct. Some BERT variants have achieved more than 88 percent right on this task (Wang et al. 2019; Wang et al. 2020).

Let's try BERT out on another task, called *sentiment analysis*. We'll classify short movie reviews as being either positive or negative in tone. The data comes from a database of almost 7,000 movie reviews called *SST2*, where each review has been labeled as positive or negative (Socher et al. 2013a; Socher et al. 2013b).

For this run, we used a pretrained BERT model called DistillBERT (Sanh et al. 2020; Alammar 2019) (the term *distilling* is often used when we carefully trim a trained neural network to make it smaller and faster without losing much performance). We're again doing a classification task, so we can reuse the model of Figure 20-28.

Here are six examples verbatim from the test data (there's no indication of what movies they each refer to). DistillBERT properly classified the first three reviews as positive and the second three as negative (the reviews are all lowercase, and commas are treated as their own tokens).

- a beautiful , entertaining two hours
- this is a shrewd and effective film from a director who understands how to create and sustain a mood
- a thoroughly engaging , surprisingly touching british comedy
- the movie slides downhill as soon as macho action conventions assert themselves
- a zombie movie in every sense of the word mindless , lifeless , meandering , loud , painful , obnoxious
- it is that rare combination of bad writing , bad direction and bad acting the trifecta of badness

Of the 1,730 reviews in the test set, DistillBERT correctly predicted the sentiment of about 82 percent of them.

To recap, models based on the BERT architecture are united by their use of a sequence of encoder blocks. They create an embedding of a sentence that captures enough information that downstream applications can perform a wide range of operations upon it. With an appropriate downstream model, BERT can be used to perform many of the NLP tasks we mentioned at the start of Chapter 19.

If we're willing to get clever, we can make BERT generate language, but it's not easy (Mishra 2020; Mansimov et al. 2020). A better solution is to use decoder blocks, as we'll see next.

GPT-2

We've seen how transformers use a series of decoder blocks to generate words for a translation. We can also use a sequence of decoder blocks to generate new text.

Since we don't have an encoder stage to receive KV values from, as in the full transformer of Figure 20-24, let's remove the Q/KV multi-head attention layer from each decoder block, leaving us with just masked self-attention and a pair of 1×1 convolutions. The first system to do this in a big way was called the *Generative Pre-Training model 2*, or simply *GPT-2* (Radford et al. 2019). Its architecture is shown in Figure 20-29.

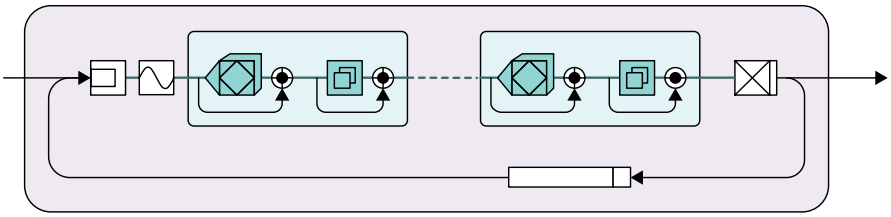


Figure 20-29: A block diagram of GPT-2, made out of transformer decoder blocks without the Q/KV layer. The dashed lines stand for more repeated, identical decoder blocks. Note that because these are versions of decoder blocks, the first multi-head attention layer in each block is a masked attention layer.

Like BERT, we start with token embedding followed by positional embedding for each input word. The self-attention layer in each decoder block uses masking as before so that as we compute attention for any given word, we can only use information from that word and those that precede it.

The original GPT-2 model was released in several different sizes, the largest of which processed 512 tokens at a time through 48 decoder blocks with 12 heads in each, for a total of 1,542 million parameters. That's 1.5 *billion* parameters. GPT-2 was trained on a dataset called *WebText*, which contained about eight million documents for a total of about 40GB of text (Radford et al. 2019).

We typically use GPT-2 by starting with the pretrained model, and then we fine-tune it by providing an additional dataset to learn from, adjusting all of the weights in the process (Alammar 2018).

We started each of our text generators in Chapter 19 with a seed, but that's only one way to get them started. A simpler approach starts the system with general guidance and a prompt. This is called a *zero-shot* scenario, since the system has been given zero "shots," or examples, for it to use as a model for new text.

For instance, suppose we built a system to advise us on what to wear each day. A zero-shot scenario might start with the instruction, Describe today's outfit, followed by the prompt, Today I should wear: The generator takes it from there. It has no examples or context to work from, so it might suggest a suit of armor, a spacesuit, or a bear skin.

Alternatively, we can provide one or more examples, or shots. In a one-shot scenario, we might give the instruction Describe today's outfit, followed by the example, Yesterday I wore a blue shirt and black pants, and conclude with the prompt, Today I should wear: The thinking is that the text that's provided before the prompt can help guide the system into the kind of output we want. In this case, the bear skin would be less likely.

If we give the system two or three shots, but not many more, we usually call it a *few-shot* scenario (these terms don't have sharp cutoffs). People usually prefer generators that require as few shots as possible in order to provide the output we want.

Let's see GPT-2 in action, using a medium-sized, pretrained GPT-2 model (von Platen 2020). We won't do any fine tuning, so the system will generate text based only on its core training data. Let's take a zero-shot approach, and give it no information except the starting prompt, I woke up this morning to the roar of a hippopotamus. Here's a typical output, verbatim:

> I woke up this morning to the roar of a hippopotamus. I was in the middle of a long walk, and I saw a huge hippopotamus. I was so excited. I was so excited. I was so excited. I was so excited.

At this point the system kept repeating I was so excited endlessly. That's not the generator's fault, it's ours. The system is producing grammatical output, which is what we trained it to do. The problem with the output is that, despite its emphasis on excitement, it's boring. The end of one sentence happened to lead back to the start of that same sentence, and we got locked in a loop. The system as it is now has no idea that such output is boring or undesirable.

To make output more interesting, we can chip away at the problem, removing characteristics of the output we see as undesirable. Let's look at just two such changes (Vijayakumar et al. 2018; Shao et al. 2017).

First, let's do away with that repetition. We can penalize the system if it generates the same group of words repeatedly. This is called an *n-gram penalty* because a sequence of *n* words is called an *n-gram* (Paulus, Xiong, and Socher 2017; Klein et al. 2017). Let's take it easy on the system and only punish repeated pairs of words, or 2-grams.

Second, rather than always picking the word with the highest probability, let's pick several of them and see where they lead. That is, for each word, we will generate more words following on from that. When we've followed several paths for a while, we will score each of them to determine which one has the highest probability of being generated. We use the first word of that branch as our predicted word, and throw away the rest of that branch and all of the other branches. Then we repeat the process for the next word, and the next, and so on. This is called a *beam search* (Russell and Norvig 2009). This should give us longer sequences that hang together. And instead of always choosing the very highest probability word, we'll pick among several of the most probable words, according to their probabilities, which will help make the output less predictable. This is called increasing the *temperature* of the word selection process (a temperature of zero corresponds to always picking the most probable choice).

With those two changes in place, let's try generating text again using the same starting sentence. Here is one verbatim result:

> I woke up this morning to the roar of a hippopotamus. "Oh, my God," I said. "I'm so sorry. I didn't mean to hurt you. It's just that I don't know how to deal with it. You know what I mean? I'm not going to let you hurt me, okay? You're my friend, and I love you so much. And I want you to know that you're not alone in this. We've been together for a long time now, so it's not like there's anything we can do about it, but I can't help but feel sorry for you, because I know you'll never be alone with me again.

Wow. That's remarkable. We could keep improving the output by using more processing techniques (Welleck et al. 2020), but this is already pretty great, particularly considering that we're using a stripped-down GPT-2 model.

For comparison, let's revisit the same Sherlock Holmes stories from Chapter 19. We started with a small, pre-trained GPT-2, fine-tuned it on these stories (Woolf 2019), and then gave it the prompt, I asked Holmes to explain.

We used a slightly warm temperature, so each time we ran the algorithm, we got a different result. Here's just one, which we arbitrarily cut off after a few paragraphs.

> I asked Holmes to explain.
>
> "It is very simple really," he replied. "The short answer is, of course, that the law does not apply to these matters, and that the case should be left entirely in the discretion of the Crown Prosecution and the police. But the long answer is that the case clearly does not rest with us, and that it is very much in the power of the police to prevent it.
>
> "You remember, Watson, that it was upon this very day that the young man, Mr. Latimer, picked up that long revolver which he used to brand me. The revolver which he used to brand me, is probably the best known weapon in the world. You remember, Watson, that I said at the time that it was better not to know than to hate the fear of it. Now, we have heard, and we must not let ourselves be drawn into a dilemma. The matter has been sufficiently complicated, and it is not necessary that the facts should be remarked upon in this fashion. The man who does not know is not himself a danger to himself. Let us take the man who does.

These results are grammatical and even refer to themselves. Compare this output to what we got from character-based autoregression with RNNs.

GPT-2 can do lots of other tasks well, such as running a version of the cloze test, predicting the next word of a phrase where essential information appears at least 50 tokens before, answering questions about text, summarizing documents, and translating from one language to another.

Generators Discussion

GPT-2 shows that if we process 512 tokens at a time through 48 decoder layers with 12 attention heads in each, for a total of 1.5 billion parameters, we can produce some pretty good text. What if we scaled everything up? That is, we won't modify the basic architecture at all, but just use a lot more of everything.

This was the plan of the successor to GPT-2, which was called (surprise) *GPT-3*. The block diagram for GPT-3 looks generally like that of GPT-2 in Figure 20-29 (aside from some efficiency improvements). There's just more of everything. A lot more. GPT-3 processes 2,048 tokens at a time, on 96 decoder layers, with 96 attention heads in each layer, for a total of 175 billion parameters (Brown et al. 2020). 175 billion. Training this behemoth required an estimated 355 GPU years at an estimated cost of US$4.6 million (Alammar 2018).

GPT-3 was trained using a database called the *Common Crawl* dataset (Common Crawl 2020). It started with about a trillion words from books and the web. After removing duplications and cleaning the database, the database still had about 420 billion words (Raffel et al. 2020).

GPT-3 is capable of creating lots of different kinds of data. It was made available to the public for a period as a kind of beta test, but it's now a commercial product (Scott 2020). During the beta test, people used GPT-3 for many applications, such as writing code for web layouts, writing actual computer programs, taking imaginary employment interviews, rewriting legal text in plain language, writing new text that looks like legal language, and, of course, writing in creative genres like fiction and poetry (Huston 2020).

All this power is a mixed bag. Fine-tuning such a system requires enormous resources, and it becomes harder and harder to fine tune, as that requires finding task-specific data that wasn't in the original data.

If bigger is better, would even bigger still be even better still? The researchers behind GPT-3 have estimated that we can extract everything we need to know about any text (at least from the point of view of NLP-type tasks) with a model that uses 1 trillion parameters trained on 1 trillion tokens (Kaplan et al. 2020). These numbers are rough predictions and could be far off, but it's interesting to think that there could be a point at which a stack of decoder blocks (and some support mechanisms) could extract almost all the information we need from a piece of text. We'll probably know the answer soon, as other huge firms with enormous resources are sure to produce their own gargantuan NLP systems trained on their own colossal databases. Training these vast systems is a game only the big and rich can play.

On a light-hearted note, we can play an interactive text-based fantasy game online, driven by a GPT-3 implementation (Walton 2020). The system was trained on a variety of genres, ranging from fantasy and cyberpunk to spy stories. Perhaps the most fun way to play with this system is to treat the AI as an improv partner, agreeing with and expanding on whatever the system throws at us. Let the AI set the flow and go with it.

Generated text can often hold up well in short doses, but how well does it do when we look closer? A recent study asked many language generators, including GPT-3, to perform 57 tasks, based on topics from humanities like law and history, to social sciences like economics and psychology, and STEM subjects like physics and mathematics (Hendrycks et al. 2020). Most output never came near human performance. The systems fared especially poorly on important social issues like morality and law.

This shouldn't be a surprise. These systems are simply producing words based on their probabilities of belonging together. In a real and fundamental sense, they have no idea what they're talking about.

For all their power, text generators like those we've seen here have no common sense. Worse, they blindly reiterate the stereotypes and prejudices inherited wholesale from the gender, racial, social, political, age, and other biases in their training data. Text generators have no idea of accuracy, fairness, kindness, or honesty. They don't know when they're stating facts or

making things up. They just generate words that follow the statistics of the training data, and perpetuate every prejudice and limitation to be found there.

Data Poisoning

We saw in Chapter 17 that adversarial attacks can trick convolutional neural networks into generating incorrect results. Natural language processing algorithms are also susceptible to intentional attacks, called *data poisoning*.

The idea behind data poisoning is to manipulate the training data for an NLP system in such a way that the system produces a desired type of inaccurate result, perhaps consistently, or perhaps only in the presence of a triggering word or phrase. For example, one can insert sentences or phrases into the training data that suggest that strawberries are made of cement. If these new entries are not discovered, then if the system is later used to generate stocking orders for a supermarket or a building contractor, they may find that their inventories end up being consistently and mysteriously wrong.

This is particularly concerning because, as we've seen, NLP systems are typically trained on massive databases of millions or billions of words, so nobody is carefully reviewing the database for misleading phrases. Even if one or more people carefully read the entire training set, the poisoning texts can be designed so that they never explicitly refer to their targets, making them essentially indetectable and their effects unpredictable.

Returning to our previous example, such phrases can convince a system that strawberries are made of cement, while never referring to fruit or building materials at all. This is called *concealed data poisoning*, and it can be fiendishly hard to detect and prevent (Wallace et al. 2020).

Another kind of attack is based on changing the training data in a seemingly benign way. Suppose we're working with a system that classifies news headlines into different categories. Any given headline can be subtly rewritten so that the obvious meaning is not changed, but the story is incorrectly classified. For instance, the original headline, Turkey is put on track for EU membership, would be correctly classified under "World." But if an editor rephrases this into the active voice—EU puts Turkey on track for full membership—this would now be misclassified as "Business" (Xu, Ramirez, and Veeramachaneni 2020).

Data poisoning is particularly nefarious for several reasons. First, it can be done by people who have no connection to the organizations building or training the NLP models. Since significant amounts of training data are usually drawn from public sources, such as the web, a poisoner only needs to publish the manipulative phrases in a public blog or other location where they're likely to be scooped up and used. Second, data poisoning can be done well ahead of any specific system's use, or indeed, even before it's conceived of.

There's no knowing how much training data has already been poisoned and is simply awaiting activation, like the sleeper agents in *The Manchurian Candidate* (Frankenheimer 1962). Finally, unlike adversarial attacks on CNNs, poisoned data compromises the NLP system from within, making its influence an inherent part of the trained model.

When a compromised system is used to make important decisions, such as evaluating school admission essays, interpreting medical notes, monitoring social media for fraud and manipulation, or searching legal records, then data poisoning can produce errors that change the course of people's lives. Before any NLP system is used in such sensitive applications, in addition to examining it for signs of bias and historical prejudice, we must also analyze it for data poisoning, and certify it as safe only if it is demonstrably not biased or poisoned. Unfortunately, no methods for robust detection or certification of any of these problems currently exist.

Summary

We started this chapter with word embedding, which assigns each word a vector in a high-dimensional space representing its use. We saw how ELMo lets us capture multiple meanings based on content.

We discussed the mechanism of attention, which lets us simultaneously find words in the input that seem related, and build combinations of versions of the vectors describing those words.

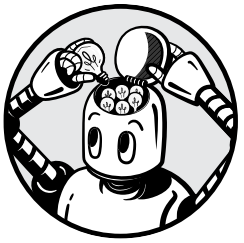
Then we looked at transformers, which do away with recurrent cells entirely and replace them with multiple attention networks. This change allows us to train in parallel, which is of huge practical value.

Finally, we saw how to use multiple transformer encoder blocks to build BERT, a system for high-quality encoding, and how to use multiple decoder blocks to build GPT-2, a high-quality text generator.

In the next chapter we'll turn our attention to reinforcement learning, which offers a way to train neural networks by evaluating their guesses, rather than expecting them to predict a single correct answer.

21

REINFORCEMENT LEARNING



There are many ways to train a machine learning system. When we have a set of labeled samples, we can use supervised learning to teach the computer to predict the right label for each sample. When we can't offer any feedback, we can use unsupervised learning and let the computer do its best. But sometimes we're somewhere in between these two extremes. Perhaps we know *something* about what we want the system to learn, but it's not as clear-cut as having labels for samples. Perhaps all we know is how to tell a better solution from a worse one.

For example, we might be trying to teach a new kind of humanoid robot how to walk on two legs. We don't know exactly how it ought to balance and how it should move, but we know we want it to be upright and not falling over. If the robot tries to slither on its belly, or hop on one leg, we

can tell it that's not the right way to proceed. If it starts with both legs on the ground and then uses them to make some forward progress, we can tell it that it's on the right track and keep exploring those kinds of behaviors. This strategy of rewarding what we recognize as progress is called *reinforcement learning (RL)* (Sutton and Baro 2018). The term describes a general approach to learning, rather than a specific algorithm.

In this chapter we cover some of the basic ideas of this vast field. The key idea is that RL breaks up the simulated world into one entity that takes action and the rest of the world that responds to that action. To make this concrete, we will use RL to learn how to play a simple one-player game, and then dig into the details of the technique. We'll begin with a simple algorithm that has some flaws and upgrade it into something that learns efficiently and well.

Basic Ideas

Suppose that you're playing a game of checkers with a friend, and it's your turn. At this moment, you can move one of your pieces, and your friend has to wait. In reinforcement learning, we say that you're the *actor* or *agent* since you have the choice of action. Everything else in the universe—the board, the pieces, the rules, and even your friend—are lumped together as the *environment*. These roles aren't fixed. When it's your friend's turn to move, then they're the agent, and everything else—the board, the pieces, the rules, and even you—are now part of the environment.

When the actor or agent chooses an action, they change the environment. In our checkers game, you're the agent, so you move one of your pieces, and maybe you remove some of your opponent's pieces. The result is that the world has changed. In reinforcement learning, after an agent's action, they're given a piece of *feedback,* also called a *reward,* that tells them how "good" their action was, using whatever criteria we like. The feedback or reward is usually just a single number. Since we're creating this world, the feedback can mean anything we want. In a game of checkers, for instance, a move that wins the game would be assigned a huge positive reward, whereas a move that loses the game would be assigned a huge negative reward. In between, the more a move seems to lead to victory, the bigger the reward.

Through trial and error, an agent can discover which actions are better than others in different situations, and can thus gradually make better and better choices as it gains experience. This approach works particularly well for situations in which we don't already know the best thing to do at all times. For example, consider the problem of scheduling the elevators in a tall and busy office building. Even just figuring out where elevator cars ought to go when they're empty is hard. Should the cars always return to the ground floor? Should some wait at the top? Should they wait at floors evenly distributed between the top and bottom? Maybe these policies should change over time so in the early morning and just after lunch, the cars should be on the ground floor, waiting for people arriving off the

street, but in the late afternoon, they should be higher up, ready to help people descend and head home. There's no obvious answer to how we should schedule a particular building. It all depends on the average traffic pattern for that building (and that pattern itself might depend on the time, season, or weather).

This is an ideal problem for reinforcement learning. The elevator's control system can try out a policy for directing the empty cars, and then use feedback from the environment (such as the number of people waiting for elevators, their average waiting time, the density of the elevator cars, etc.) to help it adjust that policy to perform as well as it can on the metrics we're measuring.

Reinforcement learning can help us with problems for which we don't know the best result. We may not have a measurement as clear as the winning conditions of a game, but only better and worse outcomes. This is a key point: we may not be able to find any objective, consistent "right" or "best" answer. Instead, we're trying to find the best answer we can with the information we have according to whatever metrics we're measuring by. In some situations, we may not even have any idea of how well we're doing along the way. For example, in a complex game, we might not be able to tell if we're ahead or behind until the surprising moment when we win or lose. In those cases, we can only evaluate our actions in light of how things finally work out when the task is done.

Reinforcement learning offers a nice way to model uncertainty. In simple rule-based games, we can, in principle, evaluate any board and select the best move, assuming that the other player always does the same. But in the real world, other players make moves that surprise us. And when we deal with the real world, where on some days more people need an elevator than on other days, we need to have strategies that can continue to perform well in the face of surprises. Reinforcement learning can be a good choice for these kinds of situations.

Let's look at reinforcement learning in more detail with a specific example.

Learning a New Game

Let's see the steps involved in using reinforcement learning to teach a program how to play *tic-tac-toe* (also called *naughts and crosses*, or *Xs and Os*). To play, the players alternate placing an X or O in the cells of a three by three grid, and the first to get three of their symbols in a row (in any direction) is the winner. In the examples of Figure 21-1, we play O and our computer learner plays X.

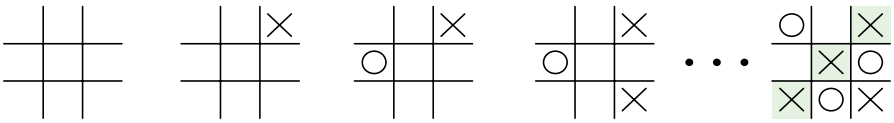


Figure 21-1: A game of tic-tac-toe, reading left to right. X moved first.

In this scenario, the program we're training is the agent. It's playing against the environment, which would probably be simulated by another program that knows all about the game and how to play. The agent doesn't know the rules of the game, how to win or lose, or even how to make moves. But our agent won't be completely in the dark. At the start of each of the agent's turns, the environment gives it two important pieces of information: the current board, and the list of available moves. This is shown in steps 1 and 2 of Figure 21-2.

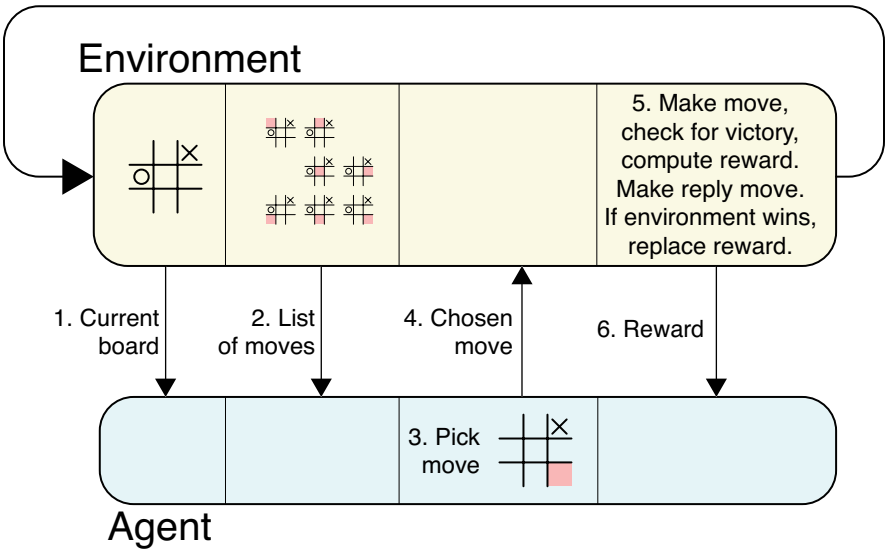


Figure 21-2: The basic information exchange loop between a player and the environment in a game of tic-tac-toe

In step 3, the agent picks a move, based on any methodology it likes. For example, it can pick at random, or consult an online resource, or use its own memory of previous games. Part of the challenge in reinforcement learning is designing an agent that can do a good job with the resources we have available for it.

Once the agent picks a move, it communicates that to the environment in step 4. The environment then follows step 5, starting with actually making the move by placing an X in the chosen cell. The environment then checks to see if the agent has won. If so, it sets the reward to something big. Otherwise, it computes a reward based on how good the move seems to be for the agent. Now the environment, simulating the other player, makes its own move. If it's won, then it changes the reward to something very low. If the game ended as a result of the environment's or agent's move, we call the reward an *ultimate reward* or *final reward*. In step 6, the environment sends the reward (sometimes this is called the *reward signal)* to the agent, so the agent can learn how good its selected move was. If nobody's won, we return to the start of the loop and the agent gets to take another turn.

In some cases, we don't give the agent the list of available moves. This might be because there are too many to list, or they have too many variations. Then we might give the agent some guidelines, or even no guidance at all.

Following this procedure, the agent will probably make useless or terrible actions when it starts learning, but using the techniques below we'd hope that the agent will gradually learn to find good actions. For our discussions, we'll keep things simple and assume that the agent is given a list of possible actions to choose from.

The Structure of Reinforcement Learning

Let's reorganize and generalize our tic-tac-toe example into a more abstract description. This will let us embrace situations that go beyond turn-taking games. We'll organize things into three steps, which we discuss in turn.

Before we begin, a bit of terminology. At the start of training, we place the environment into an *initial state*. In a board game, this is the setup for the start of a new game. In our elevator example, this might be placing all elevator cars on the ground floor. A full training cycle (such as a game from start to finish) is called an *episode*. We generally expect to teach the agent over a great many episodes.

Step 1: The Agent Selects an Action

We begin with Figure 21-3.

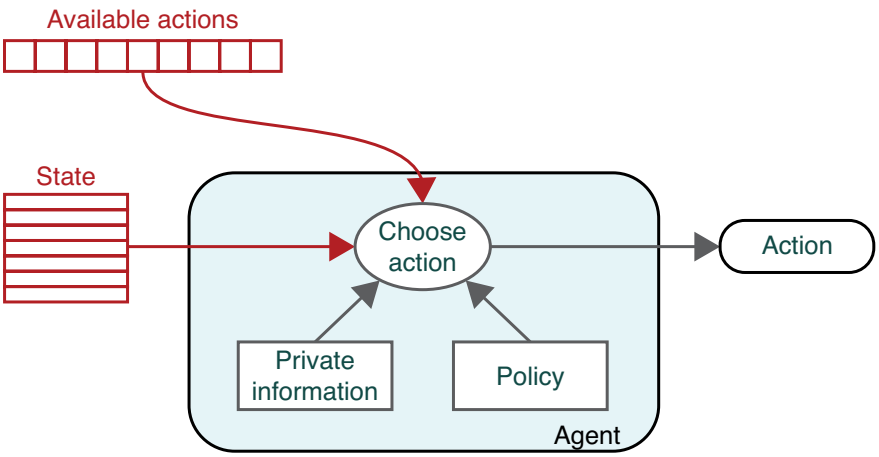


Figure 21-3: The environment provides the agent with the current world state and a choice of actions. The agent chooses an action and communicates that to the environment.

Recall that the environment is the world in which all of our agent's actions take place. The environment is completely described by a set of numbers that are collectively called the *environmental state*, the *state variables*, or simply the *state*. This might be a short list, or a very long one, depending on the complexity of the environment. In the case of a board game, the

state is commonly made up of the position of all the markers on the board, plus any game assets (such as game money, power-ups, hidden cards, etc.) held by each player.

The agent then chooses one of the available actions. We often anthropomorphize the agent and talk about how the agent "wants" to achieve some result, such as winning a game or scheduling the elevators so nobody has to wait too long. In basic reinforcement learning, the agent is idle until the environment tells it that it's time to take an action. The agent then chooses an action from the list of actions by using an algorithm called its *policy*, along with whatever *private information* the agent may have access to (including what it's learned from previous episodes).

We usually think of the agent's private information as a database. It might contain descriptions of possible strategies or some kind of history of the actions that were taken in previous states and the rewards that were returned. The policy, by contrast, is an algorithm that is usually controlled by a set of parameters. The parameters usually change over time as the agent plays and searches for improved action-choosing policies.

We usually don't think of the agent implementing its action. Instead, the chosen action is reported to the environment, and the environment takes care of performing the action. This is because the environment is in charge of the state. Returning to our elevator example, if the agent directs a car to move from the 13th floor to the 8th floor, the agent doesn't update the state to place the car at the 8th floor. Something might go wrong along the way, such as a mechanical failure causing the car to get stuck. The agent simply tells the environment what it wants to do, and the environment tries to make that happen, maintaining the state so it's always a correct picture of the current situation. In our tic-tac-toe game, the state contains the current distribution of X and O markers on the board.

Step 2: The Environment Responds

Figure 21-4 shows step 2 of our reinforcement learning overview.

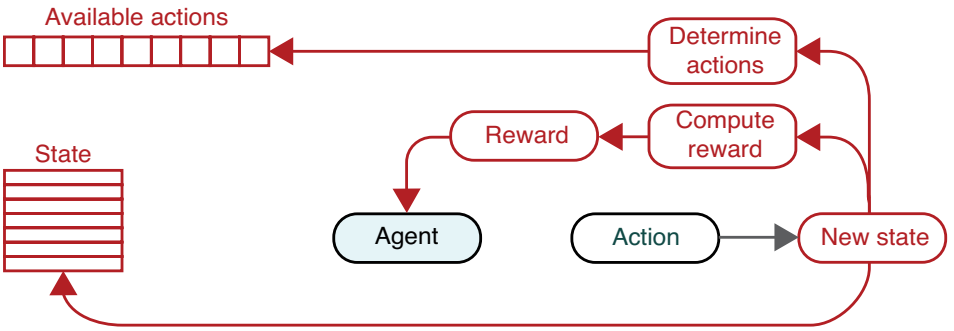


Figure 21-4: Step 2 of our reinforcement learning process. This step starts with the computation of the new state (far right).

In this step, the environment processes the agent's action to produce a new state and processes the information that follows from this change. The environment saves its new state in the state variables, so that they reflect the new environment when the agent next gets to choose an action. The environment also uses its new state to determine what actions will be available to the agent on its next move. The previous state and the available actions are entirely replaced by their new versions. Lastly, the environment provides a reward signal that tells the agent how "good" its last chosen action was. The meaning of "good" is completely dependent on what this whole system is doing. In a game, good actions are moves that lead to stronger positions or even victory. In an elevator scheduling system, good actions might be those that minimize wait times.

Step 3: The Agent Updates Itself

Figure 21-5 shows step 3 of our reinforcement learning overview.

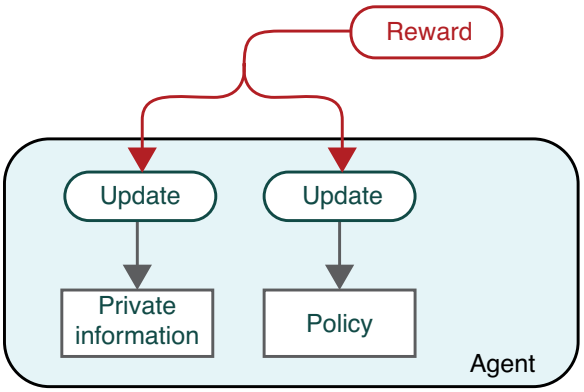


Figure 21-5: Step 3 of our reinforcement learning process, where the agent updates itself in response to the reward

In this step, the agent uses the reward value to update its private information and policy parameters so that the next time this situation comes around, it's able to build on what it has learned from this choice. After step 3, the agent might wait quietly until the environment tells it that it's time to take action again. Alternatively, it can immediately start planning for its next move. This is particularly useful for some real-time systems where the reward precedes the full calculation of the new state.

Rather than simply stashing each reward into its private information, an agent usually processes that reward in some way to extract as much value from it as possible. This might even involve changing the values for other actions. For example, if we have just won a game and received an ultimate reward, we probably want to add a little bit of that reward to each of the moves that led us to victory.

The goal of reinforcement learning is to discover ways to help the agent in this scenario learn from the feedback to choose actions that bring it the best possible rewards. Whether it's winning a game, scheduling elevators,

designing vaccines, or moving a robot, we want to create an agent that can *learn from experience* to become as good as possible at manipulating its environment to bring about positive rewards.

Back to the Big Picture

Now that we've seen the overall approach, let's look at some big-picture issues. When the agent updates its policy, it might have access to all the parameters of the state, or only some of them. If an agent gets to see the entire state, we say it has *full observability*, otherwise it has only *limited observability* (or *partial observability*). One reason we may give an agent only limited observability is that some parameters may be very expensive to compute, and we're not sure if they're relevant or not. So, we block the agent's access to those parameters to see if doing so hurts the agent's performance. If leaving them out does no harm, we can leave them off entirely from then on and save effort. Or we can only compute them and make them visible when they seem necessary. Another example of partial observability is if we're teaching the system to play a card game such as poker. We don't reveal to the system we're teaching what cards are in its opponent's hand.

As soon as we start thinking about using feedback to train agents in the way we've been discussing, we find ourselves facing two interesting problems. First, when we receive an ultimate reward (perhaps for winning or losing a game), we want to share some of that reward with every move we made along the way. Suppose we're playing a game and make a winning move. That final move gets great feedback, but the intermediate steps were essential, and we should remember that they led to victory. That way if we see those intermediate boards again, we are more likely to select the move that leads to winning. Finding a way to share the ultimate reward this way is called the *credit assignment problem*. By the same token, if we lose, we'd want to let the moves that led us there take some of the blame, so we are less likely to select them again.

Second, suppose at some point the agent sees a situation (such as a game board) that it has seen before, and that at some earlier point, it tried a move that got a reasonably good score. But it hasn't yet tried some of the other possible moves. Should it select the safe move with known returns, or risk something new that might either lead to failure or to even greater success? Somehow we need to decide, each time we pick an action, whether we want to take a risk with a new action and *explore* where it might lead, or play it safe with an action we've tried before and *exploit* what we've already learned. This is called the *explore or exploit dilemma*. Part of the task of designing a reinforcement learning system is thinking about how we want to balance these issues of the known and unknown, or guarantee and risk.

Understanding Rewards

For the agent to perform as well as possible, it should be guided by a policy that leads the agent to pick the actions that deliver the highest rewards.

Understanding the nature of rewards, and how to use them wisely, is time well spent. Let's dig in.

We can distinguish between two categories of rewards: *immediate* and *long term*. Immediate rewards are the ones we've focused on so far. The environment delivers these back to the agent right after executing an action, as we saw in Figure 21-2. Long-term rewards are more general and refer to our overall objective, like winning a game.

We'd like to understand each immediate reward in the context of all the other rewards we get during a given game, or episode. There are lots of ways to interpret rewards and what they should mean to us. Let's look at one popular approach called the *discounted future reward (DFR)*. This is a way to address the credit assignment problem, or make sure that all the actions that led us to success share in that ultimate victory.

To see how DFR works, we need to unwind the reward process a little bit. Let's imagine that we're an agent playing a game. When the game is done, we can line up the rewards we've collected for that game in a list, one after the other in the order they were received, along with the moves that earned those rewards. Adding up all the rewards gets us the *total reward* for that game, as in Figure 21-6.

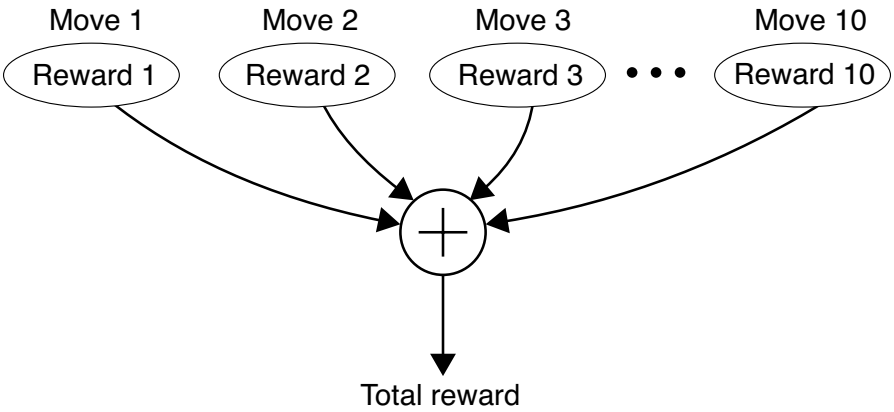


Figure 21-6: The total reward associated with any episode is the sum of all the rewards that arrive from the first to the last move of the episode.

We can add up any piece of this list, such as the first five entries, or the last eight. Let's start at move 5 and add up all the rewards from there up to the game's end, as in Figure 21-7.

Figure 21-7 shows us the *total future reward*, or *TFR*, associated with the fifth move of the game. It's that part of the total reward that comes from the fifth move and all the moves that followed it.

The very first move of a game is special, because its total future reward is the same as the game's total reward. Since our rewards so far are always zero or positive, each subsequent move's TFR is equal to or less than the TFR of the move before it.

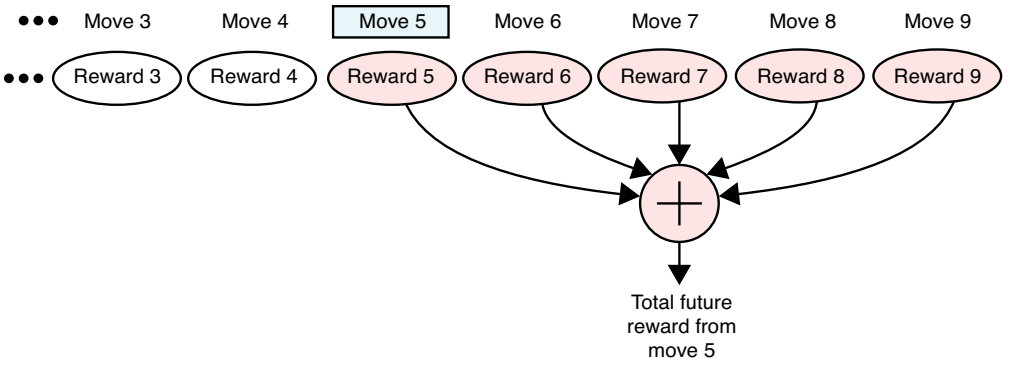


Figure 21-7: The total future reward for any move is the sum of the reward for that move and all other moves to the end of the episode.

The total future reward is a good description of how well a given move contributed to a game we just finished, but it's not as good at predicting how useful that move might be in future games, even if they start with the exact same sequence of moves. This is because real environments are unpredictable. If we're playing a multiplayer game, we can't be sure that the other player (or players) will act the same way in the next game as they did in the previous game. If they make a different move, then that can change the trajectory of the game, and thus it can also change the rewards we earn. It can even change whether we win or lose. Even if we're playing solitaire, we might be playing with a shuffled deck of cards, or a computer game with pseudorandom numbers, so we can't be sure what's going to come our way in the future, even if we play the exact same way we did in the past.

Immediate rewards are more reliable. We can imagine two types of immediate rewards. The first tells us the quality of the move we just made, *before* the environment responds. For example, in our game of tic-tac-toe, if the agent places an X in some cell, they can receive an *instant reward* describing how well the player is set up to win later on before the environment makes its move in return. This kind of reward is completely predictable. If we face the identical environment again later and make the same move, we get the same reward.

The second kind of reward tells us the quality of the move we just made *after* the environment responds, so the reward can be influenced by the environment's move. This type of reward, which we might call the *resulting reward*, isn't as predictable as the instant reward because the environment might respond in a different way each time we make the move.

Let's compare the two. Suppose we're training an agent-powered robot how to use a remote control to turn on a device. It might pick up the remote control, press the power button, and put the remote back down, doing the same thing 100 times in a row, earning high rewards. But all this time, the battery is draining, so the 101st time the agent repeats the process, the device won't turn on. If the agent receives the instant reward for pressing the button, that is, the one that is computed and returned *before* the environment responds, the agent gets a large reward, because it

did the right thing. On the other hand, the resulting reward, which is computed and returned *after* the environment responds, will be low or even 0, because the device failed to turn on.

From here on, we'll be using the resulting reward when we refer to the immediate reward.

When something works 100 times in a row but then fails the 101st time, that's a *surprise*.

It's important to deal gracefully with surprises, because most environments are unpredictable. Generally speaking, each action we take is intended to bring about a result. So waiting to see that result, even if we can't be certain about what will happen, is a big part of understanding if our action represented a good choice.

We say that real environments, with their unpredictable elements, are *stochastic*. By contrast, a perfectly predictable environment (such as a game based purely on logic) is *deterministic*. The amount of unpredictability (or *stochasticity*) can vary in amount. If the unpredictability is low (that is, the environment is largely deterministic), then we may feel pretty confident about saying that the rewards we just received are likely to be repeated, or very nearly so, in future games. With very high unpredictability (that is, in a largely stochastic environment), we have to assume that if we repeat the same actions, any predictions we make about future rewards should be considered little more than estimates.

We quantify our estimate of the stochasticity, or uncertainty, of the environment with a *discount factor*. This is a number between 0 and 1, usually written with the lowercase Greek letter γ (gamma). The value of γ that we select represents our confidence in the repeatability of the environment. If we think the environment is close to being deterministic and that we'll get about the same reward for a given action every time, we set γ to a value near 1. If we think the environment is chaotic and unpredictable, we set γ to a value nearer to 0.

We need to somehow accommodate unexpected surprises into our learned rewards in a principled way. One way to do that is to create a modified version of the total future reward that accounts for how confident we are that the game will proceed in just the same way again. We generally attach high values of this modified TFR to moves we feel confident about, and lower values to the others.

We can use the discount factor to create a version of the total future reward called the discounted future reward (DFR). Rather than adding up all the rewards that come after an action, as we do for the TFR, we start with the immediate reward, and then we reduce the values of the subsequent rewards by multiplying them by γ one time for each step they are in our future. The reward for one step in the future is multiplied by γ once, the reward after that is multiplied by γ twice, and so on. This accounts for the fact that we consider future rewards increasingly less reliable. The technique is illustrated graphically in Figure 21-8.

Notice that in Figure 21-8 each successive value gets multiplied by γ one more time than the one before. These increased multiplications can have a significant effect on the amount by which each reward contributes to the sum.

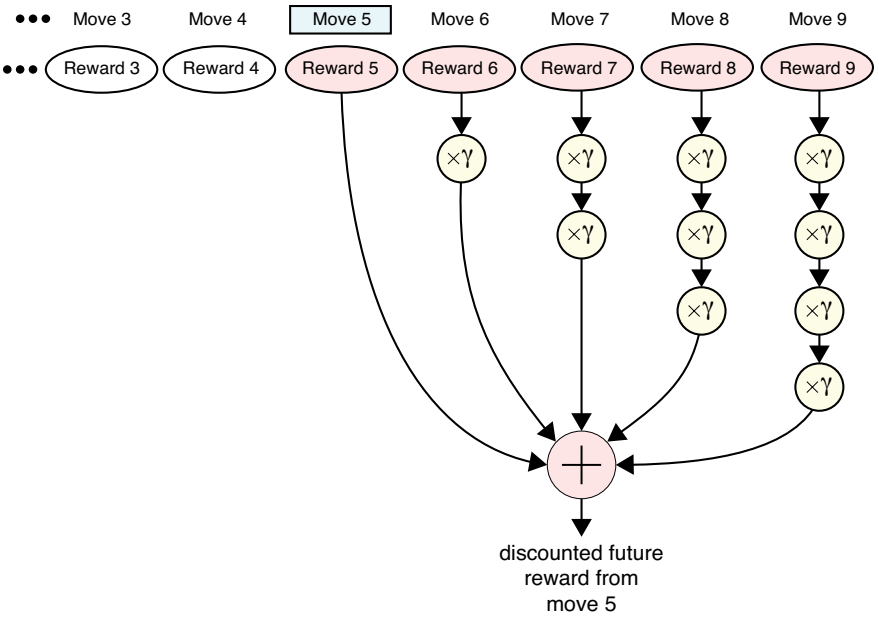


Figure 21-8: The DFR is found by adding together the immediate reward, the next reward after multiplying it by gamma, the reward after that multiplied by gamma twice, and so on.

Let's see this in action. We can consider the reward and the discounted future reward we'd get from our opening move in a game, using several values for γ. Figure 21-9 shows a set of immediate rewards for an imaginary game with 10 moves.

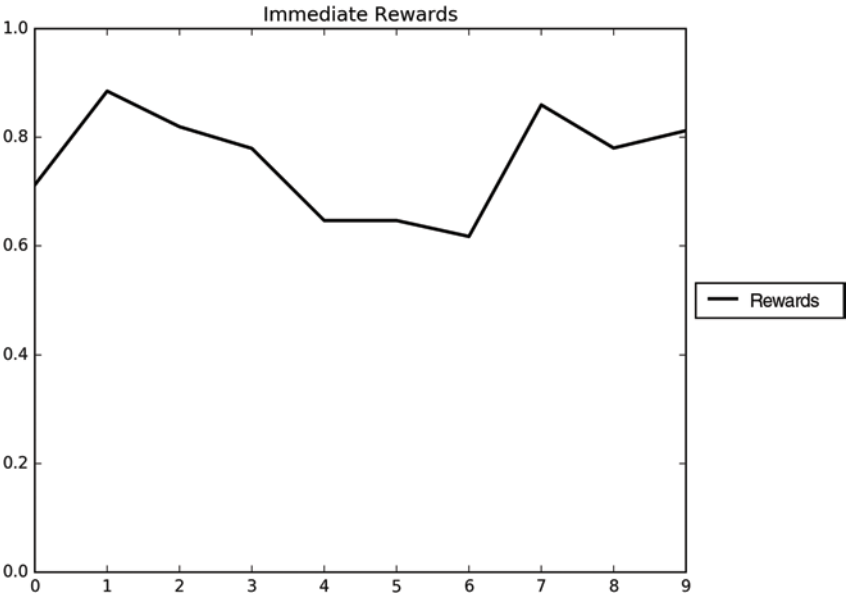


Figure 21-9: Immediate rewards for a game with 10 moves. The game ended without a clear winner.

Applying different future discounts to these rewards following Figure 21-8 gives us the curves of Figure 21-10. Notice how quickly the rewards drop to 0 as the discount factor γ decreases. This means that we're less sure of our predictions of the future.

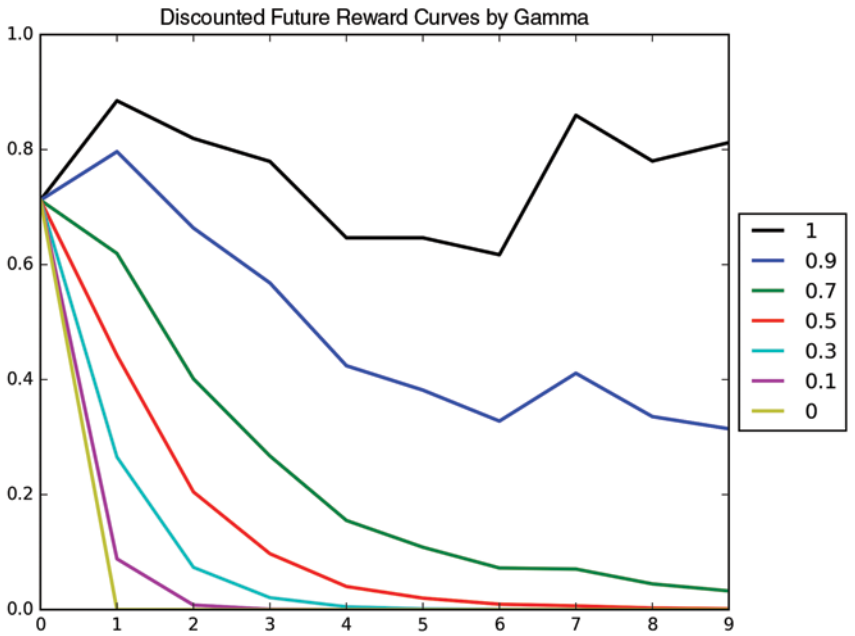


Figure 21-10: The rewards of Figure 21-9 as discounted by different values of γ

If we add up the values of each curve in Figure 21-10, we get the discounted future reward for the first move for different values of γ. These DFRs are shown in Figure 21-11. Notice that as we think of the future as being increasingly unpredictable (that is, γ gets smaller), the DFR also becomes smaller because we're less confident of getting those future rewards.

When γ has a value near 1, the future rewards aren't diminished much, so the DFR is close to the TFR. In other words, we're saying that the total rewards we got for making this move are likely to be similar to the total rewards we'll get if we play this move again.

But when γ has a value near 0, then the future rewards are scaled way down to the point where they practically don't matter, and we're left with just the immediate reward. In other words, we're saying that we have little confidence that the game will continue again as it did this time, so the only reward we can be sure of is the immediate reward.

In many reinforcement learning scenarios, we often pick a value of γ around 0.8 or 0.9 to get started, and then adjust the value as we discover more about how stochastic our system is and how well our agent is learning.

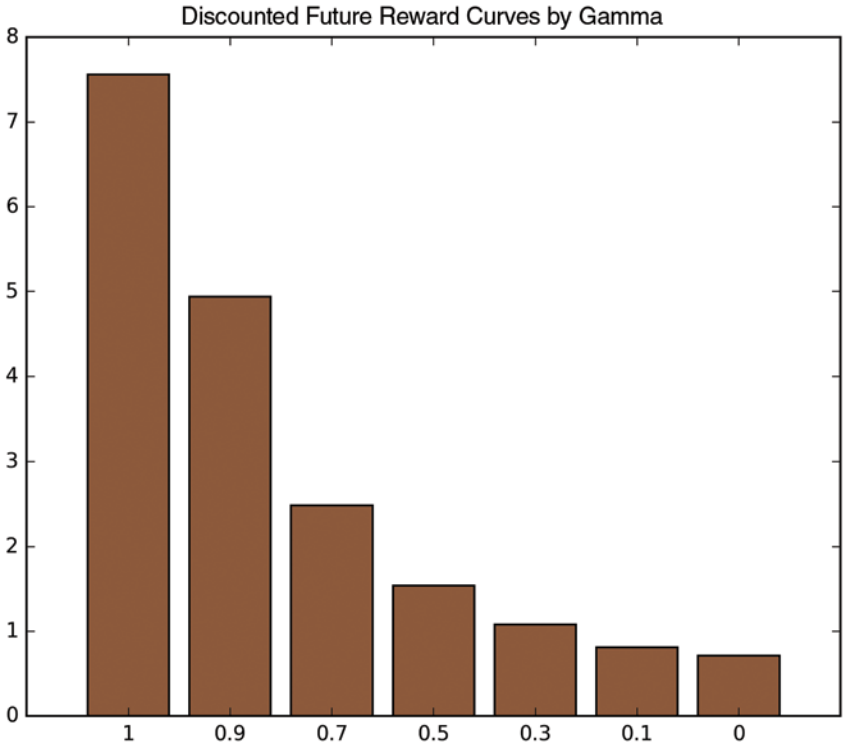


Figure 21-11: The DFR from Figure 21-10 for different values of γ

So far, we've been discussing principles and ideas, but we still don't have a specific algorithm for the agent to use when it picks an action. To develop such an algorithm, let's start with a description of an environment.

Flippers

In the following sections, we're going to look at actual algorithms for learning a game. To keep our focus on the algorithms and not the game, let's pare down tic-tac-toe into a new single-player game that we'll call *Flippers*.

We play Flippers on a square grid of size three by three. Each cell holds a little tile that pivots around a bar, as in Figure 21-12.

One side of each tile is blank, while the other side holds a dot. On each move, the player pushes one tile to flip it over. If it was showing a dot, the dot disappears, and vice versa.

The game begins with the tiles in a random state. Victory comes from having exactly three blue dots showing, arranged in either a vertical column or horizontal row, with all the other tiles showing blanks. This may not be the most intellectually demanding game ever invented, but it'll help make our algorithms clear.

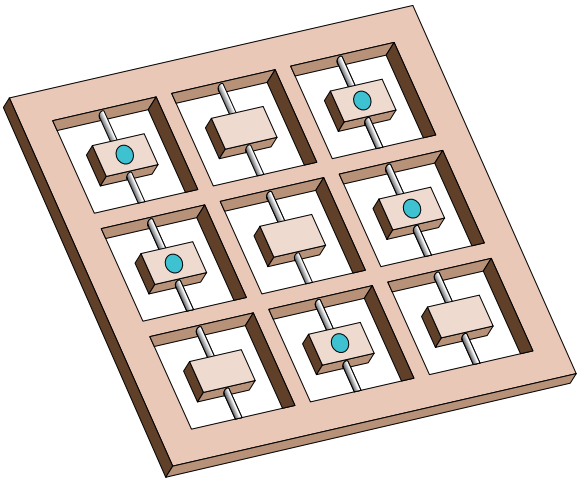


Figure 21-12: The board for the game of Flippers. Each tile is blank on one side and has a dot on the other. A move in the game consists of flipping (or rotating) one tile.

Starting from a random board, we want to get to victory in the smallest number of flips. Since diagonal lines don't count as victory, there are six different boards that satisfy our conditions for winning: three with horizontal rows and three with vertical columns.

Figure 21-13 shows an example game, along with our notation for showing the moves. We read the game left to right. Each board but the last shows the starting configuration for that move, with one cell highlighted in red. That's the cell that is going to be flipped over.

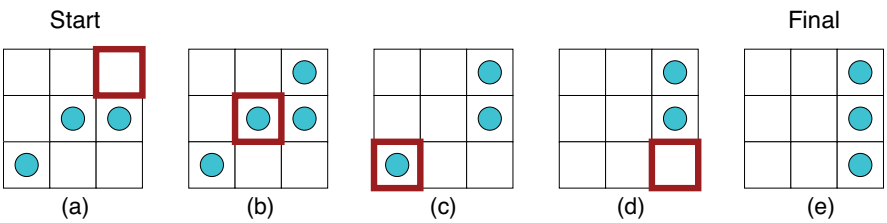


Figure 21-13: Playing a game of Flippers. (a) The initial board, showing three dots. The red square shows the tile we intend to flip for this move. (b) The resulting board is like part (a), but the tile in the upper-right has gone from blank to dot. Our move for this board is to flip the center tile. (c) through (e) show subsequent steps in game play. Board (e) is a winning board.

Now that we have a game to play, we can look at how to use reinforcement learning to win it.

L-Learning

Let's build a complete system for learning how to play Flippers. Although we will make this algorithm much better in the next section, this starting version is going to perform so badly that we call it *L-learning*, where L stands for "lousy." Note that L-learning is a stepping-stone that we invented to help us get to something better and not a practical algorithm that appears in the literature. It is, after all, lousy.

The Basics

To make things easy, we're going to use a very simple reward system. Every move we make in Flippers gets an immediate reward of 0, except for the final move that wins the game. Because Flippers is such an easy game, every game can be won. To prove this, we can take any starting board and flip over all the tiles that are showing a dot, so that there are no dots showing. Then we can flip over three tiles in any row or column, and we've won. Thus, no game should take more than 12 moves at most.

Our goal is not simply to win, however, but to win in the smallest number of moves. The final, winning move gets a reward that depends on the length of the game. If it takes one move to win the game, the reward is 1. If it takes more moves, this final reward drops off quickly with the number of moves that were taken. The specific formula for this curve is less important than the fact that it drops off fast and is always getting smaller. Figure 21-14 shows a graph of our final reward versus game length curve.

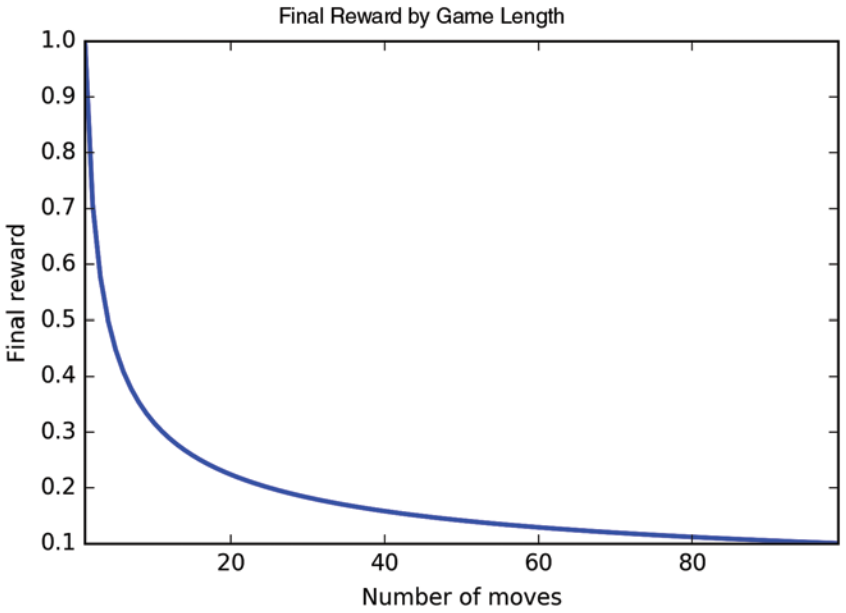


Figure 21-14: The reward for victory in Flippers starts at 1 for an immediate win but drops quickly with the number of moves required to win the game.

At the heart of our system is a grid of numbers that we call the *L-table*. Each row of the L-table represents one state of the board. Each column represents one of the nine actions we can make in response to that board. The content of each cell in the table is a single number, which we call an *L-value*. Figure 21-15 shows this schematically.

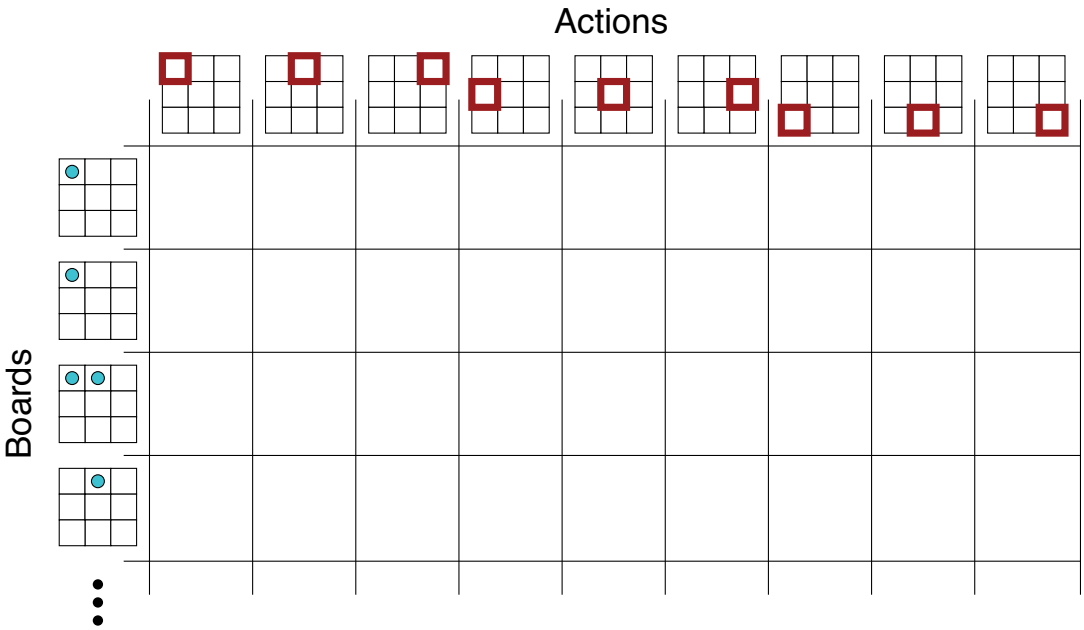


Figure 21-15: The L-table contains one row for each of the 512 possible patterns of blanks and dots on a Flipper board and one column for each of the 9 possible actions.

This table is big, but not too big. The board has only 512 possible configurations, so we need 512 rows. Each row has 9 columns, for a total of $512 \times 9 = 4{,}608$ cells. We're going to use the L-table to help us choose the highest-rewarding action in response to each board. To make that happen, we're going to fill each cell in the table with a score: a number, based on experience, that tells us how good the corresponding move is.

We save values into the L-table as we learn how good moves are, and we read those values back to guide our choice of moves as we play. Before we start assigning values to the L-table, we initialize it with a 0 in every cell. As we play a game, we will keep a record of all the moves we've played. When the game is over, we will look back through our moves for the whole game, and determine a value for each. Then we will combine this value with the number already in that move's cell to produce a new value for that move (we'll get to the mechanics for this in a moment). The way we combine the old and new values is called the *update rule*.

As we play a game (either during the learning phase, or later for real), we pick an action by looking at the corresponding row for the board at the start of that move. We use a policy to tell us which of the actions in that row we want to select.

Let's make these steps concrete. First, after each game (or episode), we need to determine the score we want to assign to each action we played. Let's use the total future reward, or TFR, that we discussed earlier. Recall that the TFR comes from lining up all our actions and their rewards, and then summing up all the rewards that came after that action.

While we play the game, every move along the way gets an immediate reward of 0, but the final move gets a positive reward based on the game's length: the shorter the game, the larger the reward. This means that the TFR for each action we took along the way is the same as this final reward.

Second, let's pick a simple update rule that says that after each game, the TFR we compute for each cell merely replaces whatever was in there before. In other words, the TFR for each action in this game becomes the new value in the cell at the intersection of the row of the board we were looking at when we took that action, and the column of the action we chose to take.

This simple update rule is good for getting familiar with how the L-learning system works. But because it doesn't combine our new experience with what we've learned before, this rule is a big reason that this algorithm isn't going to perform well.

Now that we have values in our L-table, we need a policy that tells us which move to play in response to a given configuration of the board. Let's say that we choose the action corresponding to the largest L-value in the row. If multiple cells have the same maximum value, we pick one at random. Figure 21-16 shows this graphically.

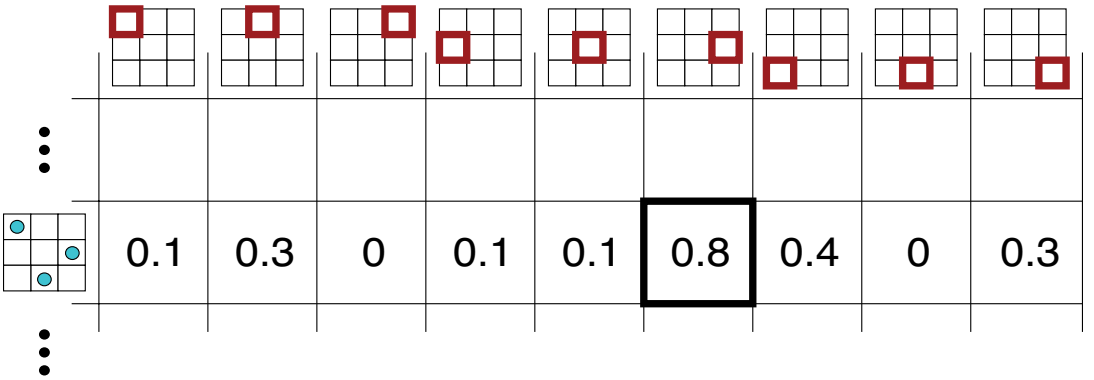


Figure 21-16: The policy step involves choosing one action in response to a board.

In Figure 21-16, we see a row of the L-table that lists the possible moves we can take in response to the board state shown at the far left. Each column holds the most recently computed TFR that resulted when that action was taken at that board state. Note that two columns hold 0, because we haven't tried those actions yet. In L-learning, we choose the largest value. Here that means we flip the center-right tile.

The L-Learning Algorithm

We now have all the steps required for L-learning. Let's combine them into a functional, but lousy, reinforcement learning algorithm. We start our agent with a private memory that contains a 512 by 9 table filled with zeros, representing the L-table.

In the first move of the first game, the agent sees a board. It finds the row for that board in its L-table and scans the nine entries there for the move with the largest score. They're all zero, so it picks one at random. This will happen frequently for quite a while because the agent will see lots of boards it has never seen before. The tile flips, the agent considers the new board, picks a new action, and so on, until it finally wins the game (the computer will produce a winning board eventually, even if the actions are selected entirely at random).

When the game ends, the agent wants to distribute the final reward among all the moves that got it to victory. To do so, while the game is played, the system will need to maintain a list of each move it plays, in order.

We'll later find that list more useful if each entry saves more than just the selected move. Anticipating that need, let's say that after each move, the agent retains a small bundle consisting of the starting board, the action the agent took, the immediate reward it received, and the board that resulted from that move. Figure 21-17 shows this visually. The agent saves these bundles in a list that starts empty at the start of the game and grows by one bundle after every move.

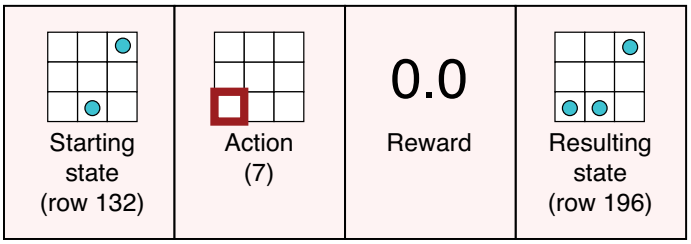


Figure 21-17: Each time we make a move, we append a bundle of four values to the end of a growing list of bundles: the starting state, our chosen action, the reward we received, and the final state the environment returned to us after taking that action.

As Figure 21-17 shows, we can save this bundle as a list of four numbers: the row number of the starting state, the column number of the action, the value of the reward, and the row number of the resulting state.

To make our first move, we look at the row of the L-table corresponding to our starting board, and the nine numbers we find along that row. Our policy will be to usually pick the largest value in the row, but sometimes pick one of the others for the sake of exploration. If all values are the same, as they are when we start out, we pick one at random.

The environment flips that tile for us, either making a dot appear or disappear. The environment then gives us back a reward and the new

board. We make a little bundle to represent this move: the board we started with, the action we just took, the reward we got back, and the new state that resulted. We stick that bundle onto the end of our list of moves.

Because we're playing solo, the environment isn't going to make any moves on its own. As soon as it has sent us our feedback, the environment tells us to take a new action. So again, we look at the current board, find its row in the L-table, select the largest cell in that row, and report that as our action. We get back a reward and a new state, and we add a new bundle of the four items describing this move to our list.

This goes on until the game is over. In the final piece of feedback, we get our only nonzero reward. It's the final reward based on the number of moves we played in the game, which drops off quickly, as we saw in Figure 21-14. With that final, nonzero reward, we know the game is over, so it's time to learn from our experience.

We start by looking at our bundles from our list of moves. Conceptually, we line up our board states and resulting moves, along with their rewards, as in Figure 21-18. One by one, we look at each move and find its TFR by adding up all the rewards that came after that move. In Figure 21-18 the calculations aren't very interesting, since all the immediate rewards except the last are zero. But it's worth seeing the steps here, as we'll have nonzero immediate rewards later on.

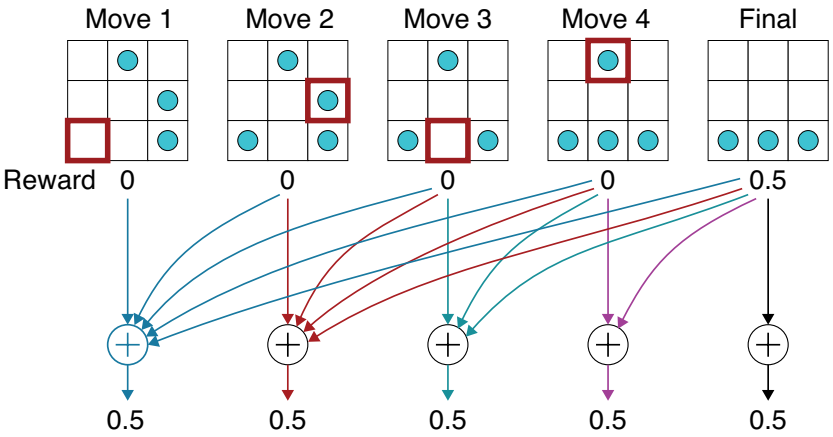


Figure 21-18: Finding the TFR for each move. We add up the immediate reward for each move (shown directly underneath it) with the immediate rewards for all following moves. In our game where every immediate reward is zero except for the final reward, these sums are all the same.

We then use our simple update rule and the list of moves we made, and plunk each action's TFR into the cell of the L-table corresponding to that action for that board, as shown in Figure 21-19.

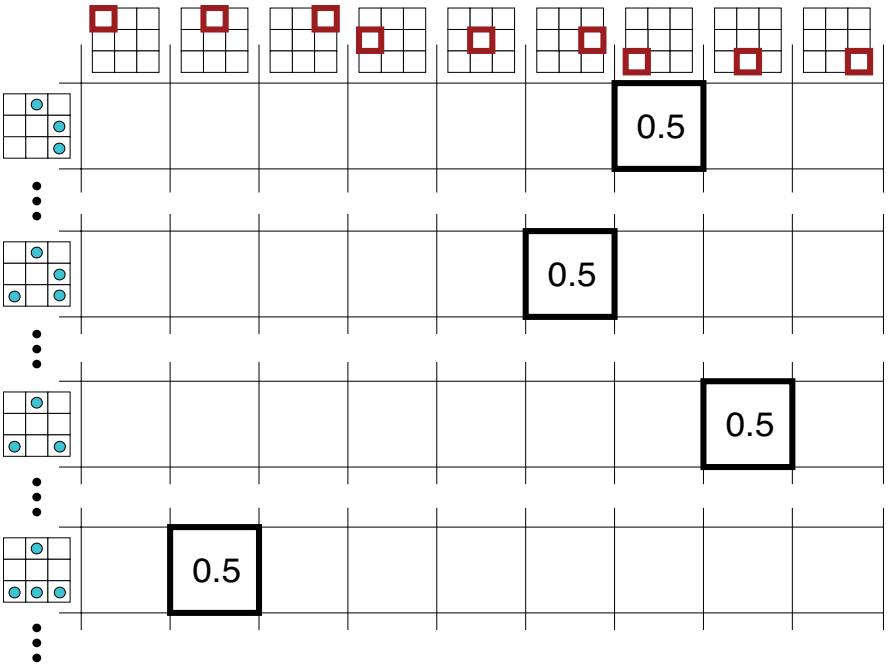


Figure 21-19: Updating our L-table with the new TFR for each action we took in the game. We find the row corresponding to the board we were looking at when we took each action and the column corresponding to the action we made. The new TFR replaces whatever was in that cell before.

If we want to learn some more, we go back up to the start of the process and play a new game. When we're done, we compute a TFR value for each action we played and store that in its corresponding cell (overwriting whatever was there before). Note that we don't reset the L-table after each game, though, so it gradually fills up with TFRs as we play more episodes.

When it's time to stop training and start playing, we use the L-table to pick our moves. That is, at each move, we're presented with a board, so we find that row of the table, pick the largest L-value in that row, and select the action corresponding to that column.

Testing Our Algorithm

Let's see how well our system works. Let's start out by playing 3,000 episodes of Flippers from start to finish so the L-table can get filled up pretty well. Figure 21-20 shows a game of Flippers played from start to finish after these 3,000 episodes of training. It's not a very nice result. There's a simple two-move solution that any human would spot: flip the left-middle cell, and then the upper-left cell (or do it in the other order). Instead, our algorithm seems to meander randomly until it finally stumbles on a solution after six moves.

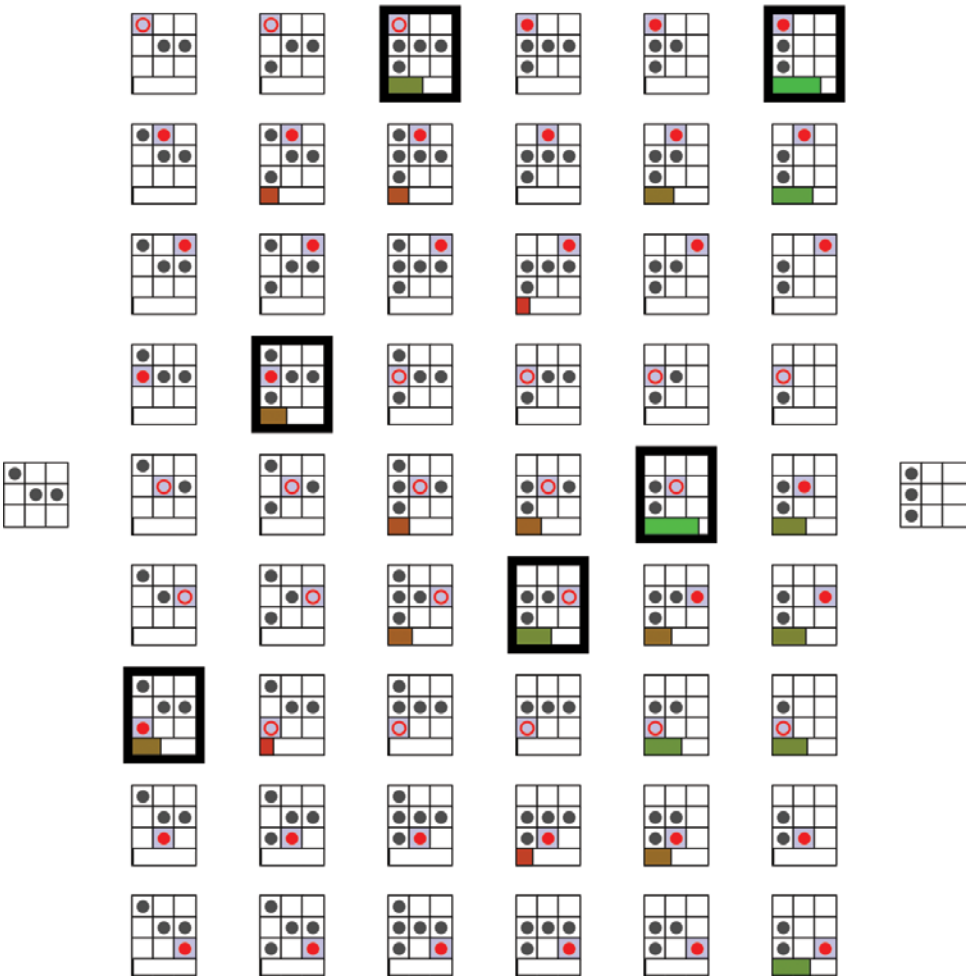


Figure 21-20: Playing a game of Flippers after 3,000 episodes of training with the L-table algorithm. Read the game left to right.

The arrangement shown in Figure 21-20 shows rows of the L-table arranged as columns to better fit the page. Each column represents one board configuration (or state). The nine possible actions are shown in each row, highlighted in red. The thick black outline shows the action that the agent selected from that list, leading to the new board in the column to its right. The shaded cell shows the action taken. If the move causes a dot to appear, the move is shown as a solid red dot. If the move causes the dot to go away, it's shown as an outlined red dot. The colored bar below each board shows its L-value from the table. Larger and greener bars correspond to larger L-values calculated with discounted future rewards.

Boards near the right have larger L-values than those near the left. That's because those boards were sometimes the randomly chosen starting board for a game. If we picked a good move and won immediately, or in just a few moves, the final reward was large.

Returning to this game, starting from the position on the far left, the algorithm's first move was to flip the cell in the lower left, introducing a new dot. From that result, it then flipped the square in the middle of the left-most column, again introducing a dot. From that position it then flipped the upper-left square, removing the dot that was there. The game continued in this way until it found a solution.

We'd expect our algorithm to improve with more training, and it does. Figure 21-21 shows the same game as Figure 21-20 after doubling the length of the training run to 6,000 episodes.

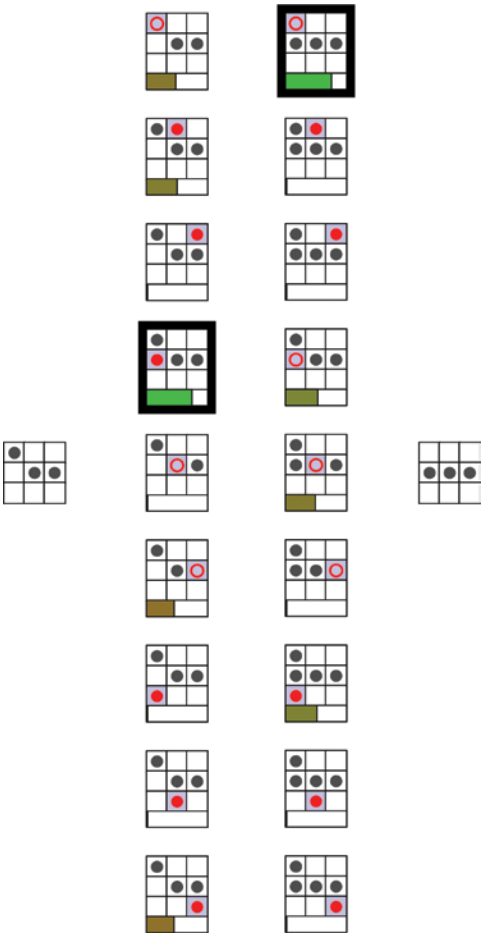


Figure 21-21: The same game as Figure 21-20 after 6,000 episodes of training

This is very nice. The algorithm found the easy answer and went right for it.

We seem to have created a great algorithm for learning and playing. So why did we label everything with "L" for lousy? It seems to be working just fine.

It *is* just fine, as long as the environment remains completely predictable. Remember that earlier in this chapter we discussed unpredictable

environments. In reality, most environments are unpredictable. Logic-based single-player games, such as the Flippers game we've been looking at, are one of the few activities that are completely deterministic. If our goal is to play only single-player games in completely deterministic environments where we are able to execute every intended move perfectly and the environment responds identically every time, then this algorithm isn't so lousy. But such deterministic games and environments are rare. For example, as soon as there's a second player, there's uncertainty, and the game becomes unpredictable. In any situation in which the environment is not perfectly deterministic, the L-learning algorithm flounders.

Let's see why, and then we will see how to fix it.

Handling Unpredictability

Because we don't have an opponent when playing Flippers on the computer, we have a completely deterministic system. Every time we make a move, we are guaranteed to get back the same result. But in the real world, even single-player activities can have unpredictable events. Video games throw random surprises at us, a lawnmower can hit a rock and jump to the side, or an internet connection can stutter and cause us to miss making the winning bid in an auction.

Since handling unpredictability is so important, let's introduce some artificial randomness into Flippers and see how our L-learning algorithm responds. Our model of randomness takes the form of a big truck that drives by our playing area every now and then, shaking our board. Sometimes it's enough to cause one or more random tiles to spontaneously flip over. Of course, we still want to play good games and win, even in the face of such surprises, but our L-learning system is helpless in the face of this kind of event.

It's the combination of our policy and update rule that causes trouble. Remember that before we start learning, each row starts out with all zeros. When a training game is finally won, every action gets the same score, based on the length of the game, as we saw in Figure 21-19. As we continue to play our training games, the next time we come to that board, we pick the cell with the largest value.

Suppose we're in the midst of a training game. We're looking at a board that we once received as a starting board and we won it in two moves. The L-table values for each of those moves have large scores, so we select the high-scoring move, preparing to win on the next flip. But just after our first move, the big truck comes rumbling by, shaking our board and flipping a tile. Playing from this board forward, we end up requiring lots of moves before winning. This means that the TFR that ultimately comes from playing that action is less than if the truck hadn't come by.

And here's the problem: that smaller value overwrites the previous value in every cell that led to this long game. In other words, because of that event, every action we played sees its L-value lowered. In particular, that great starting move that led to victory in just one more move now has a low score. When we encounter this board again in a later game, we might find that one of the other cells has a larger value than the cell that formerly held the great move.

The result is that this one-time random event causes us to stop making the best move we've found up to that point. We have "forgotten" that this was a great move, because a random event turned it into a bad move once. That low score makes it unlikely that we'll ever choose that move again.

Let's see this problem in action. Figure 21-22 shows an example where there are no unpredictable events. We start at the top with a board with three dots, and we find that the largest value in that row of the table is 0.6, corresponding to a flip of the center square. We make that move, and supposing the next move is also well-chosen, we have a victory in two moves, as shown in the center row. The reward of 0.7 replaces the 0.6 that was there for our first move, cementing that move's status as the one to make. Everything went right.

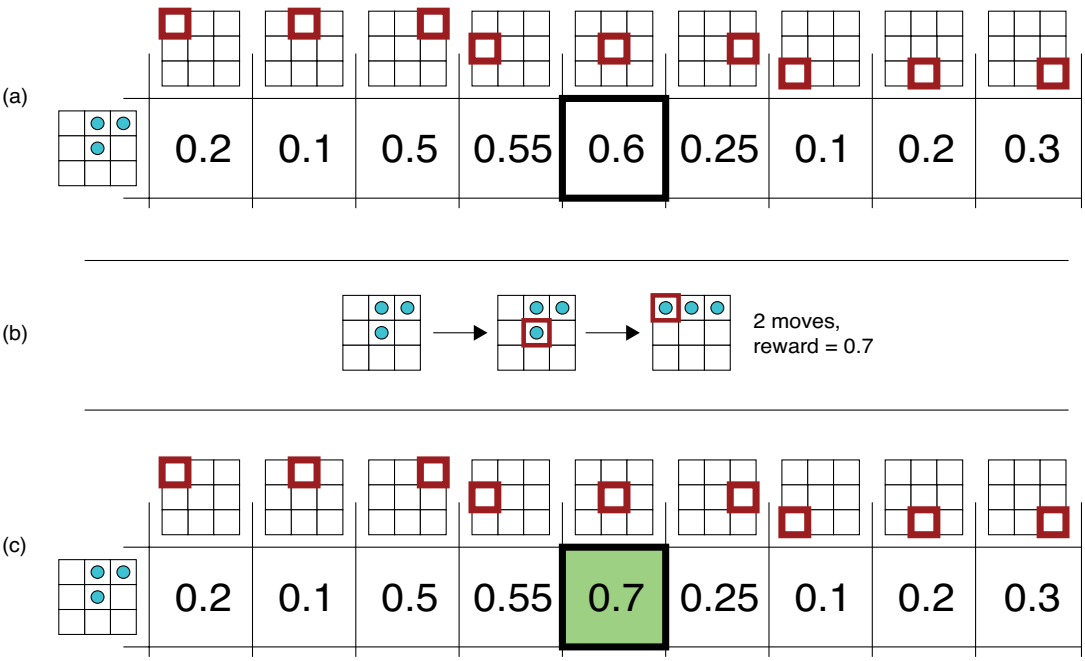


Figure 21-22: When there are no surprises, our algorithm works well. (a) The row of the L-table for the starting board. (b) The game plays out and is won in just a total of two moves. (c) The value of 0.7 overwrites the previous value for all table entries that led to this success.

In Figure 21-23 we introduce our rumbling truck. Just after we flip the center tile, the truck shakes the board and the bottom-right tile flips. This puts us on a whole new path. Let's suppose the algorithm finally finds victory after four more moves. The total is five moves, and the reward of 0.44 is placed in every cell that led to this success.

This is terrible. In one quick stroke, we have "forgotten" our best move. In this example, two other actions now have better scores. The next time we come to this board, the cell with score 0.55 will be picked, which will not place us one move away from victory as before. In other words, our best move is now forgotten, and we're going to always play a worse move.

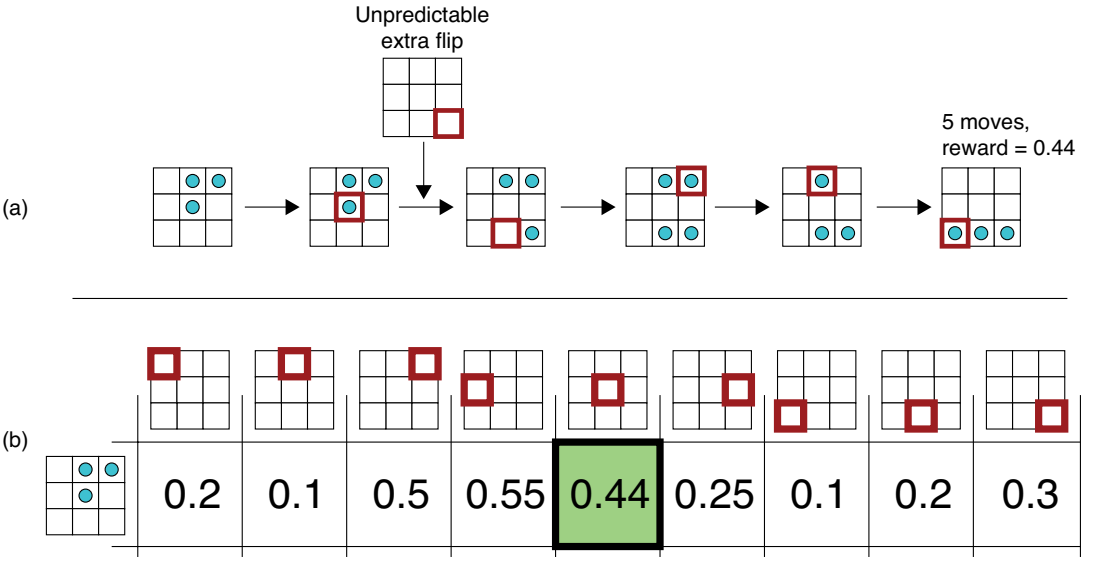


Figure 21-23: (a) When a truck rumbles by, it flips the lower-right square, causing the game to take five moves to win. (b) The new reward of 0.44 overwrites the old value of 0.6. This cell is no longer the highest-scoring cell in the row.

Recall that we said that occasionally during training we'll pick one of the cells in the row at random, just to explore what might happen. So someday we might make a new choice, or the truck might rumble by again and help us remember this cell, but that might not happen for a long time. And by the time the truck does come by and sets this move right again, others will have gone wrong. The L-table is almost always inferior to what it ought to be, and thus, on average, games powered by L-learning are longer and we get lower rewards. One surprise caused us to forget how to play this board well.

That's why we called this algorithm lousy.

But all is not lost. We looked at this algorithm because the lousy version can be improved. Most of the algorithm is fine. We only need to fix how it fails in the face of unpredictability. From now on, we assume that when we play Flippers, that big truck may come thundering along, creating unpredictability in the form of occasionally flipping a random tile. In the next section, we'll see how to handle this kind of unpredictable event gracefully and produce an improved learning algorithm that works well.

Q-Learning

Without too much effort, we can upgrade L-learning to a much more effective algorithm that is commonly used today, called *Q-learning* (the Q is for quality) (Watkins 1989; Eden, Knittel, and van Uffelen 2020). Q-learning

looks a lot like L-learning, but, naturally, it instead fills up Q-tables with Q-values. The big improvement is that Q-learning performs well in stochastic, or unpredictable, environments.

To get from L-learning to Q-learning we make three upgrades: we improve how we compute new values for Q-table cells, how we update existing values, and the policy we use for choosing an action.

The Q-table algorithm starts with two important principles. First, we *expect* uncertainty in our results, so we build it in from the start. Second, we work out new Q-table values as we go, rather than waiting for the final reward. This second idea lets us work with games (or processes) that go on for a very long time, or perhaps never reach a conclusion (like scheduling elevators). By updating as we go, we're able to develop our table of useful values even if we never get a final reward.

To make this work, we need to also upgrade the environment's super simple rewarding process from the last section. Rather than always rewarding zero except for the final move, the environment will instead return immediate rewards that estimate the quality of each action as soon as it's taken.

Q-Values and Updates

Q-values are a way to approximate the total future reward even when we don't know how things are going to end up. To find a Q-value, we add together the immediate reward, plus all the other rewards that are yet to come. So far, that's nothing more than the definition of the total future reward. The change is that now we find the future rewards by using the reward from the next state.

In Figure 21-17, we saved four pieces of information for every move: the starting state, the action we chose, the reward we got, and the new state that action landed us in. We saved that new state so that we could use it now, where we will use it to compute the rest of the future rewards.

The key insight is to notice that our next move begins with that new state, and by following our policy we will always select the action whose cell has the greatest Q-value. If that cell's Q-value is the total future reward for *that* action, then adding together that cell's value with our immediate reward gives us the current cell's total future reward. This works because our policy guarantees us that we always pick the cell with the largest Q-value for any given board state.

If multiple cells in the next state share the maximum value, then it doesn't matter which one we pick when we get there. All we care about now is the total future reward that comes from the next action.

Figure 21-24 shows this idea visually. Note that the value we compute in this step isn't the final Q-value, but it's almost there.

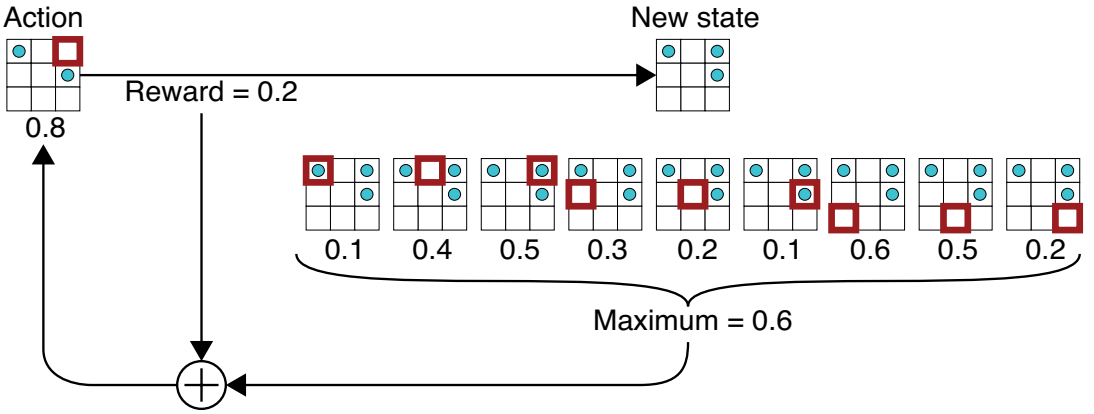


Figure 21-24: Part of the process for computing a new Q-value for a cell. The new value is the sum of two others. The first value is the immediate reward for taking the action that cell corresponds to, here 0.2. The second value is the largest Q-value of all the actions belonging to the new state, here 0.6.

The step that's missing is where Q-learning accounts for randomness. Rather than use the value in the cell for our next action, we use the discounted value of that cell. Recall that this means we multiply it by our discount factor, a number from 0 to 1, often written as γ (gamma). As we discussed earlier, the smaller the value of γ, the less certain we are that unpredictable events in the future won't change this value. Figure 21-25 shows the idea.

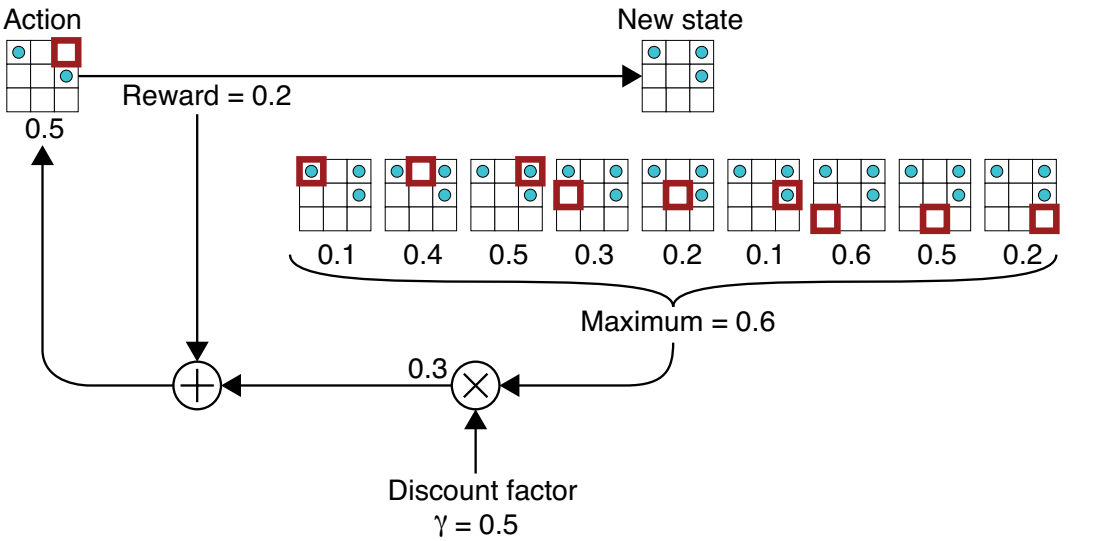


Figure 21-25: To find the Q-value, we modify Figure 21-24 to include the discount factor γ, which reduces the future rewards based on how confident we are that they won't be changed by future, unpredictable events.

Note that the many multiplications in the discounted future reward shown in Figure 21-8 are automatically handled by this scheme. The first multiplication is included here explicitly. The multiplication for the states beyond that are accounted for when the Q-values in the cells for the next state are evaluated.

Now that we've calculated a new value, how do we update the current value? We saw during L-learning that simply replacing the current value with the new one is a poor choice in the face of uncertainty. But we want to update the cell's Q-value in some way, or we'll never improve.

The Q-learning solution to this puzzle is to update the new cell's value as a blend of the old and new values. The amount of blending is left up to us as a parameter that we specify. That is, the blend is controlled by a single number between 0 and 1, usually written as the lowercase Greek letter α (alpha). At the extreme value of $\alpha = 0$, the value in the cell doesn't change at all. At the other extreme value of $\alpha = 1$, the new value replaces the old one, as in L-learning. Values of α between 0 and 1 blend, or mix, the two values, as shown in Figure 21-26.

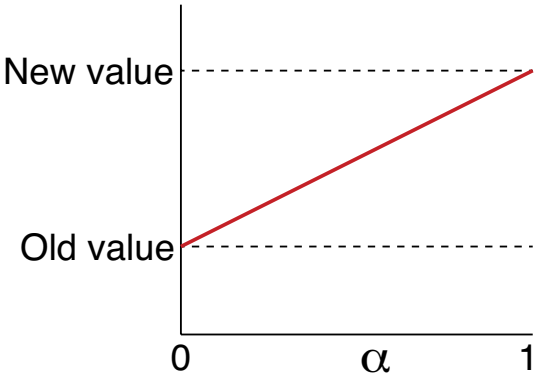


Figure 21-26: The value of α lets us blend smoothly from the old value (when α = 0) to the new value (when α = 1), or any value in between.

The parameter α is called the *learning rate*, and it's left up to us to set it. It's unfortunate that this is the same term that's used by the update step of backpropagation, but usually context makes it clear which type of "learning rate" we're referring to.

In practice, we usually set α to a value close to 1, such as 0.9 or even 0.99. These values near 1 cause the new values to dominate the value stored in the cell. For instance, when $\alpha = 0.9$, the new value stored in the cell is 10 percent of the old value, and 90 percent of the new value. But even a value of 0.99 is very different than 1, because remembering even 1 percent of the old value is often enough to make a difference.

Using our value for α, we run our system through some training and see how it does. Then we can adjust the value based on what we see and try

again, repeating the process until we've found the value of α that seems to work best. We usually automate this search so we don't have to do it ourselves.

The elephant in the room is that this whole argument has been based on having the correct Q-values in the next state, even before we get there. But where did they come from? And if we have the correct Q-values already, then why do any of this in the first place?

These are fair questions, and we will return to them after we look at the new policy rule.

Q-Learning Policy

Recall that the policy rule tells us which action to select when we're given a state of the environment. We use this policy while learning, and later, when playing actual games. The policy we used in L-learning was to usually select the action with the highest L-value in the row of the table corresponding to the current board. That makes sense, since we've learned that this is the action that brings us the highest rewards. But this policy doesn't explicitly address the explore or exploit dilemma. In an unpredictable environment, the move that brings us the best rewards sometimes may not bring us the best reward at other times. And completely untried moves can be far better, if only we give them a chance.

Still, we don't want to pick moves at random, because we do want to favor the ones that we know lead to high rewards. We just don't want to do that every time. Q-learning picks a middle road. Instead of always picking the action with the highest Q-value, we *almost* always pick the action with the highest Q-value. The rest of the time we pick one of the other values. Let's look at two popular policies for doing this.

The first approach we'll look at is called *epsilon-greedy* or *epsilon-soft* (these refer to the Greek lowercase letter ε, epsilon, so they sometimes appear as ε-greedy and ε-soft). The algorithms are almost the same. We pick some number ε between 0 and 1, but usually it's a small number quite close to 0, such as 0.01 or less.

Each time we're at a row and ready to choose an action, we ask the system for a random number between 0 and 1, chosen from a uniform distribution. If the random number is greater than ε, then we proceed as usual and pick the action with the greatest Q-value in the row. But in that occasional case when the random number is less than ε, we select an action at random out of all the other actions in the row. In this way, we usually pick the most promising choice, but infrequently, we select one of the other actions and see where it leads us. Figure 21-27 shows this idea graphically.

The other policy we'll look at is called *softmax*. This works in a way similar to the softmax layer that we discussed in Chapter 13. When we apply softmax to the Q-values in a row, they are transformed in a complex way so that they add up to 1. This lets us treat the resulting values as a discrete probability distribution, and then we randomly select one of entries according to those probabilities.

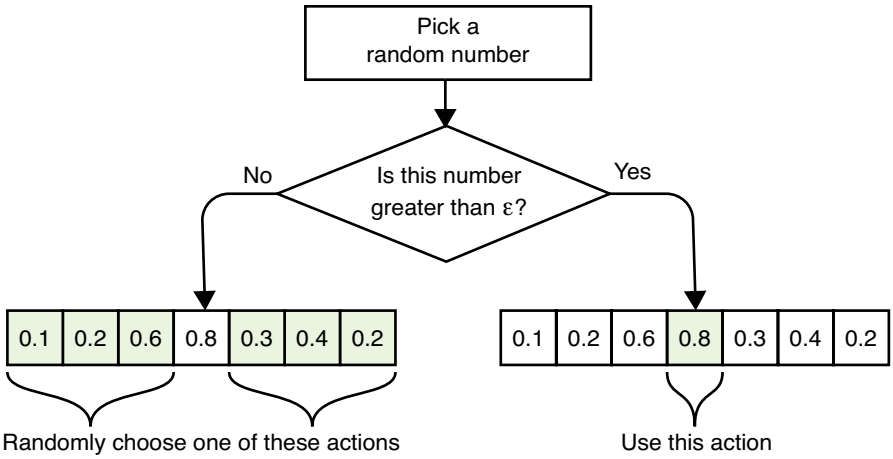


Figure 21-27: The epsilon-greedy policy

In this way, we usually get the action with the largest score. Infrequently, we get the value with the second-highest score. Even less frequently, we get the value with the third-highest score, and so on. Figure 21-28 illustrates the idea.

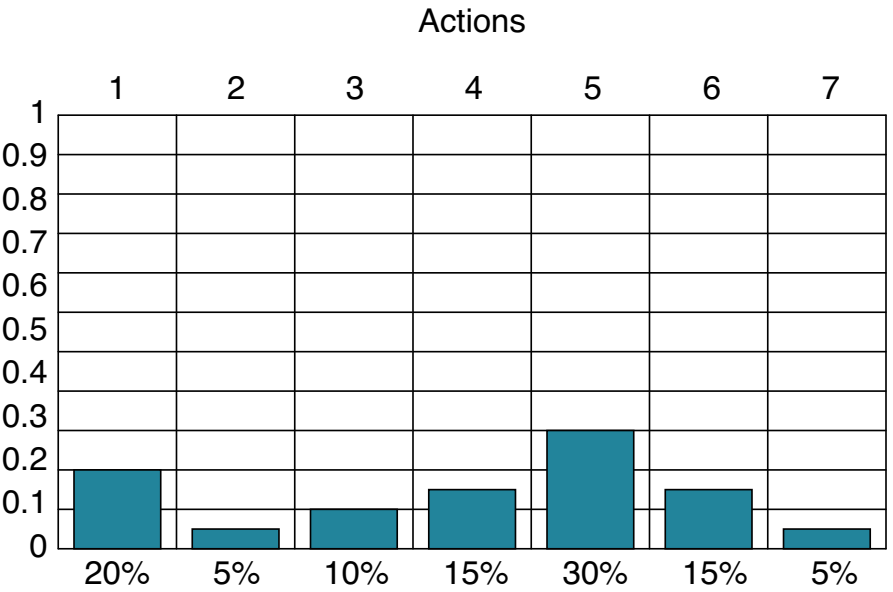


Figure 21-28: The softmax policy for picking an action temporarily scales all the actions in the row so that they add up to 1.

An attractive quality of this scheme is that the probabilities of choosing each action always reflect the most current Q-values of all the actions associated with a given state. So as the values change over time, so too do the probabilities of picking the actions.

The particular calculations carried out by softmax can sometimes lead to the system not settling down on a good set of Q-values. An alternative is the *mellowmax* policy, which uses slightly different math (Asadi and Littman 2017).

Putting It All Together

We can summarize the Q-learning policy and update rule in a few words and a diagram. In words, when it's time for a move, we use the current state to find the appropriate row of the Q-table. We then select an action from that row according to our policy (either epsilon-greedy or softmax). We take that action and get back a reward and a new state. Now we want to update our Q-value to reflect what we've learned from the reward. We look at the Q-values in that new state and select the largest one. We discount that by how much we think the environment is unpredictable, add it to the immediate reward we just got, and blend that new value with the current Q-value, producing a new Q-value for the action we just took, which we save.

Figure 21-29 summarizes the process.

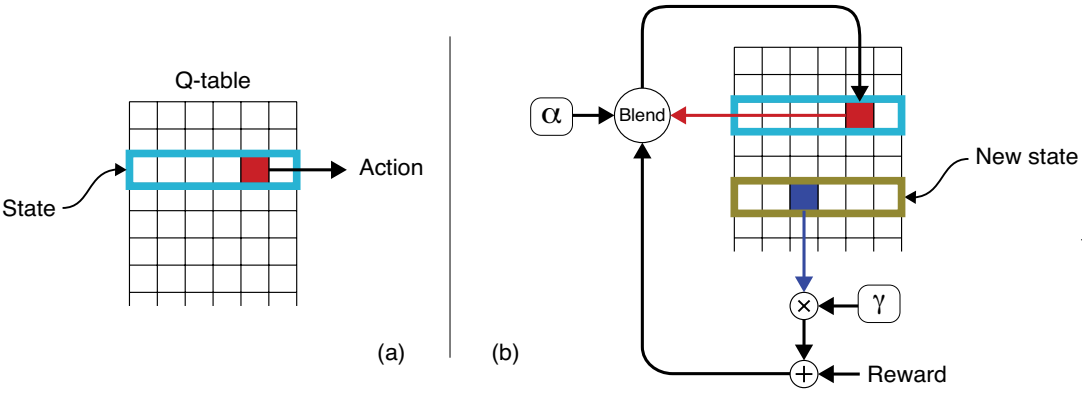


Figure 21-29: The Q-learning policy and update procedure. (a) Choosing an action. (b) Finding a new Q-value for that action.

When we start a move, shown in Figure 21-29(a), we look at the Q-table row for the current state, and use our policy to pick an action, here shown in red. That action is communicated to the environment. The environment responds with a reward, and a new state. As in Figure 21-29(b), we find the row of the Q-table corresponding to the new state and select the largest reward there (this assumes that we're going to pick the largest action when we get to that new state, which we know won't always be the case. We will return to this issue soon). We discount this reward by multiplying it by γ, and then we add it to the immediate reward for this move, giving us a new value for the action we originally chose. We blend the old value and new value using α, and that new value is placed into the original action's cell in the Q-table.

The best values for the policy parameter ε, the learning rate α, and the discount factor γ have to be found by trial and error. These factors depend intimately on the specific nature of the task we're performing, the nature of the environment, and the data we're working with. Experience and intuition often give us good starting points, but nothing beats traditional trial-and-error to find the best values for any particular learning system.

The Elephant in the Room

Earlier we promised to return to the problem that we needed to have accurate Q-values in order to evaluate the update rule, but those values themselves were computed by the update rule using the values that came after them, and so on. Each step seems to depend on the data from the following step. How can we use data that we haven't created yet?

Here's the beautiful, simple answer to that problem: we ignore it. Incredibly enough, we can initialize the Q-table with all zeros, and then start learning. In the beginning, the system makes moves erratically because there's nothing in the Q-table to help it pick one cell over another. It picks one of the cells at random and plays that move. All of the actions in the resulting state are also zero, so the update rule, no matter what values we use for α and γ, keeps the cell's score at zero.

Our system plays games that look chaotic and foolish, making terrible choices and missing obvious good moves. But eventually, the system stumbles onto a victory. That victory gets a reward of a positive number, and that reward updates the Q-value of the action that led to it. Sometime later, an action that led us to that action incorporates some of that great reward, because of the step in Q-learning that looks ahead to the next state. That ripple effect continues to slowly work backward through the system, as new games fall into the states that lead to states that previously led to victory.

Note that the information isn't actually moving backward. Every game is played from beginning to end, and every update is made immediately after each move. The information seems to move backward because Q-learning involves the step of looking forward one move when evaluating the update rule. The score from the next move is able to influence the score for this one.

At some point, thanks to our policy that sometimes tries out new actions, every move eventually leads to a path to victory, and those values also influence earlier and earlier actions. Eventually the Q-table fills up with values that accurately predict the rewards of each action. Further playing serves to only improve the accuracy of those values. This process of settling into a consistent solution is called *convergence*. We say that the Q-learning algorithm *converges*.

We can prove mathematically that Q-learning converges (Melo 2020). This kind of proof guarantees that the Q-table gradually gets better. What we can't say is how long that will take. The larger the table, and the more unpredictable the environment, the longer the training process requires. The speed of convergence also depends on the nature of the task the system is trying to learn, the feedback provided, and, of course, our chosen

values for the policy variable ε, the learning rate α, and the discount factor γ. As always, there's no substitute for trial-and-error experimentation to learn the specific idiosyncrasies of any particular system.

Note that the Q-learning algorithm very nicely addresses two of the problems we discussed earlier. The credit assignment problem asks us to make sure that the moves that lead up to a victory are rewarded, even when the environment isn't providing that reward. The nature of the update rule takes care of this, propagating the rewards for successful moves backward from the final step that led to victory all the way back to the very first move. The algorithm also addresses the explore or exploit dilemma by using epsilon-greedy or softmax policies. They both favor choosing actions that have proven to be successful (exploitation), but they also sometimes try the other actions just to see what might come of them (exploration).

Q-learning in Action

Let's put Q-learning to work and see if it can learn how to play Flippers in an unpredictable environment. One way to measure the algorithm's performance is to have the trained model play a large number of random games and see how long they take. The better the algorithm has gotten at finding good moves and eliminating bad ones, the fewer moves each game should require before reaching victory.

The longest well-played game is the one that starts out with all nine cells showing a dot. Then we have to flip six cells to get to a victory. So, we'd like to see our algorithm win every game in six moves or less.

To see the effect of training on the algorithm, let's look at plots of the lengths of a large number of games for different amounts of training. Our plots show the results of playing games that start with each of the 512 possible patterns of dots and blanks, in an environment with a considerable degree of unpredictability. We played 10 games for each starting board, for a total of 5,120 games. We cut off any game that ran for more than 100 steps.

We set α to 0.95, so each cell retained just 5 percent of its old value when it was updated. This way we don't completely lose what we've learned before, but we are expecting new values to be better than old ones, since they'll be based on improved Q-table values when they pick the next move. To select moves, we used an epsilon-greedy policy with a relatively high ε of 0.1, encouraging the algorithm to seek out new moves 1 time out 10.

We introduced a lot of unpredictability by simulating our random truck coming by after each move with a probability of 1 in 10, flipping over a single random tile each time. To account for this, we set the discount factor γ to 0.2. This low value says we're only 20 percent sure that the future will play out the same way each time because of the influence of those random events. We set this higher than the noise level we know the truck introduces (10 percent), because we expect that most well-played games will only be three or four moves long, so they are less likely to see a random event than a game of 10 or more moves.

These values of α, γ, and ε are all basically informed guesses. In particular, γ was chosen based on our knowledge of how often random events would occur, which we rarely know ahead of time. In a real situation we'd experiment with our parameters to find what works best for this game and this amount of noise.

Figure 21-30 shows the game lengths after training for just 300 games. The algorithm already found a lot of quick wins.

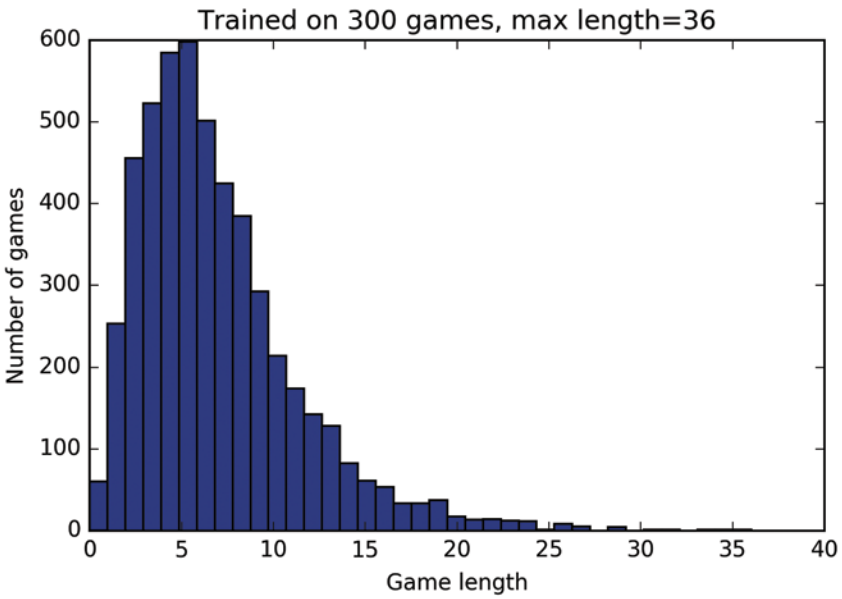


Figure 21-30: The number of games that required from 0 to 40 moves to win (we played each of the 512 starting boards 10 times) using a Q-table that had been trained for 300 games

The "instant wins" are in the first column, corresponding to zero moves. These are games whose starting boards already have just three dots, arranged in a vertical column or horizontal row. Since there are six possible winning game configurations, and we ran through all the possible board configurations 10 times each, we started with a winning board 60 times.

Since no game in Figure 21-30 hit our 100-move cutoff, we can see that the algorithm never fell into a long-lived loop. A loop might just be two states alternating forever, or a long string of them that wraps back around on itself. Loops are possible in Flippers, and there's nothing in the basic Q-learning algorithm that explicitly prevents the system from getting into a loop.

We might say that the system "discovered" that loops don't get to victory and thus don't bring any rewards, so it learned to avoid them. If at some point it did return to a previously visited state, either by making that move

or as the result of a randomly introduced flip, the relatively high value of ε meant it had a good chance of eventually picking a new action and thereby going off in a new direction.

Let's raise the number of training games to 3,000, as in Figure 21-31.

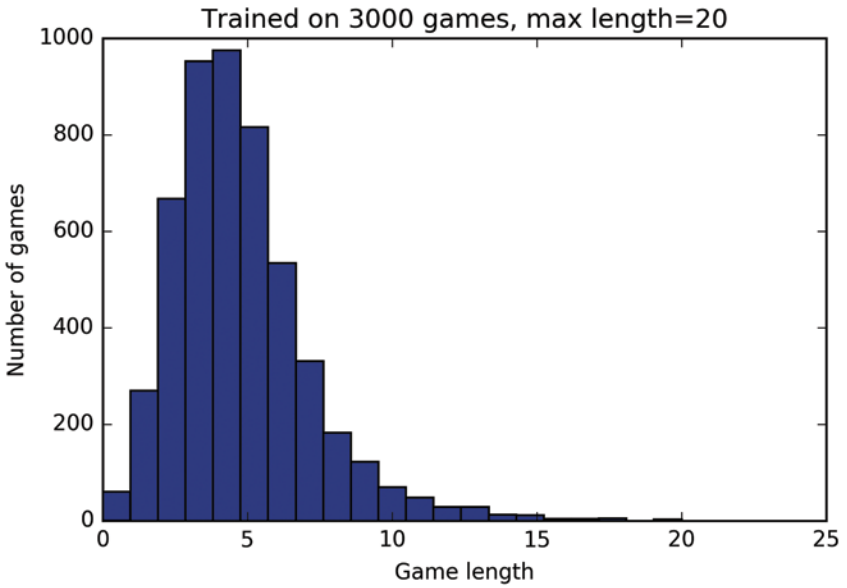


Figure 21-31: The number of games of different lengths resulting from playing 5,120 games, based on a Q-table trained by playing 3,000 games

The algorithm has learned a lot. The longest game is now just 20 moves, with most games being won in 10 moves or less. It's nice to see the denser clustering around four and five moves.

Let's look at a typical game played after these 3,000 episodes of training. Figure 21-32 shows the game, played left to right. The algorithm took eight moves to win.

Figure 21-32 is not an encouraging result. Just by looking at the starting board, we can see at least four different ways to win this game in four moves. For example, flip the lower-left square and then flip the three dots in the middle and rightmost columns. But our algorithm seems to be flipping over tiles at random. It eventually stumbles onto a solution, but it's definitely not an elegant result.

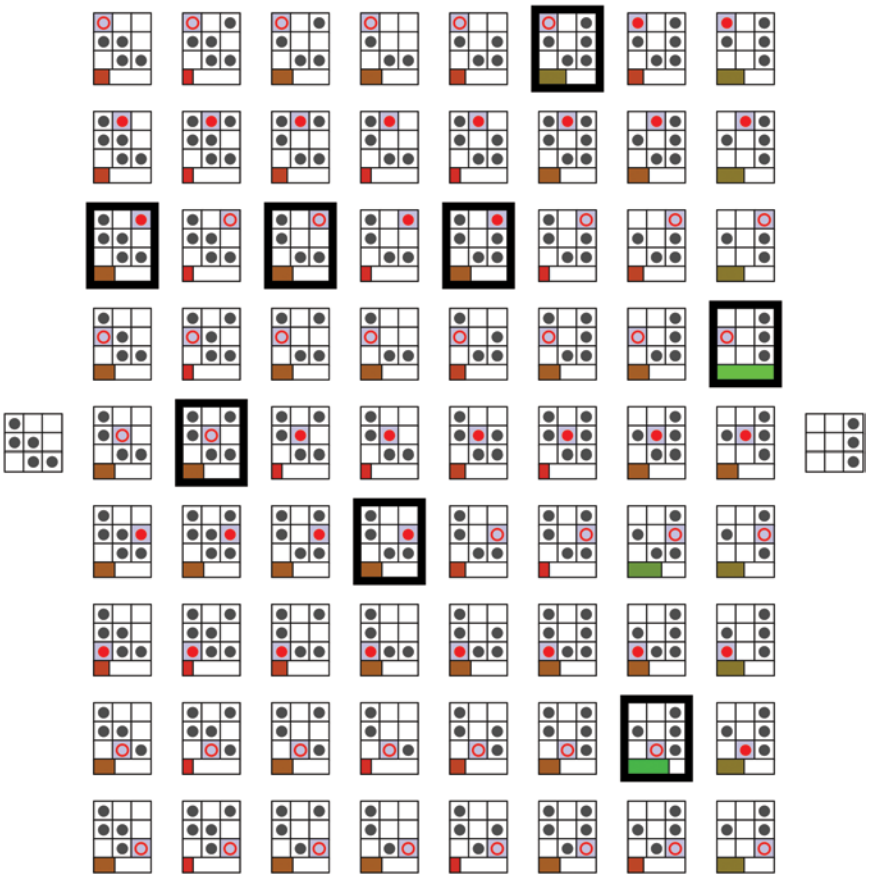


Figure 21-32: Playing a game of Flippers after training Q-learning for 3,000 episodes

If we train the algorithm for more episodes, we expect its performance to improve. After 3,000 more training episodes (for a total of 6,000), and looking at the number of games that required different numbers of moves, we get the results of Figure 21-33.

Compared to our results in Figure 21-31, after 3,000 games of training, the longest game has decreased from 20 moves to 18, and the shorter games of just 3 and 4 steps have become more frequent.

This chart suggests that the algorithm is learning, but how is it actually performing when it plays a game? In fact, the algorithm has taken a huge jump in ability.

Figure 21-34 shows the very same game as Figure 21-32, which required eight moves to win. Now it takes just four moves, which is the minimum number for this board (though there's more than one way to achieve it).

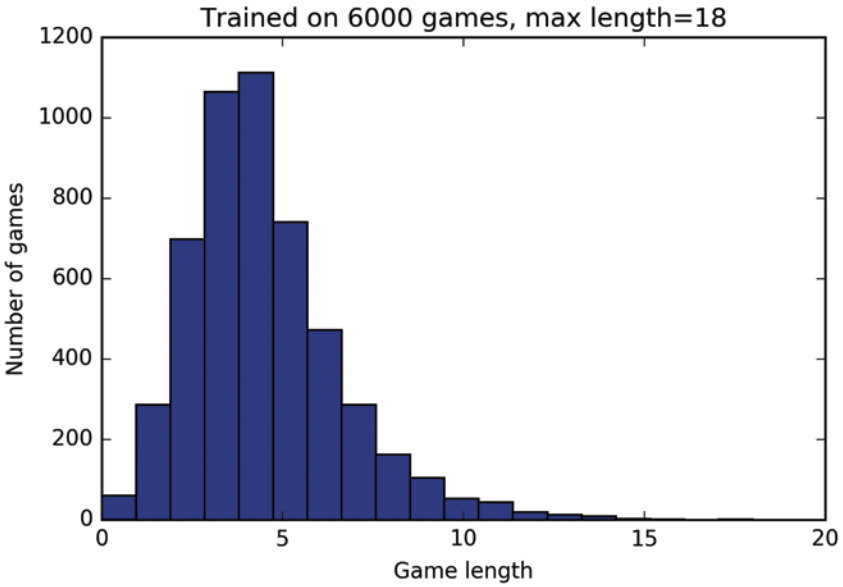


Figure 21-33: The number of games that required a given number of moves to win our 5,120 games after training the Q-table with 6,000 games

Q-learning has done remarkably well even in this highly unpredictable learning environment, where a tile is flipped over at random after 10 percent of the moves. It weathered that unpredictability and managed to find ideal solutions for most games, even with only 6,000 training runs.

SARSA

Q-learning does a great job, but it has a flaw that can reduce the accuracy of the Q-values that it relies on. It's the problem we referred to when discussing Figure 21-29, when we noted that we were basing our future reward on the score of the most likely next action, even though that's not necessarily the action that would be taken. In other words, the update rule *assumes* we're going to pick the highest-scoring action on our next move, and its calculations of the new Q-value are based on that assumption. This isn't an unreasonable assumption, because both our epsilon-greedy and softmax policies usually pick the most rewarding action. But the assumption is wrong when one of those policies chooses one of the other actions.

When our policy picks any action other than the one we used in the update rule, the calculation will have used the wrong data, and we end up with reduced accuracy in the new value that we compute for that action. Happily, we can fix that problem.

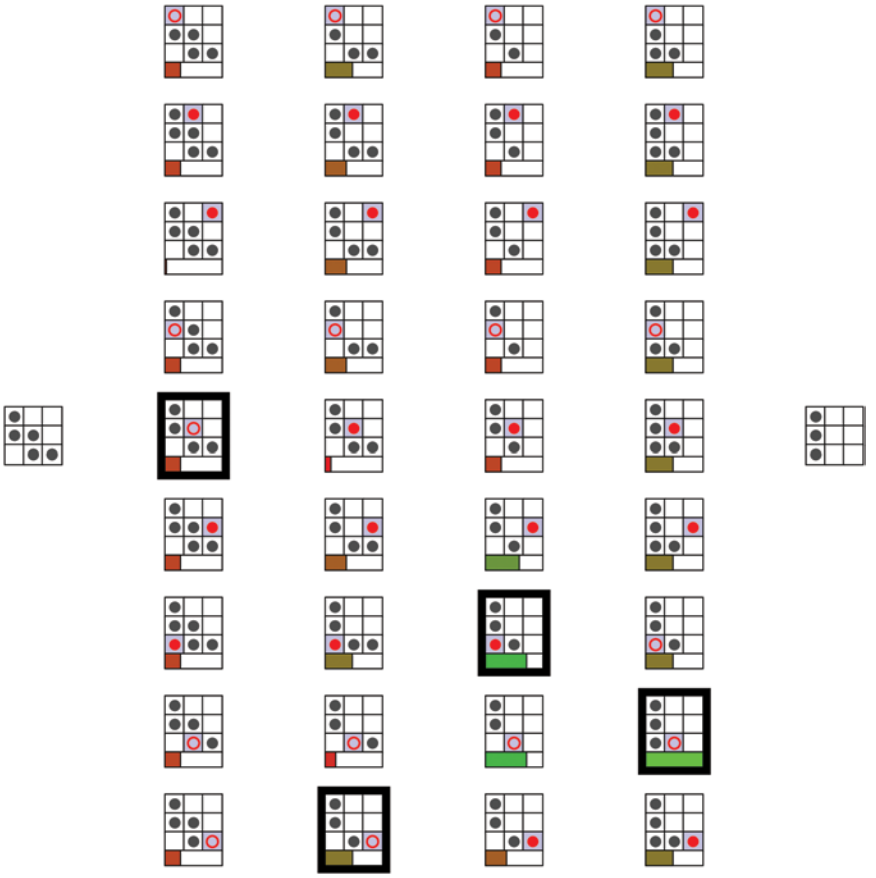


Figure 21-34: The game of Figure 21-32, solved more efficiently by Q-learning thanks to more training episodes

The Algorithm

It would be nice to keep all the virtues of Q-learning, but avoid making the mistake of calculating a move's Q-value by using the Q-value of the highest-scoring next action when there's a chance we won't actually select that action when we make our next move. We can do that by modifying Q-learning just a little, creating a new algorithm known as *SARSA* (Rummery and Niranjan 1994). This is an acronym for "state-action-reward-state-action." The "SARS" part we've had covered ever since Figure 21-17, when we saved the starting state (S), action (A), reward (R), and resulting state (S). What's new here is the extra action "A" at the end.

SARSA fixes the problem of choosing the wrong cell from the next state by choosing that next cell *with our policy* (rather than just selecting the

biggest one), and *remembering* the choice of action (that's the extra "A" at the end). Then when it's time to make our new move, we select the action that we computed previously and saved.

In other words, we've moved the time when we apply our action-choosing policy. Instead of choosing our action at the start of a move, we choose it during the previous move and remember our choice. That lets us use the value of the action we really will use when building the new Q-value.

Those two changes (moving the action-choosing step and remembering the action we chose) are all that differentiate SARSA from Q-learning, but they can make a big difference in learning speed.

Let's look at three successive moves using SARSA. The first move is shown in Figure 21-35. Because this is the first move, we use our policy to pick an action for this move in Figure 21-35(a). This is the only time we do this. Once we have our chosen action, we use our policy to pick the action for move two. We get a reward from the environment, and update the Q-value for the action we just chose, in Figure 21-35(b).

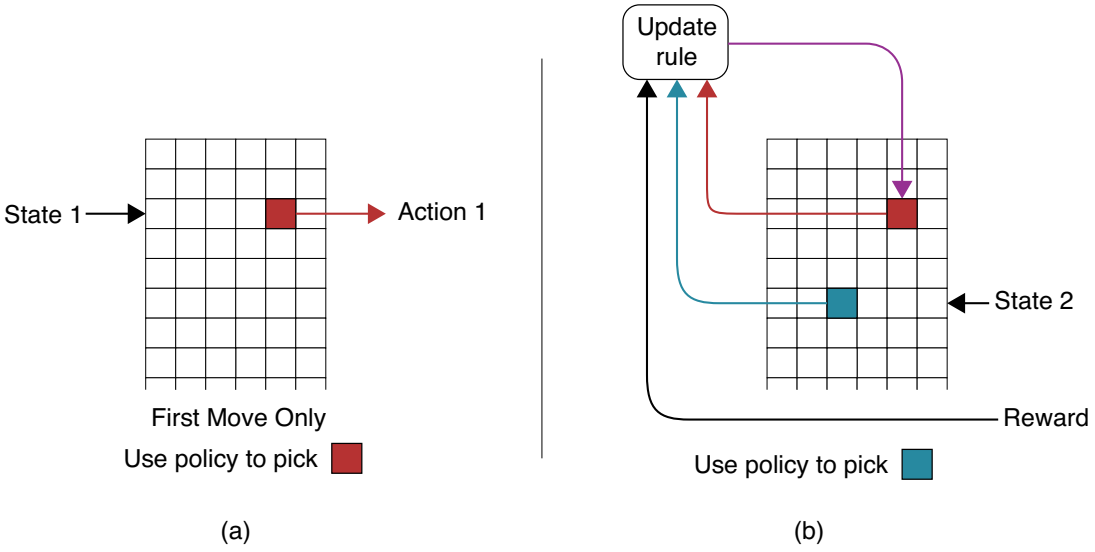


Figure 21-35: Using SARSA in the first move of our game. (a) We use our policy to pick the current action. (b) We also use our policy to pick our next action and update our current Q-value with the Q-value for that next action.

The second move is shown in Figure 21-36. Now we use the action we picked for ourselves last time and then pick the action we'll use in the third move once we get there.

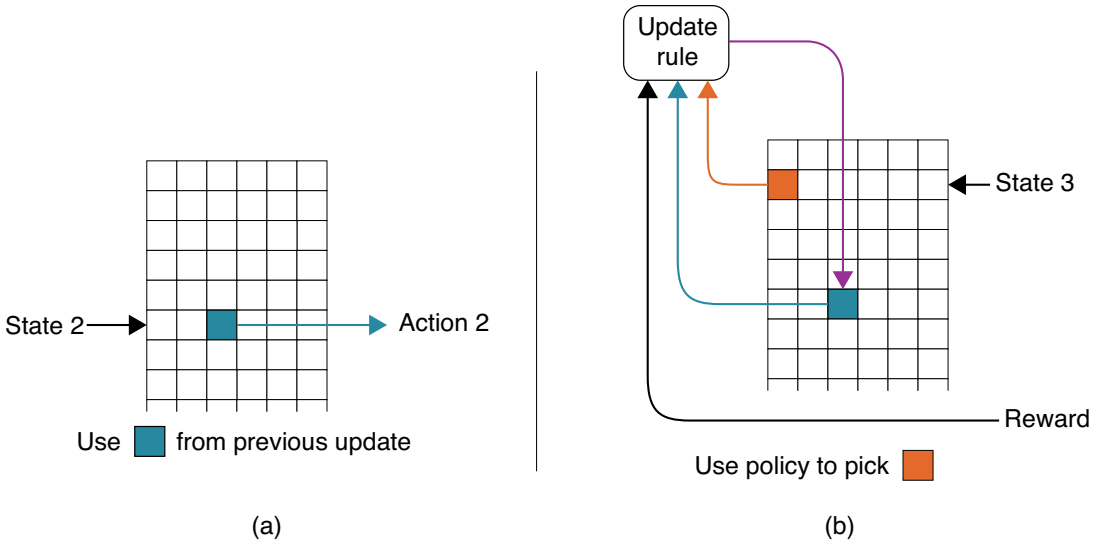


Figure 21-36: The second move using SARSA. (a) We make the action we picked for ourselves last time. (b) We pick the next action, and use its Q-value to update the current action's Q-value.

The third move is shown in Figure 21-37. Here again we take the previously determined action and work out the action for the next, fourth move.

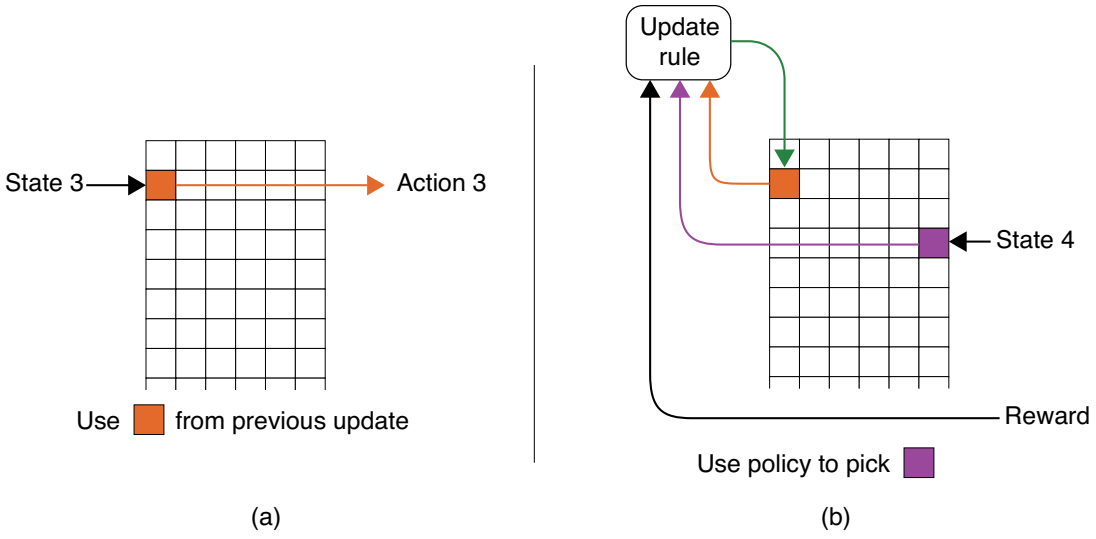


Figure 21-37: The third move using SARSA. (a) We take the action we determined during the second move. (b) We choose an action for the fourth move, and use its Q-value to improve the current action's Q-value.

Happily, we can prove that SARSA will also converge. As before, we can't guarantee how long it will take, but it usually starts producing good results sooner than Q-learning and improves them quickly after that.

SARSA in Action

Let's see how well SARSA plays Flippers, using the same approach we took to Q-learning. Figure 21-38 shows the number of moves required by our 5,120 games after 3,000 training episodes using SARSA. For this plot and those following, we continue to use the same parameters as for the Q-learning plots: the learning rate α is 0.95, we introduce a random flip with a probability of 0.1 after every move, the discount factor γ is 0.2, and we pick moves with an epsilon-greedy policy with ε set to 0.1.

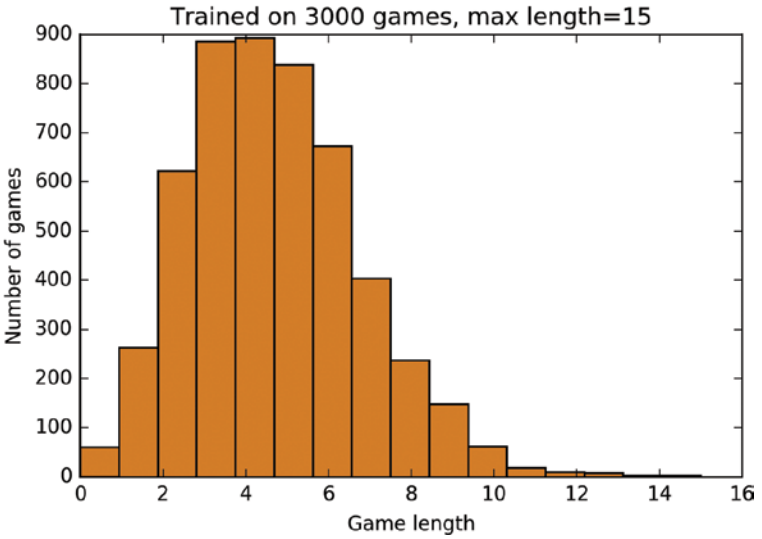


Figure 21-38: The lengths of 5,120 games using SARSA after training with 3,000 games. Note that only a few games required more than the maximum of six moves.

This is looking great, with most values clustered around 4. The longest game is only 15 steps, with very few longer than 8.

Let's look at a typical game. Figure 21-39 shows the game, played left to right. The algorithm needed seven moves to win. That's not terrible, but we know it can be solved more quickly.

As always, more training should result in better performance. As before, let's double our training to 6,000 episodes.

Figure 21-40 shows the lengths of our 5,120 games after 6,000 training episodes.

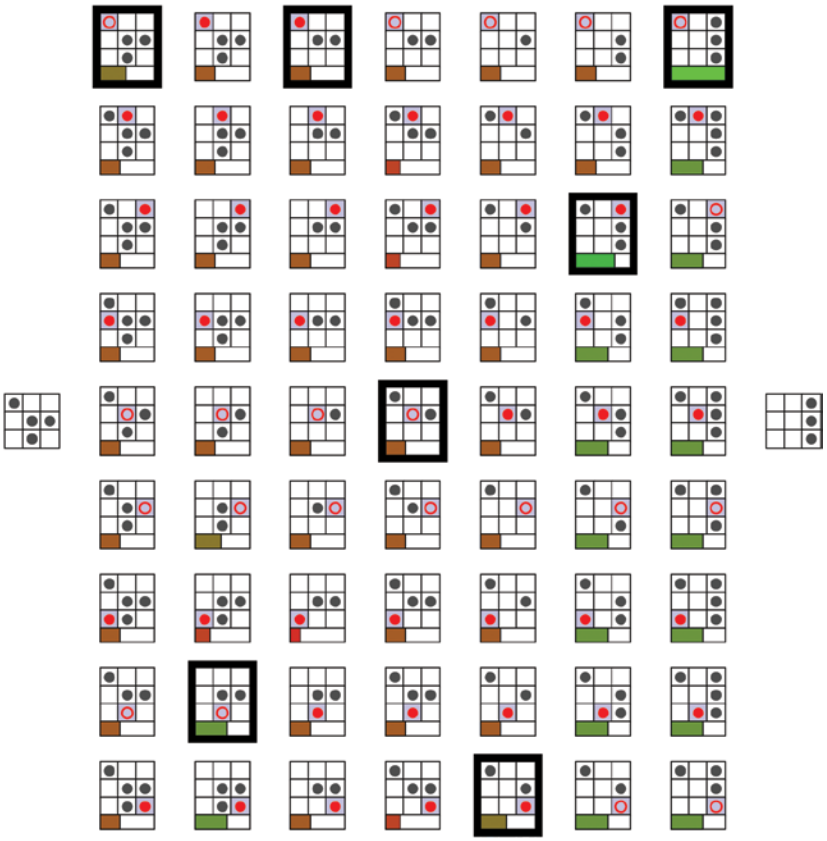


Figure 21-39: Playing a game of Flippers after 3,000 episodes of training to SARSA

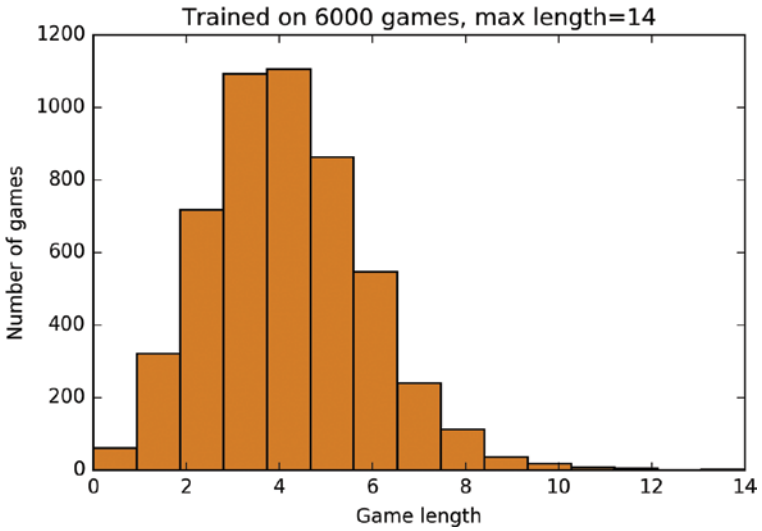


Figure 21-40: The lengths of our 5,120 games using SARSA after training for 6,000 games. Note how much shorter most of the games have become, and that none of the games got caught in a loop.

The longest game has gone down from 15 to 14, which isn't much to shout about, but the number of short games of lengths 3 and 4 is now even more pronounced. There weren't many games that required more than 6 moves.

Figure 21-41 shows the same game as Figure 21-39, which required 7 moves to win. Now it takes just 3 moves, which is the minimum for this board (though again, there's more than one way to win with just 3 moves).

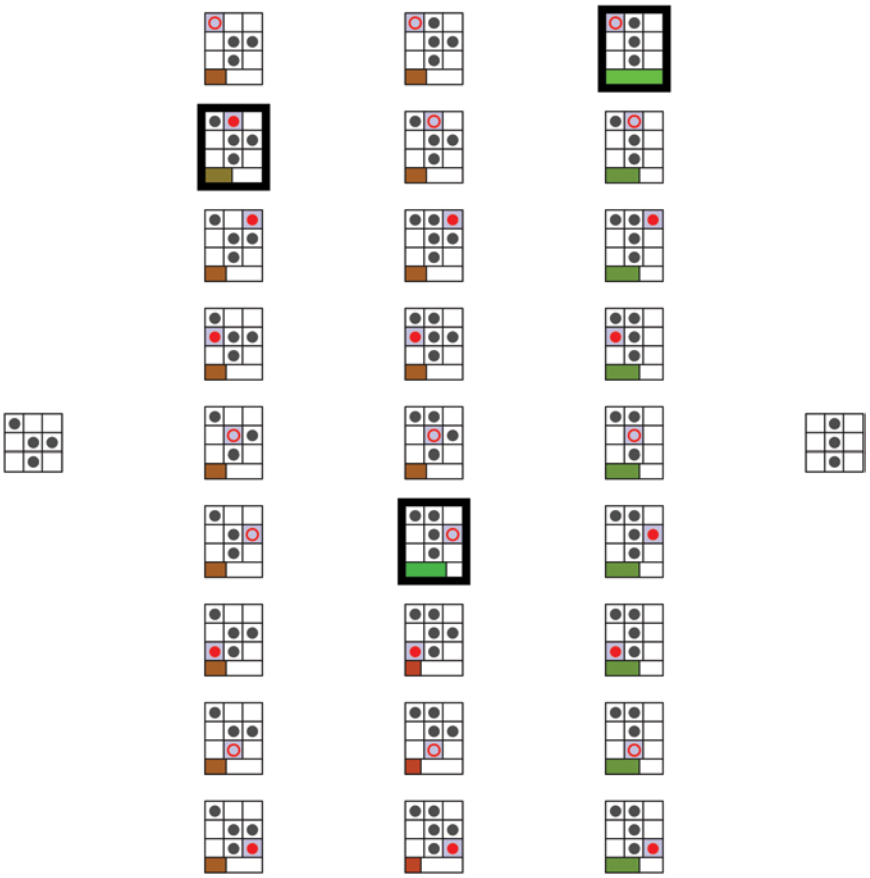


Figure 21-41: The same game as Figure 21-39, after 3,000 more training episodes

Comparing Q-Learning and SARSA

Let's compare the Q-learning and SARSA algorithms. Figure 21-42 shows the lengths of all 5,120 possible games, after 6,000 games of training by Q-learning and SARSA. These results are slightly different from the previous plots because they were generated by new runs of the algorithm, so the random events were different.

They're roughly comparable, but Q-learning produces a few games that are longer than SARSA's maximum of 12.

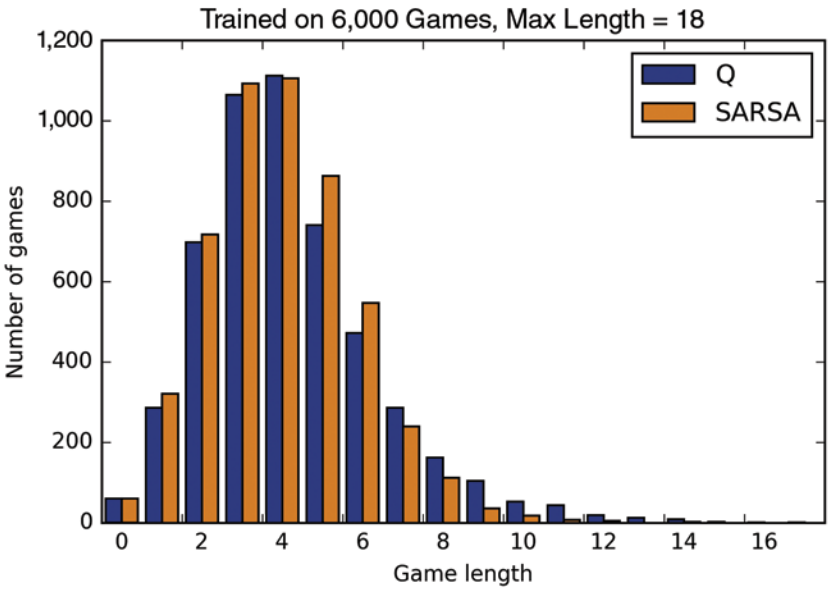


Figure 21-42: Comparing game lengths after 6,000 training games for both Q and SARSA. SARSA's longest game was 11 steps, while Q-learning went as high as 18.

More training helps. We've increased the training length by a factor of 10, for 60,000 games each. The results are shown in Figure 21-43.

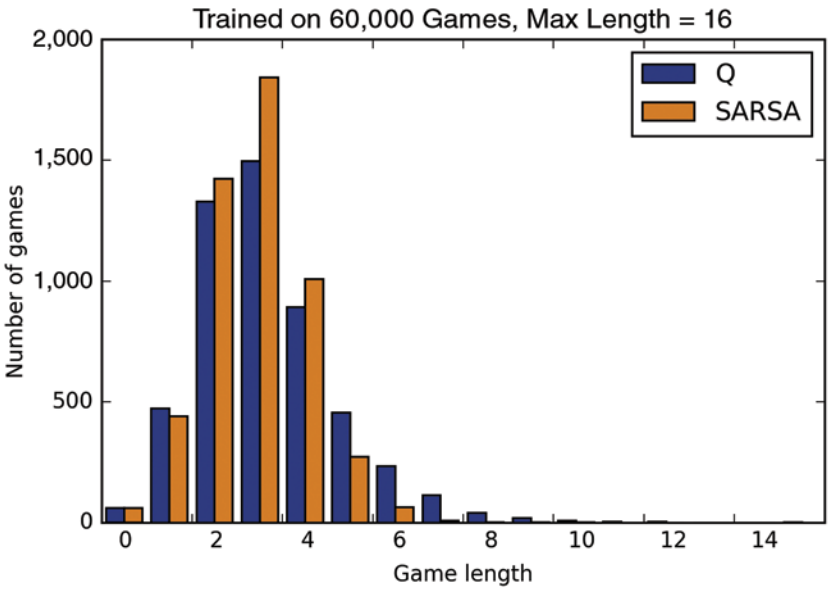


Figure 21-43: The same training scenario as in Figure 21-42, but now we've trained for 60,000 games

At this level of training, SARSA is doing an excellent job on Flippers, with almost all games coming in at 6 moves or less (very few games required 7 moves). Q-learning is faring slightly worse overall, needing up to 16 steps to solve some of its games, but it too is greatly concentrated in the region of 4 moves and under.

Another way to compare Q-learning and SARSA for this simple game is to plot the average game length after increasingly long training sessions. This gives us an idea of how effectively they're learning to win the game. Figure 21-44 shows this for our Flippers game.

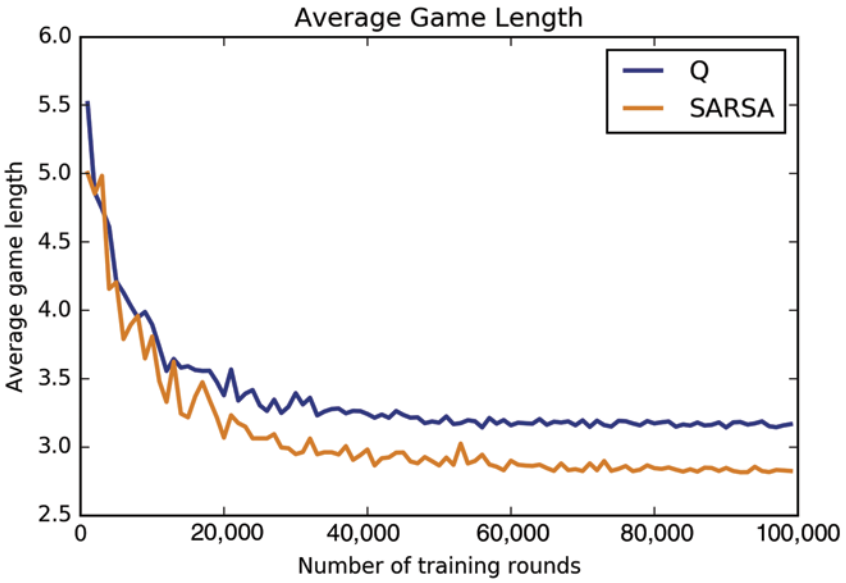


Figure 21-44: The length of the average game for training sessions from 1 to 100,000 episodes (in increments of 1,000)

The trend here is easy to see. Both algorithms drop quickly and then level off, but after a noisy start, SARSA always performs better, ultimately saving almost a half move on every game (that is, in general, it plays one less move for every two games). By the time we reach 100,000 training games, both algorithms appear to have stopped improving. It seems likely that the Q-tables of each algorithm have settled down into stable states, changing a little bit over time due to the random flips introduced by the environment.

So, although Q-learning and SARSA can both do a great job of learning to play Flippers, SARSA's games will generally be shorter.

The Big Picture

Let's step back and review the big picture of reinforcement learning.

There's an environment and an agent. The environment provides the agent with two lists of numbers (the state variables and the available

actions). Using its policy, the agent considers these two lists, along with whatever private information it has saved internally, to select one of the values from the list of actions, which it returns to the environment. In response, the environment gives the agent back a number (the reward) and two new lists.

Interpreting the lists as boards and moves was great because it lets us think of Q-learning in terms of learning to play a game. But the agent doesn't know it's in a game, or that there are rules, or really much of anything. It just knows that two lists of numbers come in, it picks a value from one of the lists, and then a reward value arrives in response. It's remarkable that this little process can do much that's interesting at all, but if we can find a way to describe our environment, and actions on that environment, using sets of numbers, and we can find even a crude way to distinguish a good action from a bad one, this algorithm can learn how to perform high-quality actions.

This worked for our simple game of Flippers, but how practical is all of this Q-table stuff in practice? In Flippers, there are nine squares and each can have a dot or not, so the game needs a Q-table with 512 rows and 9 columns, or 4,608 cells. In a game of tic-tac-toe, there are nine squares, and each can have one of three symbols: blank, X, or O. The Q-table for this game would need 20,000 rows and 9 columns, or 180,000 cells.

That's big, but not ridiculously big for a modern computer. But what if we want a slightly more challenging game? Rather than play tic-tac-toe on a 3 by 3 board, suppose we played on a 4 by 4 board. There are a bit more than 43 million such boards, so our table would have 43 million rows and 9 columns, or a bit under 390 million cells. That's getting pretty big, even for modern computers. Let's increase it just one more modest step, and play tic-tac-toe on a 5 by 5 board. That hardly seems outrageous. Yet that board has almost 850 *billion* states. If we get a little ambitious and play on a 13 by 13 board, we find that the number of states is more than the number of atoms in the visible universe (Villanueva 2009). In fact, it's roughly the number of atoms in one *billion* visible universes.

Storing the table for this game is not remotely practical, but it's an entirely reasonable thing to want to do. More reasonably, we might want to play Go. The standard board for the game of Go is a grid of 19 by 19 intersections, and each intersection can be empty, have a black stone, or a white stone. This is like our tic-tac-toe board, but unfathomably bigger. We'd need a table whose rows would have labels requiring 173 digits. Such numbers are not just wildly impractical, they're incomprehensible.

Yet this is the basic strategy that was used by the Deep Mind team to build AlphaGo, which famously beat a world champion human player (DeepMind 2020). They did it by combining reinforcement learning with deep learning. One of the key insights in this *deep reinforcement learning* approach was to eliminate explicit storage of the Q-table. We can think of the table as a function that takes a board state as input and returns a move number and Q-value as output. As we've seen, neural networks are great at learning how to predict things like this.

We can build a deep learning system that takes the board input and predicts the Q-value we'd get for each move if we really did keep the table around. With enough training, this network can become accurate enough that we can abandon the Q-table and use just the network. Training a system like this can be challenging, but it can be done, with excellent results (Mnih et al. 2013; Matiisen 2015). Deep reinforcement learning has been applied to fields as diverse as video games, robotics, and even healthcare (François-Lavet et al. 2018). It's also the central algorithm behind AlphaZero, arguably the best player of the game Go that has ever existed (Silver et al. 2017; Hassabis and Silver 2017; Craven and Page 2018).

Reinforcement learning has an advantage over supervised learning because it does not require a database that has been manually labeled, which is often a time-consuming and expensive process. On the other hand, it requires us to design an algorithm for creating rewards that guide an agent toward the desired behavior. In complex situations, this can be a difficult problem to solve.

This has necessarily been a high-level overview of a big topic. Much more information on reinforcement learning can be found in dedicated references (François-Lavet et al. 2018; Sutton and Baro 2018).

Summary

In this chapter we took a look at some of the basic ideas in reinforcement learning, or RL. We saw that the basic idea is to break the world into an agent who acts, and an environment that encompasses everything else. The agent is given a list of options, and using a policy, it selects one. The environment executes that action, along with follow-on effects (which can include making a return move in a game, or carrying out a simulation or real-world action), and then returns to the agent a reward describing the quality of its chosen action. Typically the reward describes how well the agent has succeeded in improving the environment in some way.

We applied these ideas to the one-player game of Flippers with a simple algorithm that recorded the rewards in a table, and used a simple policy to select the move with the highest reward when possible. We saw that this didn't handle the unpredictability of the real world very well, so we improved the method into the Q-learning algorithm with a better update rule and learning policy.

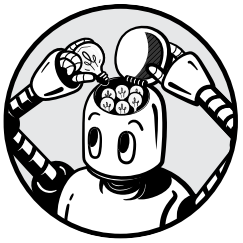
Then we improved that method again by prechoosing our next move, resulting in the SARSA algorithm. This learned to play Flippers even better.

In practice, a vast number of algorithms fall under the category of reinforcement learning, with more arriving all the time. It's a vibrant field of research and development.

In the next chapter, we'll look at a powerful method for training generators that can produce images, video, audio, text, and other kinds of data so well that we can't reliably distinguish generated data from data in the training set.

22

GENERATIVE ADVERSARIAL NETWORKS



Generating data is exciting. It lets us produce new paintings, songs, and sculptures that have a resemblance to their inputs. In Chapter 18 we saw how to use autoencoders to generate new data that was like the training data. In this chapter we explore a completely different approach to data generation. The type of system we look at is called a *Generative Adversarial Network*, or *GAN*. It's based on a clever strategy where two different deep networks are pitted against one another, with the goal of getting one network to create new samples that are not from the training data, but are so much like the training data that the other network can't tell the difference.

The GAN method is actually a technique for training a network that generates new data. That trained generator is just a neural network like any other, and the method we used to train it isn't relevant anymore. But the language of the field frequently refers to a generator trained with the GAN method as a GAN itself. It's a bit weird to name something not for what it does, but for how it learned to do its job, but we do. Thus we use the GAN technique to train a generator, which is often called a generator, but also often called a GAN.

Let's begin our discussion of the GAN method by looking at how a two-person team can learn to forge money by helping each other learn. Then we can replace the two people with neural networks. One of these networks becomes better and better at spotting forgeries, and the other becomes better and better at making forgeries. When the training process is over, the forger is able to make as many new and different bills as we like, and the detector can't reliably distinguish counterfeits from real bills. The process works for any kind of data, from pictures of dogs to the sound of someone speaking.

We'll see how to build, train, and use these coupled networks to synthesize new data, using different kinds of layers. We wrap up the chapter by discussing issues to look out for when training and using the data-generating part of these networks.

Forging Money

The usual way to introduce a GAN is by analogy to a counterfeiting operation. We will present a variation on the typical presentation to better expose the key ideas.

The story begins with two conspirators, Glenn and Dawn. Glenn's name starts with G because he plays the role of the *generator*, in this case forging new money. Dawn's name begins with D because she plays the role of the *discriminator*, in this case tasked with determining whether any given bill is real or one of Glenn's forgeries. Glenn and Dawn are going to both improve over time, thereby pushing the other to improve as well.

As the generator, Glenn sits in a back room all day, meticulously creating metal plates and printing false currency. Dawn is the quality-control half of the operation. It's her job to take a mixed-up pile of real bills along with Glenn's forgeries, and decide which is which. The penalty for forgeries in their country is life in prison, so they're both highly motivated to produce bills that nobody can tell from the real thing. Let's say that the currency of their country is called the Solar, and they want to counterfeit the 10,000 Solar bill.

An important thing to note is that all 10,000 Solar bills are not the same. At the very least, each bill has a unique serial number. But real bills are also scuffed, folded, drawn on, torn, dirtied, and otherwise handled. Since new, crisp bills stand out, Glenn and Dawn want to produce currency that looks just like all the other, worn currency in circulation so that it blends in and doesn't catch anyone's eye. And like real bills, every counterfeit bill should look unique.

In a real situation, Glenn and Dawn would surely start off with a huge stack of real bills, and pore over every detail, learning everything they could. But we're just using their operation as a metaphor, so we're going to put in some restrictions to make this situation better match the algorithms this chapter is dedicated to. First, let's simplify things a little and say that we only care about one side of the bill. Second, we're not going to give Glenn and Dawn each a stack of bills to study before they begin. In fact, let's assume that neither Dawn nor Glenn has any idea what a real 10,000 Solar bill looks

like. Clearly this is going to make things a *lot* harder. We'll justify this in a moment. The one thing we do give them goes to Glenn: a big stack of blank rectangles of paper that match the shape and size of a 10,000 Solar bill.

They each follow a daily routine. Every morning, Glenn sits down and makes a few forgeries using all the information he has so far. In the beginning, he doesn't know anything, so he may just splash different colors of inks around on the paper. Or maybe he draws some faces or numbers. He basically just draws random stuff. At the same time, Dawn goes to the bank and withdraws a stack of real 10,000 Solar notes. Very lightly, she writes the word *Real* on the back of each one in pencil. Then, when Glenn is done, she collects Glenn's forgeries for the day and writes the word *Fake* lightly on the back of each. She then shuffles the two piles together. Figure 22-1 shows the idea.

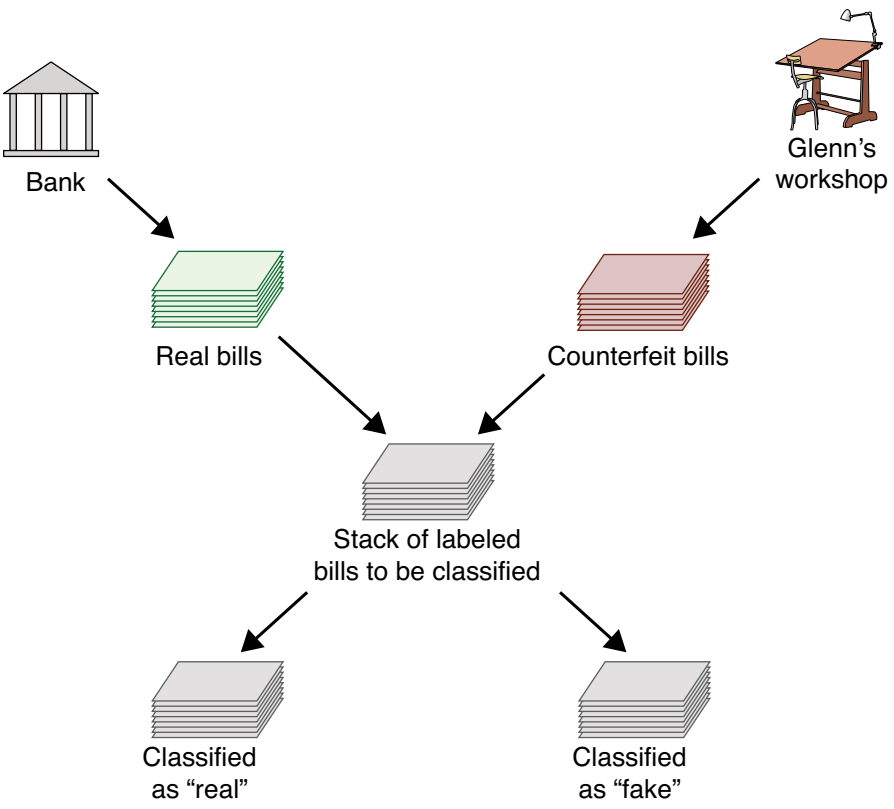


Figure 22-1: Dawn gets real bills from the bank and forgeries from Glenn, shuffles them together (the pile in the middle), and sorts them into real and fake.

Now Dawn does her main job. One by one, she goes through the bills, and without looking at the backs, she categorizes each one as real or fake. Let's say she asks herself, "Is this bill real?" We call an answer of "yes" a *positive* response to that bill and an answer of "no" a *negative* response to that bill.

Dawn carefully sorts her starting stack into two piles: the reals and the fakes. Since each bill can be real or fake, there are four possibilities, summarized in Figure 22-2.

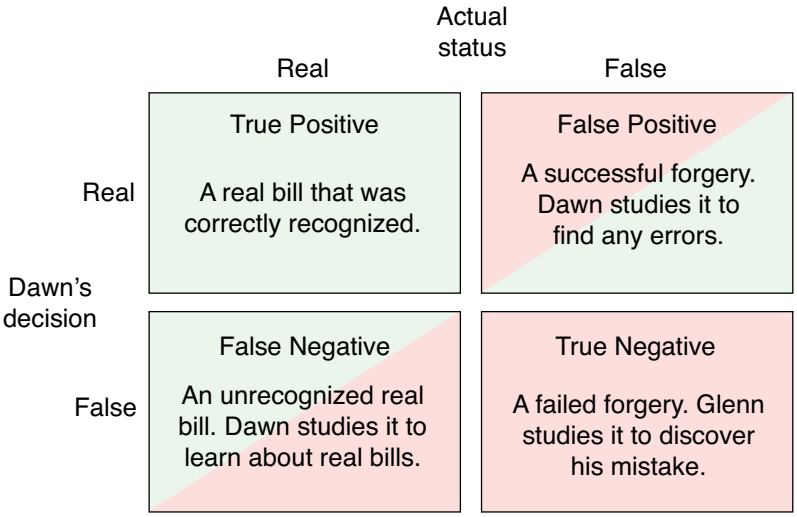


Figure 22-2: When Dawn examines a bill, it might be real or fake, and she might declare it to be real or fake. This gives us four combinations.

When Dawn looks at a bill, if it's real and she says it's real, then her "positive" decision is accurate, and we have a true positive (TP). If the bill is real but her decision is "negative" (she thinks it's fake), then it's a false negative (FN). If the bill is fake but she thinks it's real, that's a false positive (FP). Finally, if it's fake and she correctly identifies it as fake, that's a true negative (TN). In all cases but true positive, either Dawn or Glenn uses that example to improve their work.

Learning from Experience

We've mentioned that Dawn and Glenn are just human stand-ins for the neural networks called the *discriminator* and *generator*, respectively.

The discriminator is a classifier. It places each input into one of two classes: real or fake. When the prediction is wrong, that network's error function has a large value. We then train the discriminator in the usual way with backprop and optimization, so that the class is more likely to be right the next time.

The generator's job is quite different. It never sees the training data at all. Instead, we give it a random input (like a list of a few hundred numbers), and from that it produces an output. That's all it does. If the discriminator thinks that output is real (that is, from the training set), then the generator got away with this forgery and doesn't need to improve. But if the discriminator catches the output as fake (that is, synthetic, or from the generator), then the generator gets an error signal and we use backprop and optimization so that it moves away from results like this one that get caught by the discriminator. Each time we run the generator, we give it new, random, starting values. The generator has a daunting task: turn this small list of numbers into an output that fools the discriminator. For example, the intended output could be a song that sounds like it was written by Bach, a

piece of speech that sounds like a person, a face that looks like a person, or a used piece of currency that's worth 10,000 Solars.

How can we possibly train such a system? The generator never sees the data it's trying to emulate, so it can't learn from it. It only knows when it's wrong.

The approach that works surprisingly well is trial and error. We start out, as described earlier, with a generator and a discriminator that are both entirely untrained. When we give the discriminator some data, it basically just assigns each piece of data to a random class. At the same time, the generator is making random output. They're both flailing and essentially producing meaningless outputs.

Slowly, though, the discriminator starts to learn, because we're giving it the proper labels for the data it's classifying. And as the discriminator gets a little better, the generator tries a bunch of different variations on its output until something gets past the discriminator (that is, the discriminator thinks it was real data, and not from the generator). The generator hangs onto that as its best work so far. Then the discriminator gets a little better, and in turn, the generator gets a little better. As time goes on, the tiny improvements in each network accumulate, until the discriminator is very sensitive to the differences between real and generated data, and the generator is very skilled at making those differences as small as possible.

Forging with Neural Networks

Figure 22-2 showed the four possible situations that can arise after Dawn makes a decision for each bill. Let's look more closely at how we train the discriminator and generator in such a way that they each force the other to improve. Note that this discussion is meant to cover the concepts, so we will proceed one sample at a time. In practice, we often implement these ideas in ways that are more complex, but more efficient (for example, by training in mini-batches rather than one sample at a time).

Let's look more closely at the four possibilities in Figure 22-2 in flowchart form.

Starting with the true positive case, the discriminator correctly reports that the image of a real bill at its input is, indeed, a real bill. Since this is just what we want the discriminator to do in this case, there's no learning to be done. Figure 22-3 shows this process graphically.

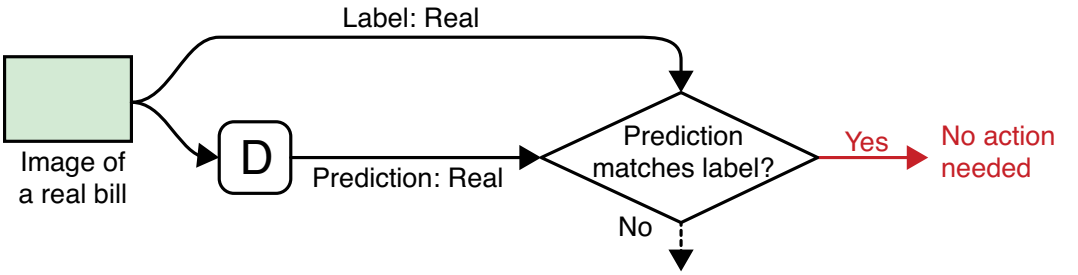


Figure 22-3: In the true positive (TP) case, the discriminator (D) receives a real bill and correctly predicts it to be real. Nothing needs to happen as a result.

Next, we have the false negative, when the discriminator incorrectly declares a real bill to be a fake. As a result, the discriminator needs to learn more about real bills so it doesn't repeat this error. Figure 22-4 shows the situation.

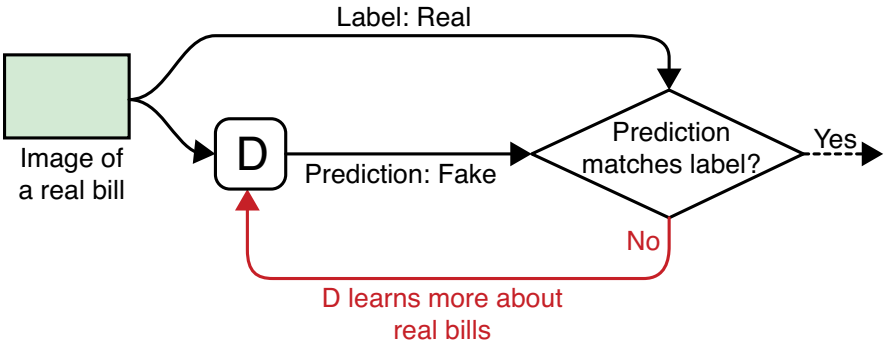


Figure 22-4: We get a false negative (FN) when the bill is real but the discriminator says it's a fake. The discriminator needs to learn more about real bills so it doesn't repeat this mistake.

The false positive case comes when the discriminator gets fooled by the generator and declares a forged bill to be real. In this case, the discriminator needs to study the bill more carefully and find any errors or inaccuracies so that it won't get fooled again. Figure 22-5 shows how this goes.

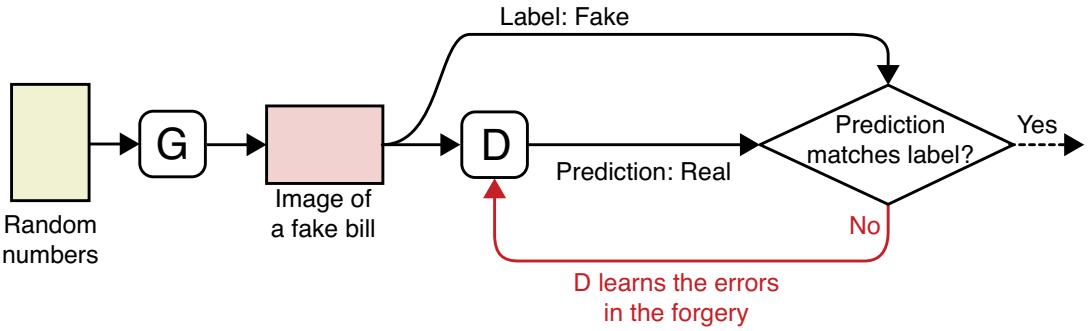


Figure 22-5: In the false positive (FP) situation, the discriminator receives a fake bill from the generator but classifies it as real. To force the generator to get even better, the discriminator learns from its mistake so that this particular forgery won't sneak through again.

Finally, the true negative case is when the discriminator correctly identifies a forgery. In this case, shown in Figure 22-6, the generator needs to learn how to improve its output.

Note that out of these four possibilities, one of them (TP) has no effect on either network, two of them (FN and FP) cause the discriminator to improve its ability to recognize real and fake bills, and only one (TN) causes the generator to learn and avoid repeating mistakes.

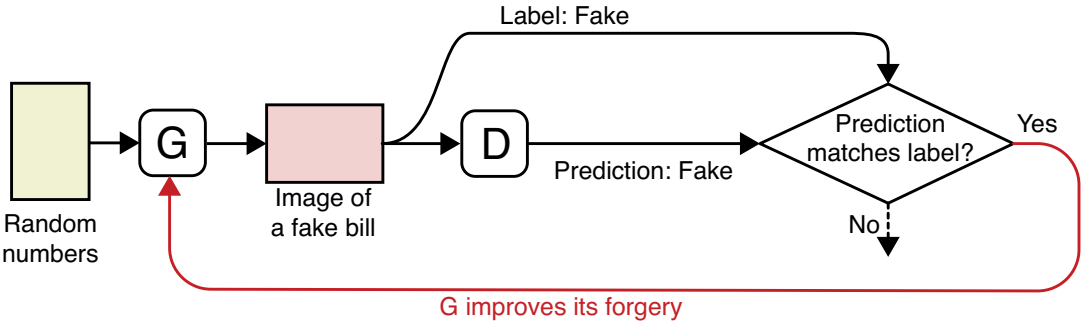


Figure 22-6: In the true negative (TN) scenario, we give the discriminator a fake bill from the generator, and the discriminator correctly identifies it as fake. In this case, the generator learns that its output was not good enough, and it has to improve its forging skills.

A Learning Round

Let's now assemble the feedback loops from the last section into a single step of training for both the discriminator and the generator. Generally, we repeat a set of four steps over and over. In each step, we give the discriminator either a real or fake bill, and then based on its response, follow one of the four flowcharts we just saw.

First we train the discriminator, then the generator, then the discriminator again, and then the generator again. The idea is to test for each of the three situations in which one or the other network needs to learn. The true negative case, where the generator learns, is repeated twice for reasons we'll get to in a moment. Figure 22-7 summarizes the four steps.

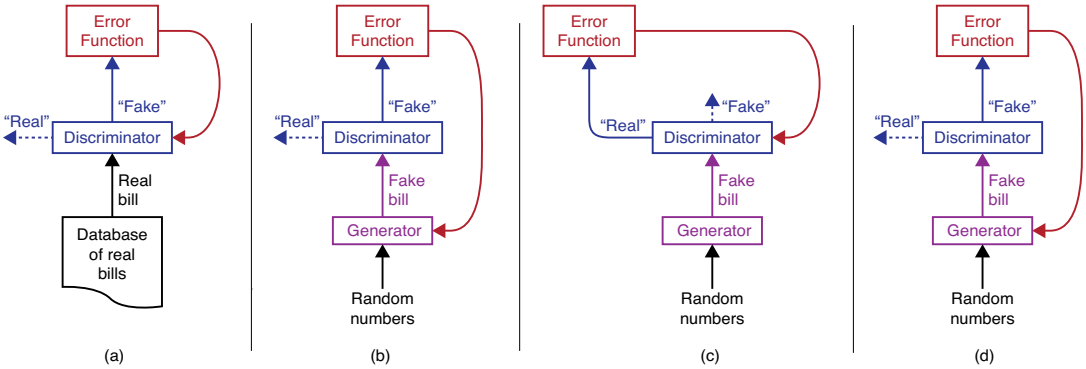


Figure 22-7: The four steps of a learning round

First, in part (a), we try to learn from false negatives. We give the discriminator a random bill from the dataset of real bills. If it misclassifies it as a forgery, we tell the discriminator to learn from that mistake.

Second, in part (b), we look for true negatives. We give some random numbers to the generator, produce a fake bill, and hand that to the discriminator. If the discriminator catches the forgery, we tell the generator, which attempts to learn to produce a better forgery.

Third, in part (c), we look for false positives. We give a new batch of random values to the generator and have it produce a new, fake bill, which we hand to the discriminator. If the discriminator is fooled and says the bill is real, the discriminator learns from its mistake.

Finally, in part (d), we repeat the true negative test from the second step. We give new random numbers to the generator, make a new fake bill, and if the discriminator catches the forgery, the generator learns.

The reason for repeating the generator's learning step twice is that practice has shown that in many cases, the most efficient learning schedule is to update both networks at roughly the same rate. Since the discriminator learns from two types of errors, while the generator learns from only one, we double the number of learning opportunities for the generator, allowing both to learn at about the same pace.

Through this process, the discriminator gets better and better at identifying real bills and spotting the errors in the counterfeits, and the generator, in turn, gets better and better at finding out how to create a counterfeit that cannot be spotted. This pair of networks, taken together, make up a single GAN. We can picture the two networks in a "learning battle" (Geitgey 2017). As the discriminator gets better and better at spotting fakes, the generator must get correspondingly better to get one through, causing the discriminator to get better at finding the forgery, causing the generator to get even better at making fakes, and so on.

The ultimate goal is to have a discriminator that is as good as it can be, with deep and broad knowledge of every aspect of the real data, and yet also have a generator that can still get forgeries past the discriminator. That tells us that the counterfeits, despite being different from the real examples, are statistically indistinguishable from them, which was our goal all along.

Why Adversarial?

The name *Generative Adversarial Network (GAN)* may seem strange in light of the preceding description. The two networks we just described seem to be cooperative, not adversarial. The choice of *adversarial* comes from looking at the situation in a slightly different way. Instead of the cooperation we described between Dawn and Glenn, we can imagine that Dawn is a detective with the police, and Glenn is working alone. To make the metaphor work, we have to also imagine that there's some way for Glenn to discover which of his forged bills were detected (perhaps he has an accomplice in Dawn's office who forwards this information to him).

If we picture the forger and the detective as opposed to one another, then indeed they are adversarial. This was how the subject of GANs was phrased in the original paper on the subject (Goodfellow et al. 2014). The adversarial view doesn't change anything about how we set up or train the networks, but it offers a different way to think about them (Goodfellow 2016).

The word *adversarial* comes from a branch of mathematics called *game theory* (Watson 2013), in which we view the discriminator and generator as opponents in a game of deception and detection.

The field of game theory is devoted to studying how competitors can maximize their advantages (Chen, Lu, and Vekhter 2016; Myers 2002).

Our goal with GAN training is to develop each network to its peak ability, despite the other network's abilities to thwart it. Game theorists call this state a *Nash equilibrium* (Goodfellow 2016).

Now that we know the general technique for training, let's see how to actually build a discriminator and a generator.

Implementing GANs

When we talk about GANs, we're discussing three distinct networks: the discriminator, the generator, and the generator and the discriminator together. We saw two of these structures in Figure 22-7. In part (a) we had just the discriminator. In parts (b) through (d) we had the generator and discriminator combined. As we'll see later, when training is done and we want to make new data, we discard the discriminator and use just the generator by itself.

It's usually clear from context which of these networks is being discussed. As mentioned earlier, when someone speaks of a GAN, they usually mean only the trained generator, after it's been taught by the adversarial process. The word GAN is used flexibly in the field. It can refer to the training method we just described, or the combined generator-discriminator network used during training, or the standalone generator that we end up with after training. Often people say that they will "train a GAN," meaning that they will use the GAN method to train a generator that might then itself be called a GAN. It's not as confusing as it sounds, as the right interpretation is usually clear from context.

Enough background! Let's build and train a GAN.

The Discriminator

The discriminator is the simplest of the three models, as shown in Figure 22-8. It takes a sample as input, and its output is a single value that reports the network's confidence that the input is from the training set rather than an attempted forgery.

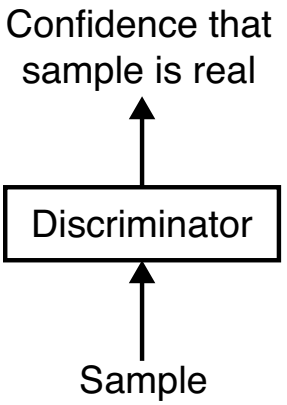


Figure 22-8: The block diagram of a discriminator

There aren't any other restrictions on how we make the discriminator. It can be shallow or deep and use any kinds of layers: fully connected layers, convolutional layers, recurrent layers, transformers, and so on. In our currency forging example, the input is an image of a bill, and the output is a real number reflecting the network's decision. A value of 1 means that the discriminator is sure that the input is a real bill, and a value of 0 means that the discriminator is sure that it's a fake. A value of 0.5 means that the discriminator just can't tell either way.

The Generator

The generator takes a bunch of random numbers as input. The output of the generator is a synthetic sample. The block diagram is in Figure 22-9.

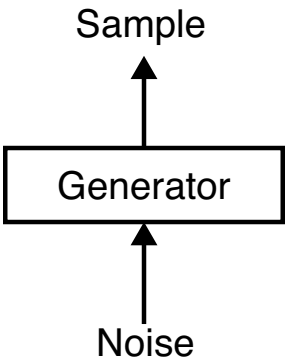


Figure 22-9: The block diagram for a generator

As with the discriminator, there aren't any constraints on how we build the generator. It can be shallow or deep, and use any kinds of layers we like.

In our example of forging currency, the output would be an image.

The loss function for the generator of Figure 22-9 all by itself is irrelevant, and in some implementations, we never even define one. As we'll see in the next section, we train the generator by hooking it up to the discriminator, so the generator learns from the loss function for the combined network.

Once our GAN is fully trained, we often discard the discriminator and keep the generator. After all, the discriminator's purpose was to train the generator so that we could use it to make new data. When the generator has been disconnected from the discriminator, we can use the generator to make an unlimited amount of new data for us to use any way we like.

Now that we have the block diagrams of the generator and discriminator, we can look more closely at the actual training process. With that in place, we'll look at implementations of both networks. Then we'll train them and see how they do.

Training the GAN

Let's now look at how to train our GAN. We'll expand the four steps in the learning round shown in Figure 22-7 to show where the updates get applied.

Our first step is to look for false negatives, so we feed real bills to the discriminator, as in Figure 22-10. In this step, we don't involve the generator at all. The error function here is designed to punish the discriminator if it reports a real bill as a fake. If that happens, the error drives a backpropagation step through the discriminator, updating its weights, so that it gets better at recognizing real bills.

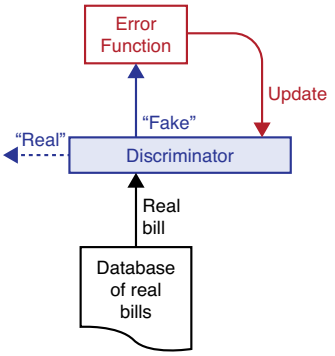


Figure 22-10: In the false negative step, the discriminator is hooked up to an error function that punishes it for categorizing a real bill as a fake.

The second step looks for true negatives. In this step, we hook up the output of the generator directly to the input of the discriminator to create one big model. We start with random numbers going into the generator, as shown in Figure 22-11. The generator's output is a fake bill, which is then fed to the discriminator. The error function is designed to have a large value if this fake bill is correctly identified as fake, meaning that the generator got caught making a forgery.

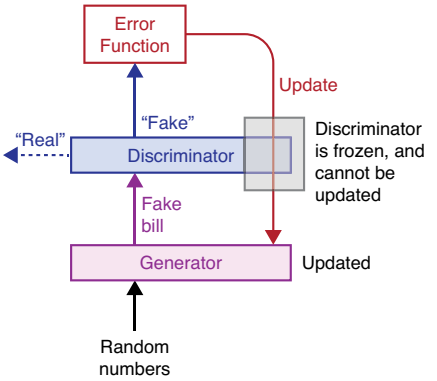


Figure 22-11: In the true negative step, random numbers feed the generator, which produces a fake bill. If the discriminator labels it as fake, we push gradients through the discriminator, but only update the generator.

In Figure 22-11 we've grayed-out the update step for the discriminator, yet the arrow labeled Update is apparently going through the discriminator. What's going on here is that our Update arrow combines backprop and

optimization. Recall that backprop computes the gradient for each weight but doesn't actually change anything. It's the optimization step that updates the weights, based on their gradients. In Figure 22-11, we want to apply optimization to the generator, which means we need to find its gradients. But because backprop computes the gradients from the end of the network to the start, the only way to find the gradients in the generator is to first compute them for the discriminator. Although we find the gradients in both networks, we only change the weights in the generator. We say that the discriminator is *frozen*, meaning that its weights are not changed, even though we computed their gradients. This insures that at any given time, we're training only the generator or only the discriminator.

Improving the generator's weights lets it learn to better fool the discriminator.

Now we look for false positives. We generate a fake bill and punish the discriminator if it classifies it as real, as in Figure 22-12.

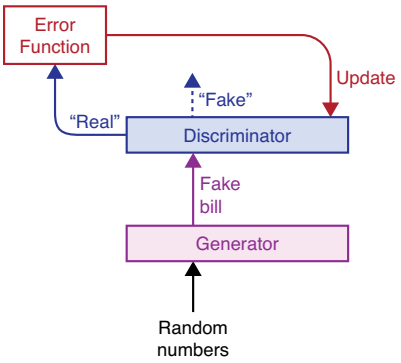


Figure 22-12: In the false positive step, we give the discriminator a fake bill. If it classifies it as real, then we update the discriminator to better spot the fakes.

Finally, we repeat the true negative step of Figure 22-11, so that both the discriminator and generator have two opportunities to get updated in each round of training.

GANs in Action

Enough theory! Let's build a GAN system and train it. We'll pick something very simple so that we can draw meaningful illustrations of the process in 2D.

Let's picture all the samples in our training set as a cloud of points in some abstract space. After all, each sample is ultimately a list of numbers, and we can treat those as coordinates in a space that has as many dimensions as there are numbers. Our set of "real" samples will be points that belong to a 2D cloud that has a Gaussian distribution. Recall from Chapter 2 that a Gaussian curve has a big bump in the center, so we expect most of our points to be near the bump, with fewer and fewer points as we move outward. Each sample is a single point from that distribution. Let's center the 2D blob at (5,5), and give it a standard deviation of 1. Figure 22-13 shows this distribution.

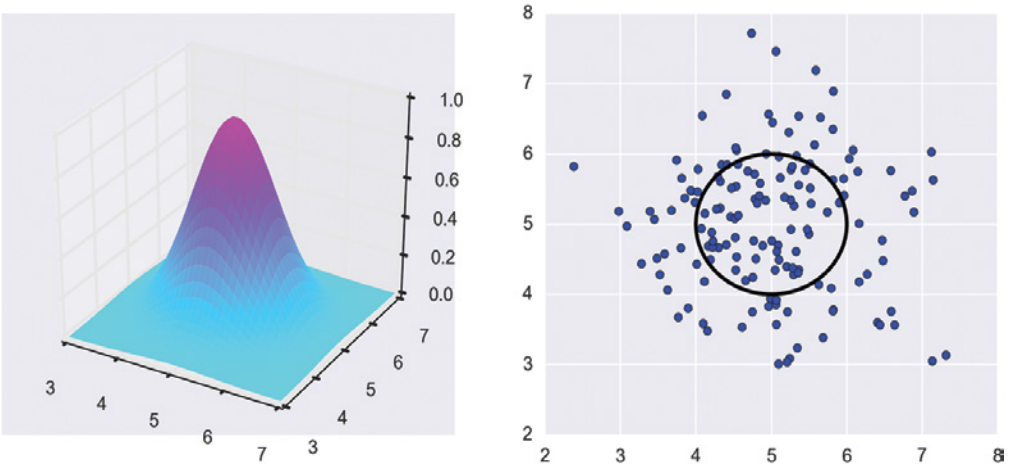


Figure 22-13: Our starting distribution is a Gaussian bump centered at (5,5) with a standard deviation of 1. Left: The blob in 3D. Right: A circle showing the location of one standard deviation of the blob in 2D, and some representative points randomly drawn from this distribution.

Our generator will try to learn how to turn the random numbers that it's given into points that seem to belong to this distribution. The goal is to do that so well that the discriminator can't tell real points from synthetic ones created by the generator. In other words, we want the generator to take in random numbers and produce output points that *could* have been the result of drawing random points from our original Gaussian bump centered at (5,5).

Given only a single point, as in Figure 22-14, it's a challenge for the discriminator to say with any certainty if it's an original sample drawn from our Gaussian distribution or a synthetic sample created by the generator.

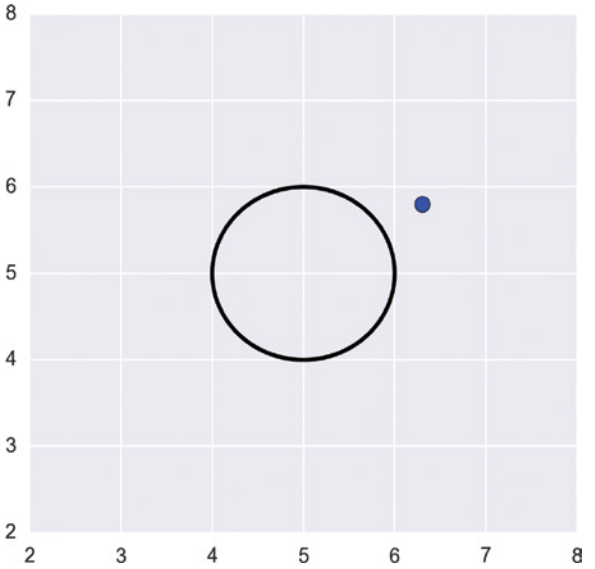


Figure 22-14: We have a single sample and we want to determine if it was drawn from the Gaussian distribution.

We can make things easier on the discriminator by using an old friend from Chapter 15: the mini-batch (or often just the batch). Rather than run one sample at a time through the system, we can run through a lot of them, often a power of two in the range 32 to 128. Given a whole bunch of points, it's easier to decide if they were plucked from our Gaussian cloud or not. Figure 22-15 shows a few sets of points that the generator might produce. We hope that the distributor will be able to easily realize that these points are unlikely to have been drawn from our original distribution.

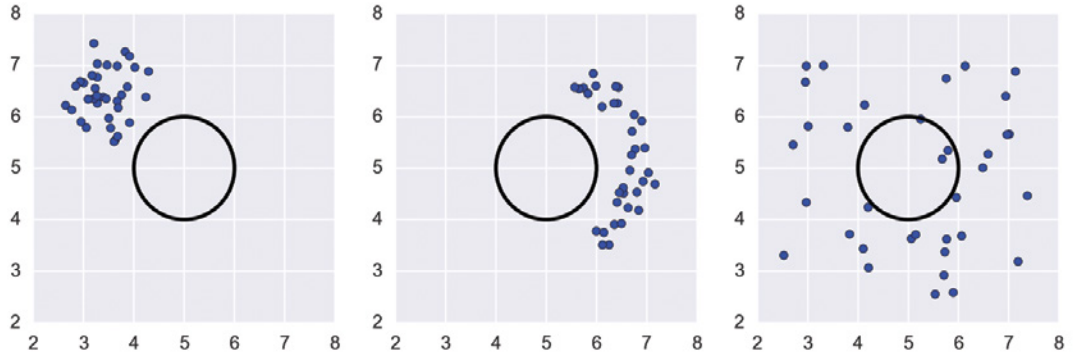


Figure 22-15: Some sets of points that are unlikely to have been the result of picking random values from our starting Gaussian

We want our generator to produce points more like the those on the right of Figure 22-13 than any of those in Figure 22-15. And we want the discriminator to classify the sets of points in Figure 22-15 as fakes, since they're so unlikely to have been part of the original Gaussian data.

Building a Discriminator and Generator

Let's build discriminator and generator networks for this problem. Because our original distribution (the 2D Gaussian bump) is so simple, our networks can be simple also.

A word of warning before we dig into the mechanics, though. GANs are famous for being finicky and sensitive. They are notoriously hard to train (Achlioptas et al. 2018). Minor changes in the architecture of the generator or discriminator, or even small changes to some of the hyperparameters (such as learning rates or dropout rates) can turn a practically useless GAN into a star performer, and vice versa. Worse, we have to train not one network but two, *and* get them to work together, so the number of choices of hyperparameters to search through and fine-tune can become overwhelming (Bojanowski et al. 2019). So, while we develop a GAN, it's essential to experiment using the specific data we want to learn from and try to home in on a good design and good hyperparameters as quickly as we can. This often means trying lots of little experiments with small excerpts from the training data, as we hunt for good networks and hyperparameters.

In the following discussion, we skip the many dead-ends and badly performing models that we tried. Instead, we'll jump right to models that we found work well for this dataset. It's very possible that with further changes, or perhaps even just small tweaks in the right places, we could significantly improve the architectures we show (that is, enable them to learn faster and more accurately).

Let's start with a simple generator, shown in Figure 22-16.

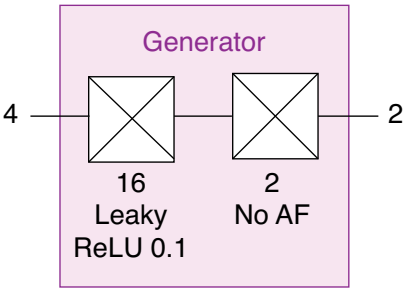


Figure 22-16: A simple generator. It takes in four random numbers and computes an (x,y) pair.

The model takes in four random values, uniformly selected from the range 0 to 1. We start with a fully connected layer with 16 neurons and a leaky ReLU activation (recall from Chapter 13 that a leaky ReLU is like a normal ReLU, but instead of returning 0 for negative values it scales them by a small number, here 0.1).

This is followed by another fully connected layer with just two neurons and no activation function. And that's it for the generator. The two values that are produced are the x and y coordinates of a point.

We're asking quite a lot of these two layers with only 18 neurons and 54 weights. We want them to learn how to convert a set of four uniformly distributed random numbers into a 2D point that could have been drawn from a Gaussian cloud with a center at (5,5) and a standard deviation of 1, but we'll never tell it anything about that goal. We only tell it when a mini-batch of its points isn't a credible match to what we want, and leave it to the neurons to figure out where they went wrong and how to make it right.

Our discriminator is in Figure 22-17.

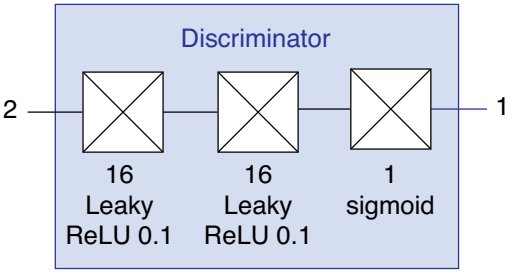


Figure 22-17: A simple discriminator. It takes in an (x,y) point and tells us if it's real or fake.

This starts with two layers of the same form as the start of the generator. Each is a fully connected layer of 16 neurons with a leaky ReLU activation. At the end is a fully connected layer with just 1 neuron and a sigmoid activation function. The output is a single number with the network's confidence that the input is from the same dataset as the training data.

Finally, we put the generator and discriminator together to make the combined model, which is sometimes referred to as the *generator-discriminator*. Figure 22-18 shows this combination.

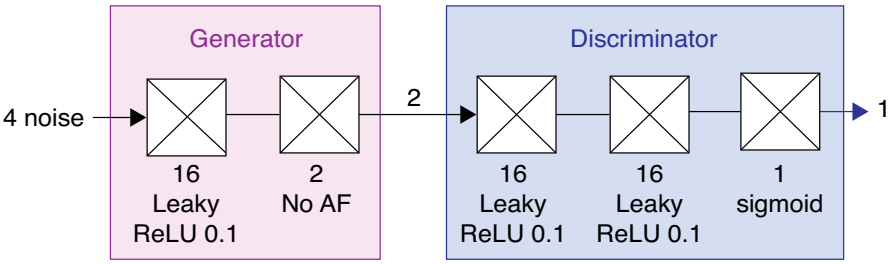


Figure 22-18: Putting the generator and discriminator together.

Since the generator presents an (x,y) pair at its output, and the discriminator takes an (x,y) pair at its input, the two networks go together perfectly. The generator's input is a set of four random numbers, and the discriminator's output tells us how likely it is that the point created by the generator is from the training set's distribution.

It's important to keep in mind that the models marked "generator" and "discriminator" in Figure 22-18 are not copies of the models in Figure 22-16 and Figure 22-17, but they are in fact the *very same models*, just connected together one after the other to make one big model. In other words, there's just one generator model and one discriminator model. When we make the combined model of Figure 22-18, we just chain together those two existing models. Modern deep-learning libraries let us make multiple models out of shared components for just this kind of application. Using the same models in these different configurations makes sense, since the combined model needs to use the most up-to-date versions of the generator and discriminator.

Training Our Network

When we train the generator using the combined model of Figure 22-18, *we don't want to train the discriminator as well*. We saw this in Figure 22-11 where we grayed out the discriminator during the update step. We need to run backprop through the discriminator, since it's part of the network, and helps create the gradients for the generator, but we only apply the update step to the weights in the generator.

Remember that we want to train the discriminator and generator in alternating passes. If we were to apply backprop to the entire network of Figure 22-18, then we'd update the weights in the discriminator as well as the generator. Because we want to train both models at about the same rate, and we know we're going to train the discriminator separately (since it also

needs to be trained on real data), we want to tell our library to update the weights in the generator *only*.

The mechanics for controlling whether or not a layer should have its weights updated are library specific, but generally speaking, they use terms like *freeze*, *lock*, or *disable* to prevent updates on a given layer. Then we can *unfreeze*, *unlock*, or *enable* updates later when we train the discriminator, and we want those layers to be able to learn.

To summarize the training process, we start with a mini-batch of points from the training set. We then follow the four-stage process in Figure 22-7, training the discriminator and generator alternately.

Testing Our Network

Let's look at some results. To train our GAN, we made a training set by drawing 10,000 random points from our starting Gaussian distribution. Then we trained the networks using mini-batches of 32 points. Running all 10,000 points through the system makes up one epoch.

Results for epochs 1 through 13 are shown in Figure 22-19.

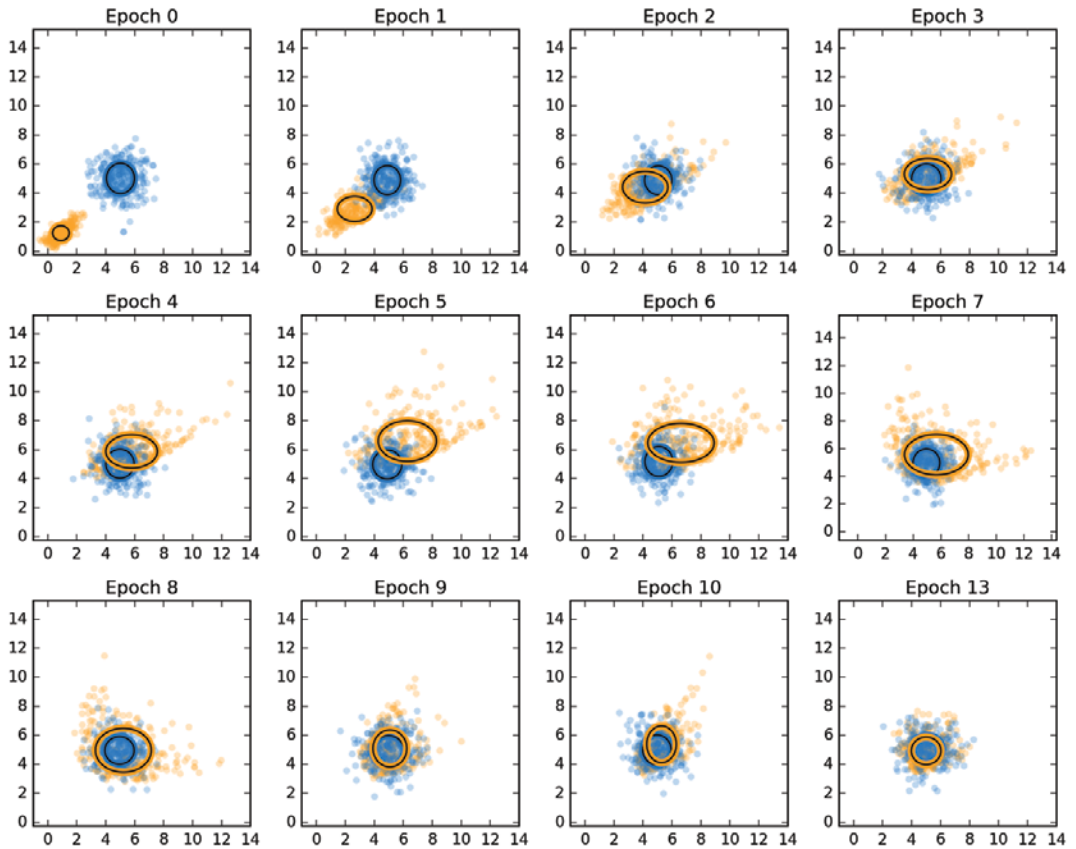


Figure 22-19: Our simple GAN in action. The blue points are the original dataset. The orange points were produced by the generator. Read the plots left to right, top to bottom. Epoch 0 refers to results after the first epoch of training.

Our starting Gaussian is shown with blue points, and a blue circle showing its mean and standard variation. The distribution that is being learned by the GAN is shown in orange, with an ellipse showing the center and standard deviation of the mini-batch of points that were generated. The plots show the results after 0 to 10 epochs of training, and then epoch 13. To keep the plots legible, we only show a randomly selected subset of the original and generated data in each plot.

We can see that after one epoch, the GAN's generated points form a smudgy line in the southwest-northeast direction, roughly centered around (1,1). With each epoch of training, they move closer to the original data's center and shape. Around epoch 4 the generated samples overshoot the center, and become increasingly elliptical rather than circular. But they come back and correct both qualities, until the match is looking very good by epoch 13.

Figure 22-20 shows the loss curves for the discriminator and generator.

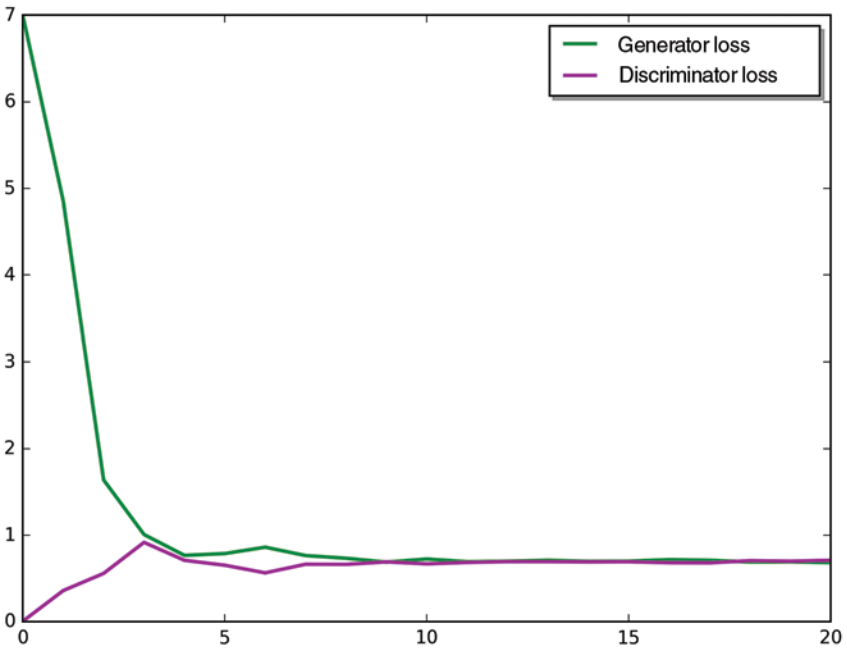


Figure 22-20: The loss for our GAN. They seem to meet and remain at a value a little above the ideal of 0.5.

Ideally the discriminator would plateau at about 0.5, meaning that it was never sure whether an input was from the real dataset or produced by the generator. In this tiny example, it got pretty close.

DCGANs

We said that we could build our discriminator and generator using any kind of architecture we like. Up to this point, our simple models have been

made of dense layers that performed nicely for our little 2D dataset. But if we want to work with images, then we'd probably prefer to use convolutional layers since, as we saw in Chapter 16, they're well suited to processing images. A GAN that's built from multiple convolution layers has its own acronym, *DCGAN*, standing for *Deep Convolutional Generative Adversarial Network*.

Let's train a DCGAN on the MNIST data we've seen in previous chapters. We'll use a model proposed by Gildenblat (2020). The generator and discriminator are shown in Figure 22-21.

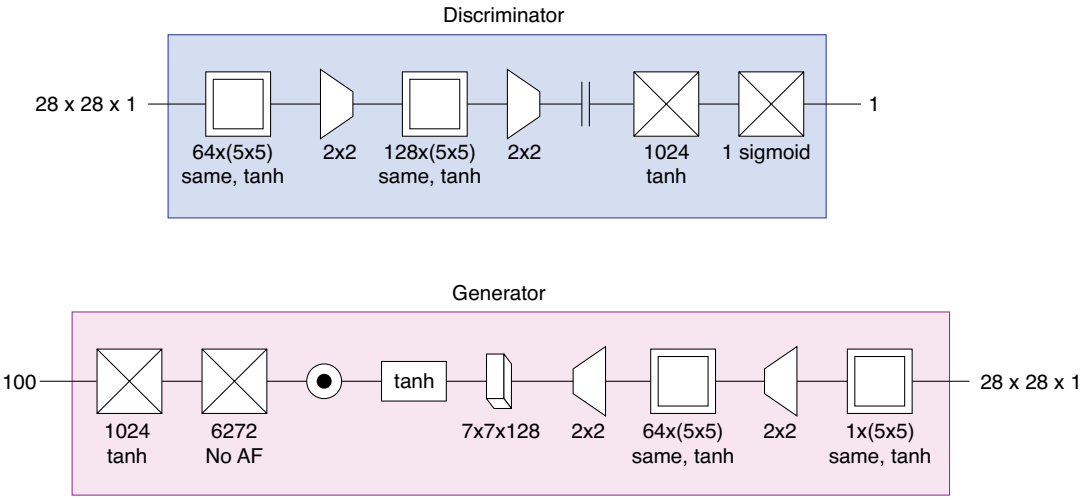


Figure 22-21: Top: The discriminator of a DCGAN for MNIST. Bottom: The generator.

In this network, we're using explicit downsampling (or pooling) layers in the discriminator and upsampling (or expanding) layers in the generator rather than making them part of the convolution steps because that's how the network was originally proposed. The circle with a dot inside it in the generator is a batchnorm layer, which helps to prevent overfitting. The little 3D box after the tanh activation function is a *reshaping* layer that converts the 1D tensor coming out of the second fully connected layer into a 3D tensor, appropriate for the following upsampling and convolution layers. We trained with a standard binary cross entropy loss function and a Nesterov SGD optimizer set to a learning rate of 0.0005 and a momentum of 0.9.

The second dense layer in the generator uses 6,272 neurons. This number might seem mysterious, but it gives the generator and discriminator equal amounts of data to work with. The output of the second downsampling layer in the discriminator has a shape of $7 \times 7 \times 128$, or 6,272 elements. By giving the second fully connected layer in the generator 6,272 values, we can provide a tensor of the same shape to its first upsampling layer. In other words, the end of the convolution stage of the discriminator is a tensor of shape $7 \times 7 \times 128$, so we provide a tensor of shape $7 \times 7 \times 128$ to the start of the convolution stages of the generator.

The discriminator and the generator both follow roughly the same steps, but in opposite order.

The results of the generator after one epoch of training are pretty unintelligible, as we might expect. Figure 22-22 shows what they look like.

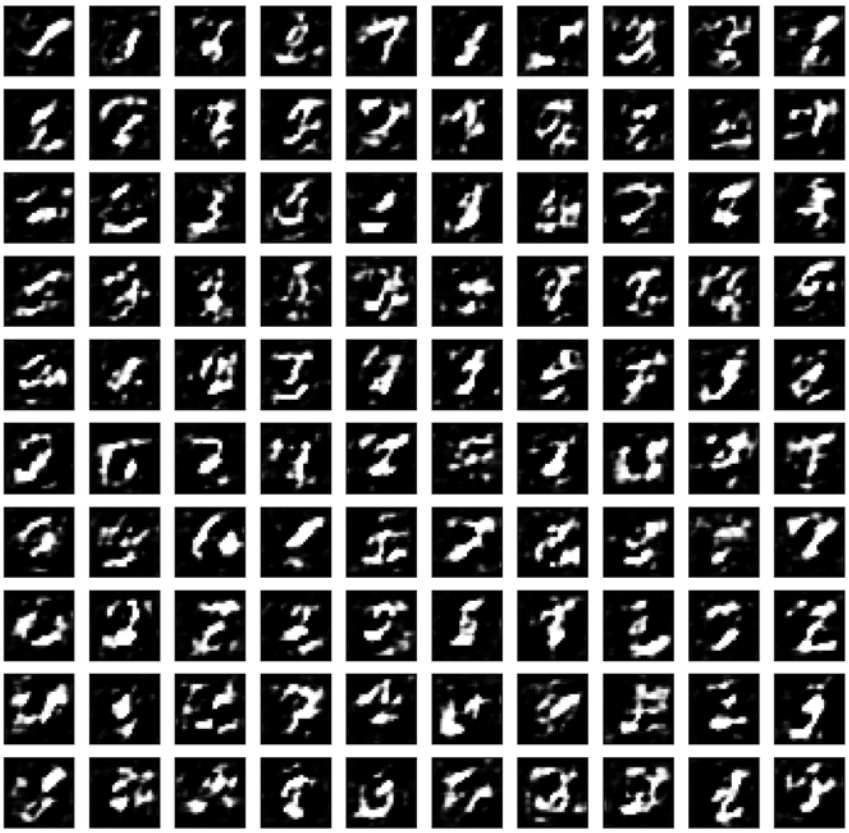


Figure 22-22: The blotches from the generator after one epoch of training

After 100 epochs of training, the generator produced the results of Figure 22-23. We could have trained longer since the discriminator was still sometimes identifying the generator's output, but this seemed a good place to stop because it shows the generator's progress.

When we step back to consider the process, this is a startling result. Remember that the generator has never seen the dataset. It has no idea what the MNIST data looks like. All it's ever done is create 3D tensors of real numbers, and then receive feedback that told it how good or bad the values in those tensors were. Over time, it produced tensors that look like digits. Somehow, the generator managed to find a way to turn random numbers into recognizable digits. Amazing. There are some misfires, but most of the digits are easily recognizable.

This wraps up our basic discussion of GANs.



Figure 22-23: The output of the deep convolutional GAN of Figure 22-21 after 100 epochs of training on the MNIST dataset

Before we move on, it's worth reviewing a bit of practical advice. We mentioned earlier that GANs are very sensitive to their specific architecture and training variables. A famous paper investigated DCGANs, and found a few rules of thumb that seem to lead to good results (Radford, Metz, and Chintala 2016). As always, experimentation is the key to success. Small changes often make the difference between a GAN that learns efficiently and one that learns slowly, or not at all.

Challenges

Perhaps the biggest challenge to using GANs in practice is their sensitivity to both structure and hyperparameters. Playing a game of cat and mouse requires both parties to be closely matched at all times. If either the discriminator or generator gets better than the other too quickly, the other will never be able to catch up. As we mentioned earlier, getting the right combination of all of these values is essential to getting good performance out of a GAN, but finding that combination can be challenging (Arjovsky

and Bottou 2017; Achlioptas et al. 2017). Following the rules of thumb given earlier is generally recommended for giving us a good starting point when training a new DCGAN.

A theoretical issue with GANs is that we currently have no proof that they will *converge*. Recall our lone perceptron of Chapter 13, which finds the dividing line between two linearly separable sets of data. We can *prove* that the perceptron will, given enough training time, always find that dividing line. But for GANs, such proofs are nowhere to be found. All we can say is that many people have found ways to make at least some of their GANs train properly, but there's no guarantee beyond that.

Using Big Samples

The basic structure of a GAN can run into trouble when we try to train a generator to produce large images, such as 1,000 by 1,000 pixels. The computational problem is that with all that data, it's easy for the discriminator to tell the generated fakes from the real images. Trying to fix all these pixels simultaneously can lead to error gradients that cause the generator's output to move in almost random directions, rather than getting closer to matching the inputs (Karras et al. 2018). On top of that, there's the practical problem of finding enough compute power, memory, and time to process large numbers of these big samples. Recall that every pixel is a feature, so every image that's 1,000 pixels on a side has one million features (or three million if it's a color photo).

Because we want our final, high-resolution images to stand up to scrutiny, we're going to want to use a large training set. The time required to crunch through big collections of giant images is going to add up fast. Even fast hardware might not be able to do the job in the time we have available.

A practical approach to building big images is called the *Progressive GAN* or *ProGAN* (Karras et al. 2018). To start with this technique, resize the images in the training set into a variety of smaller sizes, for example 512 pixels on a side, then 128, then 64, and so on, down to 4 pixels on a side. Then build a small generator and discriminator, each with just a few layers of convolution. Train these small networks with the 4 by 4 images. When they are doing a great job, add a few more convolution layers to the end of each network, and gradually blend in their contribution until the networks are doing well with 8 by 8 images. Then add some more convolution layers to the end of each network and train them on 16 by 16 images, and so on.

In this way, the generator and discriminator are able to build on their training as they grow. This means that by the time we work our way up to full size images that are 1,024 pixels on a side, we already have a GAN that can do a great job at generating and discriminating images that are 512 pixels on a side. We won't have to do too much additional training with the larger images until the system performs well with them, too. This process takes much less time to complete than if we'd trained with only the full-sized images from the start.

Modal Collapse

GANs have an interesting way to exploit a loophole in our training. Recall that we want the generator to learn to fool the discriminator. It can succeed at this task in a way that is almost useless to us.

Let's suppose that we're trying to train our GAN to produce pictures of cats. Suppose that the generator manages to create one cat image that the discriminator accepts as real. A sneaky generator could then just produce that image every time. No matter what values we use for the noise inputs, we always get back that one image. The discriminator tells us that every image it gets is plausibly real, so the generator has accomplished its goal and stops learning.

This is another example of neural networks finding sneaky solutions to doing what we ask for, but not necessarily what we want. The generator has accomplished exactly what we requested, since it is turning random numbers into brand-new samples that the discriminator cannot tell apart from real samples. The problem is that every sample made by the generator is identical. A very sneaky kind of success.

This problem of producing just one successful output over and over is called *modal collapse* (note that the first word is *modal*, pronounced "mode'-ull," referring to a mode, or a way of working, and not "model"). If the generator settles into just a single sample (in this case, a single picture of a cat), the situation is described as *full modal collapse*. Much more common is when the system produces the same few outputs, or minor variations of them. This situation is called *partial modal collapse*.

Figure 22-24 shows one run of our DCGAN after three epochs of training using some poorly chosen hyperparameters. It's pretty clear that the system is collapsing toward a mode where it's going to output some kind of 1 a lot more than anything else.

There are schemes for addressing this problem. Perhaps the best recommendation begins with using mini-batches of data, as we did earlier. Then we can extend the discriminator's loss function with some additional terms to measure the diversity of the outputs produced in that mini-batch. If the outputs fall into a few groups where they're all the same, or nearly the same, the discriminator can assign a larger error to the result. The generator then diversifies because that action reduces the error (Arjovsky, Chintala, and Bottou 2017).

Training with Generated Data

The most common use of GANs is to train a generator that fools the discriminator. Then we discard the discriminator, leaving us with a generator capable of creating as much new data as we like, all of it seeming to come from the original dataset. Thus, we can make an unlimited number of new images of cats or sailboats, or spoken phrases, or puffs of smoke from a wood fire.

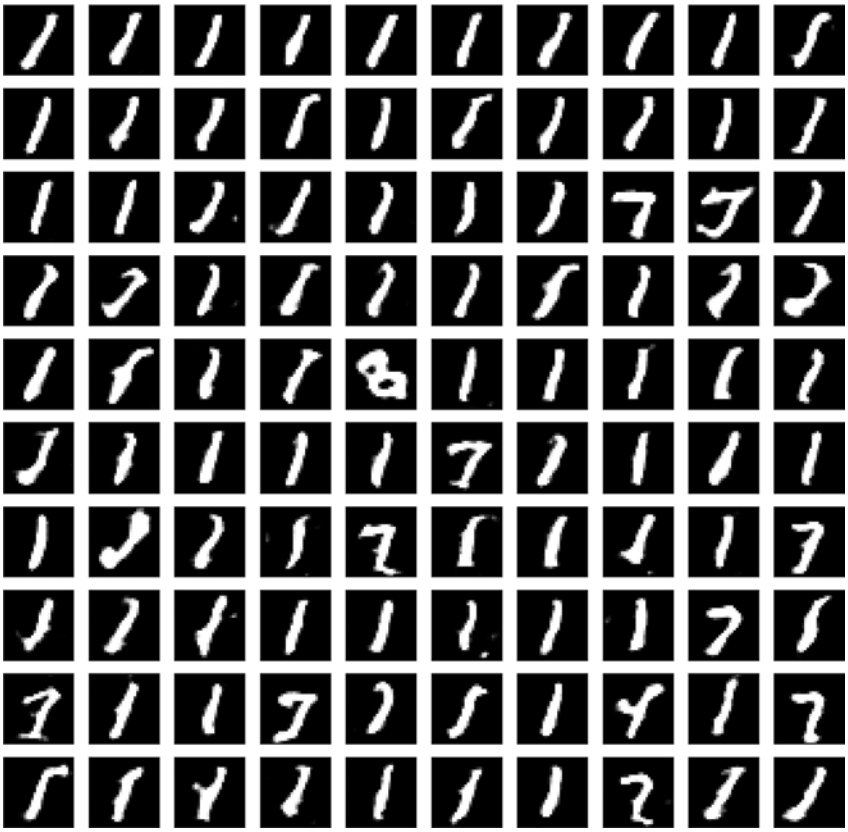


Figure 22-24: After only three epochs of training, this DCGAN is showing clear signs of modal collapse.

It may be tempting to use that generated, or synthetic, data to train another neural network. After all, huge datasets are just what we need to train neural networks. But this is a very risky practice, because our trained generators are rarely perfect. One problem is that it's very hard to make a discriminator that is robust enough to notice every possible detail in the generator's output. The output from the generator might always be slightly skewed in some way that the discriminator was unable to notice, or to which it assigned a very low penalty. Another problem is that the generator's output can be incomplete. As we saw in our example of modal collapse, the generator's results might not span the whole range of the inputs. For example, tasked with generating new paintings in a given artist's style, a generator might always produce landscapes, or portraits, or still lifes, even when the artist's body of work has a much wider range of subjects.

It's very difficult to completely catch every issue that can come up. As hard as we might try to build a perfect generator, it always seems to be able to find another sneaky way to satisfy our desired criteria (as expressed by the discriminator) while still producing data that isn't quite as diverse or realistic as we were hoping for. Another problem is that our criteria themselves often aren't as clear or as broad as we think they are.

In short, the generator's output can contain errors and biases that get past the discriminator. If we train a new system with that data, it inherits those errors and biases, which we may be completely unaware of. The differences may be subtle, but they can still influence the results in practice. This can create a dangerous situation in which we believe we have trained a robust neural network capable of making important decisions, without realizing that it has blind spots and biases. When we use trained networks for critical safety or medical applications, or we use them in social situations like hiring interviews, school admissions, or granting bank loans, we may get back seriously flawed or unfair decisions due to perpetuated errors we're not aware of. The biases, errors, prejudices, misjudgments, and other common problems with databases become the baked-in basis on which the generator creates new data. The result is a self-perpetuating, self-fulfilling, but erroneous system. We can summarize this with a simple credo: *bias in, bias out.*

In short, training neural networks on excellent data can still produce flawed results. Training networks on flawed data can produce much more deeply flawed results. As a general rule, it's usually a good idea to resist the temptation to train a network on synthetic, or generated data.

Summary

In this chapter, we saw how to build a generative adversarial network, or GAN, out of two smaller pieces. The generator learns how to create new data that is plausibly like data from a given data set, and the discriminator learns how to distinguish the generator's output from real data in the given set. They both learn from the other as they train, improving their respective skills. When a successful training is complete, the discriminator is unable to reliably distinguish between the synthetic data and the real data. At that point, we usually throw away the discriminator and use the generator for any application where we want arbitrary amounts of new data.

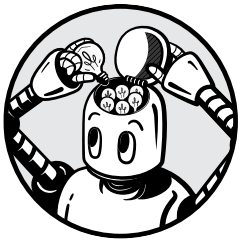
We saw that training takes place in alternating steps so that the generator and discriminator learn at roughly the same pace. We then looked at building a simple GAN with fully connected layers to learn how to make data points in 2D and then a convolutional GAN that learned to generate new image data from MNIST.

Because GANs are notoriously hard to train, we discussed some rules of thumb for convolutional GANs that usually give us a good start. We saw that we can generate large outputs by working our way up in size during training, and we saw that we can use mini-batches to avoid modal collapse, where the generator always produces the same output (or a small number of outputs).

We closed with a brief consideration of the perils of training with synthetic data.

23

CREATIVE APPLICATIONS



We've reached the end of the book. Before we go, let's relax and have some fun. In this chapter we look at some creative ways to use neural networks to create art. We explore two image-based applications: *deep dreaming*, which turns images into wild, psychedelic art, and *neural style transfer*, which allows us to transform photographs into what appear to be paintings in the styles of different artists. At the very end, we take a quick dip into *text generation* and use deep learning to generate even more of this book.

Deep Dreaming

In deep dreaming, we use some ideas that were invented to help us visualize filters in convolutional networks, but we use them to make art. The result is that we modify images to excite different filters, causing those images to explode in psychedelic patterns.

Stimulating Filters

In Chapter 17, we made images, or visualizations, of the filters in a convolutional neural network. Both deep dreaming and style transfer build on that visualization technique, so let's look at it a little more closely. We can make our discussion specific by again using VGG16 as we did in Chapter 17 (Simonyan and Zisserman 2020), though we could substitute just about any CNN image classifier. Our only interest here is in the convolution stages, so although we will use the whole network as described in Chapter 17, the drawings in this chapter show just the convolution and pooling layers, as shown in Figure 23-1.

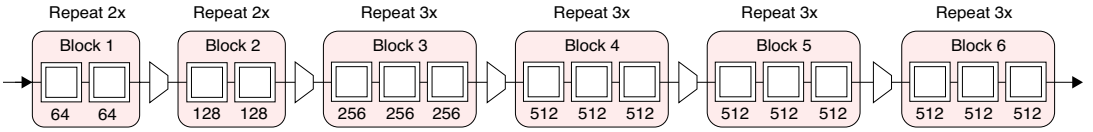


Figure 23-1: A simplified diagram of VGG16, showing just the convolution and pooling layers

We're leaving out the last few stages of VGG16 because their job is to help the network predict the proper class of the output. In this application, we don't care about the network's output. Our only interest here is running an image through the network so that the filters in the convolution layers will evaluate their inputs. Our goal is to modify a starting image so it excites some chosen layers as much as possible. For example, if a few pixels are darker in the middle, that may cause the filter that looks for eyes to respond a little bit. Our goal is to modify those pixels so that they excite that filter more and more, which means that they look more and more like eyes.

We don't need any new tools to do this. All we have to do is pick which filter outputs we want to maximize. We can pick just one filter, or several filters from different parts of the network. Our choice of which filters to use is entirely personal and artistic. Typically, we hunt around, trying out different filters, until we see our input images changing in a way that we like.

Let's see the steps. Suppose we pick one filter on each of three different layers, as in Figure 23-2. We start things off by providing the network with an image, which it processes.

We take the feature map from the first filter we've chosen, add up all of its values, and determine how much influence this sum will have by multiplying it by a weight that we choose. Though we're using the word *weight,* this isn't a weight inside the network. It's just a value we use to control the impact of each filter in the deep dream process. We sum up and weight the other filters we've chosen. Now we add up those results. This gives us a single number, telling us how strongly our chosen filters are responding to the input image, weighted by how much influence we want to give to each layer's filters. We call this number the *multifilter loss,* or the *multilayer loss.*

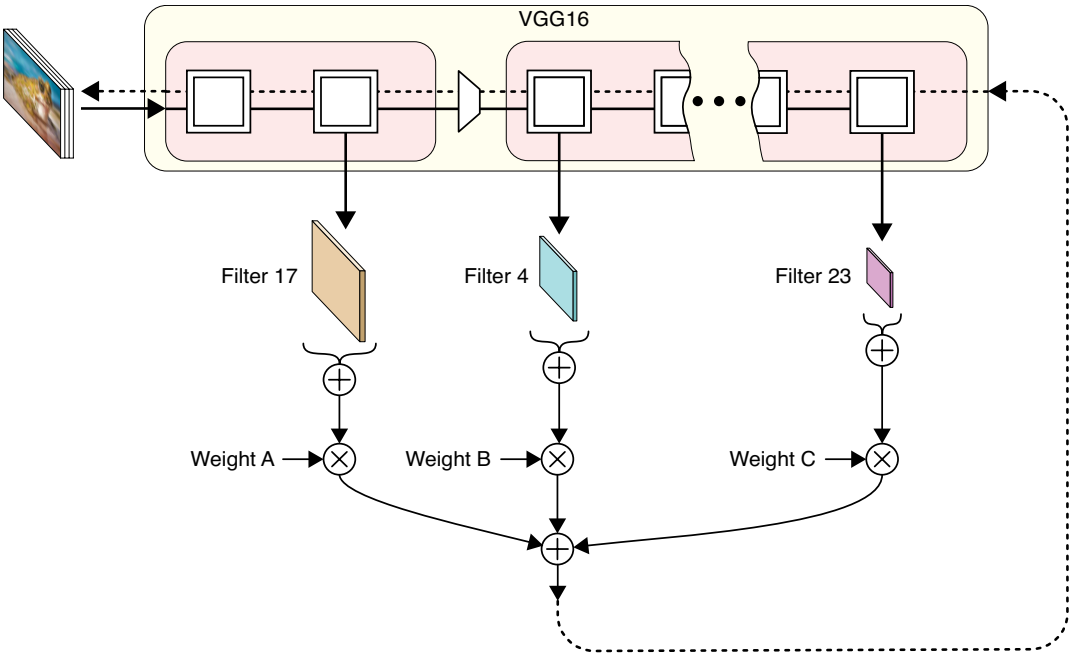


Figure 23-2: The deep dream algorithm uses a loss built from multiple layers

Now comes the tricky part: the multifilter loss becomes the network's "error." In previous chapters, we used the error to drive backprop, which computed gradients for all of the network's weights, starting at the final layer and working our way backward to the first. Then we used those gradients to modify the network's weights to minimize the error. But that's not what we do here. Instead, we want the error (the filter responses) to be as big as we can make them. And we don't want to do this by changing the network, since we're not training it. We're going to *freeze* the network, so its weights can't change. Instead, we're going to modify the colors of the pixels themselves.

So, starting with this error, we use backprop as usual to find the gradients on all the weights in the network, but when we reach the first hidden layer, we take one more step backward to the input layer, which holds the pixels themselves. Then we use backprop as usual to find the gradients for the pixels. After all, changing the input pixels causes the values computed by the network to change, and thus causes a change to our error. Just as we can use backprop to learn how to change the network's weights to reduce the error in a typical training setup, we can use the same backprop algorithm to find out how to change the pixel values to increase this error.

Now, as usual, we apply the optimization step. Since we're not training and the network is frozen, we don't touch the network weights. But we do use the gradients on the pixels to modify their color values so that they *maximize* the error, or more strongly stimulate our selected filters.

The result is that the pixel colors change just a little, in such a way that filters respond even more, creating a bigger error, which we use to find new

gradients on the pixels, causing them to excite the filters even more, and around it goes, with the picture changing more and more each time we repeat the loop.

Since this is an artistic process, we usually watch the output after every update (or every few updates), and stop it when we like what we're seeing.

Running Deep Dreaming

Let's put this algorithm to work, using the frog in Figure 23-3 as our starting point.



Figure 23-3: A calm and thoughtful frog

Figure 23-4 shows some "dreams" from our frog image, using some filters (and weights on them) that we chose by trial and error.

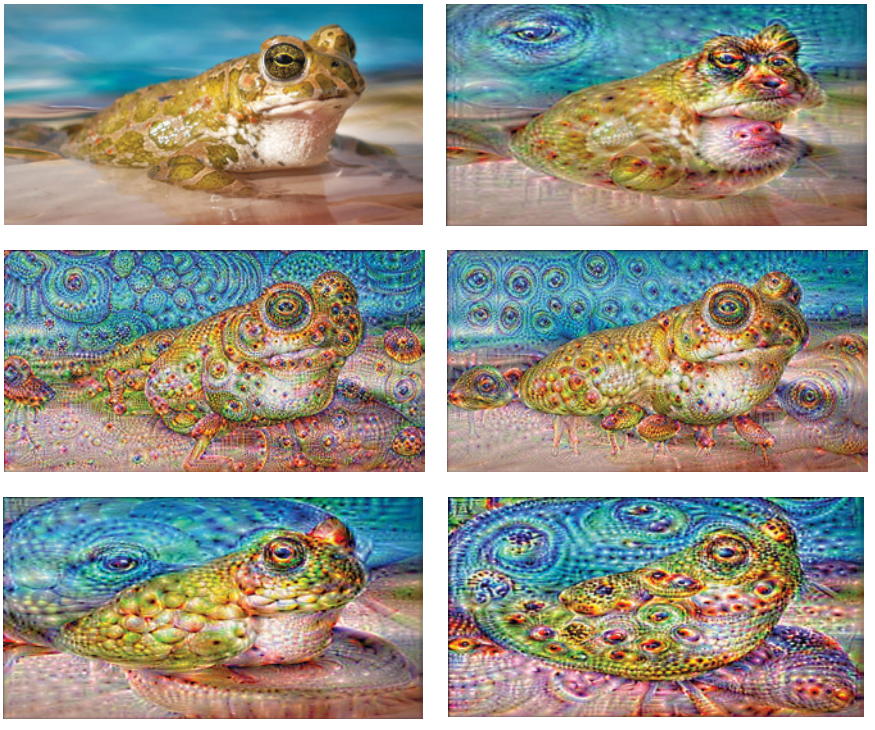


Figure 23-4: Some deep dream results from the starting frog image (in the upper-left corner)

We're seeing lots of eyes in Figure 23-4 because some of our selected filters responded to eyes. If we had selected filters that responded to, say, horses and shoes, then we would expect to see lots of horses and shoes in our images.

Figure 23-5 shows the results starting from an image of a dog. The changes to the image are mostly of a finer texture because the dog image is about 1,000 pixels on each side, more than four times the size of the images that the network trained on. The image in the lower-right used a version of the dog image that was scaled to the same size as the network's training data, 224 by 224.



Figure 23-5: Deep dreaming about a dog

The original name for this technique was *inceptionism* (Mordvintsev, Olah, and Tyka 2015) in honor of the movie *Inception,* but it has come to be more frequently known as *deep dreaming*. The name is a poetic suggestion that the network is "dreaming" about the original image, and the image we get back shows us where the network's dream went. Deep dreaming has become popular not only because of the wild images it creates, but because it's not hard to implement using modern deep learning libraries (Bonaccorso 2020).

Many variations on this basic algorithm have been explored (Tyka 2015), but they only scratch the surface. We can imagine schemes to automatically determine the weights on the layers or even apply weights to the individual filters on each layer. We can "mask" the activation maps before

we add them up so that some areas (like the background) are ignored, or we can mask the updates to the pixels so that some pixels in the original image are not changed at all in response to one set of layer outputs, but are instead allowed to change a lot in response to some other set of layer outputs. We can even apply different combinations of layers and weights to different regions of the input image. The deep dreaming approach to making art has lots of room left for new discoveries.

There's no "right" or "best" way to do deep dreaming. It's a creative exercise in which we follow our aesthetics, hunches, or wild guesses to hunt for images that appeal to us. It can be hard to predict what's going to come out from any particular combination of network, layers, and weights, so the process rewards patience and a lot of experimenting.

Neural Style Transfer

We can use a variation on the deep dreaming technique to do something remarkable: transfer one artist's style onto another image. This process is called *neural style transfer*.

Representing Style

Cultures often celebrate the idiosyncratic visual style of artists. Let's focus on paintings. What characterizes the style of a painting? That's a big question, because "style" can include someone's world view, which influences choices as diverse as their subject matter, composition, materials, and tools. Let's focus strictly on visual appearance. Even narrowed down this way, it's hard to precisely identify what "style" means for a painting, but we might say that it refers to how colors and shapes are used to create forms, and the types and distributions of those forms across the canvas (Art Story Foundation 2020; Wikipedia 2020).

Rather than try to refine this description, let's see if we can find something that seems like it's in the ballpark, while also being something we can formalize in terms of the layers and filters of a deep convolutional network.

Our goal in this section is to take a picture we want to modify, called the *base image*, and a second picture whose style we want to match, called the *style reference*. For example, our frog could be our base image, and any painting could be the style reference. We want to use these to create a new image, called the *generated image*, which has the content of the base image expressed in the style of the style reference.

To get started, we will make an assertion that comes out of nowhere. We say that we can characterize the style of an image (such as a painting) by looking at the layer activations it produces and finding pairs of layers that activate in roughly the same way. This idea comes from a seminal paper published in 2015 (Gatys, Ecker, and Bethge 2015). Without getting into the details, the process begins by running the style reference through a deep convolutional network. As with deep dreaming, we ignore its output, and instead focus on just the convolution filters.

All of the activation maps in a given layer have the same size, so we can easily compare them to one another. Let's start with the first activation map in a layer (that is, the first filter's output). We can compare that to the activation map produced by the second filter and produce a score for the pair. If the two maps are very similar (that is, the filters are firing in the same places), we assign the pair a high score, and if the maps are very different, we give that pair a low score. We then compare the first map to the third map and compute their score, then compare the first and the fourth map, compute their score, and so on. Then we can start with the second map and compare it to every other map in the layer. We can organize the results in a grid, with as many cells on each side as there are filters in the layer. The value in each cell of the grid tells us the score for that pair of layers. This grid is called a *Gram matrix*. Let's make one such Gram matrix for every layer.

Now we can restate our notion of style more formally: the style of an image is represented by the Gram matrices for all the layers. That is, each style produces its own particular form of Gram matrices.

Let's see if this claim is true. Figure 23-6 shows a famous self-portrait by Pablo Picasso from 1907. There's a ton of style here, such as big blocks of color and thick dark lines. Let's run this through VGG16 and save the Gram matrix at each layer. We'll call those the *style matrices*, and save them as the representation of the style for this image.

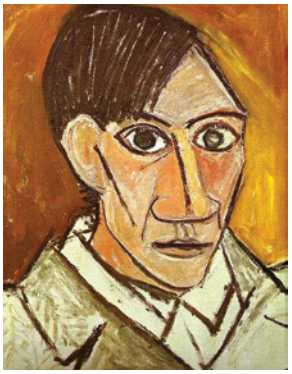


Figure 23-6: A 1907 self-portrait by Pablo Picasso

If the Gram matrices represent style, then we can use them to modify a starting image of random noise. We run the noisy input image through the network and compute its Gram matrices, which we call the *image matrices*. If the style matrices really do somehow represent the style of the Picasso image, then if we can change the colors of the pixels in the noisy image so that eventually the image matrices come close to the style matrices, the noisy image should take on the style of the painting.

Let's do just that. We'll run the noise through the network, compute the image matrix at each layer, and compare that to the style matrix we saved for that layer. We'll add up the differences between these two matrices so the more different they are, the bigger the result. Then we'll add

together these differences for all the layers, and this is the error for our network. As with deep dreaming, we use this error to compute the gradients for the entire network, including the pixels at the start, but we only modify the colors of the pixels. Unlike deep dreaming, our goal now is to minimize the error, and thereby change the pixels so that their colors produce Gram matrices that are like those of the style reference.

Figure 23-7 shows the result of this process. For this visualization, we computed each layer's error as the sum of the differences of all the matrices in all layers up to and including that layer.

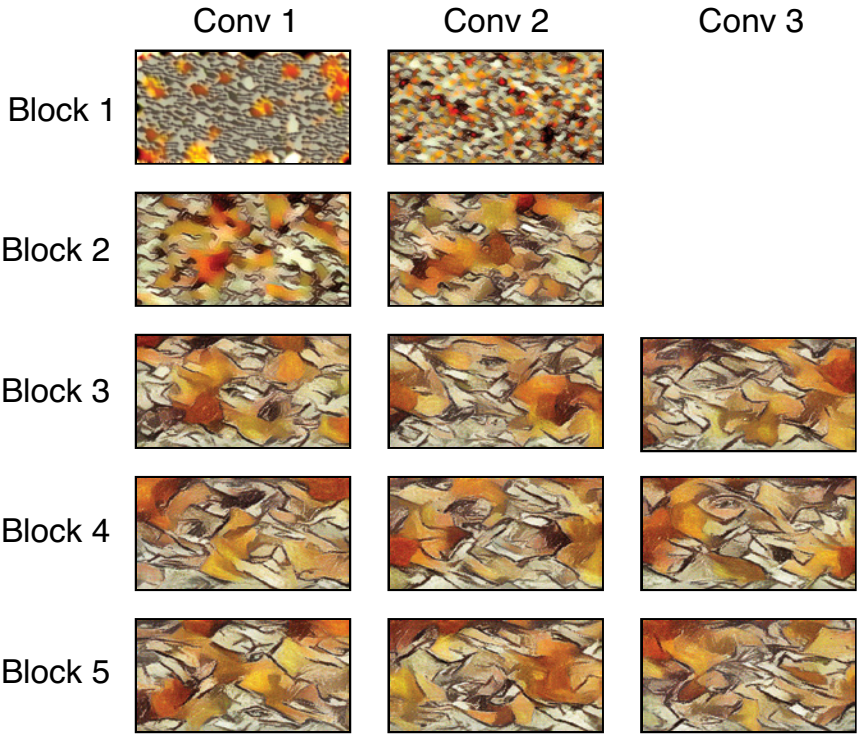


Figure 23-7: The result of getting noise to match the Gram matrices in VGG16

This is remarkable. By the time we get to the three convolution layers in Block 3, we're generating abstracts that are very similar to our original style reference in Figure 23-6. The splotches of color show similar gradual changes in color. There are dark lines between some regions of different colors, and we can even see brushstroke textures.

The Gram matrices have indeed captured the style of Picasso's painting. But why? The anticlimactic answer is that nobody really knows (Li et al. 2017). We have different ways to write down the mathematics of what the Gram matrices are measuring, but that doesn't help us understand why this technique captures this elusive idea we call style. Neither the original paper on neural style transfer (Gatys, Ecker, and Bethge 2015), nor a somewhat more detailed follow-up (Gatys, Ecker, and Bethge 2016) explains how the authors hit on this idea or why it works so well.

Representing Content

In deep dreaming, we started with an image and manipulated it by changing its pixels. If we try the same thing with neural transfer and start with an image rather than noise, the image quickly gets lost. The effect of minimizing the differences between the Gram matrices causes big changes to the input image, moving it toward the style we want, but losing the content of the image in the process.

A solution to this problem is to still start with noise, because it works so well (as shown by Figure 23-7), but retain the essence of the original image by adding a second error term. In addition to imposing a *style loss* that punishes the input for being a poor match to the style reference (as measured by the difference in the Gram matrices), we also impose a *content loss* that punishes the input for being too much unlike the base image (the picture we want to stylize). By starting with noise and adding together these two error terms (usually with different emphasis), we cause the pixels in the noise to change so that they simultaneously more closely match the colors of the picture we want to modify and the style in which we want it to be shown.

Gathering the content loss is easy. We take our base image, like our frog in Figure 23-3, and run it through the network. Then we save the activation map of every filter. From then on, any time we feed a new image to the network, the content loss is just the difference between the filter responses for that input and the responses we got from our base image. After all, if all the filters respond to an input the same way as the starting image, then the input is the starting image (or something very close to it).

Style and Content Together

To recap, we feed our style reference through the network and save the Gram matrix for every pair of filters after each layer. Next, we find a base picture we'd like to stylize, run that through the network, and save the feature maps produced by every filter.

Using this saved data, we can create a stylized version of our picture. We start with noise and feed it to the network. The block diagram of the whole process is shown in Figure 23-8.

Let's start with the content loss. In the light blue rounded-corner rectangle at the far left, we gather up the feature maps from the filters on the first convolution layer and compute the difference between these maps and the ones we saved from the base image (such as the frog). We do the same with the feature maps from the second convolution layer. We can do this for all the layers, but for this figure and the examples that follow we stopped after two (this is another personal choice, guided by experimentation). We add together all of these differences, or content losses, and scale their sum by some value that lets us control how much the content of the picture should influence the changes we ultimately make to the colors of the input image.

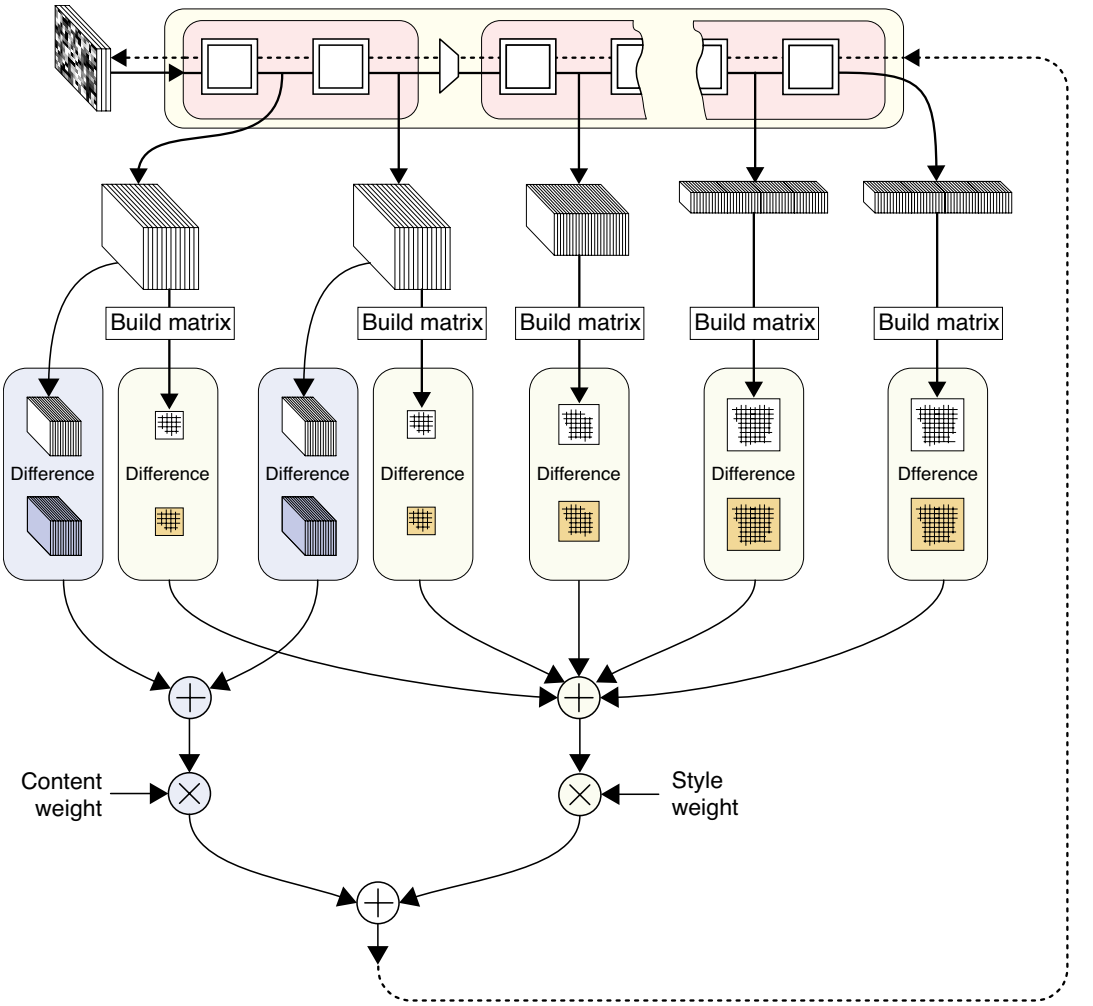


Figure 23-8: A block diagram of neural style transfer

Now we address the style. For each layer, in the light yellow rounded-corner rectangles, we compute the Gram matrix that tells us how much each filter's output corresponds with every other filter's output. We then compare those matrices with the style matrices we saved earlier. We add up all of these differences to get the style loss and scale them by some value that let us control how much influence the style has when we modify the pixel colors.

The sum of the content and style losses is our error. As with deep dreaming, we compute the gradients throughout the network and all the way back to the pixels. And again, we leave the weights in the network untouched. Unlike deep dreaming, we modify the values of the pixels to *minimize* this total error, because we want the input to match the content

and style information we previously saved. The result is that the original noise slowly changes so that it is simultaneously more like the original image and also has the filter relationships of the style.

As with deep dreaming, implementing neural style transfer is straightforward with modern deep learning libraries (Chollet 2017; Majumdar 2020).

Running Style Transfer

Let's see how well this works in practice. We again used VGG16 as our network and followed the process summarized in Figure 23-8.

Figure 23-9 shows nine images, each with a distinctive style. These are our style references.

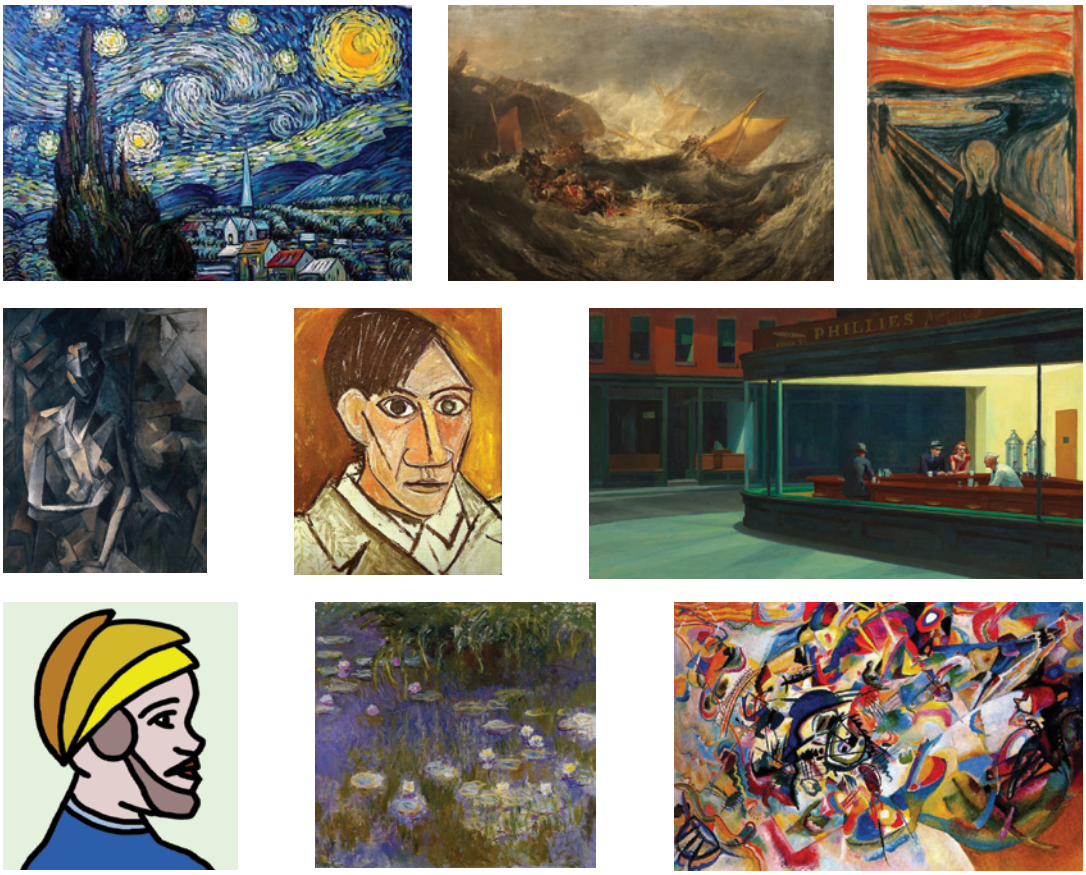


Figure 23-9: Nine images with different styles, which serve as our style references. From left to right and top down, they are Starry Night*, by Vincent van Gogh,* The Shipwreck of the Minotaur*, by J. M. W. Turner,* The Scream*, by Edvard Munch,* Seated Female Nude*, by Pablo Picasso,* Self-Portrait 1907*, by Pablo Picasso,* Nighthawks*, by Edward Hopper,* Sergeant Croce*, by the author,* Water Lilies, Yellow and Lilac*, by Claude Monet, and* Composition VII *by Wassily Kandinsky.*

Let's apply these styles to our old friend the frog. Figure 23-10 shows the results.

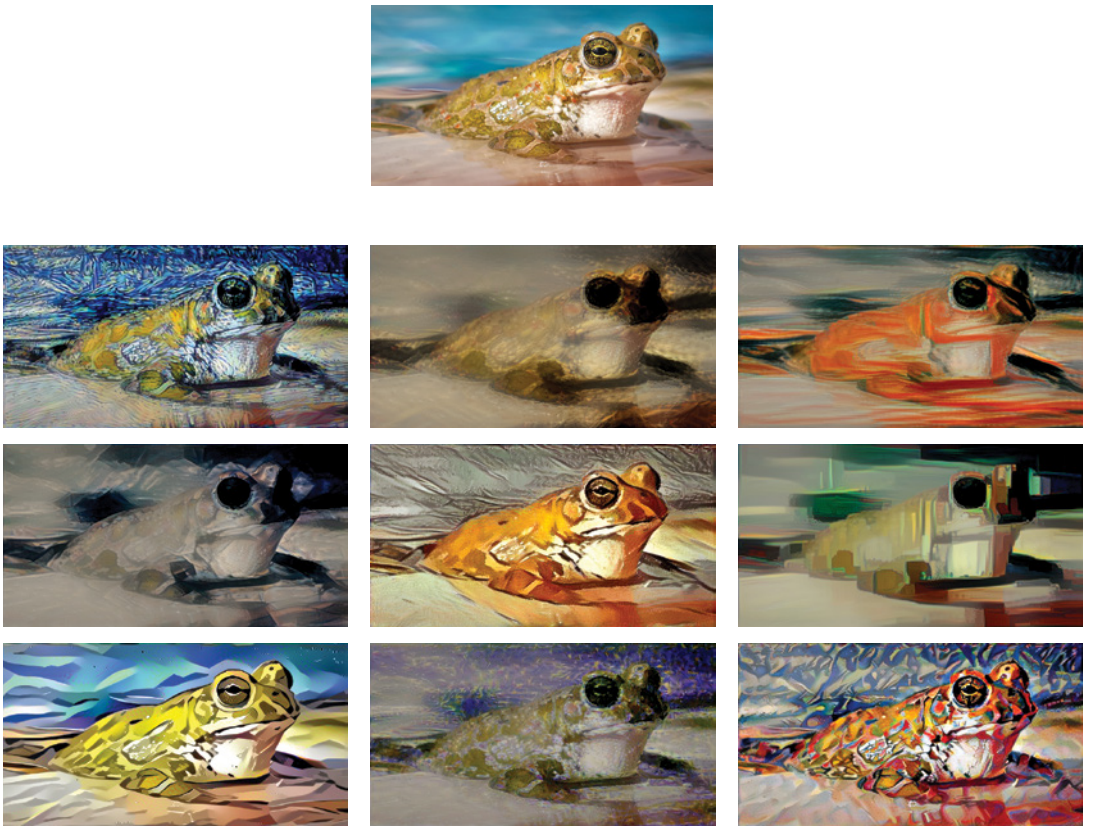


Figure 23-10: Applying the nine styles in Figure 23-9 to a photograph of a frog (top)

Wow. That worked great. These images bear close examination, because they have a lot of detail. At a first glance, we can see that the color palette of each style reference has been transferred to the frog photo. But notice the textures and edges, and how blocks of color are shaped. These images are not just color shifted frogs, or some kind of overlay or blend of two images. Instead, these are high-quality, detailed images of the frog in the different styles. To see this more clearly, Figure 23-11 shows the same zoomed-in region from each frog.

These images are significantly different, and all match the style they're based on.

Let's see another example. Figure 23-12 shows our styles applied to a photograph of a town.

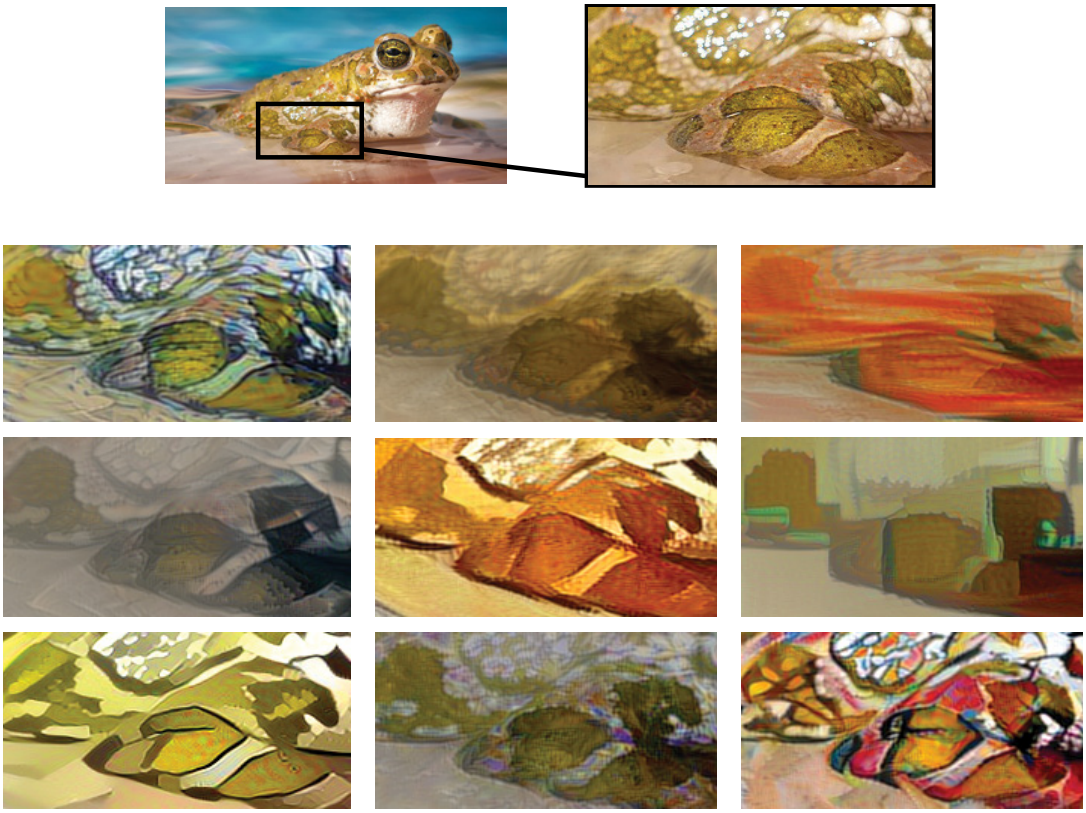


Figure 23-11: Details for the nine styled frogs of Figure 23-10

These images are all the more remarkable when we remember that every one of them started out as random noise. For each image, we weighted the content loss by 0.025 and the style loss by 1, so the style had 40 times more influence on the changes to the pixels than the content did. In these examples, a little bit of content went a long way.

As Figures 23-10 through 23-12 show, the basic algorithm of neural style transfer produces terrific results. The technique has been extended and modified in many ways to improve the flexibility of the algorithm, the types of results it produces, and the range of control that artists can apply to create the results they want (Jing et al. 2018). It's even been applied to video and spherical images that completely surround a viewer (Ruder, Dosovitskiy, and Brox 2018).

As with deep dreaming, neural style transfer is a general algorithm that allows for a lot of variation and exploration. There are surely many interesting and beautiful artistic effects waiting to be discovered.

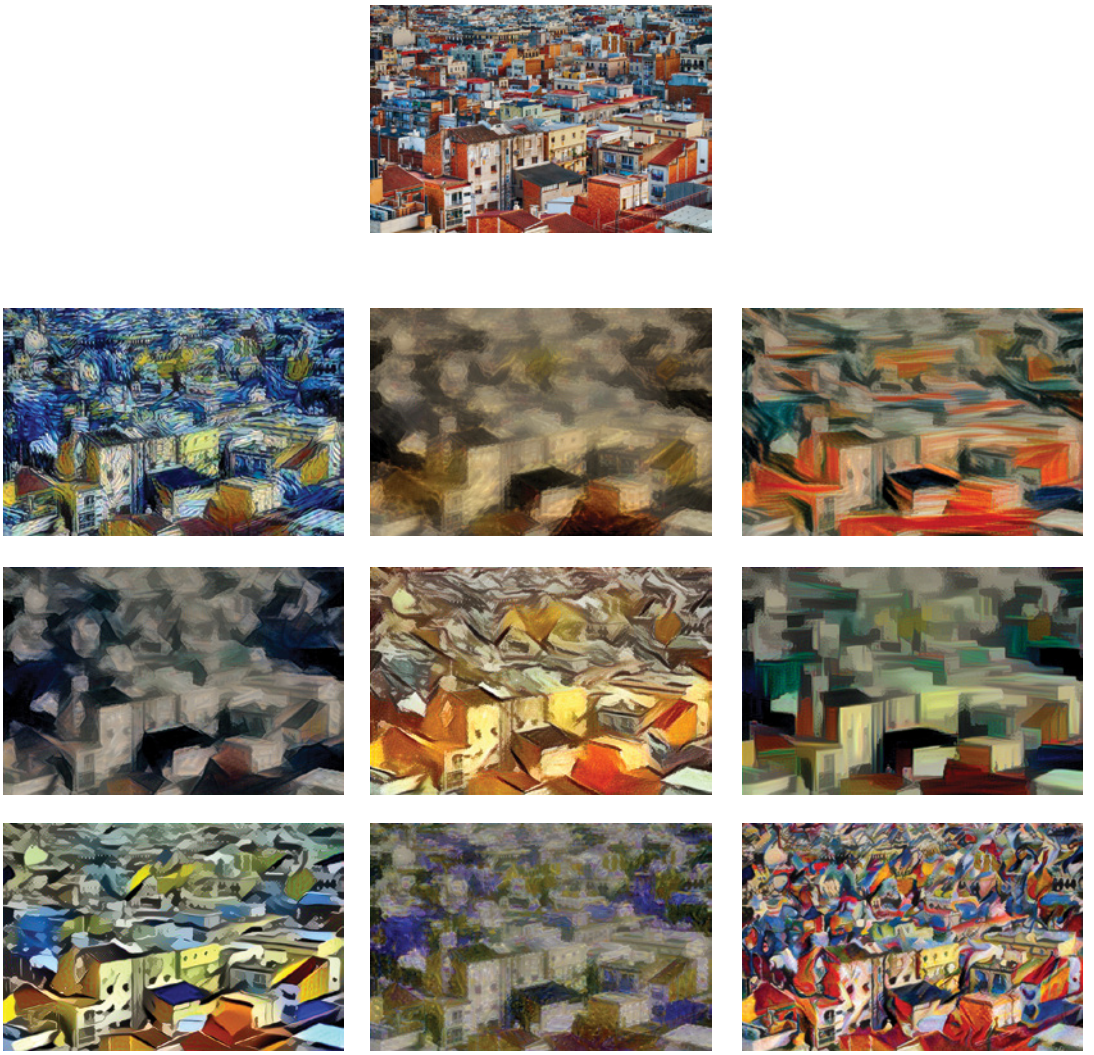


Figure 23-12: Applying our nine styles of Figure 23-9 to a photograph of a town seen from above (top)

Generating More of This Book

Just for fun, we ran the text of the first edition of this book (except for this section) through an RNN that generates new text word by word, as discussed in Chapter 19. The full text contained about 427,000 words, drawn from a vocabulary of about 10,300 words. To learn this text, we used a network built from two layers of LSTMs, with 128 cells each.

The algorithm develops its output autoregressively, by finding the next most likely word based on the text it has created so far, then the next most likely word, then the next, and so on, until we stop it. Generating text this

way is like the game of creating messages by repeatedly choosing just one of the three or four words suggested by a cell phone when you are typing a text (Lowensohn 2014).

Here are a few sentences manually selected from the output after 250 iterations. They are included here exactly as generated, including punctuation.

- The responses of the samples in all the red circles share two numbers, like the bottom of the last step, when their numbers would influence the input with respect to its category.
- The gradient depends on the loss are little pixels on the wall.
- Let's look at the code for different dogs in this syllogism.

It's surprising how close these come to making sense!

Whole sentences are fun, but some of the most entertaining and poetic bits came out soon after the start of training, when the system was generating only fragments. Here are some manually selected excerpts after just 10 epochs, again presented exactly as they were produced:

- Set of of apply, we + the information.
- Suppose us only parametric.
- The usually quirk (alpha train had we than that to use them way up).

These are mostly incoherent, but from these synthetic phrases we can distill a grain of truth: one of the primary goals of this book has definitely been to "+ the information."

RNNs are great, but transformer-based generators are even better. We fine-tuned a medium-sized instance of the GPT-2 generator on the current edition of this book (again, except for this section). Here are a few hand-picked fragments of output, selected for their creative range (the second set appear to be figure captions).

- This is the neural network that's been hailed as the queen of artificial neurons. It's no surprise that her name is Christine, but it does speak volumes about the state of the field.
- We can chain together several of these versions into a single tensor of a classifier that is essentially a jack-in-the-box.
- Let's use a small utility to take a shortcut that will let us make word predictions on the fly, even if we aren't the right person online or offline.
- In this view, a 1 in this range is a perfect integer, while a 0 in this range is a hyper-realized string of numbers. The approach we adopted in Chapter 6 is to treat all 0's as incomplete, and all 1's as incomplete, since we still have some information about the system we're evaluating.
- Figure 10-7: A grid of negative images that don't have labels.
- Figure 10-10: A deep learning system learns how to create and remove license plates from a dataset.

Summary

We started this chapter by looking at deep dreaming, a method for manipulating an image to stimulate chosen filters in a network and create wild, psychedelic images. Then we looked at neural style transfer. Using this technique, we slowly change an input of random noise to become simultaneously more like an input image and like a style reference, such as works by various painters. Finally, we used a little RNN and a transformer to produce new text from the manuscript of this book. It's fun to make new text that seems familiar or reasonable on first glance!

Final Thoughts

This book has covered only the basic ideas of deep learning. The field is developing at a startling pace. Every year new breakthroughs seem to defy all expectations of what computers can recognize, analyze, predict, and synthesize.

Just keeping up with new work can be a full-time job. One way to stay up on new developments is to watch a website called the arXiv (pronounced "archive") server at *https://arxiv.org/*, or more specifically, the machine learning section at *https://arxiv.org/list/cs.LG/recent*. This site hosts preprints of new papers before they appear in journals and conferences. But even watching arXiv can be overwhelming, so many people use the arXiv Sanity Preserver at *http://www.arxiv-sanity.com* and the Semantic Sanity project at *https://s2-sanity.apps.allenai.org/cold-start*. Both sites help filter the repository for just those papers involving specific keywords and ideas.

In this book we've focused on the technical background of deep learning. It's important to keep in mind that when we use these systems in ways that can affect people, we need to take into account much more than just algorithms (O'Neil 2016). Because of their efficiency, deep learning systems are being widely and rapidly deployed, often with little oversight or consideration of their impact on the societies they affect.

Deep learning systems are being used today to influence or determine job offers, school admissions, jail sentences, personal and business loans, and even the interpretation of medical tests. Deep learning systems control what people see in their news and social media feeds, choosing items not to build a healthy and well-informed society, but to increase the profits of the organizations delivering those feeds (Orlowski 2020). Deep learning systems in "smart speakers" and "smart displays" (which also contain microphones and cameras) listen and watch people in their homes without pause, sometimes sending their captured data up to a remote server for analysis. Cultures used to fear such constant surveillance, but now people pay for these devices and willingly put them in their previously private spaces, such as their homes and bedrooms. Deep learning is being used to single out potentially troublesome children in schools, evaluate the honesty of people answering questions at border crossings, and identify individuals at protests and other gatherings by recognizing their faces. Mistakes made by these

algorithms can vary from annoying to fundamentally life-changing. Even when the results are not incorrect, these systems are increasingly making profound decisions that affect our public and private lives.

Deep learning systems are only as good as their training and their algorithms, and time after time, we find that biases, prejudices, and outright errors in the training data are perpetuated and even enforced by the resulting algorithms. Such systems fall far short of the kinds of accuracy and fairness we expect when dealing with humans and other living creatures. Our algorithms completely lack empathy and compassion. They have no sense of exceptional circumstances, and have no conception of the joy or suffering their decisions can cause. They cannot understand love, kindness, fear, hope, gratitude, sorrow, generosity, wisdom, or any of the other qualities that we value in one another. They cannot admire, cry, smile, grieve, celebrate, or regret.

Deep learning holds great promise to help us as individuals and societies. But any tool can be used as a weapon, benefiting the owners of that tool to the disadvantage of those affected. Machine learning systems are often effectively invisible, so any systematic errors can go undetected for long periods. Even once problems are uncovered, it can require enormous social and political action to hold accountable the organizations that benefit by selling or using these systems, and even more effort to bring about change.

Another peril of machine learning systems is their insatiable need for enormous amounts of training data. This creates a market for organizations that do nothing but collect, organize, and sell previously private information about people's lives, from their friendships and family relationships to where and when they like to travel, what food they like to eat, what medications they're taking, and what their DNA reveals about them. This data can be used to harass, intimidate, threaten, and harm individuals.

The demand for massive quantities of data also means that as an organization grows larger (and often less accountable), the more data it can gather, and the more powerful its algorithms grow, making their decisions more influential, which means they can gather more data, and thereby solidify the organization's power in a feedback loop. Every imperfection in such a system becomes magnified, and because of their scale, negative effects on individuals can go entirely unnoticed by the people running these systems. In general, the only competition to such organizations will come from other organizations of equivalent size, offering systems with their own enormous databases containing their own biases and errors, leading to the battles of the biased behemoths we're familiar with today. This continuous and increasing concentration of power is a dangerous force in a free society when not subject to significant and strongly enforced control and regulation. Sadly, such controls are rarely to be seen today.

Deep learning has produced algorithms that make it possible to take any person's appearance, from an entertainer to a politician, and produce images, audio, and video that seem to realistically portray that person saying or doing anything the person wielding the software desires. Societies have come to rely on captured audio, photographs, and video to enforce contracts, reconcile conflicts, exalt or shame a public person, influence

elections, and serve as evidence in courts. The end of that era is very near, returning us to a time before reliable photographs, recordings, and video, when hearsay, memory, and opinion replace objective historical recordings. Without reliable audio and visual evidence documenting what someone actually said or did, the loudest, richest, or most persuasive voices in the room will determine many outcomes in public opinion, elections, and courts of law, because objective facts will be increasingly harder to find or trust.

Deep learning is a fascinating field, and we're just starting to understand how it will affect our culture and society. Learning algorithms will surely continue to grow in scope and impact. They have the chance to create enormous good by helping people be happier, enabling societies to be more fair and supportive, and creating healthier and more diverse personal, social, political, and physical environments. It's important to strive for these positive outcomes, even when they curtail corporate profits or governmental control.

We should remember to always use deep learning, like all of our tools, to bring out the best in humanity, and make the world a better place for everyone.

REFERENCES

Whenever possible, I have preferred to use references that are available online so that you can immediately access them. The exceptions are usually books and other printed matter, but occasionally I've included an important online reference even if it's behind a paywall. Every link here was live as of the time of writing. But the Internet is nothing if not volatile, and some of these links are sure to change, or simply stop working. If you find a link doesn't work, I suggest using a search engine to look for the title and/or author of the reference you want to locate. Often you'll find that it's only moved to a new location. If it's gone, you can try finding a saved copy on the Wayback Machine at *https://archive.org/web/*. You can also try searching for the reference using the Google search engine, which will sometimes offer a cached version of a page that is no longer live.

If you're reading this as an ebook, you can simply click on any link to follow it. If you're reading this on paper, you can type any of these links into your favorite browser, but a better approach is to go to the online copy of this chapter at No Starch Press, where the links are live: *https://nostarch.com/deep-learning-visual-approach/*.

Chapter 1

Bishop, Christopher M. 2006. *Pattern Recognition and Machine Learning*. New York: Springer-Verlag. Available at *https://docs.google.com/viewer?a=v&pid=sites&srcid=aWFtYW5kaS5ldXxpc2N8Z3g6MjViZDk1NGI1NjQzOWZiYQ*.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2017. *Deep Learning*. Cambridge, MA: MIT Press. Available at *https://www.deeplearningbook.org/*.

Saba, Luca, Mainak Biswas, Venkatanareshbabu Kuppili, Elisa Cuadrado Godia, Harman S. Suri, Damodar Reddy Edla, Tomaž Omerzu, John R. Laird, Narendra N. Khanna, Sophie Mavrogeni, et al. 2019. "The Present and Future of Deep Learning in Radiology." *European Journal of Radiology* 114 (May): 14–24.

Chapter 2

Anscombe, F. J. 1973. "Graphs in Statistical Analysis." *American Statistician* 27, no. 1 (February): 17–21.

Banchoff, Thomas F. 1990. *Beyond the Third Dimension: Geometry, Computer Graphics, and Higher Dimensions*. Scientific American Library Series. New York: W. H. Freeman.

Brownlee, Jason. 2017. "How to Calculate Bootstrap Confidence Intervals for Machine Learning Results in Python." *Machine Learning Mastery* (blog). Last updated on August 14, 2020. *https://machinelearningmastery.com/calculate-bootstrap-confidence-intervals-machine-learning-results-python/*.

Efron, Bradley, and Robert J. Tibshirani. 1993. *An Introduction to the Bootstrap*. Boca Raton, FL: Chapman and Hall/CRC, Taylor and Francis Group.

Matejka, Justin, and George Fitzmaurice. 2017. "Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics Through Simulated Annealing." In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, CO, May 6–11): 1290–94. *https://www.autodesk.com/research/publications/same-stats-different-graphs*.

Norton, John D. 2014. "What Is a Four Dimensional Space Like?" Lecture notes. Department of History and Philosophy of Science, University of Pittsburgh, PA. *https://www.pitt.edu/~jdnorton/teaching/HPS_0410/chapters/four_dimensions/index.html*.

Teknomo, Kardi. 2015. "Bootstrap Sampling Tutorial." Revoledu. *https://people.revoledu.com/kardi/tutorial/Bootstrap/index.html*.

ten Bosch, Marc. 2020. "*N*-Dimensional Rigid Body Dynamics." *ACM Transactions on Graphics* 39, no. 4 (July). *https://marctenbosch.com/ndphysics/NDrigidbody.pdf*.

Wikipedia. 2017a. "Anscombe's Quartet." Last modified June 21, 2020. *https://en.wikipedia.org/wiki/Anscombe%27s_quartet*.

Wikipedia. 2017b. "Random Variable." Last modified August 21, 2020. *https://en.wikipedia.org/wiki/Random_variable*.

Chapter 3

Glen, Stephanie. 2014. "Marginal Distribution." Statisticshowto.com. February 6, 2014. *http://www.statisticshowto.com/marginal-distribution/*.

Jaynes, Edwin Thompson. 2003. *Probability Theory: The Logic of Science*. Cambridge, UK: Cambridge University Press.

Kirby, Roger. 2011. *Small Gland, Big Problem*. London, UK: Health Press.

Kunin, Daniel, Jingru Guo, Tyler Dae Devlin, and Daniel Xiang. 2020. "Seeing Theory." Seeing Theory. Accessed September 16, 2020. *https://seeing-theory.brown.edu/#firstPage*.

Levitin, Daniel J. 2016. *A Field Guide to Lies: Critical Thinking in the Information Age*. New York: Viking Press.

Walpole, Ronald E., Raymond H. Myers, Sharon L. Myers, and Keying E. Ye. 2011. *Probability and Statistics for Engineers and Scientists*, 9th ed. New York: Pearson.

Wikipedia. 2020. "Sensitivity and Specificity." Last updated October 20, 2020. *https://en.wikipedia.org/wiki/Sensitivity_and_specificity*.

Chapter 4

Cthaeh, The. 2016a. "Bayes' Theorem: An Informal Derivation." *Probabilistic World*. February 28, 2016. *http://www.probabilisticworld.com/anatomy-bayes-theorem/*.

Cthaeh, The. 2016b. "Calculating Coin Bias with Bayes' Theorem." *Probabilistic World*. March 21, 2016. *http://www.probabilisticworld.com/calculating-coin-bias-bayes-theorem/*.

Genovese, Christopher R. 2004. "Tutorial on Bayesian Analysis (in Neuroimaging)." Paper presented at the Institute for Pure and Applied Mathematics Conference, University of California, Los Angeles, July 20, 2004. *http://www.stat.cmu.edu/~genovese/talks/ipam-04.pdf*.

Kruschke, John K. 2014. *Doing Bayesian Data Analysis: A Tutorial with R, JAGS, and Stan*, 2nd ed. Cambridge, MA: Academic Press.

Stark, P. B., and D. A. Freedman. 2016. "What Is the Chance of an Earthquake?" UC Berkeley Department of Statistics, Technical Report 611, October 2016. *https://www.stat.berkeley.edu/~stark/Preprints/611.pdf*.

VanderPlas, Jake. 2014. "Frequentism and Bayesianism: A Python-Driven Primer." Cornell University, Astrophysics, arXiv:1411.5018, November 18, 2014. *https://arxiv.org/abs/1411.5018*.

Chapter 5

3Blue1Brown. 2020. 3Blue1Brown home page. Accessed September 1, 2020. *https://www.3blue1brown.com*.

Apostol, Tom M. 1991. *Calculus, Vol. 1: One-Variable Calculus, with an Introduction to Linear Algebra*, 2nd ed. New York: Wiley.

Berkey, Dennis D., and Paul Blanchard. 1992. *Calculus*. Boston: Houghton Mifflin Harcourt School.

Chapter 6

Bellizzi, Courtney. 2011. "A Forgotten History: Alfred Vail and Samuel Morse." Smithsonian Institution Archives. May 24, 2011. *http://siarchives.si.edu/blog/forgotten-history-alfred-vail-and-samuel-morse*.

Ferrier, Andrew. 2020. "A Quick Tutorial on Generating a Huffman Tree." *Andrew Ferrier* (tutorial). Accessed November 12, 2020. *https://www.andrewferrier.com/oldpages/huffman_tutorial.html*.

Huffman, David A. 1952. "A Method for the Construction of Minimum-Redundancy Codes." In *Proceedings of the IRE* 40, no. 9. *https://web.stanford.edu/class/ee398a/handouts/papers/Huffman%20-%20Min%20Redundancy%20Codes%20-%20IRE52.pdf*.

Kurt, Will. 2017. "Kullback-Leibler Divergence Explained." *Probably a Probability* (blog), Count Bayesie. May 10, 2017. *https://www.countbayesie.com/blog/2017/5/9/kullback-leibler-divergence-explained*.

Longden, George. 1987. "G3ZQS' Explanation of How FISTS Got Its Name." FISTS CW Club. Accessed September 1, 2020. *https://fists.co.uk/g3zqsintroduction.html*.

McEwen, Neal. 1997. "Morse Code or Vail Code?" The Telegraph Office. *http://www.telegraph-office.com/pages/vail.html*.

Pope, Alfred. 1887. "The American Inventors of the Telegraph, with Special References to the Services of Alfred Vail." *The Century Illustrated Monthly Magazine* 35, no. 1 (November). *http://tinyurl.com/jobhn2b.*

Serrano, Luis. 2017. "Shannon Entropy, Information Gain, and Picking Balls from Buckets." *Medium*. November 5, 2017. *https://medium.com/udacity/shannon-entropy-information-gain-and-picking- balls-from-buckets-5810d35d54b4.*

Seuss, Dr. 1960. *Green Eggs and Ham*. Beginner Books.

Shannon, Claude E. 1948. "A Mathematical Theory of Communication." *Bell Labs Technical Journal* (July). *http://people.math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf.*

Stevenson, Robert Louis. 1883. *Treasure Island*. Project Gutenberg. Available at *https://www.gutenberg.org/ebooks/120.*

Thomas, Andrew. 2017. "An Introduction to Entropy, Cross Entropy and KL Divergence in Machine Learning." *Adventures in Machine Learning* (blog). March 29, 2017.

Twain, Mark. 1885. *Adventures of Huckleberry Finn (Tom Sawyer's Comrade)*. Charles L. Webster and Company. Available at *https://www.gutenberg.org/ebooks/32325.*

Wikipedia. 2020. "Letter Frequency." Wikipedia. Last modified August 31, 2020. *https://en.wikipedia.org/wiki/Letter_frequency.*

Chapter 7

Aggarwal, Charu C., Alexander Hinneburg, and Daniel A. Keim. 2001. "On the Surprising Behavior of Distance Metrics in High Dimensional Space." Conference paper presented at the International Conference on Database Theory 2001 in London, UK, January 4–6, 2001. *https://bib.dbvis.de/uploadedFiles/155.pdf.*

Arcuri, Lauren. 2019. "How to Candle an Egg." *The Spruce* (blog). April 20, 2019. *https://www.thespruce.com/definition-of-candling-3016955.*

Bellman, Richard Ernest. 1957. *Dynamic Programming*. Princeton, NJ: Princeton University Press.

Domingos, Pedro. 2012. "A Few Useful Things to Know About Machine Learning." *Communications of the ACM* 55, no. 10 (October). *https://homes.cs.washington.edu/~pedrod/papers/cacm12.pdf.*

Hughes, G. F. 1968. "On the Mean Accuracy of Statistical Pattern Recognizers." *IEEE Transactions on Information Theory* 14, no. 1: 55–63.

Nebraska Extension. 2017. "Candling Eggs." Nebraska Extension in Lancaster County, University of Nebraska-Lincoln. *http://lancaster.unl.edu/4h/embryology/candling.*

Numberphile. 2017. "Strange Spheres in Higher Dimensions." YouTube. September 18, 2017. *https://www.youtube.com/watch?v=mceaM2_zQd8.*

Spruyt, Vincent. 2014. "The Curse of Dimensionality." *Computer Vision for Dummies* (blog). April 16, 2014. *http://www.visiondummy.com/2014/04/curse-dimensionality-affect-classification/.*

Chapter 8

Muehlhauser, Luke. 2011. "Machine Learning and Unintended Consequences." *LessWrong* (blog). September 22, 2011. *http://lesswrong.com/lw/7qz/machine_learning_and_unintended_consequences/.*

Schneider, Jeff, and Andrew W. Moore. 1997. "Cross Validation." In *A Locally Weighted Learning Tutorial Using Vizier 1.0*. Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, February 1, 1997. *https://www.cs.cmu.edu/~schneide/tut5/node42.html*.

Chapter 9

Bishop, Christopher M. 2006. *Pattern Recognition and Machine Learning*. New York: Springer-Verlag.

Bullinaria, John A. 2015. "Bias and Variance, Under-Fitting and Over-Fitting." *Neural Computation, Lecture 9* (lecture notes), University of Birmingham, United Kingdom. *http://www.cs.bham.ac.uk/~jxb/INC/l9.pdf*.

Domke, Justin. 2008. "Why Does Regularization Work?" *Justin Domke's Weblog* (blog). December 12, 2008. *https://justindomke.wordpress.com/2008/12/12/why-does-regularization-work/*.

Domingos, Pedro. 2015. *The Master Algorithm*. New York: Basic Books.

Foer, Joshua. 2012. *Moonwalking with Einstein: The Art and Science of Remembering Everything*. New York: Penguin Books.

Macskassy, Sofus A. 2008. "Machine Learning (CS 567) Notes." (PowerPoint presentation, Bias-Variance. Fall 2008. *http://www-scf.usc.edu/~csci567/17-18-bias-variance.pdf*.

Proctor, Philip, and Peter Bergman. 1978. "Brainduster Memory School," from *Give Us A Break*, Mercury Records. *https://www.youtube.com/watch?v=PD2Uh_TJ9X0*.

Chapter 10

Centers for Disease Control and Prevention. 2020. "Body Mass Index (BMI)." CDC.gov. June 30, 2020. *https://www.cdc.gov/healthyweight/assessing/bmi/*.

Crayola. 2020. "What Were the Original Eight (8) Colors in the 1903 Box of Crayola Crayons?" Accessed on September 10, 2020. *http://www.crayola.com/faq/your-history/what-were-the-original-eight-8-colors-in-the-1903-box-of-crayola-crayons/*.

Turk, Matthew, and Alex Pentland. 1991. "Eigenfaces for Recognition." *Journal of Cognitive Neuroscience* 3, no. 1. *http://www.face-rec.org/algorithms/pca/jcn.pdf*.

Chapter 11

Bishop, Christopher M. 2006. *Pattern Recognition and Machine Learning*. New York: Springer.

Raschka, Sebastian. 2015. *Python Machine Learning*. Birmingham, UK: Packt Publishing.

Steinwart, Ingo, and Andreas Christmann. 2008. *Support Vector Machines*. New York: Springer.

VanderPlas, Jake. 2016. *Python Data Science Handbook*. Sebastopol, CA: O'Reilly.

Chapter 12

Bonab, Hamed, R., and Fazli Can. 2016. "A Theoretical Framework on the Ideal Number of Classifiers for Online Ensembles in Data Streams." In *Proceedings of the 25th ACM International Conference on Information and Knowledge Management (CIKM), 2016* (October): 2053–56.

Ceruzzi, Paul. 2015. "Apollo Guidance Computer and the First Silicon Chips." Smithsonian National Air and Space Museum, October 14, 2015. *https://airandspace.si.edu/stories/editorial/apollo-guidance-computer-and-first-silicon-chips.*

Freund, Y., and R. E. Schapire. 1997. "A Decision-Theoretic Generalization of On-line Learning and an Application to Boosting." *Journal of Computer and System Sciences* 55 (1): 119–39.

Fumera, Giorgio, Roli Fabio, and Serrau Alessandra. 2008. "A Theoretical Analysis of Bagging as a Linear Combination of Classifiers." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 30, no. 7: 1293–99.

Kak, Avinash. 2017. "Decision Trees: How to Construct Them and How to Use Them for Classifying New Data." RVL Tutorial Presentation at Purdue University, August 28, 2017. *https://engineering.purdue.edu/kak/Tutorials/DecisionTreeClassifiers.pdf.*

RangeVoting.org. 2020. "Glossary of Voting-Related Terms." Accessed on September 16, 2020. *http://rangevoting.org/Glossary.html.*

Schapire, Robert E., and Yoav Freund. 2012. *Boosting Foundations and Algorithms.* Cambridge, MA: MIT Press.

Schapire, Robert E. 2013. "Explaining Adaboost." in *Empirical Inference: Festschrift in Honor of Vladimir N. Vapnik.* Berlin, Germany: Springer-Verlag. Available at *http://rob.schapire.net/papers/explaining-adaboost.pdf.*

Chapter 13

Clevert, Djork-Arné, Thomas Unterthiner, and Sepp Hochreiter. 2016. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)." Cornell University, Computer Science, arXiv:1511.07289, February 22, 2016. *https://arxiv.org/abs/1511.07289.*

Estebon, Michele D. 1997. "Perceptrons: An Associative Learning Network," Virginia Institute of Technology. *http://ei.cs.vt.edu/~history/Perceptrons.Estebon.html.*

Furber, Steve. 2012. "Low-Power Chips to Model a Billion Neurons." *IEEE Spectrum* (July 31). *http://spectrum.ieee.org/computing/hardware/lowpower-chips-to-model-a-billion-neurons.*

Glorot, Xavier, and Yoshua Bengio. 2010. "Understanding the Difficulty of Training Deep Feedforward Neural Networks," In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS), 2010* (Chia Laguna Resort, Sardinia, Italy): 249–56. *http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf.*

Goldberg, Joseph. 2015. "How Different Antidepressants Work." WebMD Medical Reference (August). *http://www.webmd.com/depression/how-different-antidepressants-work.*

Goodfellow, Ian J., David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. 2013. "Maxout Networks." In *Proceedings of the 30th International Conference on Machine Learning (PMLR)* 28, no. 3: 1319–27. *http://jmlr.org/proceedings/papers/v28/goodfellow13.pdf.*

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." Cornell University, Computer Science, arXiv:1502.01852, February 6, 2015. *https://arxiv.org/abs/1502.01852.*

Julien, Robert M. 2011. *A Primer of Drug Action*, 12th ed. New York: Worth Publishers.

Khanna, Asrushi. 2018. "Cells of the Nervous System." *Teach Me Physiology* (blog). Last modified August 2, 2018. *https://teachmephysiology.com/nervous-system/components/cells-nervous-system.*

Kuphaldt, Tony R. 2017. "Introduction to Diodes and Rectifiers, Chapter 3 - Diodes and Rectifiers." In *Lessons in Electric Circuits, Volume III. All About Circuits.* Accessed on September 18, 2020. *https://www.allaboutcircuits.com/textbook/semiconductors/chpt-3/introduction-to-diodes-and-rectifiers/.*

LeCun, Yann, Leon Bottou, Genevieve B. Orr, and Klaus-Rober Müller. 1998. "Efficient BackProp." In *Neural Networks: Tricks of the Trade*, edited by Grégoire Montavon, Genevieve B. Orr, and Klaus-Rober Müller. Berlin: Springer-Verlag. 9–48. *http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf.*

Limmer, Steffen, and Slawomir Stanczak. 2017. "Optimal Deep Neural Networks for Sparse Recovery via Laplace Techniques." Cornell University, Computer Science, arXiv:1709.01112, September 26, 2017. *https://arxiv.org/abs/1709.01112.*

Lodish, Harvey, Arnold Berk, S. Lawrence Zipursky, Paul Matsudaira, David Baltimore, and James Darnell. 2000. *Molecular Cell Biology*, 4th edition. New York: W. H. Freeman; 2000. *http://www.ncbi.nlm.nih.gov/books/NBK21535/.*

McCulloch, Warren S., and Walter Pitts. 1943. "A Logical Calculus of the Ideas Immanent in Nervous Activity." *Bulletin of Mathematical Biophysics* 5, no. 1/2: 115–33. *http://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf.*

Meunier, David, Renaud Lambiotte, Alex Fornito, Karen D. Ersche, and Edward T. Bullmore. 2009. "Hierarchical Modularity in Human Brain Functional Networks." *Frontiers in Neuroinformatics* (October 30). *https://www.frontiersin.org/articles/10.3389/neuro.11.037.2009/full.*

Minsky, Martin, and Seymour Papert. 1969. *Perceptrons: An Introduction to Computational Geometry.* Cambridge, MA: MIT Press.

Oppenheim, Alan V., and S. Hamid Nawab. 1996. *Signals and Systems*, 2nd ed. Upper Saddle River, NJ: Prentice Hall.

Purves, Dale, George J. Augustine, David Fitzpatrick, Lawrence C. Katz, Anthony-Samuel LaMantia, Jomes O. McNamara, and S. Mark Williams. 2001. *Neuroscience*, 2nd edition, Sunderland, MA: Sinauer Associates. *http://www.ncbi.nlm.nih.gov/books/NBK11117/.*

Ramachandran, Prajit, Barret Zoph, and Quoc V. Le. 2017. "Swish: A Self-Gated Activation Function." Cornell University, Computer Science, arXiv:1710.05941, October 27, 2017. *https://arxiv.org/abs/1710.05941.*

Rosenblatt, Frank. 1962. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.* Washington, DC: Spartan.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams, 1986. "Learning Representations by Back-Propagating Errors." *Nature* 323, no. 9: 533–36. *https://www.cs.utoronto.ca/~hinton/absps/naturebp.pdf.*

Serre, Thomas. 2014. "Hierarchical Models of the Visual System." (Research notes), Cognitive Linguistic and Psychological Sciences Department, Brain Institute for Brain Sciences, Brown University. *https://serre-lab.clps.brown.edu/wp-content/uploads/2014/10/Serre-encyclopedia_revised.pdf.*

Seung, Sebastian. 2013. *Connectome: How the Brain's Wiring Makes Us Who We Are.* Boston: Mariner Books.

Sitzmann, Vincent, Julien N. P. Martel, Alexander W. Bergman, David B. Lindell, and Gordon Wetzstein. 2020. "Implicit Neural Representations with Periodic Activation Functions." Cornell University, Computer Science, arXiv:2006.09661, June 17, 2020. *https://arxiv.org/abs/2006.09661*.

Sporns, Olaf, Giulio Tononi, and Rolf Kötter. 2005. "The Human Connectome: A Structural Description of the Human Brain." *PLoS Computational Biology* 1 no. 4 (September 2005). *http://journals.plos.org/ploscompbiol/article/file?id=10.1371/journal.pcbi.0010042&type=printable*.

Timmer, John. 2014. "IBM Researchers Make a Chip Full of Artificial Neurons." *Ars Technica*. August 7, 2014. *https://arstechnica.com/science/2014/08/ibm-researchers-make-a-chip-full-of-artificial-neurons/*.

Trudeau, Richard J. 1994. *Introduction to Graph Theory*, 2nd ed. Garden City, NY: Dover Books on Mathematics.

Wikipedia. 2020a. "History of Artificial Intelligence." Last modified September 4, 2020. *https://en.wikipedia.org/wiki/History_of_artificial_intelligence*.

Wikipedia. 2020b. "Neuron." Last modified September 11, 2020. *https://en.wikipedia.org/wiki/Neuron*.

Wikipedia. 2020c. "Perceptron." Last modified August 28, 2020. *https://en.wikipedia.org/wiki/Perceptron*.

Chapter 14

Dauphin, Yann, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. 2014. "Identifying and Attacking the Saddle Point Problem in High-Dimensional Non-Convex Optimization." Cornell University, Computer Science, arXiv:1406.2572, June 10, 2014. *http://arxiv.org/abs/1406.2572*.

Fullér, Robert. 2010. "The Delta Learning Rule Tutorial." Institute for Advanced Management Systems Research, Department of Information Technologies, Åbo Adademi University, November 4, 2010. *http://uni-obuda.hu/users/fuller.robert/delta.pdf*.

NASA. 2012. "Astronomers Predict Titanic Collision: Milky Way vs. Andromeda." *NASA Science Blog*, Production editor Dr. Tony Phillips, May 31, 2012. *https://science.nasa.gov/science-news/science-at-nasa/2012/31may_andromeda*.

Nielsen, Michael A. 2015. "Using Neural Networks to Recognize Handwritten Digits." In *Neural Networks and Deep Learning*. Determination Press. Available at *http://neuralnetworksanddeeplearning.com/chap1.html*.

Pyle, Katherine. 1918. *Mother's Nursery Tales*. New York: E. F. Dutton & Company. Available at *https://www.gutenberg.org/files/49001/49001-h/49001-h.htm#Page_207*.

Quote Investigator, 2020. "You Just Chip Away Everything That Doesn't Look Like David." Quote Investigator: Tracing Quotations. Accessed October 26, 2020. *https://quoteinvestigator.com/tag/michelangelo/*.

Seung, Sebastian. 2005. "Introduction to Neural Networks." 9.641J course notes, Spring 2005. MITOpenCourseware, Massachusetts Institute of Technology. *https://ocw.mit.edu/courses/brain-and-cognitive-sciences/9-641j-introduction-to-neural-networks-spring-2005/*.

Chapter 15

Bengio, Yoshua. 2012. "Practical Recommendations for Gradient-Based Training of Deep Architectures." Cornell University, Computer Science, arXiv:1206.5533. *https://arxiv.org/abs/1206.5533v2*.

Darken, C., J. Chang, and J. Moody. 1992. "Learning Rate Schedules for Faster Stochastic Gradient Search." In *Neural Networks for Signal Processing II, Proceedings of the 1992 IEEE Workshop* (September): 1–11.

Dauphin, Y., R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio. 2014. "Identifying and Attacking the Saddle Point Problem in High-Dimensional Non-convex Optimization." Cornell University, Computer Science, arXiv:1406.2572, June 10, 2014. *http://arxiv.org/abs/1406.2572*.

Duchi, John, Elad Hazan, and Yoram Singer. 2011. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." *Journal of Machine Learning Research* 12, no. 61: 2121–59. *http://jmlr.org/papers/v12/duchi11a.html*.

Hinton, Geoffrey, Nitish Srivastava, and Kevin Swersky. 2015. "Neural Networks for Machine Learning: Lecture 6a, Overview of Mini-Batch Gradient Descent." (Lecture slides). University of Toronto, Computer Science. *https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf*.

Ioffe, Sergey, and Christian Szegedy. 2015. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." Cornell University, Computer Science, arXiv:1502.03167, March 2, 2015. *https://arxiv.org/abs/1502.03167*.

Karpathy, Andrej. 2016. "Neural Networks Part 3: Learning and Evaluation." (Course notes for Stanford CS231n.) Stanford CA: Stanford University. *http://cs231n.github.io/neural-networks-2/*.

Kingma, Diederik. P., and Jimmy L. Ba. 2015. "Adam: A Method for Stochastic Optimization." Conference paper for the 3rd International Conference on Learning Representations (San Diego, CA, May 7–9): 1–13.

Nesterov, Y. 1983. "A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence o(1/k2)." *Doklady ANSSSR* (trans. as *Soviet Mathematics: Doclady*) 269: 543–47.

Orr, Genevieve. 1999a. "Learning Rate Adaptation." In *CS-449: Neural Networks.* (Course notes.) Salem, OR: Willamette University. *https://www.willamette.edu/~gorr/classes/cs449/intro.html*.

Orr, Genevieve. 1999b. "Momentum." In *CS-449: Neural Networks.* (Course notes.) Salem, OR: Willamette University. *https://www.willamette.edu/~gorr/classes/cs449/intro.html*.

Qian, N. 1999. "On the Momentum Term in Gradient Descent Learning Algorithms." *Neural Networks* 12(1): 145–51. *http://www.columbia.edu/~nq6/publications/momentum.pdf*.

Ruder, Sebastian. 2017. "An Overview of Gradient Descent Optimization Algorithms." Cornell University, Computer Sciences, arXiv:1609.04747. June 15, 2017. *https://arxiv.org/abs/1609.04747*.

Srivasta, Nitish, Geoffrey Hinton, Alex Krizhevsky, and Ilya Sutskever. 2014. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research* 15 (2014): 1929–58. *https://jmlr.org/papers/volume15/srivastava14a.old/srivastava14a.pdf*.

Wolpert, David H. 1996. "The Lack of A Priori Distinctions Between Learning Theorems." *Neural Computation* 8, 1341–90. *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.51.9734&rep=rep1&type=pdf*.

Wolpert, D. H., and W. G. Macready. 1997. "No Free Lunch Theorems for Optimization." *IEEE Transactions on Evolutionary Computation* 1, no. 1: 67–82. *https://ti.arc.nasa.gov/m/profile/dhw/papers/78.pdf*.

Zeiler, Matthew D. 2012. "ADADELTA: An Adaptive Learning Rate Method." Cornell University, Computer Science, arXiv:1212.5701. *http://arxiv.org/abs/1212.5701*.

Chapter 16

Aitken, Andrew, Christian Ledig, Lucas Theis, Jose Caballero, Zehan Wang, and Wenzhe Shi. 2017. "Checkerboard Artifact Free Sub-pixel Convolution: A Note on Sub-pixel Convolution, Resize Convolution and Convolution." Cornell University, Computer Science, arXiv:1707.02937, July 10, 2017. *https://arxiv.org/abs/1707.02937*.

Britz, Denny. 2015. "Understanding Convolutional Neural Networks for NLP," *WildML* (blog), November 7, 2015. *http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/*.

Canziani, Alfredo, Adam Paszke, and Eugenio Culurciello. 2017. "An Analysis of Deep Neural Network Models for Practical Applications." Cornell University, Computer Science, ArXiv:1605.07678, April 4, 2016. *https://arxiv.org/abs/1605.07678*.

Culurciello, Eugenio. 2017. "Neural Network Architectures." *Medium: Towards Data Science*, March 23, 2017. *https://medium.com/towards-data-science/neural-network-architectures-156e5bad51ba*.

Dumoulin, Vincent, and Francesco Visin. 2018. "A Guide to Convolution Arithmetic for Deep Learning." Cornell University, Computer Science, arXiv:1603.07285, January 11, 2018. *https://arxiv.org/abs/1603.07285*.

Esteva, Andre, Brett Kuprel, Roberto A. Novoa, Justin Ko, Susan M. Swetter, Helen M. Blau, and Sebastian Thrun. 2017. "Dermatologist-Level Classification of Skin Cancer with Deep Neural Networks." *Nature* 542 (February 2): 115–18. *http://cs.stanford.edu/people/esteva/nature/*.

Ewert, J. P. 1985. "Concepts in Vertebrate Neuroethology." *Animal Behaviour* 33, no. 1 (February): 1–29.

Glorot, Xavier, and Yoshua Bengio. 2010. "Understanding the Difficulty of Training Deep Feedforward Neural Networks." In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (Sardinia, Italy, May 13–15): 249–56. *http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf*.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." Cornell University, Computer Science, arXiv:1502.01852, February 6, 2015. *https://arxiv.org/abs/1502.01852v1*.

Kalchbrenner, Nal, Edward Grefenstette, and Phil Blunsom. 2014. "A Convolutional Neural Network for Modelling Sentences." Cornell University, Computer Science, arXiv:1404.2188v1, April 8, 2014. *https://arxiv.org/abs/1404.2188*.

Karpathy, Andrej. 2016. "Optimization: Stochastic Gradient Descent." Course notes for Stanford CS231n, Stanford, CA: Stanford University. *http://cs231n.github.io/neural-networks-2/*.

Kim, Yoon. 2014. "Convolutional Neural Networks for Sentence Classification." Cornell University, Computer Science, arXiv:1408.5882, September 3, 2014. *https://arxiv.org/abs/1408.5882*.

Levi, Gil, and Tal Hassner. 2015. "Age and Gender Classification Using Convolutional Neural Networks." IEEE Workshop on Analysis and Modeling of

Faces and Gestures (AMFG), at the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (Boston, June). *http://www.openu.ac.il/home/hassner/projects/cnn_agegender/.*

Lin, Min, Qiang Chen, and Shuicheng Yan. 2014. "Network in Network." Cornell University, Computer Science, arXiv:1312-4400v3, March 4, 2014. *https://arxiv.org/abs/1312.4400v3.*

Mao, Xiao-Jiao, Chinhua Shen, and Yu-Bin Yang. 2016. "Image Restoration Using Convolutional Auto-encoders with Symmetric Skip Connections." Cornell University, Computer Science, arXiv:16.06.08921v3, August 30, 2016. *https://arxiv.org/abs/1606.08921.*

Memmott, Mark. "Do You Suffer from RAS Syndrome?" *'Memmos': Memmott's Missives and Musings*, NPR, 2015. *https://www.npr.org/sections/memmos/2015/01/06/605393666/do-you-suffer-from-ras-syndrome.*

Odena, Augustus, Vincent Dumoulin, and Chris Olah. 2016. "Deconvolution and Checkerboard Artifacts." *Distill.* October 17, 2016. *https://distill.pub/2016/deconv-checkerboard/.*

Oppenheim, Alan V., and S. Hamid Nawab. 1996. *Signals and Systems*, 2nd ed. Upper Saddle River, NJ: Prentice Hall.

Quiroga, R. Quian, L. Reddy, G. Kreiman, C. Koch, and I. Fried, 2005. "Invariant Visual Representation by Single Neurons in the Human Brain." *Nature* 435 (June 23): 1102–07. *https://www.nature.com/articles/nature03687.*

Serre, Thomas. 2014. "Hierarchical Models of the Visual System." In Jaeger D., Jung R. (eds), *Encyclopedia of Computational Neuroscience.* New York: Springer. *https://link.springer.com/referenceworkentry/10.1007%2F978-1-4614-7320-6_345-1.*

Snavely, Noah. "CS1114 Section 6: Convolution." Course notes for Cornell CS1114, Introduction to Computing Using Matlab and Robotics, Ithaca, NY: Cornell University, February 27, 2013. *https://www.cs.cornell.edu/courses/cs1114/2013sp/sections/S06_convolution.pdf.*

Springenberg, Jost Tobias, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. 2015. "Striving for Simplicity: The All Convolutional Net." Cornell University, Computer Science, arXiv:1412.6806, April 13, 2015. *https://arxiv.org/abs/1412.6806.*

Sun, Y., X. Wang, and X. Tang. 2014. "Deep Learning Face Representation from Predicting 10,000 Classes." Conference paper, for the *2014 IEEE Conference on Computer Vision and Pattern Recognition* (Columbus, OH, June 23–28): 1891–98.

Zeiler, Matthew D., Dilip Crishnana, Graham W. Taylor, and Rob Fergus. 2010. "Deconvolutional Networks." Conference paper for the Computer Vision and Pattern Recognition conference (June 13–18). *https://www.matthewzeiler.com/mattzeiler/deconvolutionalnetworks.pdf.*

Zhang, Richard. 2019. "Making Convolutional Networks Shift-Invariant Again." Cornell University, Computer Science, arXiv:1904.11486, June 9, 2019. *https://arxiv.org/abs/1904.11486.*

Chapter 17

Chollet, François. 2017. "Keras-team/Keras." GitHub. *https://keras.io/.*

Fei-Fei, Li, Jia Deng, Olga Russakovsky, Alex Berg, and Kai Li. 2020. "Download." ImageNet website. Stanford Vision Lab. Stanford University/Princeton University. Accessed October 4, 2020. *http://image-net.org/download.*

ImageNet. 2020. "Results of ILSVRC2014: Classification + Localization Results." Stanford Vision Lab. Stanford University/Princeton University. Accessed October 4, 2020. *http://image-net.org/challenges/LSVRC/2014/results*.

LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. "Backpropagation Applied to Handwritten Zip Code Recognition." *Neural Computing* 1(4): 541–51. Available at *http://yann.lecun.com/exdb/publis/pdf/lecun-89e.pdf*.

Moosavi-Dezfooli, Seyed-Mohsen, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. 2017. "Universal Adversarial Perturbations." Cornell University, Computer Science, arXiv:1610.08401, March 9, 2017. *https://arxiv.org/abs/1610.08401*.

Rauber, Jonas, and Wieland Brendel. 2017. "Welcome to Foolbox Native." Foolbox. *https://foolbox.readthedocs.io/en/latest*.

Rauber, Jonas, Wieland Brendel, and Matthias Bethge. 2018. "Foolbox: A Python Toolbox to Benchmark the Robustness of Machine Learning Models." Cornell University, Computer Science, arXiv:1707.04131, March 20, 2018. *https://arxiv.org/abs/1707.04131*.

Russakovsky, Olga, et al. 2015. "ImageNet Large Scale Visual Recognition Challenge." Cornell University, Computer Science, arXiv:1409.0575, January 30, 2015. *https://arxiv.org/abs/1409.0575*.

Simonyan, Karen, and Andrew Zisserman. 2015. "Very Deep Convolutional Networks for Large-Scale Image Recognition." Cornell University, Computer Science, arXiv:1409.1556, April 10, 2015. *https://arxiv.org/abs/1409.1556*.

Zeiler, Matthew D., and Rob Fergus. 2013. "Visualizing and Understanding Convolutional Networks." Cornell University, Computer Science, arXiv: 1311.2901, November 28, 2013. *https://arxiv.org/abs/1311.2901*.

Chapter 18

Altosaar, Jann. 2020. "Tutorial: What Is a Variational Autoencoder?" *Jaan Altosaar* (blog). Accessed on September 30, 2020. *https://jaan.io/what-is-variational-autoencoder-vae-tutorial/*.

Audio Mountain. 2020. "Audio File Size Calculations." AudioMountain.com Tech Resources. Accessed on September 30, 2020. *http://www.audiomountain.com/tech/audio-file-size.html*.

Bako, Steve, Thijs Vogels, Brian McWilliams, Mark Meyer, Jan Novák, Alex Harvill, Prdeep Sen, Tony DeRose, and Fabrice Rousselle. 2017. "Kernel-Predicting Convolutional Networks for Denoising Monte Carlo Renderings." In *Proceedings of SIGGRAPH 17, ACM Transactions on Graphics* 36, no. 4 (Article 97). *https://s3-us-west-1.amazonaws.com/disneyresearch/wp-content/uploads/20170630135237/Kernel-Predicting-Convolutional-Networks-for-Denoising-Monte-Carlo-Renderings-Paper33.pdf*.

Chaitanya, Chakravarty R. Alla, Anton Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. 2017. "Interactive Reconstruction of Monte Carlo Image Sequences Using a Recurrent Denoising Autoencoder." In *Proceedings of SIGGRAPH 17, ACM Transactions on Graphics* 36, no. 4 (July 1). *http://research.nvidia.com/publication/interactive-reconstruction-monte-carlo-image-sequences-using-recurrent-denoising*.

Chollet, François. 2017. "Building Autoencoders in Keras." *The Keras Blog*, March 14, 2017. *https://blog.keras.io/building-autoencoders-in-keras.html*.

Doersch, Carl. 2016. "Tutorial on Variational Autoencoders." Cornell University, Statistics, arXiv:1606.05908, August 13, 2016. *https://arxiv.org/abs/1606.05908*.

Donahue, Jeff. 2015. "mnist_autoencoder.prototxt." BVLC/Caffe. GitHub. February 5, 2015. *https://github.com/BVLC/caffe/blob/master/examples/mnist/mnist_autoencoder.prototxt*.

Dürr, Oliver. 2016. "Introduction to Variational Autoencoders." Presentation at Datalab-Lunch Seminar Series, Winterthur, Switzerland, May 11, 2016. *https://tensorchiefs.github.io/bbs/files/vae.pdf*.

Frans, Kevin. "Variational Autoencoders Explained." (Tutorial) Kevin Frans website, August 6, 2016. *http://kvfrans.com/variational-autoencoders-explained/*.

Jia, Yangqing, and Evan Shelhamer, 2020. "Caffe." Berkeley Vision online documentation, Accessed October 1, 2020. *http://caffe.berkeleyvision.org/*.

Kingma, Diederik P., and Max Welling, "Auto-Encoding Variational Bayes." Cornell, Statistics, arXiv:1312.6114, May 1, 2014. *https://arxiv.org/abs/1312.6114*.

Raffel, Colin. 2019. "A Few Unusual Autoencoders." Slideshare.net. February 24, 2019. *https://www.slideshare.net/marlessonsa/a-few-unusual-autoencoder-colin-raffel*.

Rezende, Danilo Jimenez, Shakir Mohamed, and Daan Wierstra. 2014. "Stochastic Backpropagation and Approximate Inference in Deep Generative Models." In *Proceedings of the 31st International Conference on Machine Learning (ICML), JMLR: W\&CP* 32 (May 30). *https://arxiv.org/abs/1401.4082*.

Wikipedia. 2020a. "JPEG." Accessed on September 30, 2020. *https://en.wikipedia.org/wiki/JPEG*.

Wikipedia. 2020b. "MP3." Accessed on September 30, 2020. *https://en.wikipedia.org/wiki/MP3*.

Chapter 19

Barrat, Robbie. 2018. "Rapping-Neural-Network." GitHub, October 29, 2018. *https://github.com/robbiebarrat/rapping-neural-network*.

Bryant, Alice. 2019. "A Simple Sentence with Seven Meanings." *VOA Learning English: Everyday Grammar* (blog). May 16, 2019. *https://learningenglish.voanews.com/a/a-simple-sentence-with-seven-meanings/4916769.html*.

Chen, Yutian, Matthew W. Hoffman, Sergio Gómez Colmenarejo, Misha Denil, Timothy P. Lillicrap, Matt Botvinick, and Nando de Freitas. 2017. "Learning to Learn Without Gradient Descent by Gradient Descent." Cornell University, Statistics, arXiv:1611.03824v6, June 12, 2017. *https://arxiv.org/abs/1611.03824*.

Chen, Qiming, and Ren Wu. 2017. "CNN Is All You Need." Cornell University, Computer Science, arXiv:1712.09662, December 27, 2017. *https://arxiv.org/abs/1712.09662*.

Chollet, Francois. 2017. "A Ten-Minute Introduction to Sequence-to-Sequence Learning in Keras." *The Keras Blog*, September 29, 2017. *https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html*.

Chu, Hang, Raquel Urtasun, and Sanja Fidler. 2016. "Song from PI: A Musically Plausible Network for Pop Music Generation." Cornell University, Computer Science, arXiv:1611.03477, November 10, 2016. *https://arxiv.org/abs/1611.03477*.

Deutsch, Max. 2016a. "Harry Potter: Written by Artificial Intelligence." *Deep Writing* (blog), Medium. July 8, 2016. *https://medium.com/deep-writing/harry-potter-written-by-artificial-intelligence-8a9431803da6*.

Deutsch, Max. 2016b. "Silicon Valley: A New Episode Written by AI." *Deep Writing* (blog), Medium, July 11, 2016. *https://medium.com/deep-writing/silicon-valley-a-new-episode-written-by-ai-a8f832645bc2*.

Dickens, Charles. 1859. *A Tale of Two Cities*. Project Gutenberg. *https://www.gutenberg.org/ebooks/98*.

Dictionary.com. 2020. "How Many Words Are There in the English Language?" Accessed October 29, 2020. *https://www.dictionary.com/e/how-many-words-in-english/*.

Doyle, Arthur Conan. 1892. *The Adventures of Sherlock Holmes*. Project Gutenberg. *https://www.gutenberg.org/files/1661/1661-0.txt*.

Full Fact. 2020. "Automated Fact Checking." Accessed October 29, 2020. *https://fullfact.org/automated*.

Geitgey, Adam. 2016. "Machine Learning Is Fun Part 6: How to Do Speech Recognition with Deep Learning." Medium. December 23, 2016. *https://medium.com/@ageitgey/machine-learning-is-fun-part-6-how-to-do-speech-recognition-with-deep-learning-28293c162f7a*.

Google. "Google Translate." 2020. *https://translate.google.com/*.

Graves, Alex, Abdel-rahman Mohamed, and Geoffrey Hinton, "Speech Recognition with Deep Recurrent Neural Networks." *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Vancouver, BC, Canada, May 26–31. *https://www.cs.toronto.edu/~fritz/absps/RNN13.pdf*.

Heerman, Victor, dir. 1930. *Animal Crackers*. Written by George S. Kaufman, Morrie Ryskind, Bert Kalmar, and Harry Ruby. Paramount Studios. *https://www.imdb.com/title/tt0020640/*.

Hochreiter, Sepp, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. 2001. "Gradient Flow in Recurrent Nets: The Difficulty of Learning Long-Term Dependencies." in S. C. Kremer and J. F. Kolen, eds. *A Field Guide to Dynamical Recurrent Neural Networks*. New York: IEEE Press. *https://www.bioinf.jku.at/publications/older/ch7.pdf*.

Hughes, John. 2020. "English-to-Dutch Neural Machine Translation via Seq2Seq Architecture." GitHub. Accessed October 29, 2020. *https://colab.research.google.com/github/hughes28/Seq2SeqNeuralMachineTranslator/blob/master/Seq2SeqEnglishtoDutchTranslation.ipynb#scrollTo=8q4ESVzKJgHd*.

Johnson, Daniel. 2015. "Composing Music with Recurrent Neural Networks." *Daniel D. Johnson* (blog). August 3, 2015. *https://www.danieldjohnson.com/2015/08/03/composing-music-with-recurrent-neural-networks/*.

Jurafsky, Dan. 2020. "Language Modeling: Introducing N-grams." Class notes, Stanford University, Winter 2020. *https://web.stanford.edu/~jurafsky/slp3/slides/LM_4.pdf*.

Kaggle. 2020. "Sunspots." Dataset, Kaggle.com. Accessed October 29, 2020. *https://www.kaggle.com/robervalt/sunspots*.

Karim, Raimi. 2019. "Attn: Illustrated Attention." Post in *Towards Data Science* (blog), Medium, January 20, 2019. *https://towardsdatascience.com/attn-illustrated-attention-5ec4ad276ee3*.

Karpathy, Andrej, and Fei-Fei Li. 2013. "Automated Image Captioning with ConvNets and Recurrent Nets." Presentation slides, Stanford Computer Science Department, Stanford University. *https://cs.stanford.edu/people/karpathy/sfmltalk.pdf*.

Karpathy, Andrej. 2015. "The Unreasonable Effectiveness of Recurrent Neural Networks." *Andrej Karpathy blog*, GitHub, May 21, 2015. *http://karpathy.github.io/2015/05/21/rnn-effectiveness/*.

Kelly, Charles. 2020. "Tab-Delimited Bilingual Sentence Pairs." Manythings.org. Last updated August 23, 2020. *http://www.manythings.org/anki/*.

Krishan. 2016. "Bollywood Lyrics via Recurrent Neural Networks." *From Data to Decisions* (blog), December 8, 2016. *https://iksinc.wordpress.com/2016/12/08/bollywood-lyrics-via-recurrent-neural-networks/*.

LISA Lab, 2018. "Modeling and Generating Sequences of Polyphonic Music with the RNN-RBM." Tutorial, Deeplearning.net. Last updated June 15, 2018. *http://deeplearning.net/tutorial/rnnrbm.html#rnnrbm*.

Mao, Junhua, Wei Xu, Yi Yang, Jiang Wang, Zhiheng Huang, and Alan Yuille. 2015. "Deep Captioning with Multimodal Recurrent Neural Networks (m-RNN)." Cornell University, Computer Science, arXiv:1412.6632, June 11, 2015. *https://arxiv.org/abs/1412.6632*.

McCrae, Pat. 2018. Comment on "How Many Nouns Are There in English?" Quora. November 15, 2018. *https://www.quora.com/How-many-nouns-are-there-in-English*.

Moocarme, Matthew. 2020. "Country Lyrics Created with Recurrent Neural Networks." *Matthew Moocarme* (blog). Accessed October 29, 2020. *http://www.mattmoocar.me/blog/RNNCountryLyrics/*.

Mooney, Raymond J. 2019. "CS 343: Artificial Intelligence: Natural Language Processing." Course notes, PowerPoint slides, University of Texas at Austin. *http://www.cs.utexas.edu/~mooney/cs343/slides/nlp.ppt*.

O'Brien, Tim, and Irán Román. 2017. "A Recurrent Neural Network for Musical Structure Processing and Expectation." Report for CS224d: Deep Learning for Natural Language Processing, Stanford University, Winter 2017. *https://cs224d.stanford.edu/reports/O%27BrienRom%C2%B4an.pdf*.

Olah, Christopher. 2015. "Understanding LSTM Networks." *Colah's Blog*, GitHub, August 27, 2015. *http://colah.github.io/posts/2015-08-Understanding-LSTMs/*.

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio. 2013. "On the Difficulty of Training Recurrent Neural Networks." Cornell University, Computer Science, arXiv:1211.5063, February 16, 2013. *https://arxiv.org/abs/1211.5063*.

R2RT. 2016. "Written Memories: Understanding, Deriving and Extending the LSTM." *R2RT* (blog), July 26, 2016. *https://r2rt.com/written-memories-understanding-deriving-and-extending-the-lstm.html*.

Rajpurkar, Pranav, Robin Jia, and Percy Liang. 2018. "Know What You Don't Know: Unanswerable Questions for SQuAD." Cornell University, Computer Science, arXiv:1806.03822, June 11, 2018. *https://arxiv.org/abs/1806.03822*.

Roberts, Adam, Colin Raffel, and Noam Shazeer. 2020. "How Much Knowledge Can You Pack into the Parameters of a Language Model?" Cornell University, Computer Science, arXiv:2002.08910, October 5, 2020. *https://arxiv.org/abs/2002.08910*.

Robertson, Sean. 2017. "NLP from Scratch: Translation with a Sequence to Sequence Network and Attention." Tutorial, PyTorch. *https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html*.

Schuster, Mike, and Kuldip K. Paliwal. 1997. "Bidirectional Recurrent Neural Networks." *IEEE Transactions on Signal Processing*, 45, no. 11 (November). *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.331.9441&rep=rep1&type=pdf*.

Sturm, Bob L. 2015a. "The Infinite Irish Trad Session." *High Noon GMT* (blog), Folk the Algorithms, August 7, 2015. *https://highnoongmt.wordpress.com/2015/08/07/the-infinite-irish-trad-session/.*

Sturm, Bob L. 2015b. "'Lisl's Stis': Recurrent Neural Networks for Folk Music Generation." *High Noon GMT* (blog), Folk the Algorithms, May 22, 2015. *https://highnoongmt.wordpress.com/2015/05/22/lisls-stis-recurrent-neural-networks-for-folk-music-generation/.*

Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le. 2014. "Sequence to Sequence Learning with Neural Networks." Cornell University, Computer Science, arXiv:1409.3215, December 14, 2014. *https://arxiv.org/abs/1409.3215.*

Unicode Consortium. 2020. Version 13.0.0, March 10, 2020. *https://www.unicode.org/versions/Unicode13.0.0/.*

van den Oord, Äaron, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. 2016. "WaveNet: A Generative Model for Raw Audio." Cornell University, Computer Science, arXiv:1609.03499, September 19, 2016. *https://arxiv.org/abs/1609.03499.*

Vicente, Agustin, and Ingrid L. Falkum. 2017. "Polysemy." *Oxford Research Encyclopedias: Linguistics*. July 27, 2017. *https://oxfordre.com/linguistics/view/10.1093/acrefore/9780199384655.001.0001/acrefore-9780199384655-e-325.*

Chapter 20

Alammar, Jay. 2018. "How GPT3 Works - Visualizations and Animations." *Jay Alammar: Visualizing Machine Learning One Concept at a Time* (blog). GitHub. Accessed November 5, 2020. *http://jalammar.github.io/how-gpt3-works-visualizations-animations/.*

Alammar, Jay. 2019. "A Visual Guide to Using BERT for the First Time." *Jay Alammar: Visualizing Machine Learning One Concept at a Time* (blog). GitHub. November 26, 2019. *http://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/.*

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. 2016. "Neural Machine Translation by Jointly Learning to Align and Translate." Cornell University, Computer Science, arXiv:1409.0473, May 19, 2016. *https://arxiv.org/abs/1409.0473.*

Brown, Tom B., et al., 2020. "Language Models Are Few-Shot Learners." Cornell University, Computer Science, arXiv:2005.14165, July 22, 2020. *https://arxiv.org/pdf/2005.14165.pdf.*

Cer, Daniel, et al. 2018. "Universal Sentence Encoder." Cornell University, Computer Science, arXiv:1803.11175 April 12, 2018. *https://arxiv.org/abs/1803.11175.*

Cho, Kyunghyun, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. "Learning Phrase Representations Using RNN Encoder–Decoder for Statistical Machine Translation." In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (Doha, Qatar, October 25–29, 2014): 1724–34. *http://emnlp2014.org/papers/pdf/EMNLP2014179.pdf.*

Chromiak, Michał. 2017. "The Transformer—Attention Is All You Need." *Michał Chromiak's Blog*, GitHub, October 30, 2017. *https://mchromiak.github.io/articles/2017/Sep/12/Transformer-Attention-is-all-you-need/.*

Common Crawl. 2020. Common Crawl home page. Accessed November 15, 2020. *https://commoncrawl.org/the-data/.*

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." Cornell University, Computer Science, arXiv:1810.04805, May 24, 2019. *https://arxiv.org/abs/1810.04805.*

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. 2020. "Google-Research/bert." GitHub. Accessed November 5, 2020. *https://github.com/google-research/bert*.

El Boukkouri, Hicham. 2018. "Arithmetic Properties of Word Embeddings." *Data from the Trenches* (blog), Medium, November 21, 2018. *https://medium.com/data-from-the-trenches/arithmetic-properties-of-word-embeddings-e918e3fda2ac*.

Facebook Open Source. 2020. "fastText: Library for Efficient Text Classification and Representation Learning." Open source software. Accessed November 5, 2020. *https://fasttext.cc/*.

Frankenheimer, John, dir. 1962. *The Manchurian Candidate*, written by George Axelrod, based on a novel by Richard Condon., M. C. Productions. *https://www.imdb.com/title/tt0056218/*.

Gluon authors. 2020. "Extracting Sentence Features with Pre-trained ELMo." Tutorial, Gluon, Accessed November 5, 2020. *https://gluon-nlp.mxnet.io/examples/sentence_embedding/elmo_sentence_representation.html*.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. "Deep Residual Learning for Image Recognition." Cornell University, Computer Science, arXiv:1512.03385, December 10, 2015. *https://arxiv.org/abs/1512.03385*.

Hendrycks, Dan, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. "Measuring Massive Multitask Language Understanding." Cornell University, Computer Science, arXiv:2009.03300, September 21, 2020. *https://arxiv.org/abs/2009.03300*.

Howard, Jeremy and Sebastian Ruder. 2018. "Universal Language Model Fine-Tuning for Text Classification." Cornell University, Computer Science, arXiv:1801.06146, May 23, 2018. *https://arxiv.org/abs/1801.06146*.

Huston, Scott. 2020. "GPT-3 Primer: Understanding OpenAI's Cutting-Edge Language Model." *Towards Data Science* (blog), August 20, 2020. *https://towardsdatascience.com/gpt-3-primer-67bc2d821a00*.

Kaplan, Jared, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. "Scaling Laws for Neural Language Models." Cornell University, Computer Science, arXiv:2001.08361, January 23, 2020. *https://arxiv.org/abs/2001.08361*.

Kazemnejad, Amirhossein. 2019. "Transformer Architecture: The Positional Encoding." *Amirhossein Kazemnejad's Blog*, September 20, 2019. *https://kazemnejad.com/blog/transformer_architecture_positional_encoding/*.

Kelly, Charles. 2020. "Tab-Delimited Bilingual Sentence Pairs." Manythings.org. Accessed November 6, 2020. *http://www.manythings.org/anki/*.

Klein, Guillaume, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M. Rush. 2017. "OpenNMT: Open-Source Toolkit for Neural Machine Translation." Cornell University, Computer Science, arXiv:1701.02810, March 6, 2017. *https://arxiv.org/abs/1701.02810*.

Liu, Yang, Lixin Ji, Ruiyang Huang, Tuosiyu Ming, Chao Gao, and Jianpeng Zhang. 2018. "An Attention-Gated Convolutional Neural Network for Sentence Classification." Cornell University, Computer Science, arXiv:2018.07325. December 28, 2018. *https://arxiv.org/abs/1808.07325*.

Mansimov, Elman, Alex Wang, Sean Welleck, and Kyunghyun Cho. 2020. "A Generalized Framework of Sequence Generation with Application to Undirected Sequence Models." Cornell University, Computer Science, arXiv:1905.12790, February 7, 2020. *https://arxiv.org/abs/1905.12790*.

McCormick Chris, and Nick Ryan, 2020. "BERT Fine-Tuning Tutorial with PyTorch." *Chris McCormick* (blog). Last updated March 20, 2020. *https://mccormickml.com/2019/07/22/BERT-fine-tuning/.*

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013a. "Efficient Estimation of Word Representations in Vector Space." Cornell University, Computer Science, arXiv:1301.3781, September 7, 2013. *https://arxiv.org/abs/1301.3781.*

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013b. "Distributed Representations of Words and Phrases and Their Compositionality." Cornell University, Computer Science, arXiv:1310.4546, October 16, 2013. *https://arxiv.org/abs/1310.4546.*

Mishra, Prakhar. 2020. "Natural Language Generation Using BERT Introduction." *TechViz: The Data Science Guy* (blog). Accessed November 6, 2020. *https://prakhartechviz.blogspot.com/2020/04/natural-language-generation-using-bert.html.*

Paulus, Romain, Caiming Xiong, and Richard Socher, 2017. "A Deep Reinforced Model for Abstractive Summarization." Cornell University, Computer Science, arXiv:1705.04304, November 13, 2017. *https://arxiv.org/abs/1705.04304.*

Pennington, Jeffrey, Richard Socher, and Christopher D. Manning. 2014. "GloVe: Global Vectors for Word Representation." in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (October): 1532–43. *https://nlp.stanford.edu/pubs/glove.pdf.*

Peters, Matthew E., Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. "Deep Contextualized Word Representations." Cornell University, Computer Science, arXiv:1802.05365. March 22, 2018. *https://arxiv.org/abs/1802.05365.*

Radford, Alec, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. "Language Models Are Unsupervised Multitask Learners." OpenAI, San Francisco, CA, 2019. *https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.*

Raffel, Colin, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer." Cornell University, Computer Science, arXiv:1910.10683, July 28, 2020. *https://arxiv.org/abs/1910.10683.*

Rajasekharan, Ajit. 2019. "A Review of BERT Based Models." *Towards Data Science* (blog), June 17, 2019. *https://towardsdatascience.com/a-review-of-bert-based-models-4ffdc0f15d58.*

Reisner, Alex. 2020. "What's It Like to Be an Animal?" SpeedofAnimals.com. Accessed November 6, 2020. *https://www.speedofanimals.com/.*

Russell, Stuart, and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach*, 3rd ed. New York: Pearson Press.

Sanh, Victor, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2020. "DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter." Cornell University, Computer Science, arXiv:1910.01108, March 1, 2020. *https://arxiv.org/abs/1910.01108.*

Scott, Kevin. 2020. "Microsoft Teams Up with OpenAI to Exclusively License GPT-3 Language Model." *Official Microsoft Blog*, September 22, 2020. *https://blogs.microsoft.com/blog/2020/09/22/microsoft-teams-up-with-openai-to-exclusively-license-gpt-3-language-model/.*

Shao, Louis, Stephan Gouws, Denny Britz, Anna Goldie, Brian Strope, and Ray Kurzweil. 2017. "Generating High-Quality and Informative Conversation Responses with Sequence-to-Sequence Models." Cornell University, Computer Science, arXiv:1701.03185, July 31, 2017. *https://arxiv.org/abs/1701.03185*.

Singhal, Vivek. 2020. "Transformers for NLP." *Research/Blog*, CellStrat, May 19, 2020. *https://www.cellstrat.com/2020/05/19/transformers-for-nlp/*.

Socher, Richard, Alex Perelygin, Jean Y. Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. 2013a. "Deeply Moving: Deep Learning for Sentiment Analysis—Dataset." Sentiment Analysis. August 2013. *https://nlp.stanford.edu/sentiment/index.html*.

Socher, Richard, Alex Perelygin, Jean Y. Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. 2013b. "Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank." Oral presentation at the Conference on Empirical Methods in Natural Language Processing (EMNLP) (Seattle, WA, October 18–21). *https://nlp.stanford.edu/~socherr/EMNLP2013_RNTN.pdf*.

spaCy authors. 2020. "Word Vectors and Semantic Similarity." spaCy: Usage. *https://spacy.io/usage/vectors-similarity*.

Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le. 2014. "Sequence to Sequence Learning with Neural Networks." Cornell University, Computer Science, arXiv:1409.3215, December 14, 2014. *https://arxiv.org/abs/1409.3215*.

Tay, Yi, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2020. "Efficient Transformers: A Survey." Cornell University, Mathematics, arXiv:2009.0673, September 1, 2020. *https://arxiv.org/abs/2009.0673*.

Taylor, Wilson L. 1953. "'Cloze Procedure': A New Tool for Measuring Readability." *Journalism Quarterly*, 30(4): 415–33. *https://www.gwern.net/docs/psychology/writing/1953-taylor.pdf*.

TensorFlow authors. 2018. "Universal Sentence Encoder." Tutorial, TensorFlow model archives, GitHub, 2018. *https://colab.research.google.com/github/tensorflow/hub/blob/master/examples/colab/semantic_similarity_with_tf_hub_universal_encoder.ipynb#scrollTo=co7MV6sX7Xto*.

TensorFlow authors, 2019a. "Why Add Positional Embedding Instead of Concatenate?" *tensorflow/tensor2tensor* (blog). May 30, 2019. *https://github.com/tensorflow/tensor2tensor/issues/1591*.

TensorFlow authors. 2019b. "Transformer Model for Language Understanding." Documentation, TensorFlow, GitHub. *https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/text/transformer.ipynb*.

TensorFlow authors. 2020a. "Elmo." TensorFlow Hub, November 6, 2020. *https://tfhub.dev/google/elmo/3*.

TensorFlow authors. 2020b. "Transformer Model for Language Understanding." Tutorial, TensorFlow Core documentation. Last updated November 2, 2020. *https://www.tensorflow.org/tutorials/text/transformer*.

Thiruvengadam, Aditya. 2018. "Transformer Architecture: Attention Is All You Need." *Aditya Thiruvengadam* (blog), Medium. October 9, 2018. *https://medium.com/@adityathiruvengadam/transformer-architecture-attention-is-all-you-need-aeccd9f50d09*.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jacob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhim. 2017. "Attention Is All You Need." Cornell University, Computer Science, arXiv:1706.03762v5, December 6, 2017. *https://arxiv.org/abs/1706.03762v5*.

Vijayakumar, Ashwin K., Michael Cogswell, Ramprasath R. Selvaraju, Qing Sun, Stefan Lee, David Crandall, and Dhruv Batra. 2018. "Diverse Beam Search: Decoding Diverse Solutions from Neural Sequence Models." Cornell University, Computer Science, arXiv:1610.02424. October 22, 2018. *https://arxiv.org/abs/1610.02424*.

von Platen, Patrick. 2020. "How to Generate Text: Using Different Decoding Methods for Language Generation with Transformers." *Huggingface* (blog), GitHub, May 2020. *https://huggingface.co/blog/how-to-generate*.

Wallace, Eric, Tony Z. Zhao, Shi Feng, and Sameer Singh. 2020. "Customizing Triggers with Concealed Data Poisoning." Cornell University, Computer Science, arXiv:2010.12563, October 3, 2020. *https://arxiv.org/abs/2010.12563*.

Walton, Nick. 2020. "AI Dungeon: Dragon Model Upgrade." *Nick Walton* (blog) July 14, 2020. *https://medium.com/@aidungeon/ai-dungeon-dragon-model-upgrade-7e8ea579abfe* and *https://play.aidungeon.io/main/home*.

Wang, Alex, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. "GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding." Cornell University, Computer Science, arXiv:1804.07461, February 22, 2019. *https://arxiv.org/abs/1804.07461*.

Wang, Alex, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2020. "GLUE Leaderboards." GLUE Benchmark. Accessed November 6, 2020. *https://gluebenchmark.com/leaderboard/submission/zlssuBTm5XRs0aSKbFYGVIVdvbj1/-LhijX9VVmvJcvzKymxy*.

Warstadt, Alex, Amanpreet Singh, and Sam Bowman. 2018. "CoLA: The Corpus of Linguistic Acceptability." NYU-MLL. *https://nyu-mll.github.io/CoLA/*.

Warstadt, Alex, Amanpreet Singh, and Sam Bowman. 2019. "Neural Network Ability Judgements." Cornell University, Computer Science, arXiv:1805.12471, October 1, 2019. *https://arxiv.org/abs/1805.12471*.

Welleck, Sean, Ilia Kulikov, Jaedeok Kim, Richard Yuanzhe Pang, and Kyunghyun Cho. 2020. "Consistency of a Recurrent Language Model with Respect to Incomplete Decoding." Cornell University, Computer Science, arXiv:2002.02492, October 2, 2020. *https://arxiv.org/abs/2002.02492*.

Woolf, Max. 2019. "Train a GPT-2 Text-Generating Model w/ GPU." Google Colab Notebook, 2019. *https://colab.research.google.com/drive/1VLG8e7YSEwypxU-noRNhsv5dW4NfTGce#scrollTo=-xInIZKaU104*.

Xu, Lei, Ivan Ramirez, and Kalyan Veeramachaneni. 2020. "Rewriting Meaningful Sentences via Conditional BERT Sampling and an Application on Fooling Text Classifiers." Cornell University, Computer Science, arXiv:2010.11869, October 22, 2020. *https://arxiv.org/abs/2010.11869*.

Zhang, Aston, Zachary C. Lipton, Mu Li, and Alexander J. Smola. 2020. "10.3: Transformer." In *Dive into Deep Learning. https://d2l.ai/chapter_attention-mechanisms/transformer.html*.

Zhang, Han, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena. 2019. "Self-Attention Generative Adversarial Networks." Cornell University, Statistics, arXiv:1805.08318. June 14, 2019. *https://arxiv.org/abs/1805.08318*.

Zhu, Yukun, Ryan Kiros, Richard Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. "Aligning Books and Movies: Towards Story-Like Visual Explanations by Watching Movies and Reading Books." Cornell University, Computer Science, arXiv:1506.06724, June 22, 2015. *https://arxiv.org/abs/1506.06724*.

Chapter 21

Asadi, Kavosh, and Michael L. Littman. 2017. "An Alternative Softmax Operator for Reinforcement Learning." In *Proceedings of the 34th International Conference on Machine Learning* (Sydney, Australia, August 6–11). *https://arxiv.org/abs/1612.05628*.

Craven, Mark, and David Page. 2018. "Reinforcement Learning with DNNs: AlphaGo to AlphaZero." CS 760 course notes, Spring, School of Medicine and Public Health, University of Wisconsin-Madison. *https://www.biostat.wisc.edu/~craven/cs760/lectures/AlphaZero.pdf*.

DeepMind team. 2020. "Alpha Go." *DeepMind* (blog). Accessed October 8, 2020. *https://deepmind.com/research/alphago/*.

Eden, Tim, Anthony Knittel, and Raphael van Uffelen. 2020. "Reinforcement Learning." University of New South Wales. Accessed October 8, 2020. *http://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html*.

François-Lavet, Vincent, Peter Henderson, Riashat Islam, Marc G. Bellemare, Joelle Pineau, "An Introduction to Deep Reinforcement Learning." Cornell University, Machine Learning, arXiv:1811.12560, December 3, 2018. *https://arxiv.org/abs/1811.12560*.

Hassabis, Demis, and David Silver. 2017. "AlphaGo Zero: Learning from Scratch." *DeepMind* (Blog), October 18, 2017. *https://deepmind.com/blog/alphago-zero-learning-scratch/*.

Matiisen, Tambet. 2015. "Demystifying Deep Reinforcement Learning." Computational Neuroscience Lab, Institute of Computer Science, University of Tartu, December 15, 2015. *https://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/*.

Melo, Francisco S. 2020. "Convergence of Q-Learning: A Simple Proof." Institute for Systems and Robotics, Instituto Superior Técnico, Portugal. Accessed October 9, 2020. *http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf*.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. "Playing Atari with Deep Reinforcement Learning." NIPS Deep Learning Workshop, December 19, 2013. *https://arxiv.org/abs/1312.5602v1*.

Rummery, G. A., and M. Niranjan. 1994. "On-Line Q-Learning Using Connectionist Systems." Engineering Department, Cambridge University, UK, September 1994. *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.17.2539&rep=rep1&type=pdf*.

Silver, David, et al. 2017. "Mastering the Game of Go Without Human Knowledge." *Nature* 550 (October 19, 2017): 354–59. *https://www.nature.com/articles/nature24270.epdf*.

Sutton, Richard S., and Andrew G. Baro. 2018. *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA: MIT Press. Available at *http://www.incompleteideas.net/book/the-book-2nd.html*.

Villanueva, John Carl. 2009. "How Many Atoms Are There in the Universe?" *Universe Today*, July 30, 2009. *http://www.universetoday.com/36302/atoms-in-the-universe/*.

Watkins, Christopher. 1989. "Learning from Delayed Rewards." PhD thesis, Cambridge University, UK. *http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf*.

Chapter 22

Achlioptas, Panos, Olga Diamanti, Ioannis Mitliagkas, and Leonidas Guibas. 2018. "Representation Learning and Adversarial Generation of 3D Point Clouds." Cornell University, Computer Science, arXiv:1707.02392, June 12, 2018. *https://arxiv.org/abs/1707.02392v1*.

Arjovsky, Martin, and Léon Bottou. 2017. "Towards Principled Methods for Training Generative Adversarial Networks." Cornell University, Statistics, arXiv:1701.04862, January 17, 2017. *https://arxiv.org/abs/1701.04862v1*.

Arjovsky, Martin, Soumith Chintala, and Léon Bottou. 2017. "Wasserstein GAN." Cornell University, Statistics, arXiv:1701.07875, December 6, 2017. *https://arxiv.org/abs/1701.07875v1*.

Bojanowski, Piotr, Armand Joulin, David Lopez-Paz, and Arthur Szlam. 2019. "Optimizing the Latent Space of Generative Networks." Cornell University, Statistics, arXiv 1717.05776, May 20, 2019. *https://arxiv.org/abs/1707.05776*.

Chen, Janet, Su-I Lu, and Dan Vekhter. 2020. "Strategies of Play." In *Game Theory*, Stanford Department of Computer Science, Stanford University, Stanford, CA. Accessed October 6, 2020. *https://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/game-theory/Minimax.html*.

Geitgey, Adam. 2017. "Machine Learning Is Fun Part 7: Abusing Generative Adversarial Networks to Make 8-bit Pixel Art." Medium. February 12, 2017. *https://medium.com/@ageitgey/abusing-generative-adversarial-networks-to-make-8-bit-pixel-art-e45d9b96cee7#.v1o6o0dyi*.

Gildenblat, Jacob. 2020. "KERAS-DCGAN." GitHub. Accessed October 6, 2020. *https://github.com/jacobgil/keras-dcgan*.

Goodfellow, Ian J., Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. "Generative Adversarial Networks." Cornell University, Statistics, arXiv:1406.2661, June 10, 2014. *https://arxiv.org/abs/1406.2661*.

Goodfellow, Ian. 2016. "NIPS 2016 Tutorial: Generative Adversarial Networks." Cornell University, Computer Science, arXiv:1701.00160, December 31, 2016. *https://arxiv.org/abs/1701.00160*.

Karras, Tero, Timo Aila, Samuli Laine, and Jaakko Lehtinen. 2018. "Progressive Growing of GANs for Improved Quality, Stability, and Variation." Cornell University, Computer Science, arXiv:1710.10196, February 26, 2018. *https://arxiv.org/abs/1710.10196*.

Myers, Andrew. 2002. "CS312 Recitation 21: Minimax Search and Alpha-Beta Pruning." Computer Science Department, Cornell University. *https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/rec21.htm*.

Radford, Alec, Luke Metz, and Soumith Chintala. 2016. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks." Cornell University, Computer Science, arXiv:1511.06434, January 7, 2016. *https://arxiv.org/abs/1511.06434*.

Watson, Joel. 2013. *Strategy: An Introduction to Game Theory*, 3rd ed. New York: W.W. Norton and Company.

Chapter 23

The Art Story Foundation, 2020. "Classical, Modern, and Contemporary Movements and Styles." Art Story site. Accessed October 7, 2020. *http://www.theartstory.org/section_movements.htm*.

Bonaccorso, Giuseppe. 2020. "Neural_Artistic_Style_Transfer." GitHub. Accessed October 7, 2020. *https://github.com/giuseppebonaccorso/keras_deepdream*.

Chollet, François. 2017. *Deep Learning with Python*. Shelter Island, NY: Manning Publications. *https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/8.3-neural-style-transfer.ipynb*.

Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge. 2015. "Neural Algorithm of Artistic Style." Cornell University, Computer Science, arXiv:1508.06576, September 2, 2015. *https://arxiv.org/abs/1508.06576*.

Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge. 2016. "Image Style Transfer Using Convolutional Neural Networks." in *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition* (Las Vegas, NV, June 27–30). *https://pdfs.semanticscholar.org/7568/d13a82f7afa4be79f09c295940e48ec6db89.pdf*.

Jing, Yongcheng, Yezhou Yang, Zunlei Feng, Jingwen Ye, and Mingli Song. 2018. "Neural Style Transfer: A Review." Cornell University, Computer Science, arXiv:1705.04058v1, October 30, 2018. *https://arxiv.org/abs/1705.04058*.

Li, Yanghao, Naiyan Wang, Jiaying Liu, and Xiaodi Hou. 2017. "Demystifying Neural Style Transfer." Cornell University, Computer Science, arXiv:1701.01036, July 1, 2017. *https://arxiv.org/abs/1701.01036*.

Lowensohn, Josh. 2014. "I Let Apple's QuickType Keyboard Take Over My iPhone." *The Verge* (blog), September 17, 2014. *https://www.theverge.com/2014/9/17/6337105/breaking-apples-quicktype-keyboard*.

Majumdar, Somshubra. 2020. "Titu1994/Neural-Style-Transfer." GitHub, Accessed October 7, 2020. *https://github.com/titu1994/Neural-Style-Transfer*.

Mordvintsev, Alexander, Christopher Olah, and Mike Tyka. 2015. "Inceptionism: Going Deeper into Neural Networks." *Google AI Blog*, June 17, 2015. *https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html*.

O'Neil, Cathy. 2016. *Weapons of Math Destruction*. New York: Broadway Books.

Orlowski, Jeff. 2020. *The Social Dilemma*. Exposure Labs, Argent Pictures, and Netflix. Accessed October 7, 2020. *https://www.thesocialdilemma.com/the-film/*.

Ruder, Manuel, Alexey Dosovitskiy, and Thomas Brox. 2018. "Artistic Style Transfer for Videos and Spherical Images." Cornell University, Computer Science, arXiv:1708.04538, August 5, 2018. *https://arxiv.org/abs/1708.04538*.

Simonyan, Karen, and Andrew Zisserman. 2020. "Very Deep Convolutional Networks for Large-Scale Visual Recognition." *Visual Geometry Group* (blog), University of Oxford. Accessed October 7, 2020. *http://www.robots.ox.ac.uk/~vgg/research/very_deep/*.

Tyka, Mike. 2015. "Deepdream/Inceptionism - recap." *Mike Tyka* (blog), July 21, 2015. *https://mtyka.github.io/code/2015/07/21/one-month-after-deepdream.html*.

Wikipedia authors. 2020. "Style (visual arts)." Wikipedia. September 2, 2020. *https://en.wikipedia.org/wiki/Style_(visual_arts)*.

IMAGE CREDITS

Artwork from Wikimedia and Wikiart is identified as being in the public domain. Images from Pixabay are covered by the Creative Commons CC0 license, which places them in the public domain. Unaccredited images are by the author.

Chapter 1

Figure 1-3, Bananas
https://pixabay.com/en/bananas-1642706

Figure 1-3, Cat
https://pixabay.com/en/cat-2360874

Figure 1-3, Camera
https://pixabay.com/en/photography-603036

Figure 1-3, Corn
https://pixabay.com/en/pop-corn-785074

Chapter 10

Figure 10-3, Cow
https://pixabay.com/en/cow-field-normande-800306

Figure 10-3, Zebra
https://pixabay.com/en/zebra-chapman-steppe-zebra-1975794

Figure 10-40, Husky
https://pixabay.com/en/husky-sled-dogs-adamczak-1105338

Figure 10-40, Husky
https://pixabay.com/en/husky-dog-outdoor-siberian-breed-1328899

Figure 10-40, Husky
https://pixabay.com/en/dog-husky-sled-dog-animal-2016708

Figure 10-40, Husky
https://pixabay.com/en/dog-husky-friend-2332240

Figure 10-40, Husky
https://pixabay.com/en/green-grass-playground-nature-2562252

Figure 10-40, Husky
https://pixabay.com/en/husky-dog-siberian-husky-sled-dog-2671006

Chapter 16

Figure 16-5, 16-7, Frog
https://pixabay.com/en/frog-toxic-yellow-netherlands-1463831

Chapter 17

Figure 17-17 to Figure 17-22, Duck
https://pixabay.com/en/duck-kaczor-animal-wild-bird-duck-268105

Figure 17-23, Tiger
https://pixabay.com/photos/tiger-animal-wildlife-mammal-165189/

Chapter 18

Figure 18-1, Cow
https://pixabay.com/en/cow-field-normande-800306

Figure 18-2, Zebra
https://pixabay.com/en/zebra-chapman-steppe-zebra-1975794

Figure 18-8 and throughout the chapter, Tiger
https://pixabay.com/photos/tiger-animal-wildlife-mammal-165189/

Chapter 23

Figure 23-3 and throughout the chapter, Frog
https://pixabay.com/en/waters-nature-frog-animal-swim-3038803/

Figure 23-5, Dog
https://pixabay.com/photos/labrador-retriever-dog-pet-1210559/

Figure 23-6, Figure 23-7, Figure 23-9, Pablo Picasso, “Self-Portrait 1907”
https://www.wikiart.org/en/pablo-picasso/self-portrait-1907

Figure 23-9, Vincent Van Gogh, “The Starry Night”
https://commons.wikimedia.org/wiki/File:VanGogh-starry_night_ballance1.jpg

Figure 23-9, J.M.W. Turner, "The Shipwreck of the Minotaur"
https://commons.wikimedia.org/wiki/File:Shipwreck_of_the_Minotaur_William_Turner.jpg

Figure 23-9, Edvard Munch, "The Scream"
https://www.wikiart.org/en/edvard-munch/the-scream-1893

Figure 23-9, Pablo Picasso, "Seated Female Nude"
https://www.wikiart.org/en/pablo-picasso/seated-female-nude-1910

Figure 23-9, Edward Hopper, "Nighthawks"
https://commons.wikimedia.org/wiki/File:Nighthawks_by_Edward_Hopper_1942.jpg

Figure 23-9, Claude Monet, "Water Lilies, Yellow and Lilac"
https://www.wikiart.org/en/claude-monet/water-lilies-yellow-and-lilac-1917

Figure 23-9, Wassily Kandinsky, "Composition VII"
https://www.wikiart.org/en/wassily-kandinsky/composition-vii-1913

Figure 23-12, Town
https://pixabay.com/en/town-building-urban-architecture-2430571/

INDEX

Symbols and Numbers

α (alpha), Q-learning blending, 629
β1 (beta 1), Adam parameter, 421
β2 (beta 2), Adam parameter, 421
δ (delta), proportion of error change, 357
Δ (Delta), proportion of error change, 357
ε (epsilon)
  numerical stability parameter, 419
  Q-learning policy, 630
η (eta), learning rate
  introduction, 377
  practical issues during optimization, 389
γ (gamma)
  Adagrad scaling parameter, 419
  discount factor, 611
  momentum scaling factor, 410
λ (lambda), regularization control, 204
σ (sigma), standard deviation, 24
θ (theta), coin bias, 103
P(A), simple probability, 50
P(A,B), joint probability, 53
P(A|B), conditional probability, 50
1-hot encoding, 226
1D convolution, 446
1×1 convolution, 447
20 Questions, 269
68-95-99.7 rule, 25

A

accuracy, 64
activation function
  exponential ReLU, 340
  gallery of functions, 344
  Heaviside step, 334
  identity, 331
  introduction, 318
  leaky ReLU, 337
  linear, 332
  logistic curve, 341
  maxout, 338
  noisy ReLU, 339
  parametric ReLU, 337
  piecewise linear, 336
  ReLU, 336
  role in preventing network collapse, 329
  shifted ReLU, 338
  sigmoid, 341
  sign function, 335
  sine wave, 343
  softmax, 345
  softplus, 340
  stairstep, 333
  step, 333
  swish, 341
  tanh, 341
  unit step, 334
actor, 602
acyclic graph, 324
Adaboost, 309
Adadelta, 418
Adagrad, 417
Adam, 420
adaptive code, 137
adaptive gradient learning, 417
adaptive moment estimation, 420
The Adventures of Huckleberry Finn, 146
adversarial perturbation, 491
adversary, 491

agent, 602
AI winter, 317
ancestor (in a neural network), 324
anchor point, 436
Animal Crackers, 559
Anscombe's quartet, 40
answer questions, 540
Apollo spacecraft, 297
arc (in a neural network), 323
archaeology coin-flipping game (scenario), 103
array, 329
artificial neuron. *See* neuron
atmospheric example, 205
atrous convolution, 457
attacks (with perturbations), 493
attention
    introduction, 574
    key, 575
    Q/KV, 579
    QKV, 575
    query, 575
    self-attention, 576
    value, 575
attention layer
    introduction, 578
    schematic symbol, 581
autoencoder
    basic structure, 495
    convolutional, 516
    latent layer, 501
automatic Bayes, 97
autoregression, 541
average
    common meaning, 16
    harmonic mean, 71
    mean, 16
    median, 16
    mode, 16
average pooling, 452

B

baby lengths (scenario), 33
backprop, 351
backpropagation, 351
backpropagation through time, 553
backward propagation, 376
bagging, 299
balanced decision tree, 270
base image, 680
batch gradient descent, 401
batchnorm
    discussion, 424
    schematic symbol, 424
Baxter, William, 139
Bayes' Rule
    discussion, 95
    evidence, 96
    hypothesis, 102
    likelihood, 96
    line fitting, 212
    multiple hypotheses, 109
    observation, 102
    posterior, 96
    prior, 96
    refining estimate, 101
    repeating, 101
    statement, 94
Bayes' Theorem. *See* Bayes' Rule
Bayes, Thomas, 83
Bayesian probability
    vs. frequentist, 85
    overview, 85
beam search, 595
bell curve, 22
Bernoulli distribution, 26
BERT, 590
bi-RNN, 559
bias
    in artificial neuron, 318
    bias trick, 320
    in families of curves, 204
    of a flipped coin, 86
bias in, bias out, 673
bias-variance tradeoff, 210
bidirectional RNN layer
    introduction, 559
    schematic symbol, 559
binary classifier network, 378
binary cross entropy, 150
binary relevance, 161
bind (to neuron), 314
bit, 136
blessing of non-uniformity, 173
blessing of structure, 173
block of numbers, 329

BMI (body mass index), 243
bold driver method, 400
boosting, 302
bootstrap, 299
bootstrap aggregation, 299
bootstrapping. *See* bootstraps
bootstraps
  in bagging (for ensembles), 299
  in statistics, 31
  linear, 39
  multiple, 39
  negative, 38
  partial, 39
  positive, 38
  simple, 39
  strong, 38
  weak, 38
bottleneck, 501
bouncing around the minimum, 394
boundary
  classifier, 156
  decision boundary, 158
  method, 156
  simple, 58
box filter, 451
BPTT (backpropagation through time), 553
branch (in decision tree), 270
building a sundae (scenario), 548

C

C (support vector machine parameter), 285
candling, 157
Cartesian space, 213
categorical data, 226
categorical distribution, 27
categorical variable decision tree, 271
categorization. *See* classifier
categorizer. *See* classifier
category, 156
channel (in a tensor), 430
cheating
  at finding tanks, 185
  when overfitting, 197
child
  in a decision tree, 271
  in a neural network, 324
choosing store background music (scenario), 198
class, 156
classification. *See* classifier
classifier
  binary, 156
  binary relevance, 161
  multiclass, 160
  non-parametric, 264
  one-against-all, 161
  one-versus-all, 161
  one-versus-one, 163
  one-versus-rest, 161
  overview, 6
  parametric, 264
  training, 182
closure (training task), 591
cloze task, 591
clustering
  k-means, 166
  overview, 8
CNN (convolutional neural network), 431
CNN-LSTM, 557
code
  adaptive, 137
  constant-length, 141
  fixed-length, 141
  Huffman, 148
  variable-bitrate, 143
coin detection (scenario), 86
coin flipping
  basic idea, 86
  Bayesian approach, 87
coin-flipping game (scenario), 87
collapse, 329
Common Crawl, 597
compression, 496
compression ratio, 148
concealed data poisoning, 598
conditional probability, 50
confidence (in ensemble voting), 299
confidence interval, 32
confusion matrix
  with Bayes' Rule, 97
  correct analysis, 77
  definition, 60
  proper use, 74
  wrong use, 76

connectome, 315
consistency in data preparation, 223
constant distribution, 22
constant-length code, 141
contaminated data, 187
content blending, 498
content loss (for style transfer), 683
context (in information), 135
context vector, 562
contextualized word embeddings, 571
continuous probability distribution, 21
continuous variable decision tree, 272
convergence
    of GAN, 670
    of Q-learning, 633
convnet, 431
convolution
    introduction, 433
    layer, 446
    layer schematic symbol, 471
    multidimensional, 443
    striding, 453
    transposed convolution, 457
convolution with downsampling layer symbol, 521
convolution with upsampling layer symbol, 521
convolutional autoencoder, 516
convolutional neural network, 431
correctness, 56
correlation, 37
correlation coefficient, 38
cost (for neural network error), 352
covariance, 35
cpd (continuous probability distribution), 21
credit assignment problem, 608
cross entropy, 145
cross-validation
    basic, 190
    information leakage, 239
    k-fold, 192
curse of dimensionality, 168
cusp, 118
cycle (in a neural network), 323

D

DAG (directed acyclic graph), 324
dart throwing (scenario), 48
data
    augmentation, 255
    cleaning, 221
    contamination, 187
    hygiene, 187
    leakage, 187
    poisoning, 598
    preparation, 221
dataset enlargement, 255
DCGAN (deep convolutional GAN), 667
dead neuron, 345
decay parameter (for learning rate), 396
decay schedule, 398
decision boundary, 158
decision node (in decision tree), 270
decision region, 158
decision stump, 303
decision tree
    balanced, 270
    branch, 270
    children, 271
    decision node, 270
    depth, 270
    depth limiting, 280
    distant children, 271
    edge, 270
    ensemble, 299
    immediate children, 271
    internal node, 270
    leaf, 270
    link, 270
    node, 270
    overfitting, 275
    overview, 269
    parent, 271
    pruning, 280
    root, 270
    sibling, 271
    stump, 303
    subtree, 271
    terminal node, 270
    unbalanced, 270
decoder
    for autoencoder, 496
    for seq2seq, 561
decoder block (transformer), 587

deconvolution (as transposed convolution), 457
deep convolutional GAN, 667
deep dreaming, 675
deep learning
  network structure, 326
  overview, 10
deep reinforcement learning, 647
delay step, 550
delayed exponential decay, 398
delta values (for neurons), 360
denoising, 519
dense layer. *See* fully connected layer
density, 169
dependent variable, 29
deployed system, 183
depth
  in decision tree, 270
  of a deep learning network, 327
  of a tensor, 430
depth limiting, 280
derivative
  definition, 119
  requirements, 123
descendent (in a neural network), 324
deterministic
  environment, 611
  function, 118
deterministic environment, 611
DFR (discounted future reward), 609
Dickens, Charles, 545
digits, 4
dilated convolution, 457
dimensionality reduction, 243
diminishing returns (in ensembles), 300
directed acyclic graph, 324
directed divergence, 151
directed graph, 323
discount factor, 611
discounted future reward, 609
discrete probability distribution, 20
discrimination information, 151
discriminator, 650
distant children (in decision tree), 271
distilling, 593
distinguishing cows and zebras (scenario), 224
dit, 137
domain, 158
downsampling, 451
downsizing, 457
downstream
  network, 542
  task, 542
Dr. Seuss, 137
dropout
  discussion, 422
  schematic symbol, 422
dual representation, 213
dummy variable, 226

E

E (neural network error), 359
early stopping, 202
edge
  in a decision tree, 270
  in a neural network, 323
eggs
  fertilized, 157
  quitter, 160
  unfertilized, 157
  winner, 160
  yolker, 160
eigendog, 256
eigenvector, 256
element
  in a neural network, 323
  in a tensor, 430
elementwise processing, 234
elevator scheduling (scenario), 9
ELMo, 571
ELU (exponential ReLU activation function), 340
embedding
  contextualized, 572
  sentence, 571
  space, 569
  token, 540
  word, 566
embedding layer schematic symbol, 574
encoder
  for autoencoder, 496
  for seq2seq, 561
encoder block (transformer), 587
encoder-decoder attention, 579

ensemble, 280
entropy, 143
environment (in reinforcement learning), 602
environmental state, 605
episode, 605
epoch, 182
epoch gradient descent, 401
epsilon-greedy policy, 630
epsilon-soft policy, 630
error (for neural network error), 352
error curve (for neural network error), 355
error-based decay, 399
event
  in information theory, 136
  in probability, 50
evidence, 96
expected value, 28
expert system, 4
explainability. *See* transparency
exploding gradient, 553
exploit, 608
explore or exploit dilemma, 608
exponential decay (of learning rate), 396
exponential ReLU activation function, 340
extra trees, 302
Extremely Randomized Trees, 302

F

f1 score, 71
fair coin, 86
false alarm, 73
false discovery rate, 73
false negative, 60
false negative rate, 73
false positive, 60
false positive rate, 73
FC. *See* fully connected layer
feature
  as part of a sample, 157
  a target of convolution filter, 438
feature bagging, 301
feature detector, 438
feature engineering, 5
feature filtering, 243
feature map, 438
feature selection, 243
featurewise processing, 233
feed-forward networks, 322
feedback, 602
feedback (in a neural network), 324
fertilized, 157
few-shot training, 594
fiber size (of a tensor), 430
fibre, 234
fidelty (of encoding), 497
fill a box with a sphere (scenario), 175
filter
  kernel, 433
  value, 433
finding animals for movie (scenario), 566
finding information, 4
fine-tuning, 542
finite distribution, 22
fist (sending Morse Code), 138
fitting wind speed data (scenario), 205
fixed-length code, 141
fixed-step decay, 399
flat distribution, 22
flatten layer
  example, 475
  illustration, 476
  schematic symbol, 474
Flippers game (scenario), 614
fold (in cross-validation), 192
footprint (of a convolution filter), 436
forest, 301
forging money (scenario), 650
fractional convolution, 457
frequentist probability
  vs. Bayesian, 85
  overview, 84
frozen weights
  deep dreaming, 677
  in a GAN, 660
full modal collapse, 671
fully connected layer
  introduction, 328
  schematic symbol, 328
fully connected network, 328
function
  argument, 118
  continuous, 118
  graphed, 124

mathematical, 118
return values, 118
single-valued, 119
smooth, 118

G

g (gradient), 410
game theory, 656
GAN (generative adversarial network), 649
garbage in, garbage out, 223
gated recurrent unit, 554
Gaussian distribution, 22
GD. *See* gradient descent
generalization
accuracy, 197
error, 197
loss, 197
generated image (for style transfer), 680
generative adversarial network, 649
generator, 650
generator-discriminator, 664
Gini impurity, 281
Glorot Normal initialization, 325
Glorot Uniform initialization, 325
GPT-2, 593
GPT-3
discussion, 596
performance, 597
gradient
definition, 126
vanishing, 129
zero, 129
gradient ascent, 481
gradient descent
Adadelta, 418
Adagrad, 417
Adam, 420
batch, 401
epoch, 401
mini-batch, 405
momentum, 409
Nesterov, 414
RMSprop, 418
stochastic, 403
typical meaning, 407
Gram matrix, 681
graph
acyclic, 324
directed, 323
introduction, 323
weighted graph, 324
graph theory, 323
Great Dane, 184
greedy algorithm (for decision trees), 274
Green Eggs and Ham, 137
grid, 329
ground truth, 57
grouping. *See* clustering
GRU (gated recurrent unit), 554
guitar data, 228

H

He Normal initialization, 325
He Uniform initialization, 325
head (in attention layer), 580
Heaviside step activation function, 334
hidden layer, 327
hidden state, 549
hierarchy
of features, 440
of filters, 461
of scales, 471
high-dimensional spaces, 42
high-dimensional weirdness
hypercube, 175
hyperorange, 178
hypersphere, 175
packing hyperspheres in hypercubes, 177
volume of a hypersphere in a hypercube, 175
hill (3D surface), 129
hit rate, 66
Holmes, Sherlock (stories)
generating with GPT2, 596
generating with RNN, 555
Hopper, Edward, 685
Huffman code, 148
husky, 185
hypercube, 175
hyperorange, 178
hyperparameter
number of clusters, 166
tuning, 168

hypersphere, 175
hypothesis (in Bayes' Rule), 102

I

i.i.d. (independent and identically distributed), 29
ice-cream shop (scenario), 55
idealized curve, 205
identically distributed, 29
identifying dog breeds (scenario), 183
identity activation function, 331
IG (information gain), 281
image classifier, 6
image matrix (for style transfer), 681
ImageNet, 478
immediate children (in decision tree), 271
impurity, 280
inceptionism, 679
independent and identically distributed, 29
independent variable, 29
inertia, 408
infinite distribution, 24
information
  definition, 137
  density, 143
  divergence, 151
  finding, 4
  gain, 281
  properties, 136
information exchange
  global context, 135
  local context, 135
  receiver, 134
  sender, 134
  surprise, 134
information leakage
  basic idea, 187
  in cross validation, 239
information theory, 133
initial state (in reinforcement learning), 605
initialization (of a neural network), 325
input layer, 327
input tensor, 329
instant reward, 610
internal node (in decision tree), 270
interval decay, 399
inverse transformation, 234
inversion (of encoding), 497

J

joint probability, 53
JPG encoder, 495
junkyard (scenario), 17

K

k
  dimensions left after PCA, 249
  number of clusters, 166
  number of folds, 192
  number of nearest neighbors, 264
k-means clustering, 166
k-nearest neighbors, 264
Kandinsky, Wassily, 685
kernel (of a convolution filter), 433
kernel trick, 287
key (for transformer), 575
kissing line (tangent), 122
KL divergence, 151
kNN (k-nearest neighbors), 264
Kullback-Leibler divergence, 151

L

L-Learning, 616
L-table, 617
L-value, 617
label, 156
language model, 541
last-minute stopping, 202
latent layer, 501
latent space, 509
latent variable, 501
law of diminishing returns in ensemble construction, 300
layer normalization, 583
layer schematic symbol
  attention, 581
  batchnorm, 425
  bidirectional recurrent, 559
  convolution, 471
  convolution with downsampling, 521
  convolution with upsampling, 521
  dropout, 423
  embedding, 574

flatten, 474
fully connected, 328
masked attention, 588
multi-head attention, 581
norm-add, 583
pooling, 451
positional embedding, 584
recurrent, 555
recurrent cell, 551
reshape, 667
self-attention, 581
layers
introduction, 322
overview, 10
lazy algorithm (kNN), 265
leaf (in decision tree), 270
leaky ReLU activation function, 337
learning rate
in neural networks, 377
in Q-learning, 629
learning rate adjustment
constant, 391
delayed exponential decay, 398
error-based decay, 399
exponential decay, 396
fixed-step decay, 399
interval decay, 399
learning the slow way (scenario), 354
LeCun Normal initialization, 325
LeCun Uniform initialization, 325
letter frequencies in English, 139
life-seeking probes (scenario), 97
likelihood, 96
line (in a neural network), 323
line fitting with Bayes' Rule (scenario), 212
linear activation function, 332
linear function, 331
linear layer. *See* fully connected layer
link (in decision tree), 270
list, 329
local receptive field, 436
logical flow, 540
logistic curve activation function, 341
long short-term memory, 553
long-term dependency problem, 563
loop (in a neural network), 324
loss, 352
lossless encoding, 496
lossy encoding, 496
lousy learning, 616
low-dimensional dog breed descriptions (scenario), 255
low-pass filter, 451
LSTM (long short-term memory), 553

M

m
modification of neuron value, 357
momentum (in gradient descent), 410
machine learning, 3
magnitude (of a gradient), 128
The Manchurian Candidate, 598
mapping (transformation), 236
marginal probability, 55
Marx, Groucho, 559
masked attention layer schematic symbol, 588
masking (attention decoder), 588
matching a face mask (scenario), 462
matrix, 329
max pooling, 452
maximum ascent, 126
maximum descent, 126
maximum of a function
finding with derivative, 125
finding with gradient, 128
global maximum, 119
local maximum, 121
maxout activation function, 338
McGlassface, Glasses, 61
mean, 16
mean normalization, 228
mean of Gaussian, 24
mean squared error, 365
mean subtraction, 228
median, 16
mellowmax, 632
messaging with mirrors (scenario), 497
mini-batch, 405
mini-batch gradient descent, 405
minimum of a function
finding with derivative, 125
finding with gradient, 128
global minimum, 119
local minimum, 121

mixing colors of paint (scenario), 574
MLP (multilayer perceptron), 328
MNIST
    used for an autoencoder, 505
    used for convolution, 473
    used for GAN, 667
modal collapse, 671
mode, 16
momentum, 408
momentum gradient descent, 409
Monet, Claude, 685
morbus pollicus (scenario), 63
Morse code, 137
Morse, Samuel, 139
MP (*morbus pollicus*), 63
MP3 encoder, 495
MSE (mean squared error), 365
multi-head attention
    introduction, 580
    schematic symbol, 581
multiclass classification, 160
multidimensional convolution, 443
multifilter loss, 676
multilayer loss, 676
multilayer perceptron, 328
multinoulli distribution, 27
multiple correlation, 39
multivariate transformation, 231
Munch, Edvard, 685
Muppets
    BERT, 590
    ELMo, 571
music system
    collecting data, 199
    good fit, 201
    overfitting, 199
    underfitting, 200

N

n-gram, 595
naive Bayes classifier, 290
Nash equilibrium, 657
natural language generation, 541
natural language processing, 540
natural language understanding, 540
naughts and crosses, 603
negative correlation, 38
negative covariance, 35
negative gradient, 128
negative predictive value, 73
neighbor (for kNN), 265
neighborhood, 120
Nesterov momentum, 414
network collapse, 329
neural network
    introduction, 315
    simple example, 321
neural style transfer, 680
neuron
    artificial, 317
    dead, 345
    overview, 11
    real, 314
neurotransmitter, 314
New Horizons spacecraft, 190
next sentence prediction, 591
NLG (natural language generation), 541
NLP (natural language processing), 540
NLU (natural language
        understanding), 540
No Free Lunch Theorem, 422
node
    in a decision tree, 270
    in a neural network, 323
node splitting
    Gini impurity, 281
    impurity, 280
    information gain, 281
    overview, 272
    purity, 280
noisy curve, 205
noisy ReLU activation function, 339
nominal data, 226
non-deterministic (variational
        autoencoder), 521
non-parametric classifier, 264
nonlinear function, 331
nonlinearity. *See* activation function
norm-add
    definition, 583
    schematic symbol, 583
normal deviate, 24
normal distribution, 22
normalization, 228
NSP (next sentence prediction), 591
numerical data, 226

O

OAA (one-against-all), 161
observability, 608
observation (in Bayes' Rule), 102
offline algorithm, 403
on-demand algorithm (kNN), 265
one-against-all, 161
one-by-one convolution, 447
one-hot encoding, 226
one-versus-all, 161
one-versus-one, 153
one-versus-rest, 161
online algorithm, 405
optimizers, 387
ordinal data, 226
outlier
    when cleaning data, 223
    when making a boundary, 201
output layer, 327
output tensor, 329
OvA (one-versus-all), 161
overfitting
    decision trees, 275
    definition, 196
    dog breeds, 185
overshoot (during gradient descent), 396
OvO (one-versus-one), 163
OvR (one-versus-rest), 161

P

padding, 440
parameter space (for autoencoder), 508
parametric blending, 498
parametric classifier, 264
parametric ReLU activation function, 337
paraphrase (text), 540
parent
    in a decision tree, 271
    in a neural network, 324
partial correlation, 39
partial modal collapse, 671
PCA. *See* principal component analysis
pdf (probability density function), 21
penalty (for neural network error), 352
perceptron
    introduction, 316
    Mark I Perceptron, 317
perfect precision, 69
perfect recall, 69
performance metrics
    illustrated, 75
    summary, 73
perturbation, 491
Picasso, Pablo, 681
piecewise linear activation function, 336
pixel, 430
planet mining (scenario), 97
plateau (3D surface), 129
plurality voting, 299
Pluto images, 190
pmf (probability mass function), 20
pointwise feed-forward layer, 587
policy, 606
polysemy, 559
poodle, 183
pooling
    average, 452
    discussion, 449
    layer, 452
    max, 452
    schematic symbol, 451
population (in statistics), 32
positional
    encoding, 584
    schematic symbol, 584
positional embedding, 585
positive correlation, 38
positive covariance, 35
positive predictive value, 64
posterior, 96
precision
    definition, 64
    perfect, 69
precision-recall tradeoff, 67
predicted value, 57
predicting traffic from temperature (scenario), 234
prediction, 6
pretraining, 542
principal component analysis
    description, 244
    line of maximum variance, 246
    projection, 244
prior, 96

probability
  conditional, 50
  dependence, 50
  event, 50
  independence, 50
  introduction, 19
  joint, 53
  marginal, 55
  simple, 50
probability density function, 21
probability distribution
  bell curve, 22
  Bernoulli, 26
  categorical, 27
  continuous, 20
  example in junkyard, 19
  finite, 22
  Gaussian, 22
  infinite, 24
  introduction, 17
  multinoulli, 27
  normal, 22
  uniform, 21
probability mass function, 20
ProGAN, 670
projection, 244
prompt (for text generation), 541
protect an orange in transit (scenario), 177
pruning, 280
purity, 280

Q

Q-Learning, 626
Q-table, 627
Q-value, 627
Q/KV (query / key, value), 579
QKV (query, key, value), 575
quadratic cost function, 365
quality control for dolls (scenario), 61
quality learning, 626
quantitative data, 226
query (for transformer), 575
quitter, 160

R

random forest, 301
random labeling, 302
random variable
  description, 20
  drawing, 20
  introduction, 17
randomness, 16
real-world data, 183
recall
  definition, 66
  perfect, 69
receptor site (neuron), 314
recipes, 6
reconstruction (of encoded signal), 496
recurrent cell
  introduction, 550
  schematic symbol, 551
recurrent layer schematic symbol, 555
recurrent neural network
  bidirectional, 558
  deep, 557
  deep bidirectional, 560
  introduction, 548
reference point (of a convolution filter), 436
regularization
  batchnorm, 424
  dropout, 422
  introduction, 203
  layer normalization, 583
reinforcement learning, 602
rejection, 73
relative entropy, 151
release data, 183
released system, 183
ReLU activation function, 336
remembering people's names (scenario), 196
reparameterization trick, 527
representation blending, 498
reshape layer
  introduction, 667
  schematic symbol, 667
residual connection, 582
resulting reward, 610
reward
  discounted future, 609
  final, 604
  instant, 610
  introduction, 602

resulting, 610
total, 609
total future, 609
ultimate, 604
reward signal. *See* reward
rigged coin, 86
RL (reinforcement learning), 602
RMSprop, 418
RNN. *See* recurrent neural network
rolled-up diagram, 550
root (in decision tree), 270
rotation validation. *See* cross-validation
rules, 4

S

saddle (3D surface), 130
sample, 156
sample set, 32
samplewise processing, 232
SARSA, 639
scenarios
archaeology clustering, 8
archaeology coin-flipping game, 103
baby lengths, 33
building a sundae, 548
choosing store background music, 198
coin detection, 86
coin-flipping game, 87
dart throwing, 48
distinguishing cows and zebras, 224
elevator scheduling, 9
fill a box with a sphere, 175
finding animals for movie, 566
fitting wind speed data, 205
Flippers game, 614
forging money, 650
ice-cream shop, 55
identifying dog breeds, 183
junkyard, 17
learning the slow way, 354
life-seeking probes, 97
line fitting with Bayes' Rule, 212
low-dimensional dog breed descriptions, 255
matching a face mask, 462
messaging with mirrors, 497
mixing colors of paint, 574
morbus pollicus, 63
planet mining, 97
predicting traffic from temperature, 234
protect an orange in transit, 177
quality control for dolls, 61
remembering people's names, 196
sorting eggs, 156
testing for disease, 63
text messaging, 135
transmitting books, 146
scientific notation, 222
seed (for text generation), 541
selection
sampling with replacement (SWR), 30
sampling without replacement (SWOR), 30
with replacement, 30
without replacement, 30
selection with replacement, 30
selection without replacement, 30
selectively persistent short-term memory, 553
self-attention, 576
self-attention layer
introduction, 578
schematic symbol, 581
semantics, 546
semi-supervised learning, 501
sensitivity, 66
sentence embedding, 571
sentiment analysis
introduction, 540
with BERT, 592
seq2seq, 561
sequence, 539
SGD (stochastic gradient descent), 401
Shannon, Claude, 133
shift invariance, 453
shifted ReLU activation function, 338
SI (slope-intercept) space, 213
Siberian husky, 185
sibling (in decision tree), 271
sigmoid activation function, 341
sign activation function, 334

simple correlation, 39
simple probability, 50
sine wave activation function, 343
skip connection, 582
sleuth (of bears), 568
slice processing
    elementwise, 234
    featurewise, 233
    samplewise, 232
sliding window, 543
slope (tangent), 122
softmax
    as activation function, 345
    in Q-learning, 630
softplus activation function, 340
sorting dolls (scenario), 61
sorting eggs (scenario), 156
spatial filter, 436
specificity, 73
splitting. *See* node splitting
stairstep activation function, 332
standard deviation, 24
standardization, 229
Starship *Theseus*, 97
state
    agent (reinforcement learning), 605
    environmental (reinforcement learning), 605
    hidden (RNN), 549
    of an RNN, 548
    starting (RNN), 548
    variable (reinforcement learning), 605
state variable, 605
statistics, 15
step activation function, 332
Stevenson, Robert Louis, 137
stochastic environment, 611
stochastic gradient descent, 401
striding, 453
strong learner, 303
style image, 680
style loss, 683
style matrix, 681
style transfer, 680
subjective Bayes, 97
subtree (of decision tree), 271
summarize (text), 540
supervised learning
    for classification, 156
    overview, 6
support layer, 327
support vector, 284
support vector machine
    kernel trick, 287
    overview, 282
    strictness parameter C, 285
    support vector, 284
surprise, 134
SVM. *See* support vector machine
sweeping (a filter), 433
swish activation function, 340
SWOR (selection without replacement), 30
SWR (selection with replacement), 30

T

A Tale of Two Cities
    with embedding, 579
    prediction, 545
tangent, 122
tangent line, 122
target (one-hot encoded label), 361
temperature (selecting outputs), 595
tensor, 328
terminal node (in decision tree), 270
test data
    definition, 186
    never learn from, 186
testing for disease (scenario), 63
text messaging (scenario), 135
TFR (total future reward), 609
threshold
    for artificial neuron, 316
    for real neuron, 314
tic-tac-toe, 603
time step, 549
toad visual system
    hierarchy, 461
    introduction, 437
token
    embedding, 566
    as a word, 540
total future reward, 609

total reward, 609
training
accuracy, 196
basic idea, 182
deep network overview, 12
error, 196
flowchart, 189
loss, 196
training set, 182
transfer function. *See* activation function
transfer learning, 542
transformation
inverse, 234
multivariate, 231
overview, 223
univariate, 231
transformer
introduction, 586
structure, 588
translation, 540
transmitting books (scenario), 146
transparency
decision trees, 274
one-versus-one classifiers, 164
transposed convolution, 457
Treasure Island, 137
triple modular redundancy, 297
true discovery rate, 73
true negative, 60
true positive, 60
true positive rate, 66
Turner, J.M.W., 685
Twain, Mark, 146
two-moons dataset, 275
type I error, 73
type II error, 73

U

unbalanced decision tree, 270
uncertainty, 143
underfitting, 197
unfair coin, 86
unfertilized, 157
uniform distribution, 21
unit (for artificial neuron), 315
unit step activation function, 334
univariate transformation, 231
Universal Language Model Fine-Tuning, 574
universal perturbation, 491
unrolled diagram, 550
unsupervised learning, 8
update rule, 617
upsampling, 457
upsizing, 457
user data, 183

V

VAE (variational autoencoder), 521
Vail, Alfred, 139
validation data. *See* validation set
validation error, 197
validation set
definition, 187
estimating error from, 189
valley (3D surface), 129
value (for transformer), 575
van Gogh, Vincent, 685
vanishing gradient
on surfaces, 128
training an RNN, 553
variable-bitrate code, 143
variance
with respect to bias, 204
in statistics, 26
variance normalization, 229
variational autoencoder, 521
vector, 329
vertex (in a neural network), 323
VGG16
for creative applications, 676
introduction, 478
visualizing filters, 480
vocabulary, 541
volume, 329
voting (ensembles)
confidence, 299
overview, 298
plurality voting, 299
weighted plurality voting, 299

W

weak learner, 303

weight
    naming convention for
        neurons, 322
    for neurons, 316
    overview in deep network, 11
weight sharing, 433
weighted coin, 86
weighted graph, 324
weighted plurality voting, 299
weirdness (high-dimensional), 175
width (of a recurrent cell), 551
winner (egg), 160
wire (in a neural network), 323
word embedding, 566

X

Xavier Normal initialization, 325
Xavier Uniform initialization, 325

Y

yolker, 160
Yorkshire terrier, 184

Z

zero gradient, 128
zero padding, 442
zero point, 436
zero-dimensional array, 328
zero-shot training, 594

Deep Learning: A Visual Approach is set in New Baskerville, Futura, and Dogma. The book was printed and bound by Versa Printing in East Peoria, Illinois. The paper is 70# White Coated (Matte), which is certified by the Forest Stewardship Council (FSC).

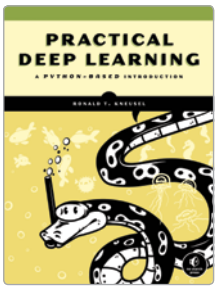
The book uses a perfect binding with polyurethane reactive (PUR) glue, the most durable bookbinding glue available. Its superior flexibility prevents the spine from cracking when the book is opened wide or pressed down flat.

The book was outlined with dozens of yellow Post-it notes on a pair of three-fold science fair display boards, each marked with a shiny star sticker when that section was completed. The manuscript was written on a MacBook Pro and an iMac using the vi text editor to produce Markdeep files. Final edits were made using Microsoft Word and then Adobe Acrobat and Adobe InDesign. Figures were hand-drawn with colored pens and then redrawn in Adobe Illustrator and Photoshop. Computer-generated figures were made with Python code written in Jupyter notebooks. Notable Python libraries included scikit-learn, scikit-image, numpy, scipy, pandas, matplotlib, TensorFlow, and PyTorch.

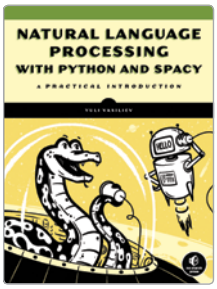
RESOURCES

Visit *https://nostarch.com/deep-learning-visual-approach/* for errata and more information.

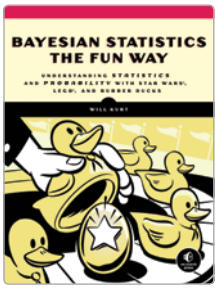
More no-nonsense books from **NO STARCH PRESS**



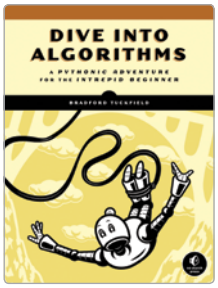
PRACTICAL DEEP LEARNING
A Python-Based Introduction
BY RONALD T. KNEUSEL
464 PP., $59.95
ISBN 978-1-7185-0074-7



NATURAL LANGUAGE PROCESSING WITH PYTHON AND SPACY
A Practical Introduction
BY YULI VASILIEV
216 PP., $39.95
ISBN 978-1-7185-0052-5



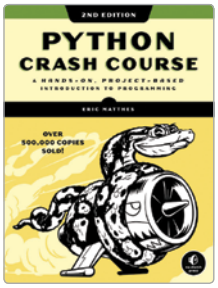
BAYESIAN STATISTICS THE FUN WAY
Understanding Statistics and Probability with Star Wars, LEGO, and Rubber Ducks
BY WILL KURT
256 PP., $34.95
ISBN 978-1-59327-956-1



DIVE INTO ALGORITHMS
A Pythonic Adventure for the Intrepid Beginner
BY BRADFORD TUCKFIELD
248 PP., $39.95
ISBN 978-1-7185-0068-6



REAL-WORLD PYTHON
A Hacker's Guide to Solving Problems with Code
BY LEE VAUGHAN
306 PP., $34.95
ISBN 978-1-7185-0062-4



PYTHON CRASH COURSE, 2ND EDITION
A Hands-On, Project-Based Introduction to Programming
BY ERIC MATTHES
544 PP., $39.95
ISBN 978-1-59327-928-8

PHONE:
800.420.7240 OR
415.863.9900

EMAIL:
SALES@NOSTARCH.COM

WEB:
WWW.NOSTARCH.COM



Never before has the world relied so heavily on the Internet to stay connected and informed. That makes the Electronic Frontier Foundation's mission—to ensure that technology supports freedom, justice, and innovation for all people—more urgent than ever.

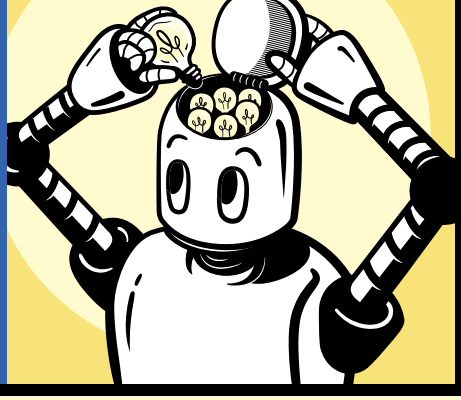
For over 30 years, EFF has fought for tech users through activism, in the courts, and by developing software to overcome obstacles to your privacy, security, and free expression. This dedication empowers all of us through darkness. With your help we can navigate toward a brighter digital future.

ELECTRONIC FRONTIER FOUNDATION EFF

LEARN MORE AND JOIN EFF AT EFF.ORG/NO-STARCH-PRESS

"MUCH MORE ACCESSIBLE THAN THE TYPICAL TREATMENTS."

—PETER SHIRLEY, NVIDIA



768 PAGES
723 FIGURES
0 COMPLEX MATH

Ever since computers began beating us at chess, they've been getting better at a wide range of human activities, from writing songs and generating news articles to helping doctors diagnose and treat diseases. Deep learning is the source of many of these breakthroughs, and its remarkable ability to find patterns in data has made it the fastest-growing field in artificial intelligence. Digital phone assistants use deep learning to understand and respond to voice commands; automotive systems use it to safely navigate roads; and online platforms use it to make personalized recommendations for movies and books.

Deep Learning will help you understand this fascinating field without requiring any advanced math or programming skills. The book's conversational style, full-color illustrations, and real-world examples expertly explain key concepts. If you want to know how deep learning tools work and use them yourself, you'll find the answers here. And when you're ready to write your own programs, the supplemental Python notebooks in the book's GitHub repository will get you started.

You'll learn how:

- Text generators create stories and articles
- Deep learning systems learn to win human games
- Image classification systems identify objects or people in a photo
- Probabilities are useful in everyday life
- Machine learning techniques contribute to modern AI

Intellectual adventurers of all kinds can use the powerful ideas covered in *Deep Learning* to build intelligent systems that help us better understand the world around us. It's the future of AI, and this book will take you there.

ABOUT THE AUTHOR

Dr. Andrew Glassner is a Senior Research Scientist at Weta Digital, where he uses deep learning to help artists produce visual effects for film and television. He was Technical Papers Chair for SIGGRAPH '94, founding editor of the *Journal of Computer Graphics Tools*, and editor-in-chief of ACM *Transactions on Graphics*. His prior books include the *Graphics Gems* series and the textbook *Principles of Digital Image Synthesis*.

FULL COLOR

no starch press

THE FINEST IN GEEK ENTERTAINMENT™

www.nostarch.com

$99.99 ($130.99 CDN)

ISBN 978-1-7185-0072-3

9 781718 500723 59999