

Volume 1

**DEEP LEARNING:
From Basics
to Practice**

Andrew Glassner

Deep Learning: From Basics to Practice

Volume 1

Copyright (c) 2018 by Andrew Glassner

www.glassner.com / @AndrewGlassner

All rights reserved. No part of this book, except as noted below, may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the author, except in the case of brief quotations embedded in critical articles or reviews.

The above reservation of rights does not apply to the program files associated with this book (available on GitHub), or to the images and figures (also available on GitHub), which are released under the MIT license. Any images or figures that are not original to the author retain their original copyrights and protections, as noted in the book and on the web pages where the images are provided.

All software in this book, or in its associated repositories, is provided “**as is**,” without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort, or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

First published February 20, 2018

Version 1.0.1 March 3, 2018

Version 1.1 March 22, 2018

Published by The Imaginary Institute, Seattle, WA.

<http://www.imaginary-institute.com>

Contact: andrew@imaginary-institute.com

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

Contents of Both Volumes

Volume 1

Preface	i
Chapter 1: An Introduction	1
1.1 Why This Chapter Is Here	3
1.1.1 Extracting Meaning from Data.....	4
1.1.2 Expert Systems	6
1.2 Learning from Labeled Data	9
1.2.1 A Learning Strategy	10
1.2.2 A Computerized Learning Strategy	12
1.2.3 Generalization	16
1.2.4 A Closer Look at Learning.....	18
1.3 Supervised Learning.....	21
1.3.1 Classification	21
1.3.2 Regression.....	22
1.4 Unsupervised Learning.....	25
1.4.1 Clustering	25
1.4.2 Noise Reduction.....	26
1.4.3 Dimensionality Reduction	28
1.5 Generators.....	32
1.6 Reinforcement Learning.....	34
1.7 Deep Learning	37
1.8 What's Coming Next	43
References	44
Image credits	45

Chapter 2: Randomness and Basic Statistics	46
2.1 Why This Chapter Is Here.....	48
2.2 Random Variables	49
2.2.1 Random Numbers in Practice.....	57
2.3 Some Common Distributions.....	59
2.3.1 The Uniform Distribution	60
2.3.2 The Normal Distribution	61
2.3.3 The Bernoulli Distribution	67
2.3.4 The Multinoulli Distribution.....	69
2.3.5 Expected Value	70
2.4 Dependence	70
2.4.1 i.i.d. Variables.....	71
2.5 Sampling and Replacement.....	71
2.5.1 Selection With Replacement	73
2.5.2 Selection Without Replacement	74
2.5.3 Making Selections.....	75
2.6 Bootstrapping	76
2.7 High-Dimensional Spaces	82
2.8 Covariance and Correlation.....	85
2.8.1 Covariance	86
2.8.2 Correlation.....	88
2.9 Anscombe's Quartet.....	93
References	95

Chapter 3: Probability	97
3.1 Why This Chapter Is Here.....	99
3.2 Dart Throwing	100
3.3 Simple Probability	103
3.4 Conditional Probability.....	104
3.5 Joint Probability.....	109
3.6 Marginal Probability.....	114
3.7 Measuring Correctness	115
3.7.1 Classifying Samples.....	116
3.7.2 The Confusion Matrix	119
3.7.3 Interpreting the Confusion Matrix	121
3.7.4 When Misclassification Is Okay.....	126
3.7.5 Accuracy.....	129
3.7.6 Precision	130
3.7.7 Recall	132
3.7.8 About Precision and Recall	134
3.7.9 Other Measures.....	137
3.7.10 Using Precision and Recall Together	141
3.7.11 f1 Score	143
3.8 Applying the Confusion Matrix	144
References	151

Chapter 4: Bayes Rule	153
4.1 Why This Chapter Is Here	155
4.2 Frequentist and Bayesian Probability	156
4.2.1 The Frequentist Approach	156
4.2.2 The Bayesian Approach	157
4.2.3 Discussion	158
4.3 Coin Flipping	159
4.4 Is This a Fair Coin?.....	161
4.4.1 Bayes' Rule	173
4.4.2 Notes on Bayes' Rule	175
4.5 Finding Life Out There	178
4.6 Repeating Bayes' Rule.....	183
4.6.1 The Posterior-Prior Loop	184
4.6.2 Example: Which Coin Do We Have?	186
4.7 Multiple Hypotheses.....	194
References	203
Chapter 5: Curves and Surfaces.....	205
5.1 Why This Chapter Is Here.....	207
5.2 Introduction	207
5.3 The Derivative.....	210
5.4 The Gradient	222
References	229

Chapter 6: Information Theory	231
6.1 Why This Chapter Is Here	233
6.1.1 Information: One Word, Two Meanings	233
6.2 Surprise and Context.....	234
6.2.1 Surprise	234
6.2.2 Context.....	236
6.3 The Bit as Unit	237
6.4 Measuring Information	238
6.5 The Size of an Event.....	240
6.6 Adaptive Codes.....	241
6.7 Entropy	250
6.8 Cross-Entropy.....	253
6.8.1 Two Adaptive Codes	253
6.8.2 Mixing Up the Codes	257
6.9 KL Divergence.....	260
References	262
Chapter 7: Classification	265
7.1 Why This Chapter Is Here	267
7.2 2D Classification.....	268
7.2.1 2D Binary Classification.....	269
7.3 2D Multi-class classification	275
7.4 Multiclass Binary Categorizing.....	277
7.4.1 One-Versus-Rest	278
7.4.2 One-Versus-One	280
7.5 Clustering	286

7.6 The Curse of Dimensionality	290
7.6.1 High Dimensional Weirdness.....	299
References	307
Chapter 8: Training and Testing.....	309
8.1 Why This Chapter Is Here	311
8.2 Training	312
8.2.1 Testing the Performance	314
8.3 Test Data.....	318
8.4 Validation Data	323
8.5 Cross-Validation	328
8.5.1 k-Fold Cross-Validation.....	331
8.6 Using the Results of Testing	334
References	335
Image Credits.....	336
Chapter 9: Overfitting and Underfitting	337
9.1 Why This Chapter Is Here.....	339
9.2 Overfitting and Underfitting	340
9.2.1 Overfitting	340
9.2.2 Underfitting.....	342
9.3 Overfitting Data	342
9.4 Early Stopping.....	348
9.5 Regularization	350

9.6 Bias and Variance	352
9.6.1 Matching the Underlying Data	353
9.6.2 High Bias, Low Variance	357
9.6.3 Low Bias, High Variance	359
9.6.4 Comparing Curves	360
9.7 Fitting a Line with Bayes' Rule	363
References	372
Chapter 10: Neurons.....	374
10.1 Why This Chapter Is Here	376
10.2 Real Neurons	376
10.3 Artificial Neurons	379
10.3.1 The Perceptron	379
10.3.2 Perceptron History	381
10.3.3 Modern Artificial Neurons	382
10.4 Summing Up.....	390
References	390
Chapter 11: Learning and Reasoning.....	393
11.1 Why This Chapter Is Here	395
11.2 The Steps of Learning.....	396
11.2.1 Representation	396
11.2.2 Evaluation	400
11.2.3 Optimization	400
11.3 Deduction and Induction	402
11.4 Deduction	403
11.4.1 Categorical Syllogistic Fallacies	410

11.5 Induction	415
11.5.1 Inductive Terms in Machine Learning	419
11.5.2 Inductive Fallacies	420
11.6 Combined Reasoning.....	422
11.6.1 Sherlock Holmes, “Master of Deduction”.....	424
11.7 Operant Conditioning.....	425
References	428
Chapter 12: Data Preparation.....	431
12.1 Why This Chapter Is Here	433
12.2 Transforming Data.....	433
12.3 Types of Data	436
12.3.1 One-Hot Encoding.....	438
12.4 Basic Data Cleaning	440
12.4.1 Data Cleaning.....	441
12.4.2 Data Cleaning in Practice	442
12.5 Normalizing and Standardizing	443
12.5.1 Normalization	444
12.5.2 Standardization	446
12.5.3 Remembering the Transformation.....	447
12.5.4 Types of Transformations	448
12.6 Feature Selection	450
12.7 Dimensionality Reduction	451
12.7.1 Principal Component Analysis (PCA)	452
12.7.2 Standardization and PCA for Images	459
12.8 Transformations	468

12.9 Slice Processing	475
12.9.1 Samplewise Processing.....	476
12.9.2 Featurewise Processing.....	477
12.9.3 Elementwise Processing	479
12.10 Cross-Validation Transforms	480
References	486
Image Credits.....	486
Chapter 13: Classifiers	488
13.1 Why This Chapter Is Here	490
13.2 Types of Classifiers.....	491
13.3 k-Nearest Neighbors (KNN).....	493
13.4 Support Vector Machines (SVMs).....	502
13.5 Decision Trees.....	512
13.5.1 Building Trees.....	519
13.5.2 Splitting Nodes.....	525
13.5.3 Controlling Overfitting	528
13.6 Naïve Bayes.....	529
13.7 Discussion.....	536
References	538
Chapter 14: Ensembles	539
14.1 Why This Chapter Is Here	541
14.2 Ensembles	542
14.3 Voting.....	543
14.4 Bagging.....	544

14.5 Random Forests.....	547
14.6 ExtraTrees	549
14.7 Boosting.....	549
References	561
Chapter 15: Scikit-learn.....	563
15.1 Why This Chapter Is Here	566
15.2 Introduction	567
15.3 Python Conventions.....	569
15.4 Estimators	574
15.4.1 Creation	575
15.4.2 Learning with fit().....	576
15.4.3 Predicting with predict()	578
15.4.4 decision_function(), predict_proba()	581
15.5 Clustering	582
15.6 Transformations	587
15.6.1 Inverse Transformations.....	594
15.7 Data Refinement	598
15.8 Ensembles	601
15.9 Automation	605
15.9.1 Cross-validation.....	606
15.9.2 Hyperparameter Searching	610
15.9.3 Exhaustive Grid Search	614
15.9.4 Random Grid Search	625
15.9.5 Pipelines.....	626
15.9.6 The Decision Boundary.....	641
15.9.7 Pipelined Transformations	643

15.10 Datasets	647
15.11 Utilities.....	650
15.12 Wrapping Up.....	652
References	653
Chapter 16: Feed-Forward Networks	655
16.1 Why This Chapter Is Here	657
16.2 Neural Network Graphs	658
16.3 Synchronous and Asynchronous Flow	661
16.3.1 The Graph in Practice	664
16.4 Weight Initialization	664
16.4.1 Initialization	667
References	670
Chapter 17: Activation Functions	672
17.1 Why This Chapter Is Here.....	674
17.2 What Activation Functions Do	674
17.2.1 The Form of Activation Functions	679
17.3 Basic Activation Functions.....	679
17.3.1 Linear Functions.....	680
17.3.2 The Stair-Step Function.....	681
17.4 Step Functions	682
17.5 Piecewise Linear Functions.....	685
17.6 Smooth Functions.....	690
17.7 Activation Function Gallery	698

17.8 Softmax.....	699
References	702
Chapter 18: Backpropagation.....	703
18.1 Why This Chapter Is Here	706
18.1.1 A Word On Subtlety.....	708
18.2 A Very Slow Way to Learn.....	709
18.2.1 A Slow Way to Learn	712
18.2.2 A Faster Way to Learn	716
18.3 No Activation Functions for Now.....	718
18.4 Neuron Outputs and Network Error.....	719
18.4.1 Errors Change Proportionally.....	720
18.5 A Tiny Neural Network.....	726
18.6 Step 1: Deltas for the Output Neurons	732
18.7 Step 2: Using Deltas to Change Weights....	745
18.8 Step 3: Other Neuron Deltas.....	750
18.9 Backprop in Action	758
18.10 Using Activation Functions	765
18.11 The Learning Rate	774
18.11.1 Exploring the Learning Rate.....	777

18.12 Discussion	787
18.12.1 Backprop In One Place	787
18.12.2 What Backprop Doesn't Do	789
18.12.3 What Backprop Does Do.....	789
18.12.4 Keeping Neurons Happy	790
18.12.5 Mini-Batches.....	795
18.12.6 Parallel Updates	796
18.12.7 Why Backprop Is Attractive	797
18.12.8 Backprop Is Not Guaranteed	797
18.12.9 A Little History	798
18.12.10 Digging into the Math.....	800
References	802
Chapter 19: Optimizers	805
19.1 Why This Chapter Is Here	807
19.2 Error as Geometry.....	807
19.2.1 Minima, Maxima, Plateaus, and Saddles.....	808
19.2.2 Error as A 2D Curve	814
19.3 Adjusting the Learning Rate	817
19.3.1 Constant-Sized Updates.....	819
19.3.2 Changing the Learning Rate Over Time	829
19.3.3 Decay Schedules	832
19.4 Updating Strategies	836
19.4.1 Batch Gradient Descent	837
19.4.2 Stochastic Gradient Descent (SGD)	841
19.4.3 Mini-Batch Gradient Descent.....	844
19.5 Gradient Descent Variations.....	846
19.5.1 Momentum	847
19.5.2 Nesterov Momentum.....	856
19.5.3 Adagrad	862
19.5.4 Adadelta and RMSprop.....	864
19.5.5 Adam	866

19.6 Choosing An Optimizer	868
References	870

Volume 2

Chapter 20: Deep Learning	872
20.1 Why This Chapter Is Here	874
20.2 Deep Learning Overview	874
20.2.1 Tensors	878
20.3 Input and Output Layers	879
20.3.1 Input Layer.....	879
20.3.2 Output Layer	880
20.4 Deep Learning Layer Survey	881
20.4.1 Fully-Connected Layer.....	882
20.4.2 Activation Functions.....	883
20.4.3 Dropout.....	884
20.4.4 Batch Normalization	887
20.4.5 Convolution	890
20.4.6 Pooling Layers	892
20.4.7 Recurrent Layers.....	894
20.4.8 Other Utility Layers	896
20.5 Layer and Symbol Summary	898
20.6 Some Examples	899
20.7 Building A Deep Learner.....	910
20.7.1 Getting Started.....	912
20.8 Interpreting Results.....	913
20.8.1 Satisfactory Explainability.....	920

References	923
Image credits:	925
Chapter 21: Convolutional Neural Networks.....	927
21.1 Why This Chapter Is Here	930
21.2 Introduction	931
21.2.1 The Two Meanings of “Depth”.....	932
21.2.2 Sum of Scaled Values.....	933
21.2.3 Weight Sharing.....	938
21.2.4 Local Receptive Field	940
21.2.5 The Kernel.....	943
21.3 Convolution.....	944
21.3.1 Filters	948
21.3.2 A Fly’s-Eye View	953
21.3.3 Hierarchies of Filters	955
21.3.4 Padding	963
21.3.5 Stride	966
21.4 High-Dimensional Convolution.....	971
21.4.1 Filters with Multiple Channels	975
24.4.2 Striding for Hierarchies.....	977
24.5 1D Convolution.....	979
24.6 1×1 Convolutions	980
24.7 A Convolution Layer	983
24.7.1 Initializing the Filter Weights	984
24.8 Transposed Convolution.....	985
24.9 An Example Convnet.....	991
24.9.1 VGG16	996
21.9.2 Looking at the Filters, Part 1.....	1001
21.9.3 Looking at the Filters, Part 2.....	1008

21.10 Adversaries	1012
References	1017
Image credits	1022
Chapter 22: Recurrent Neural Networks.....	1023
22.1 Why This Chapter Is Here.....	1025
22.2 Introduction	1027
22.3 State	1030
22.3.1 Using State.....	1032
22.4 Structure of an RNN Cell.....	1037
22.4.1 A Cell with More State	1042
22.4.2 Interpreting the State Values	1045
22.5 Organizing Inputs	1046
22.6 Training an RNN.....	1051
22.7 LSTM and GRU.....	1054
22.7.1 Gates	1055
22.7.2 LSTM	1060
22.8 RNN Structures.....	1066
22.8.1 Single or Many Inputs and Outputs	1066
22.8.2 Deep RNN	1070
22.8.3 Bidirectional RNN	1072
22.8.4 Deep Bidirectional RNN.....	1074
22.9 An Example.....	1076
References	1084

Chapter 23: Keras Part 1.....	1090
23.1 Why This Chapter Is Here.....	1093
23.1.1 The Structure of This Chapter.....	1094
23.1.2 Notebooks.....	1094
23.1.3 Python Warnings.....	1094
23.2 Libraries and Debugging.....	1095
23.2.1 Versions and Programming Style.....	1097
23.2.2 Python Programming and Debugging	1098
23.3 Overview.....	1100
23.3.1 What's a Model?.....	1101
23.3.2 Tensors and Arrays	1102
23.3.3 Setting Up Keras.....	1102
23.3.4 Shapes of Tensors Holding Images	1104
23.3.5 GPUs and Other Accelerators	1108
23.4 Getting Started	1109
23.4.1 Hello, World	1110
23.5 Preparing the Data	1114
23.5.1 Reshaping.....	1115
23.5.2 Loading the Data	1126
23.5.3 Looking at the Data	1129
23.5.4 Train-test Splitting.....	1136
23.5.5 Fixing the Data Type	1138
23.5.6 Normalizing the Data	1139
23.5.7 Fixing the Labels.....	1142
23.5.8 Pre-Processing All in One Place.....	1148
23.6 Making the Model.....	1150
23.6.1 Turning Grids into Lists.....	1152
23.6.2 Creating the Model.....	1154
23.6.3 Compiling the Model.....	1163
23.6.4 Model Creation Summary	1167

23.7 Training The Model.....	1169
23.8 Training and Using A Model.....	1172
23.8.1 Looking at the Output	1174
23.8.2 Prediction.....	1180
23.8.3 Analysis of Training History	1186
23.9 Saving and Loading.....	1190
23.9.1 Saving Everything in One File	1190
23.9.2 Saving Just the Weights	1191
23.9.3 Saving Just the Architecture	1192
23.9.4 Using Pre-Trained Models	1193
23.9.5 Saving the Pre-Processing Steps.....	1194
23.10 Callbacks.....	1195
23.10.1 Checkpoints	1196
23.10.2 Learning Rate	1200
23.10.3 Early Stopping.....	1201
References	1205
Image Credits.....	1208
Chapter 24: Keras Part 2.....	1209
24.1 Why This Chapter Is Here.....	1212
24.2 Improving the Model	1212
24.2.1 Counting Up Hyperparameters	1213
24.2.2 Changing One Hyperparameter.....	1214
24.2.3 Other Ways to Improve.....	1218
24.2.4 Adding Another Dense Layer	1219
24.2.5 Less Is More.....	1221
24.2.6 Adding Dropout	1224
24.2.7 Observations.....	1230

24.3 Using Scikit-Learn	1231
24.3.1 Keras Wrappers	1232
24.3.2 Cross-Validation	1237
24.3.3 Cross-Validation with Normalization	1243
24.3.4 Hyperparameter Searching	1247
24.4 Convolution Networks.....	1259
24.4.1 Utility Layers	1260
24.4.2 Preparing the Data for A CNN	1263
24.4.3 Convolution Layers.....	1268
24.4.4 Using Convolution for MNIST.....	1276
24.4.5 Patterns	1290
24.4.6 Image Data Augmentation	1293
24.4.7 Synthetic Data	1298
24.4.8 Parameter Searching for Convnets	1300
24.5 RNNs	1301
24.5.1 Generating Sequence Data.....	1302
24.5.2 RNN Data Preparation	1306
24.5.3 Building and Training an RNN	1314
24.5.4 Analyzing RNN Performance.....	1320
24.5.5 A More Complex Dataset.....	1330
24.5.6 Deep RNNs	1334
24.5.7 The Value of More Data	1338
24.5.8 Returning Sequences	1343
24.5.9 Stateful RNNs.....	1349
24.5.10 Time-Distributed Layers.....	1352
24.5.11 Generating Text	1357
24.6 The Functional API	1366
24.6.1 Input Layers.....	1370
24.6.2 Making A Functional Model	1371
References	1378
Image Credits.....	1379

Chapter 25: Autoencoders.....	1380
25.1 Why This Chapter Is Here.....	1382
25.2 Introduction	1383
25.2.1 Lossless and Lossy Encoding	1384
25.2.2 Domain Encoding.....	1386
25.2.3 Blending Representations.....	1388
25.3 The Simplest Autoencoder	1393
25.4 A Better Autoencoder	1400
25.5 Exploring the Autoencoder	1405
25.5.1 A Closer Look at the Latent Variables.....	1405
25.5.2 The Parameter Space.....	1409
25.5.3 Blending Latent Variables	1415
25.5.4 Predicting from Novel Input	1418
25.6 Discussion.....	1419
25.7 Convolutional Autoencoders.....	1420
25.7.1 Blending Latent Variables	1424
25.7.2 Predicting from Novel Input	1426
25.8 Denoising.....	1427
25.9 Variational Autoencoders	1430
25.9.1 Distribution of Latent Variables.....	1432
25.9.2 Variational Autoencoder Structure	1433
25.10 Exploring the VAE.....	1442
References	1455
Image credits	1457

Chapter 26: Reinforcement Learning.....	1458
26.1 Why This Chapter Is Here.....	1461
26.2 Goals.....	1462
26.2.1 Learning A New Game.....	1463
26.3 The Structure of RL.....	1469
26.3.1 Step 1: The Agent Selects an Action	1471
26.3.2 Step 2: The Environment Responds	1473
26.3.3 Step 3: The Agent Updates Itself	1475
26.3.4 Variations on The Simple Version.....	1476
26.3.5 Back to the Big Picture	1478
26.3.6 Saving Experience.....	1480
26.3.7 Rewards	1481
26.4 Flippers	1490
26.5 L-learning.....	1492
26.5.1 Handling Unpredictability	1505
26.6 Q-learning	1509
26.6.1 Q-values and Updates.....	1510
26.6.2 Q-Learning Policy	1514
26.6.3 Putting It All Together.....	1518
26.6.4 The Elephant in the Room.....	1519
26.6.5 Q-learning in Action	1521
26.7 SARSA.....	1532
26.7.1 SARSA in Action.....	1535
26.7.2 Comparing Q-learning and SARSA.....	1543
26.8 The Big Picture	1548
26.9 Experience Replay.....	1550

26.10 Two Applications	1551
References	1554
Chapter 27: Generative Adversarial Networks... 1558	
27.1 Why This Chapter Is Here	1560
27.2 A Metaphor: Forging Money	1562
27.2.1 Learning from Experience.....	1566
27.2.2 Forging with Neural Networks.....	1569
27.2.3 A Learning Round	1572
27.3 Why Antagonistic?	1574
27.4 Implementing GANs	1575
27.4.1 The Discriminator.....	1576
27.4.2 The Generator.....	1577
27.4.3 Training the GAN	1578
27.4.4 Playing the Game	1581
27.5 GANs in Action	1582
27.6 DCGANs	1591
27.6.1 Rules of Thumb.....	1595
27.7 Challenges	1596
27.7.1 Using Big Samples.....	1597
27.7.2 Modal Collapse	1598
References	1600

Chapter 28: Creative Applications.....	1603
28.1 Why This Chapter Is Here	1605
28.2 Visualizing Filters.....	1605
28.2.1 Picking A Network.....	1605
28.2.2 Visualizing One Filter	1607
28.2.3 Visualizing One Layer	1610
23.3 Deep Dreaming.....	1613
28.4 Neural Style Transfer	1620
28.4.1 Capturing Style in a Matrix	1621
28.4.2 The Big Picture.....	1623
28.4.3 Content Loss	1624
28.4.4 Style Loss	1628
28.4.5 Performing Style Transfer.....	1633
28.4.6 Discussion.....	1640
28.5 Generating More of This Book	1642
References	1644
Image Credits.....	1646
Chapter 29: Datasets.....	1648
29.1 Public Datasets	1650
29.2 MNIST and Fashion-MNIST.....	1651
29.3 Built-in Library Datasets.....	1652
29.3.1 scikit-learn	1652
29.3.2 Keras	1653
29.4 Curated Dataset Collections	1654
29.5 Some Newer Datasets.....	1655

Chapter 30: Glossary	1658
About The Glossary	1660
0-9	1660
Greek Letters	1661
A	1662
B	1665
C	1670
D	1678
E	1685
F	1688
G	1694
H	1697
I	1699
J	1703
K	1703
L	1704
M	1708
N	1713
O	1717
P	1719
Q	1725
R	1725
S	1729
T	1738
U	1741
V	1743
W	1745
X	1746
Z	1746



Preface

Welcome!

A few quick words of introduction to the book, how to get the notebooks and figures, and thanks to the people who helped me.

What You'll Get from This Book

Hello!

If you're interested in deep learning (DL) and machine learning (ML), then there's good stuff for you in this book.

My goal in this book is to give you the broad skills to be an effective practitioner of machine learning and deep learning.

When you've read this book, you will be able to:

- Design and train your own deep networks.
- Use your networks to understand your data, or make new data.
- Assign descriptive categories to text, images, and other types of data.
- Predict the next value for a sequence of data.
- Investigate the structure of your data.
- Process your data for maximum efficiency.
- Use any programming language and DL library you like.
- Understand new papers and ideas, and put them into practice.
- Enjoy talking about deep learning with other people.

We'll take a serious but friendly approach, supported by tons of illustrations. And we'll do it all without any code, and without any math beyond multiplication.

If that sounds good to you, welcome aboard!

Who This Book Is For

This book is designed for people who want to use machine learning and deep learning in their own work. This includes programmers, artists, engineers, scientists, executives, musicians, doctors, and anyone else who wants to work with large amounts of information to extract meaning from it, or generate new data.

Many of the tools of machine learning, and deep learning in particular, are embodied in multiple free, open-source libraries that anyone can immediately download and use.

Even though these tools are free and easy to install, they still require significant technical knowledge to use them properly. It's easy to ask the computer to do something nonsensical, and it will happily do it, giving us back more nonsense as output.

This kind of thing happens all the time. Though machine learning and deep learning libraries are powerful, they're not yet user friendly. Choosing the right algorithms, and then applying them properly, still requires a stream of technically informed decisions. When things often don't go as planned, we need to use our knowledge of what's going on inside the system in order to fix it.

There are multiple approaches to learning and mastering this essential information, depending on how you like to learn.

Some people like **hardcore**, detailed algorithm analysis, supported by extensive mathematics. If that's how you like to learn, there are great books out there that offer this style of presentation [Bishop06] [Goodfellow17]. This approach requires intensive effort, but pays off with a thorough understanding of how and why the machinery works. If you start this way, then you have to put in another chunk of work to translate that theoretical knowledge into contemporary practice.

At the other extreme, some people just want to know how to do some particular task. There are great books that take this cookbook approach for various machine-learning libraries [Chollet17] [Müller-Guido16] [Raschka15] [VanderPlas16]. This approach is easier than the mathematically intensive route, but you can feel like you’re missing the structural information that explains why things work as they do. Without that information, and its vocabulary, it can be hard to work out why something that you think ought to work doesn’t work, or why something doesn’t work as well as you thought it should. It can also be challenging to read the literature describing new ideas and results, because those discussions usually assume a shared body of underlying knowledge that an approach based on a single library or language doesn’t provide.

This book takes a middle road. My purpose is practical: to give you the tools to practice deep learning with confidence. I want you to make wise choices as you do your work, and be able to follow the flood of exciting new ideas appearing almost every day.

My goal here is to cover the fundamentals just deeply enough to give you a broad base of support. I want you to have enough background not just for the topics in this book, but also the materials you’re likely to need to consult and read as you actually do deep learning work.

This is not a book about programming. Programming is important, but it inevitably involves all kinds of details that are irrelevant to our larger subject. And programming examples lock us into one library, or one language. While such details are necessary to building final systems, they can be distracting when we’re trying to focus in the big ideas. Rather than get waylaid by discussions of loops and indices and data structures, we discuss everything here in a language and library independent way. Once you have the ideas firmly in place, reading the documentation for any library will be a straightforward affair.

We do put our feet on the ground in Chapters 15, 23, and 24, when we discuss the scikit-learn library for machine learning, and the Keras library for deep learning. These libraries are both Python based. In

those chapters we dive into the details of those Python libraries, and include plenty of example code. Even if you’re not into Python, these programs will give you a sense for typical workflows and program structures, which can help show how to attack a new problem.

The code in those programming chapters is available in Python notebooks. These are for use with the browser-based Jupyter programming environment [Jupyter16]. Alternatively, you can use a more classical Python development environment, such as PyCharm [JetBrains17].

Most of the other chapters also have supporting, optional Python notebooks. These give the code for every computer-generated figure in the book, often using the techniques discussed in that chapter. Because we’re not really focusing on Python and programming (except for the chapters mentioned above), these notebooks are meant as a “behind the scenes” look, and are only lightly commented.

Machine learning, deep learning, and big data are having an unexpectedly rapid and profound influence on societies around the world. What this means for people and cultures is a complicated and important subject. Some interesting books and articles tackle the topic head-on, often coming to subtle mixtures of positive and negative conclusions [Agüera y Arcas 17] [Barrat15] [Domingos15] [Kaplan16].

Almost No Math

Lots of smart people are not fans of complicated equations. If that's you, then you're right at home!

There's just about no math in this book. If you're comfortable with multiplication, you're set, because that's as mathematical as we get.

Many of the algorithms we'll discuss are based on rich sources of theory and are the result of careful analysis and development. It's important to know that stuff if you're modifying the algorithm for some new purpose, or writing your own implementation. But in practice, just about everyone uses highly optimized implementations written by experts, available in free and open-source libraries.

Our goals are to understand the principles of these techniques, how to apply them properly, and how to interpret the results. None of that requires us to get into the mathematical structure that's under the hood.

If you love math, or you want to see the theory, follow the references in each chapter. Much of this material is elegant and intellectually stimulating, and provides details that I have deliberately omitted from this book. But if math isn't your thing, there's no need to get into it.

Lots of Figures

Some ideas are more clearly communicated with pictures than with words. And even when words do the job, a picture can help cement the ideas. So this book is profusely illustrated with original figures.

All of the figures in this book are available for free download (see below).

Downloads

You can download the Jupyter/Python notebooks for this book, all the figures, and other files related to this book, all for free.

All the Notebooks

All of the Jupyter/Python notebooks for this book are available on GitHub.

The notebooks for Chapter 15 (scikit-learn) and Chapters 23 and 24 (Keras) contain all the code that's presented in those chapters.

The other notebooks are available as a kind of “behind the scenes” look at how the book’s figures were made. They’re lightly documented, and meant to serve more as references than tutorials.

The notebooks are released under the MIT license, which basically means that you’re free to use them for any purpose. There are no promises of any kind that the code is free of bugs, that it will run properly, that it won’t crash, and so on. Feel free to grab the code and adapt it as you see fit, though as the license says, keep the copyright notice around (it’s in the file named simply `LICENSE`).

<https://github.com/blueberrymusic/DeepLearningBookCode-Volume1>
<https://github.com/blueberrymusic/DeepLearningBookCode-Volume2>

All the Figures

All of the figures in this book are available on GitHub as high-resolution PNG files. You’re free to use them in classes, talks, lectures, reports, papers, even other books.

Like the code, the figures are provided under the MIT license, so you can use them as you like as long as you keep the copyright notice around. You don’t have to credit me as their creator when you use these figures, but I’d appreciate it if you would.

The filenames match the figure numbers in the book, so they're easy to find. When you're looking for something visually, it may be helpful to look at the thumbnail pages. These hold 20 images each:

[https://github.com/blueberrymusic/DeepLearningBookFigures-thumbnails](https://github.com/blueberrymusic/DeepLearningBookFigures--thumbnails)

The figures themselves are grouped into the two volumes:

<https://github.com/blueberrymusic/DeepLearningBookFigures-Volume1>

<https://github.com/blueberrymusic/DeepLearningBookFigures-Volume2>

Resources

The resources directory contains other files, such as a template for the deep learning icons we use later in the book.

<https://github.com/blueberrymusic/DeepLearningBook-Resources>

Errata

Despite my best efforts, no book of this size is going to be free of errors. If you spot something that seems wrong, please let me know at andrew@dlbasics.com. I'll keep a list of errata on the book's website at <https://dlbasics.com>.

Two Volumes

This ended up as a large book, so I've organized it into two volumes of roughly equal size.

Because the book is cumulative, the second volume picks up where the first leaves off. If you're reading the second volume now, you should have already read the first volume, or feel confident that you understand the material presented there.

Thank You!

Authors like to say that nobody writes a book alone. We say that because it's true.

For their consistent and enthusiastic support of this project, and helping me feel good about it all the way through, I am enormously grateful to Eric Braun, Eric Haines, Steven Drucker, and Tom Reike. Thank you for your friendship and encouragement.

Huge thanks are due to my reviewers, whose generous and insightful comments greatly improved this book: Adam Finkelstein, Alex Colburn, Alexander Keller, Alyn Rockwood, Angelo Pesce, Barbara Mones, Brian Wyvill, Craig Kaplan, Doug Roble, Eric Braun, Eric Haines, Greg Turk, Jeff Hultquist, Jessica Hodgins, Kristi Morton, Lesley Istead, Luis Avarado, 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 Avarado, who read all or most of the manuscript and offered terrific feedback on both presentation and structure.

Thanks to Morgan McGuire for Markdeep, which enabled me to focus on what I was saying, rather than the mechanics of how to format it. It made writing this book a remarkably smooth and fluid process.

Thanks to Todd Szymanski for insightful advice on the design and layout of the book's contents and covers, and for catching layout errors.

Thanks to early readers who caught typos and other problems: Christian Forfang, David Pol, Eric Haines, Gopi Meenakshisundaram, Kostya Smolenskiy, Mauricio Vives, Mike Wong, and Mrinal Mohit.

All of these people improved the book, but the final decisions were my own. Any problems that remain are my responsibility.

References

This section appears in every chapter. It contains references to all the documents that are referred to in the body of the chapter. There may also be other useful papers, websites, documentation, blogs, and other resources.

Whenever possible, I've preferred to use references that are available online, so you can immediately access them using the provided link. The exceptions are usually books, but occasionally I'll include an important online reference even if it's behind a paywall.

[Agüera y Arcas 17] Blaise Agüera y Arcas, Margaret Mitchell and Alexander Todorov, “Physiognomy’s New Clothes”, Medium, 2017. <https://medium.com/@blaisea/physiognomys-new-clothes-f2d4b59fdd6a>

[Barrat15] James Barrat, “Our Final Invention: Artificial Intelligence and the End of the Human Era”, St. Martin’s Griffin, 2015.

[Bishop06] Christopher M. Bishop, “Pattern Recognition and Machine Learning”, Springer-Verlag, pp. 149-152, 2006.

[Chollet17] François Chollet, “Deep Learning with Python”, Manning Publications, 2017.

[Domingos15] Pedro Domingos, “The Master Algorithm”, Basic Books, 2015.

[Goodfellow17] Ian Goodfellow, Yoshua Bengio, Aaron Courville, “Deep Learning”, MIT Press, 2017. <http://www.deeplearning-book.org/>

[JetBrains17] Jet Brains, “Pycharm Community Edition IDE”, 2017. <https://www.jetbrains.com/pycharm/>

[Jupyter16] The Jupyter team, 2016. <http://jupyter.org/>

- [Kaplan16] Jerry Kaplan, “Artificial Intelligence: What Everyone Needs to Know”, Oxford University Press, 2016.
- [Müller-Guido16] Andreas C. Müller and Sarah Guido, “Introduction to Machine Learning with Python”, O’Reilly Press, 2016.
- [Raschka15] Sebastian Raschka, “Python Machine Learning”, Packt Publishing, 2015.
- [VanderPlas16] Jake VanderPlas, “Python Data Science Handbook”, O’Reilly Media, 2016.

Chapter 1

An Introduction to Machine Learning and Deep Learning

A quick overview of the
ideas, language, and techniques
that we'll be using throughout the book.

Contents

1.1 Why This Chapter Is Here	3
1.1.1 Extracting Meaning from Data.....	4
1.1.2 Expert Systems	6
1.2 Learning from Labeled Data	9
1.2.1 A Learning Strategy.....	10
1.2.2 A Computerized Learning Strategy	12
1.2.3 Generalization	16
1.2.4 A Closer Look at Learning.....	18
1.3 Supervised Learning.....	21
1.3.1 Classification	21
1.3.2 Regression.....	22
1.4 Unsupervised Learning.....	25
1.4.1 Clustering	25
1.4.2 Noise Reduction.....	26
1.4.3 Dimensionality Reduction	28
1.5 Generators.....	32
1.6 Reinforcement Learning.....	34
1.7 Deep Learning.....	37
1.8 What's Coming Next	43
References	44
Image credits	45

1.1 Why This Chapter Is Here

This chapter is to help us get familiar with the big ideas and basic terminology of machine learning.

The phrase **machine learning** describes a growing body of techniques that all have one goal: discover meaningful information from data.

Here, “data” refers to anything that can be recorded and measured. Data can be raw numbers (like stock prices on successive days, or the mass 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), or anything else that we want to investigate.

“Meaningful information” is whatever we can extract from the data that will be useful to us in some way. We get to decide what’s meaningful to us, and then we design an algorithm to find as much of it as possible from our data.

The phrase “machine learning” describes a wide diversity of algorithms and techniques. It would be nice to nail down a specific definition for the phrase, but it’s used by so many people in so many different ways that it’s best to consider it the name for a big, expanding collection of algorithms and principles that analyze vast quantities of training data in order to extract meaning from it.

More recently, the phrase **deep learning** was coined to refer to approaches to machine learning that use specialized layers of computation, stacked up one after the next. This makes a “deep” structure, like a stack of pancakes. Since “deep learning” refers to the nature of the system we create, rather than any particular algorithm, it really refers to this particular style or approach to machine learning. It’s an approach that has paid off enormously well in recent years.

Let's look at some example applications that use machine learning to extract meaning from data.



1.1.1 Extracting Meaning from Data

The post office needs to sort an enormous number of letters and packages every day, based on hand-written zip codes. They have taught computers to read those codes and route the mail automatically, as in the left illustration in Figure 1.1.

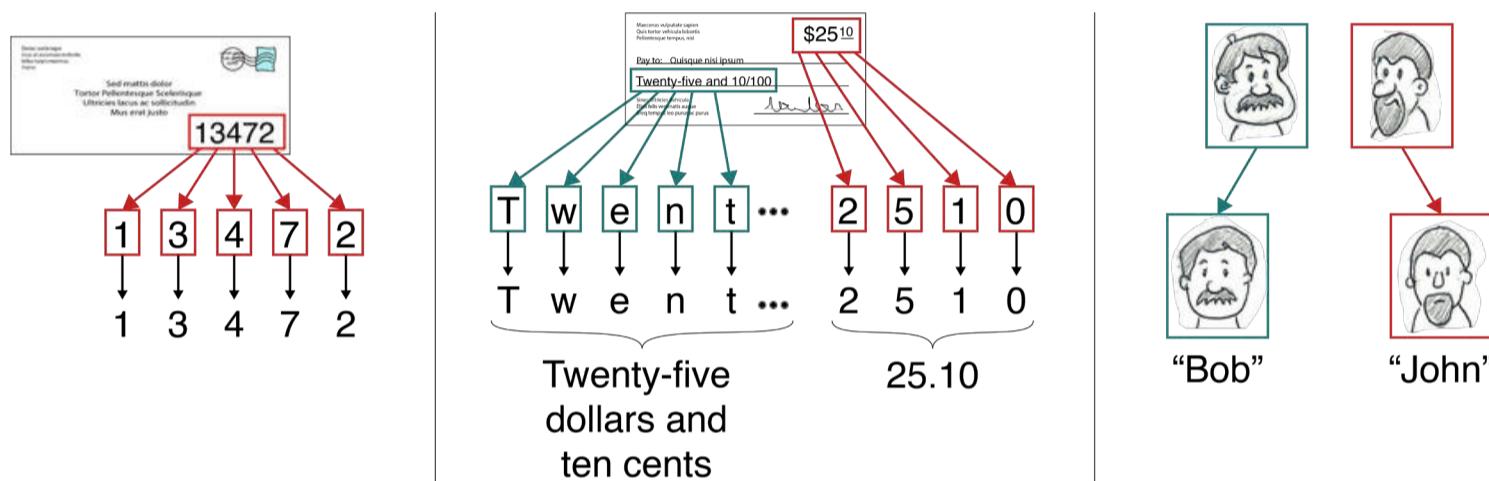


Figure 1.1: Extracting meaning from data sets. Left: Getting a zip code from an envelope. Middle: Reading the numbers and letters on a check. Right: Recognizing faces from photos.

Banks process vast quantities of hand-written checks. A valid check requires that the numerical amount hand-written into the total field (e.g., "\$25.10") matches the written-out amount on the text line (e.g., "twenty-five dollars and ten cents"). Computers can read both the numbers and words and confirm a match, as in the middle of Figure 1.1.

Social media sites want to identify the people in their member's photographs. This means not only detecting if there are any faces in a given photo, but identifying where those faces are located, and then matching up each face with previously seen faces. This is made even more difficult when we realize that virtually every photo of a person is unique: the lighting, angle, expression, clothing, and many other qualities will

be different from any previous photo. They'd like to be able to take any photo of any person, and correctly identify who it is, as in the right of Figure 1.1.

Digital assistant providers listen to what people say into their gadgets so that they can respond intelligently. The signal from the microphone is a series of numbers that describe the sound pressure striking the microphone's membrane. The providers want to analyze those numbers in order to understand the sounds that caused them, the words that these sounds were part of, the sentences that the words were part of, and ultimately the meaning of those sentences, as suggested by the left illustration in Figure 1.2.

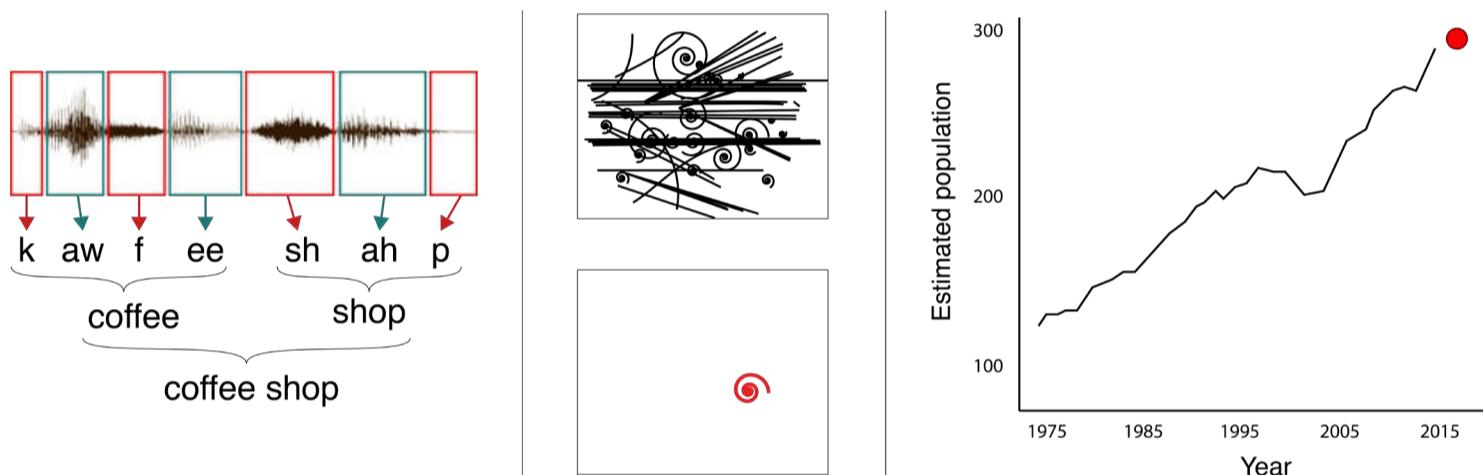


Figure 1.2: Extracting meaning from data. Left: Turning a recording into sounds, then words, and ultimately a complete utterance. Middle: Finding one unusual event in a particle smasher's output full of similar-looking trails. Right: Predicting the population of the northern resident Orca whale population off of Canada's west coast [Towers15].

Scientists create vast amounts of data from flying drones, high-energy physics experiments, and observations of deep space. From these overwhelming floods of data they often need to pick out the few instances of events that are almost like all the others, but are slightly different. Looking through all the data manually would be an effectively impossible task even for huge numbers of highly-trained specialists. It's much better to automate this process with computers that can comb the data exhaustively, never missing any details, as in the middle of Figure 1.2.

Conservationists track the populations of species over time to see how well they're doing. If the population declines over a long period of time, they may need to take action to intervene. If the population is stable, or growing, then they may be able to rest more easily. Predicting the next value of a sequence of values is something that we can train a computer to do well. The recorded annual size of a population of Orca whales off the Canadian coast, along with a possible predicted value, is shown in the right of Figure 1.2 (adapted from [Towers15]).

These six examples illustrate applications that many of us are already familiar with, but there are many more. Because of their ability to extract meaningful information quickly, machine learning algorithms are finding their way into an expanding range of fields.

The common threads here are the sheer volume of the work involved, and its painstaking detail. We might have millions of pieces of data to examine, and we're trying to extract some meaning from every one of them. Humans get tired, bored, and distracted, but computers just plow on steadily and reliably.

1.1.2 Expert Systems

A popular, early approach to finding the meaning that's hiding inside of data involved creating **expert systems**. The essence of the idea was that we would study what human experts know, what they do, and how they do it, and automate that. In essence, we'd make a computer system that could mimic the human experts it was based on.

This often meant creating a **rule-based system**, where we'd amass a large number of rules for the computer to follow in order to imitate the human expert. For instance, if we're trying to recognize digits in zip codes, we might have a rule that says that 7's are shapes that have a mostly horizontal line near the top of the figure, and then a mostly diagonal line that starts at the right edge of the horizontal line and moves left and down, as in Figure 1.3.

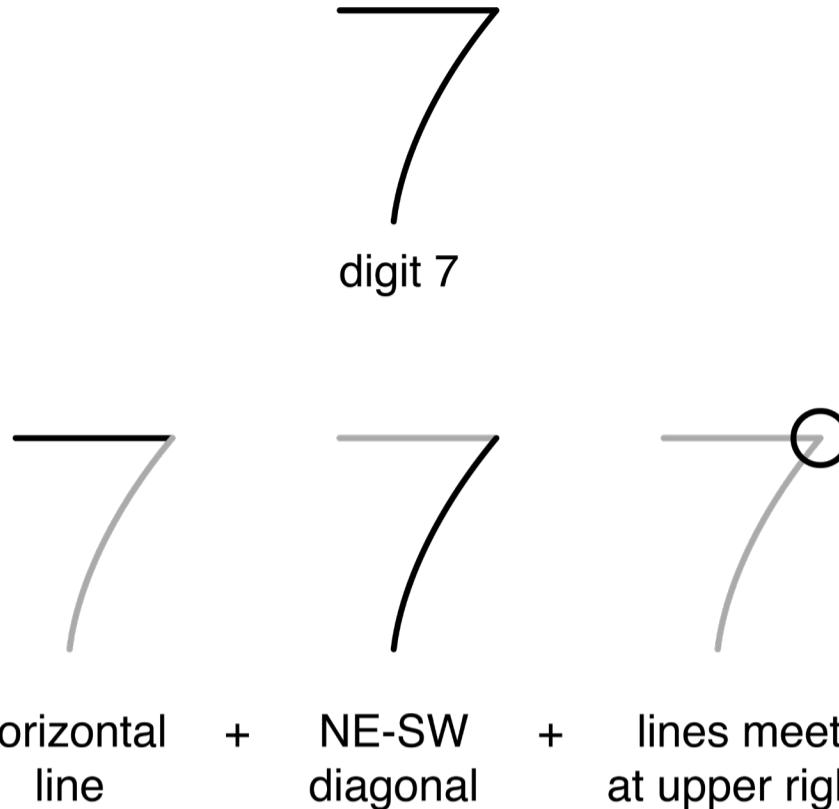


Figure 1.3: Devising a set of rules to recognize a hand-written digit 7. Top: A typical 7 we'd like to identify. Bottom: The three rules that make up a 7. A shape would be classified as a 7 if it satisfies all three rules.

We'd have similar rules for every digit. This might work well enough until we get a digit like Figure 1.4.

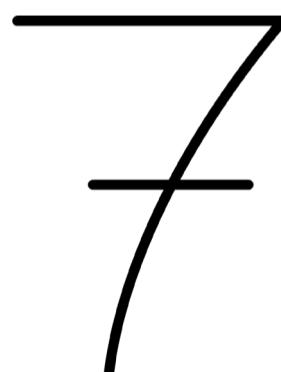


Figure 1.4: This 7 is a valid way to write a 7, but it would not be recognized by the rules of Figure 1.3 because of the extra line.

We hadn't thought about how some people put a bar through the middle of their 7's. So now we add another rule for that special case.

This process of hand-crafting the rules to understand data is sometimes called **feature engineering** (the term is also used to describe when we use the computer to find these features for us [VanderPlas16]).

The phrase describes our desire to engineer (or design) all of the features (or qualities) that a human expert uses and combines to do their job. In general, it's a very tough job. As we saw, it's easy to overlook one rule, or even lots of them. Imagine trying to find a set of rules that summarize how a radiologist determines whether a smudge on an X-ray image is benign or not, or how an air-traffic controller handles heavily scheduled air traffic, or how someone drives a car safely in extreme weather conditions.

Rule-based expert systems are able to manage some jobs, but the difficulty of manually crafting the right set of rules, and making sure they work properly across a wide variety of data, has spelled their doom as a general solution. Articulating every step in a complicated process is extremely difficult, and when we have to factor in human judgment based on experience and hunches, it becomes nearly impossible for any but the simplest scenarios.

The beauty of machine learning systems is that (on a conceptual level) they learn a dataset's relevant characteristics *automatically*. So we don't have to tell an algorithm how to recognize a 2 or a 7, because the system figures that out for itself. But to do that well, the system often needs a *lot* of data. Enormous amounts of data.

That's a big reason why machine learning has exploded in popularity and applications in the last few years. The flood of raw data provided by the Internet has let these tools extract a lot of meaning from a lot of data. Online companies are able to make use of every interaction with every customer to accumulate more data, which they can then turn around and use as input to their machine learning algorithms, providing them with even more information about their customers.

1.2 Learning from Labeled Data

There are lots of machine learning algorithms, and we'll look at many of them in this book. Many are conceptually straightforward (though their underlying math or programming could be complex). For instance, suppose we want to find the best straight line through a bunch of data points, as in Figure 1.5.

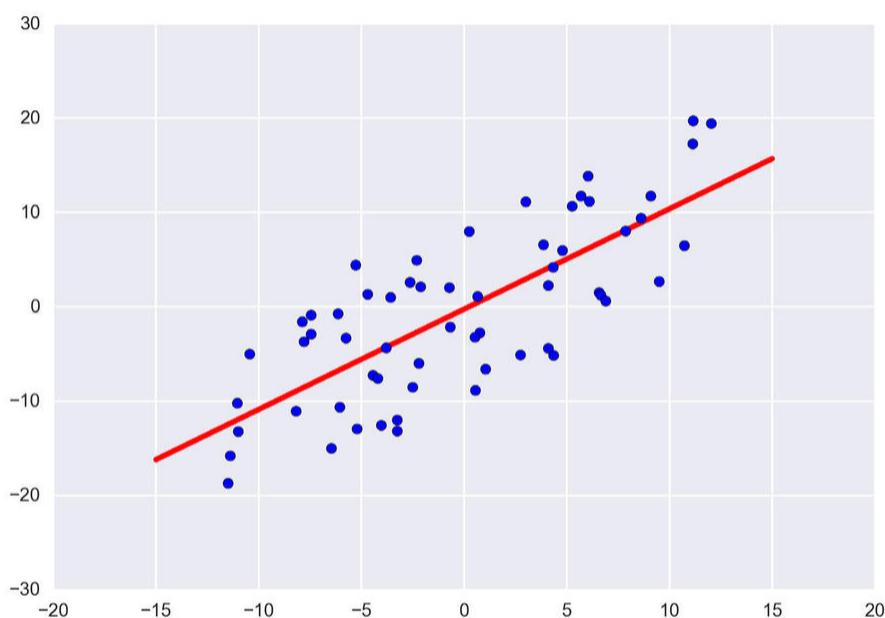


Figure 1.5: Given a set of data points (in blue), we can imagine a straightforward algorithm that computes the best straight line (in red) through those points.

Conceptually, we can imagine an algorithm that represents any straight line with just a few numbers. It then uses some formulas to compute those numbers, given the numbers that represent the input data points. This is a familiar kind of algorithm, which uses a carefully thought-out analysis to find the best way to solve a problem, and is then implemented in a program that performs that analysis. This is a strategy used by many machine learning algorithms.

By contrast, the strategy used by many deep learning algorithms is less familiar. It involves slowly learning from examples, a little bit at a time, over and over. Each time the program sees a new piece of data to be learned from, it improves its own parameters, ultimately finding a set of values that will do a good job of computing what we want. While

we're still carrying out an algorithm, it's much more open-ended than the one that fits a straight line. The idea here is that we don't know how to directly calculate the right answer, so we build a system that can figure out how to do that itself. Our analysis and programming is to create an algorithm that can work out its own answers, rather than implementing a known process that directly yields an answer.

If that sounds pretty wild, it is. Programs that can find their own answers in this way are at the heart of the recent enormous success of deep learning algorithms.

In the next few sections, we'll look more closely at this technique to get a feeling for it, since it's probably less familiar than the more traditional machine learning algorithms. Ultimately we'd like to be able to use this for tasks like showing the system a photograph, and getting back the names of everyone in the picture.

That's a tall order, so let's start with something much simpler and look at learning a few facts.

1.2.1 A Learning Strategy

Let's consider an awful way to teach a new subject to children, summarized in Figure 1.6. This is not how most children are actually taught, but it is one of the ways that we teach computers.

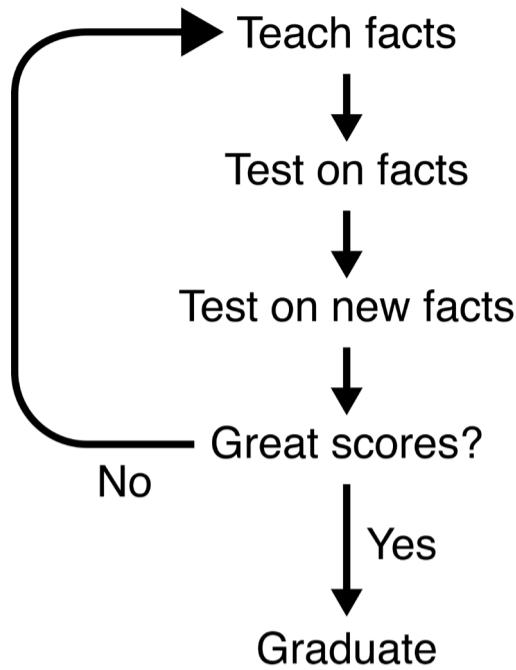


Figure 1.6: A truly terrible way to try to teach people. First, recite a list of facts. Then test each student on those facts, and on other facts they haven't been exposed to, but which we believe could be derived if the first set was well-enough understood. If the student gets great scores on the tests (particularly the second one), he or she graduates. Otherwise they go through the loop again, starting with a repeated recitation of the very same facts.

In this scenario (hopefully imaginary), a teacher stands in front of a class and recites a series of facts that the students are supposed to memorize. Every Friday afternoon they are given two tests. The first test grills them on those specific facts, to test their retention. The second test, given immediately after the first, asks new questions the students have never seen before, in order to test their overall understanding of the material. Of course, it's very unlikely that anyone would "understand" anything if they've only been given a list of facts, which is one reason this approach would be terrible.

If a student does well on the second test, the teacher declares that they've learned the subject matter and they immediately graduate.

If a given student doesn't do well on the second test, they repeat the same process again the next week: the teacher recites the very same facts as before in exactly the same way, then gives the students the same first test to measure their retention, and a new second test to

measure their understanding, or ability to generalize. Over and over every student repeats this process until they do well enough on the second test to graduate.

This would be a terrible way to teach children, but it turns out to be a great way to teach computers.

In this book we'll see many other approaches to teaching computers, but let's stick with this one for now, and look at it more closely. We'll see that, unlike most people, each time we expose the computer to the identical information, it learns a little bit more.

1.2.2 A Computerized Learning Strategy

We start by collecting the facts we're going to teach. We do this by collecting as much data as we can get. Each piece of observed data (say, the weather at a given moment) is called a **sample**, and the names of the measurements that make it up (the temperature, wind speed, humidity, etc.) are called its **features** [Bishop06]. Each named measurement, or feature, has an associated value, typically stored as a number.

To prepare our data for the computer, we hand each sample (that is, each piece of data, with a value for each feature) to a human expert, who examines its features and provides a **label** for that sample. For instance, if our sample is a photo, the label might be the name of the person in the photo, or the type of animal it shows, or whether or not the traffic in the photo is flowing smoothly or is stuck in gridlock.

Let's use weather measurements on a mountain for our example. The expert's opinion, using a score from 0 to 100, tells how confident the expert is that the day's weather would make for good hiking. The idea is shown in Figure 1.7.

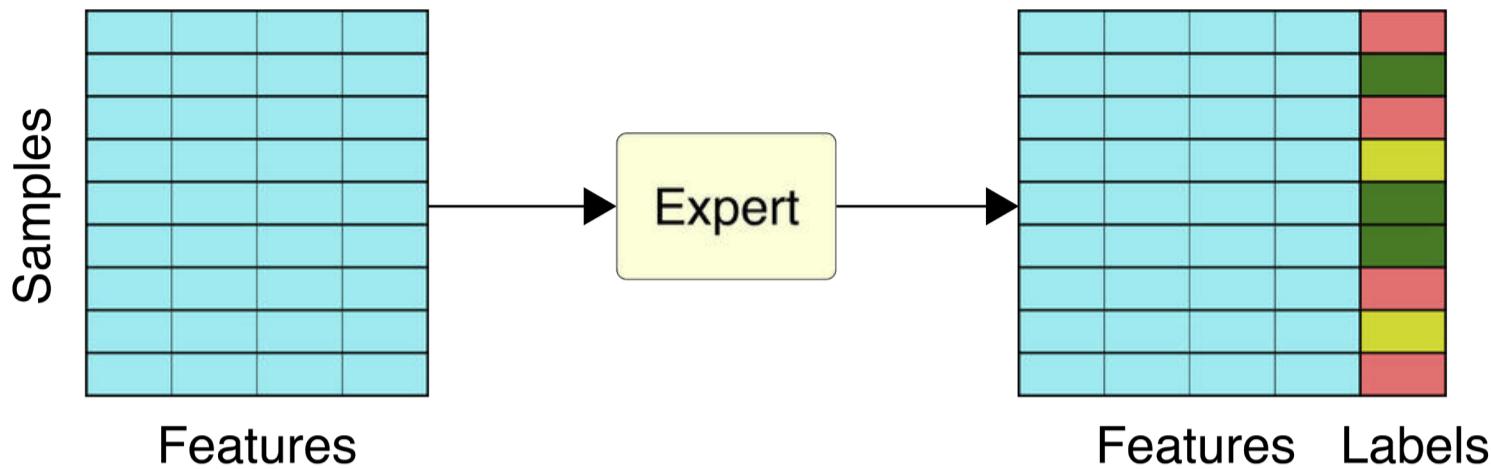


Figure 1.7: To label a dataset, we start with a list of samples, or data items. Each sample is made up of a list of features that describe it. We give the dataset to a human expert, who examines the features of each sample one by one, and assigns a label for that sample.

We'll usually take some of these labeled samples and set them aside for a moment. We'll return to them soon.

Once we have our labeled data, we give it to our computer, and we tell it to find a way to come up with the right label for each input. We do *not* tell it how to do this. Instead, we give it an algorithm with a large number of parameters it can adjust (perhaps even millions of them). Different types of learning will use different algorithms, and much of this book will be devoted to looking at them and how to use them well. But once we've selected an algorithm, we can run an input through it to produce an output, which is the computer's **prediction** of what it thinks the expert's label is for that sample.

When the computer's prediction matches the expert's label, we won't change anything. But when the computer gets it wrong, we ask the computer to modify the internal parameters for the algorithm it's using, so that it's more likely to predict the right answer if we show it this piece of data again.

The process is basically trial and error. The computer does its best to give us the right answer, and when it fails, there's a procedure it can follow to help it change and improve.

We only check the computer's prediction against the expert's label once the prediction has been made. When they don't match, we calculate an **error**, also called a **cost**, or **loss**. This is a number that tells the algorithm how far off it was. The system uses the current values of its internal parameters, the expert's prediction (which it now knows), and its own incorrect prediction, to adjust the parameters in the algorithm so that it's more likely to predict the correct label if it sees this sample again. Later on we'll look closely at how these steps are performed. Figure 1.8 shows the idea.

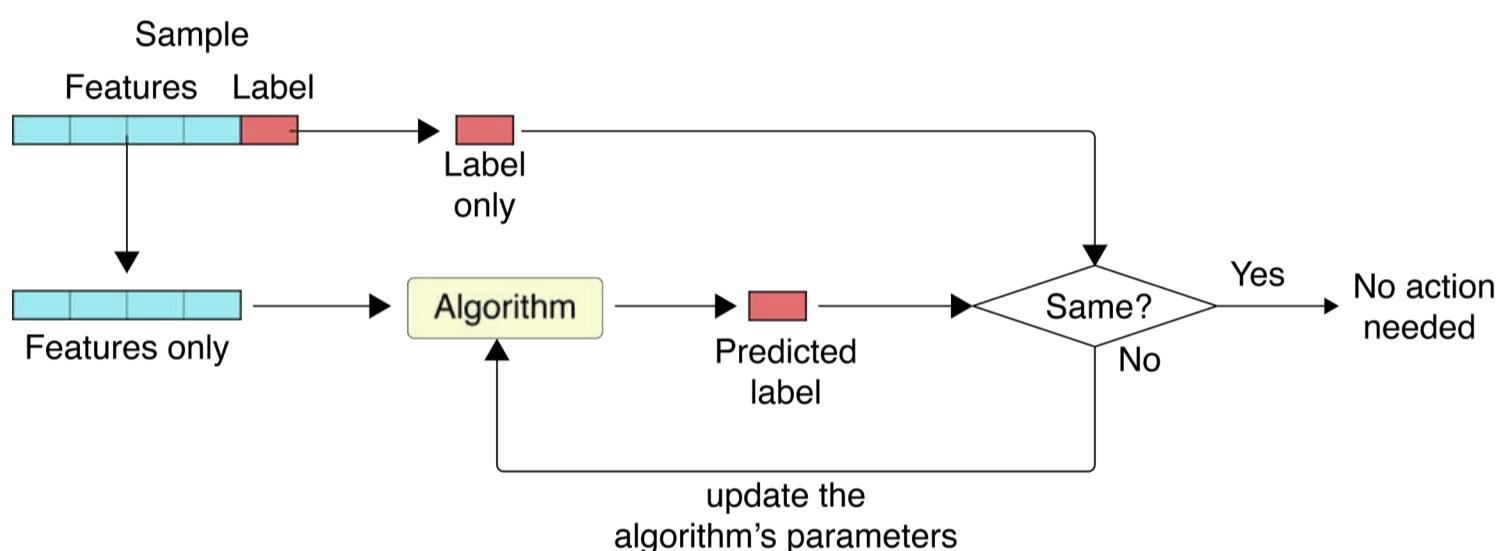


Figure 1.8: One step of training, or learning. We split the sample's features and its label. From the features, the algorithm predicts a label. We compare the prediction with the real label. If the predicted label matches the label we want, we don't do a thing. Otherwise, we tell the algorithm to modify, or update, itself so it's less likely to make this mistake again.

We say that we **train** the system to **learn** how to **predict** label data by analyzing the samples in the **training set** and **updating** its algorithm in response to incorrect predictions.

We'll get into different choices for the algorithm and update step in detail in later chapters. For now, it's worth knowing that each algorithm learns by changing the internal parameters it uses to create its predictions. It can change them by a lot after each incorrect sample, but then it runs the risk of changing them so much that it makes other predictions worse. It could change them by a small amount, but that could cause learning to run slower than it otherwise might. Finding

the right trade-off between these extremes is something we have to find by trial and error for each type of algorithm and each dataset we're training it on. We call the amount of updating the **learning rate**, so a small learning rate is cautious and slow, while a large learning rate speeds things up but could backfire.

As an analogy, suppose we're out in a desert and using a metal detector to find a buried metal box full of supplies. We'd wave the metal detector around, and if we got a response in some direction, we'd move in that direction. If we're being careful, we'd take a small step so that we don't either walk right past the box or lose the signal. If we're being aggressive, we'd take a big step so we can get to the box as soon as possible. Just as we'd probably start out with big steps but then take smaller and smaller ones the nearer we got to the box, so too we usually adjust the learning rate so that the network changes a lot during the start of training, but we reduce the size of those changes as we go.

There's an interesting way for the computer to get a great score without learning how to do anything but remember the inputs. To get a perfect score, all the algorithm has to do is memorize the expert's label for each sample, and then return that label. In other words, it doesn't need to learn how to compute a label given the sample, it only needs to look up the right answer in a table. In our imaginary learning scenario above, this is equivalent to the students memorizing the answers to the questions in the tests.

Sometimes this is a great strategy, and we'll see later that there are useful algorithms that follow just this approach. But if our goal is to get the computer to learn something about the data that will enable it to generalize to new data, this method will usually backfire. The problem is that we already have the labels that the system is memorizing, so all of our work in training isn't getting us anything new. And since the computer has learned nothing about the data itself, instead just getting its answers from a look-up table, it would have no idea how to create predictions for new data that it hasn't already seen and

memorized. The whole point is to get the system to be able to predict labels for new data it's never seen before, so that we can confidently use it in a deployed system where new data will be arriving all the time.

If the algorithm does well on the training set, but poorly on new data, we say it **doesn't generalize well**. Let's see how to encourage the computer to generalize, or to learn about the data so that it can accurately predict the labels for new data.

1.2.3 Generalization

Now we'll return to the labeled data that we set aside in the last section.

We'll evaluate how well the system can **generalize** what its learned by showing it these samples that it's never seen before. This **test set** shows us how well the system does on new data.

Let's look at a **classifier** (or **categorizer**). This kind of system assigns a label to each sample that describes which of several categories, or classes, that sample belongs to. If the input is a song, the label might be the genre (e.g., rock or classical). If it's a photo of an animal, the label might be which animal is shown (e.g., a tiger or an elephant). In our running example, we could break up each day's anticipated hiking experience into 3 categories: Lousy, Good, and Great.

We'll ask the computer to predict the label for each sample in the test set (the samples it has not seen before), and then we'll compare the computer's predictions with the expert's labels, as in Figure 1.9.

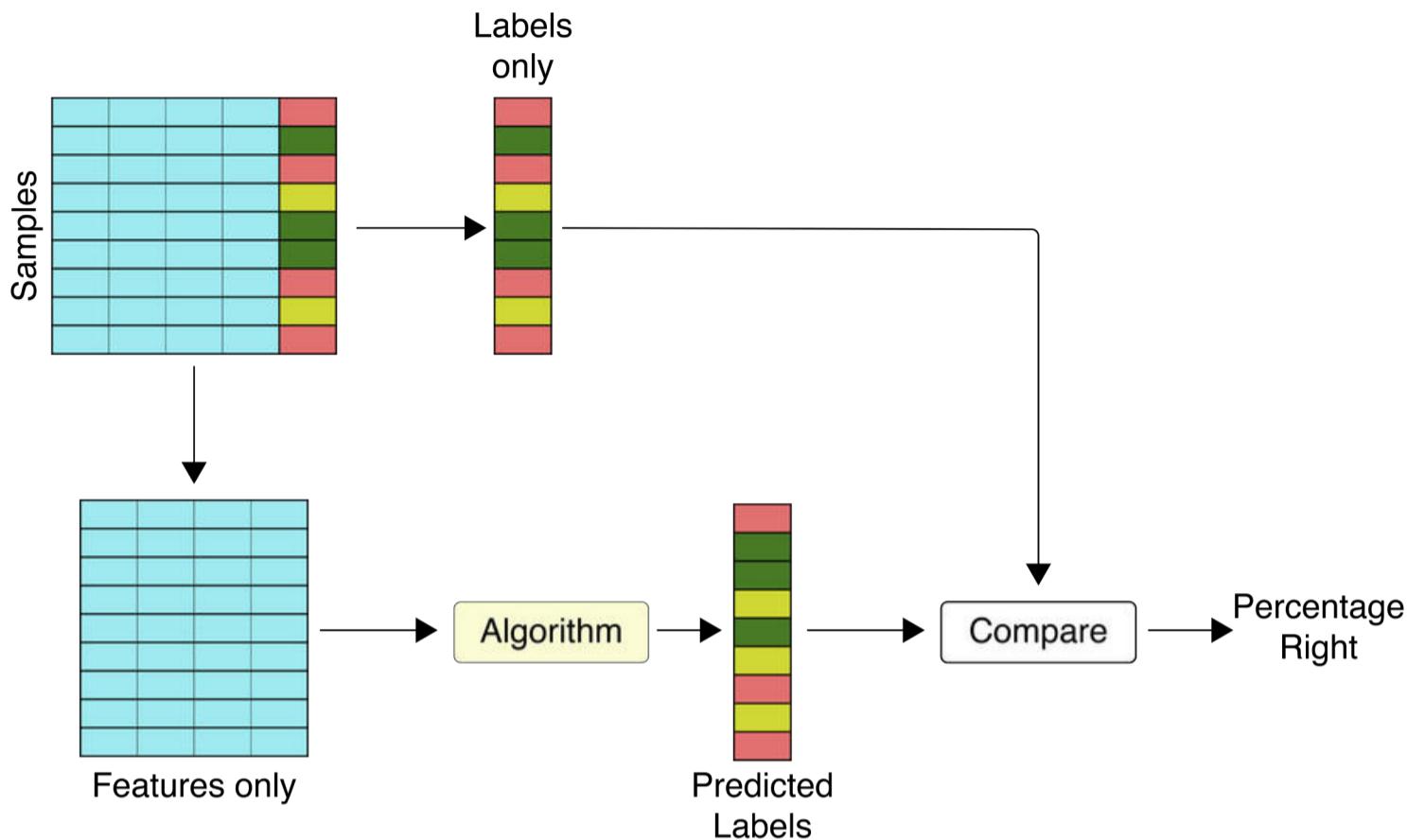


Figure 1.9: The overall process for evaluating a classifier (also called a categorizer).

In Figure 1.9, we've split the test data into features and labels. The algorithm assigns, or predicts, a label for each set of features. We then compare the predictions with the real labels to get a measurement of accuracy. If it's good enough, we can deploy the system. If the results aren't good enough, we can go back and train some more. Note that unlike training, in this process there is no feedback and no learning. Until we return to explicit training, the algorithm doesn't change its parameters, regardless of the quality of its predictions.

If the computer's predictions on these brand-new samples (well, brand-new to the algorithm) are not a sufficiently close match to the expert's labels, then we return to the training step in Figure 1.8. We show the computer every sample in the original training set again, letting it learn along the way again. Note that these are the same samples, so we're asking the computer to learn over and over again from the very same data. We usually **shuffle** the data first so that each sample arrives in a different order each time, but we're not giving the algorithm any new information.

Then we ask the algorithm to predict labels for the test set again. If the performance isn't good enough, we learn from the original training set again, and then test again. Around and around we go, repeating this process often hundreds of times, showing the computer the same data over and over and over again, letting it learn just a little more each time.

As we noted, this would be a terrible way to teach a student, but the computer doesn't get bored or cranky seeing the same data over and over. It just learns what it can and gets a little bit better each time it gets another shot at learning from the data.

1.2.4 A Closer Look at Learning

We usually believe that there are relationships in our data. After all, if it was purely random we wouldn't be trying to extract information from it. The hope of our process in the previous section is that by exposing the computer to the training set over and over, and having it learn a little bit from every sample every time, the algorithm will eventually find these relationships between the features in each sample and the label the expert assigned. Then it can apply that relationship to the new data in the test set. If it gets mostly correct answers, we say that it has high accuracy, or low **generalization error**.

But if the computer consistently fails to improve its predictions for the labels for the test set, we'll stop training, since we're not making any progress. At that point we'll typically modify our algorithm in hopes of getting better performance, and then start the training process over again. But we're just hoping. There's no guarantee that there's a successful learning algorithm for every set of data, and no guarantee that if there is one, we'll find it. The good news is that even without a mathematical guarantee, in practice we can often find solutions that generalize very well, sometimes doing even better than human experts.

One reason the algorithm might fail to learn is because it doesn't have enough computational resources to find the relationship between the samples and their labels. We sometimes think of the computer creating a kind of **model** of the underlying data. For instance, if the temperature goes up for the first three hours every day we measure it, the computer might build a "model" that says morning temperatures rise. This is a version of the data, in the same way a small plastic version of a sports car or plane is a "model" of that larger vehicle. The classifier we saw above is one example of a model.

In the computer, the model is formed by the structure of the software and the values of the parameters it's using. Larger programs, and larger sets of parameters, can lead to models that are able to learn more from the data. We say that they have more **capacity, or representational power**. We can think of this as how deeply and broadly the algorithm is able to learn. More capacity gives us more ability to discover meaning from the data we're given.

As an analogy, suppose we work for a car dealer and we're supposed to write blurbs for the cars we're selling. We'll suppose that our marketing department has sent us a list of "approved words" for describing our cars. After doing our job for a while, we will probably learn how to best represent each car with this model.

Suppose the dealer then buys its first motorcycle. The words we have available, or our model, don't have enough capacity to represent this vehicle, in addition to all the other vehicles we're already describing. We just don't have a big enough vocabulary to refer to something with 2 wheels rather than 4. We can do our best, but it's likely not to be great. If we can use a more powerful model with greater capacity (that is, a bigger vocabulary for describing vehicles), we can do a better job.

But a bigger model also means more work. As we'll see later in the book, a bigger model can often produce better results than a smaller one, though at the cost of more computation time and computer memory.

The values that the algorithm modifies itself over time are called its **parameters**. But our learning algorithms are also controlled by values that we set (such as the learning rate we saw above). These go by the rather awkward name of **hyperparameters**.

The distinction between parameters and hyperparameters is that the computer adjusts its own parameter values during the learning process, while we specify the hyperparameters when we write and run our program.

When the algorithm has learned enough to perform well enough on the test set that we're satisfied, we're ready to **deploy**, or **release**, our algorithm to the world. Our users submit data and our system returns the label it predicts.

That's how pictures of faces are turned into names, sounds are turned into words, and weather measurements are turned into forecasts.

Let's now back up and get a big picture for the field of machine learning.

A survey of the vast and growing field of machine learning would take a whole book of its own [Bishop06] [Goodfellow17]. In this section we'll review the major categories that make up the majority of today's tools. We'll be returning to these types of algorithms repeatedly throughout the book.

1.3 Supervised Learning

When our samples come with pre-assigned labels, as in the examples we saw above, we say we're doing **supervised learning**. The supervision comes from our labels. They guide the comparison step of Figure 1.8, where the algorithm is told whether it predicted the correct label or not.

There are two general types of supervised learning, called **classification** and **regression**. Classification is concerned with looking through a given collection of categories to find the one that best describes a particular input. Regression is concerned with taking a set of measurements and predicting some other value (often the one that comes next, but it could also be before the set starts or somewhere in the middle).

Let's look at these in turn.

1.3.1 Classification

Suppose we have a collection of photos of everyday objects and we want to sort them by what they show: an apple peeler, a salamander, a piano, and so on. We say that we want to **classify** or **categorize** these photos. The process is called **classification** or **categorization**.

In this approach, we start training by providing the computer with a list of all the labels (or classes, or categories) that we want it to learn. This list typically is just the combination of all the labels for all the samples in the training set, with the duplicates removed.

We then train the system with lots of photos and their labels, until we determine that it's doing a good enough job of predicting the correct label for each photo.

Now we can turn the system loose on new photos it hasn't seen before. We expect it to properly label images of those objects it saw during training. But if the shape is not recognizable, or it belongs to a category that was not part of the training set, the system will try to pick the best category from those it knows about. Figure 1.10 shows the idea.

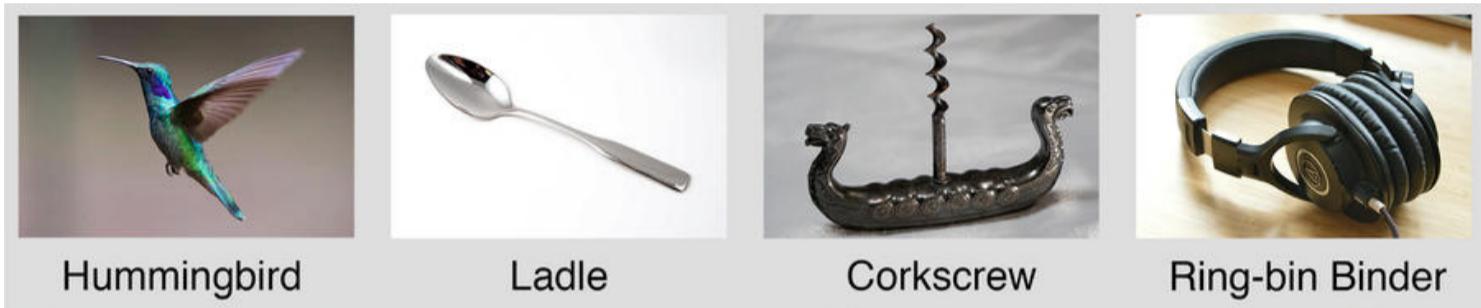


Figure 1.10: When we perform classification, we train a classifier with a set of images, each with an associated label. When training is done we can give it new images, and it will try to choose the best label for each image. This classifier had not been trained on a metal spoon or headphones, so it reported the closest match it could find.

In Figure 1.10, we used a trained classifier to identify four images it had never seen before [Simonyan14]. Remarkably, it identified the corkscrew even though the object was deliberately shaped to look like a boat. The system had not been trained on metal spoons or headphones, so in both cases it found the best match it could. To correctly identify those objects, the system would have needed to see multiple examples of them during training.

Another way to look at it is to say that the system can only understand what it has learned. Traditional classifiers will always do their best to find the closest match for every input, but they can only use the categories that they know about.

1.3.2 Regression

Suppose we have an incomplete collection of measurements, and we'd like to estimate the missing values. Perhaps we've been measuring the attendance at a series of concerts at a local arena. The band gets paid each night, receiving a fraction of the total ticket sales for that concert.

Unfortunately, we've lost the count for one evening's performance. And in order to plan our budget, we'd like to know what tomorrow's attendance is likely to be. The measurements we do have are shown in the left illustration of Figure 1.11. Our estimates for the missing values are shown on the right of Figure 1.11.

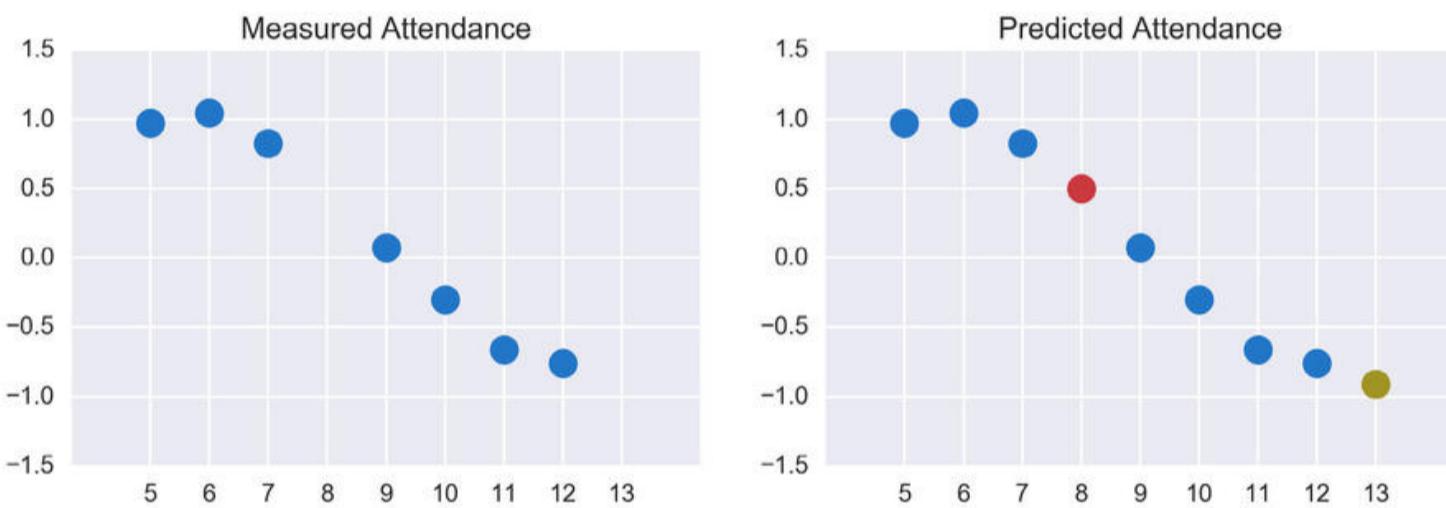


Figure 1.11: In regression, we work with sets of input and output values. Here, the input is the date of a concert from May 5 to May 13, and the output is the attendance. Left: Our measured data, which is missing a value for May 8th. Right: The red dot is the estimated value of the missing point at May 8, and the yellow dot is a prediction of the attendance on May 13.

We call this process of filling in and predicting data a **regression** problem. The name can be a bit misleading, because “regress” means to return to an earlier condition, and there doesn't seem to be any kind of backwards motion going on.

The unfamiliar usage of this word comes from a paper published in 1886, where a scientist studied the heights of children [Galton86]. He found that while some children are tall and others are short, over time people will tend to have a common height. He described this as a “regression towards mediocrity,” meaning that the measurements tended to move away from the extremes and towards an average value.

Though the phrase is usually attributed to Galton, a very similar remark was published almost 20 years earlier in an unflattering review of Darwin's *On the Origin of Species* [Darwin59]. The reviewer, named

Fleeming Jenkin, argued that variations in species would be “swamped” into oblivion by “the general impulse to bring everything back to a stable mediocrity” [Brysono4].

Today the word “mediocrity” carries some negative connotations, so the idea is now often referred to as “regression to the mean,” where “mean” is a type of average. The word “regression” is still used to convey the idea of using statistical properties of the data to estimate missing or future values.

So a “regression” problem is just one where we have a value that depends on our inputs (like the attendance as a function of the day of the month), and we predict a new value for a new input.

The most famous kind of regression is **linear regression**. The name “linear” refers to how this technique tries to match our input data with a straight line, as in Figure 1.12.

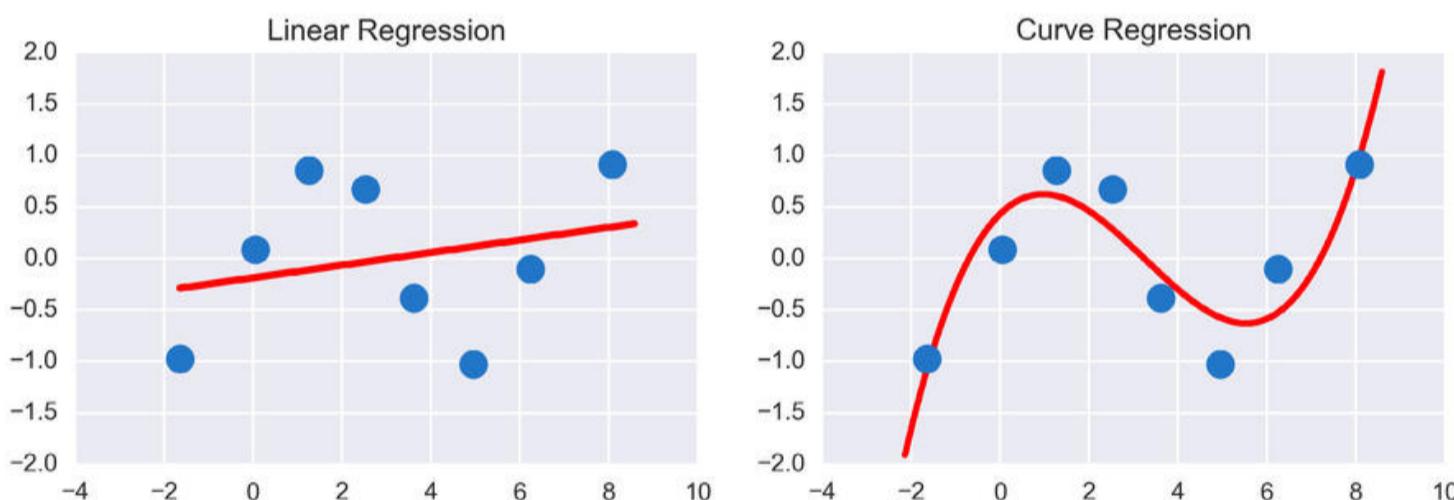


Figure 1.12: Representing points of data with mathematical shapes. Left: Linear regression fits a straight line (red) to the data (blue). The line is not a very good match to the data, but it has the benefit of being simple. Right: A more complex regression fits a curve to the same data. This is a better match to the data, but it has a more complicated form and requires more work (and thus more time) when part of a calculation.

The straight line is appealing because it's simple, but in this example it's a pretty lousy description of the data. Our data has an up and down wiggle to it that the straight line fails to capture. It's not the worst match in the world, but it's not great.

Alternatively, we can use more complex forms of regression to create more complicated types of curves, as in the right of Figure 1.12. These can provide a better fit to the data, at the cost of more computation time. As our curves become more complex, they usually require more data to work with.

1.4 Unsupervised Learning

When our input data does not come with labels, we say any algorithm that learns from the data comes from the category of **unsupervised learning**. This just means that we are not “supervising” the learning process by offering labels. The system has to figure everything out on its own, with no help from us.

Unsupervised learning is often used for solving problems that we call **clustering**, **noise reduction**, and **dimension reduction**. Let’s look at these in turn.

1.4.1 Clustering

Suppose that we’re digging out the foundation for a new house, and to our surprise we find the ground is filled with old clay pots and vases. We call an archaeologist, who realizes that we’ve found a jumbled collection of ancient pottery, apparently from many different places and maybe different times.

The archaeologist doesn’t recognize any of the markings and decorations, so she can’t declare for sure where each one came from. Some of the marks look like variations on the same theme, while others 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 of them for her to manage. 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 1.13 shows her captured marks, and the groupings that could be found automatically by an algorithm. We call this a **clustering** problem, and we say that the algorithm is a **clustering algorithm**. There are many such algorithms to choose from, and we'll see a variety of them later. Using the terminology we just saw, because our inputs are unlabeled, this archaeologist is performing clustering, using an unsupervised learning algorithm.

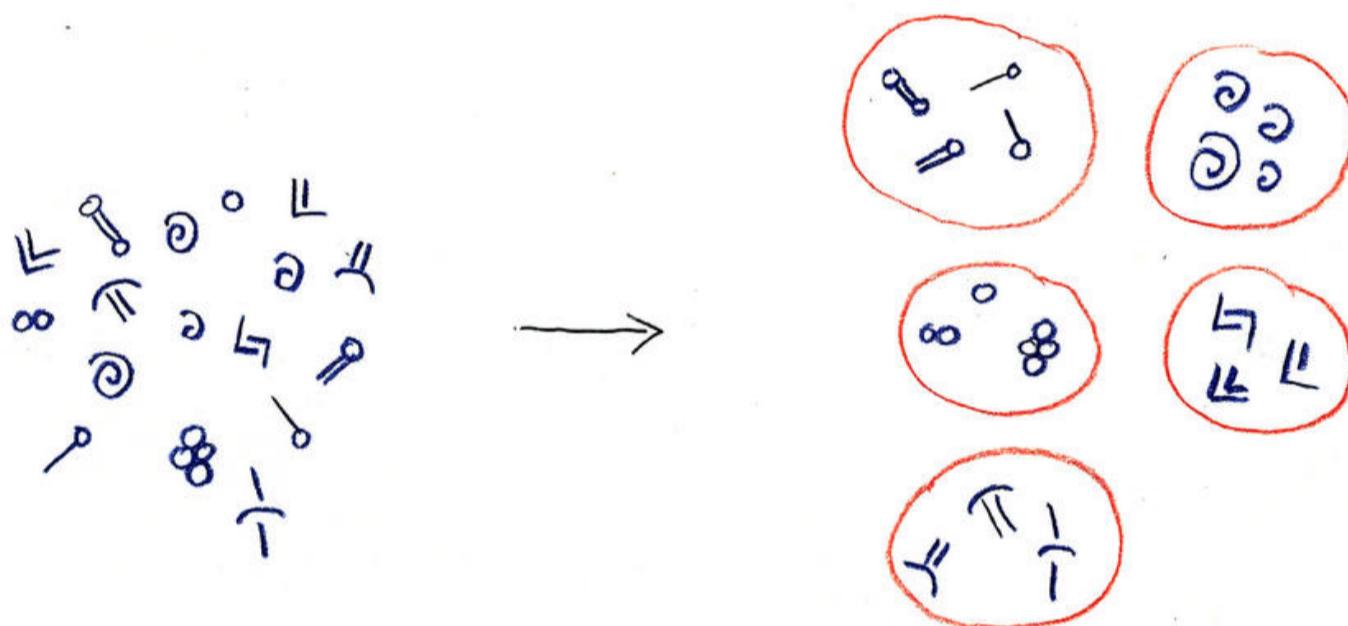


Figure 1.13: Using a clustering algorithm to organize marks on clay pots. Left: The markings from the pots. Right: The marks grouped into similar clusters.

1.4.2 Noise Reduction

Sometimes samples are **corrupted** by **noise**. If the sample is an audio clip of someone speaking into their phone, the noise can be the background noise of the street, making it hard to make out the person's words.

If the sample is an electronic signal, perhaps a radio or TV program, noise can intrude in the form of static, causing distracting sounds in the radio show or flickering spots in the TV image. In synthetic, or computer-generated images, we can save time by leaving out some of pixels, so they appear black. To the computer, this is a kind of “noise,” though it’s actually missing data rather than corrupted data.

Our eyes are sensitive to noisy and missing pixels, but the data often has more information available than we might guess by looking at it. Figure 1.14 shows an example of a noisy image, and how a de-noising algorithm might clean it up.



Figure 1.14: De-noising, or noise reduction, reduces the effects of random distortions to samples, and can even fill in some missing gaps. Left: An image of a cow degraded with a lot of noisy pixels, and some missing pixels. Right: The original image. De-noising algorithms can recover this image with different degrees of success.

This doesn’t just apply to pictures. Using data from almost any source, samples can carry inaccuracies or other distortions that can obscure the information we want to extract from them.

If there are missing values, we can turn to a regression algorithm to fill them in. Or we can treat them as another form of noise, and let the de-noising algorithm try to fill them in for us.

Because we don't have labels for our data (for example, in a noisy photo we just have pixels), de-noising is a form of unsupervised learning. By learning the statistics of the samples, the algorithm estimates what part of each sample is noise and then removes it, leaving us with cleaned-up data that's easier to learn from and interpret.

We often apply de-noising algorithms to data before we learn from it. By removing weird and missing values from our input, the learning process usually goes more quickly and smoothly.

1.4.3 Dimensionality Reduction

Sometimes our samples have more features than they need. For instance, we might be taking weather samples in the desert at the height of summer. Every day we record wind speed, wind direction, and rainfall. Given the season and locale, the rainfall value will be 0 in every sample. If we use these samples in a machine learning system, the computer will need to process and interpret this useless, constant piece of information with every sample. At best this would slow down the analysis. At worst it could affect the system's accuracy, because the computer would devote some of its finite resources of time and memory to trying to learn from this unchanging feature.

Sometimes features contain redundant data. For instance, a health clinic might take everyone's weight in kilograms when they walk into the door. Then when a nurse takes them to an examination room, she measures their weight again but this time in pounds, and places that into the chart as well. Now we have the same information repeated twice, but it might be hard to recognize that because the values are different. Like the useless rainfall measurements, this redundancy will not work to our benefit when we try to learn from this data.

A more abstract example involves data that seems more complicated than it needs to be. Suppose that a national forest has been suffering from an unknown infection that is hurting the trees. The forest service has tried to work out the spread of the infection inside a fallen tree by

putting a section of its trunk on a machine that turns the wood and shaves it away, measuring the wood's density as it goes. It might produce a 3D data set like that in the left of Figure 1.15, where each point is located in 3D, and has a value telling us how much of the infection is present. To represent this data, we need 4 numbers: 3 for the point's location in space, and 1 for its value.

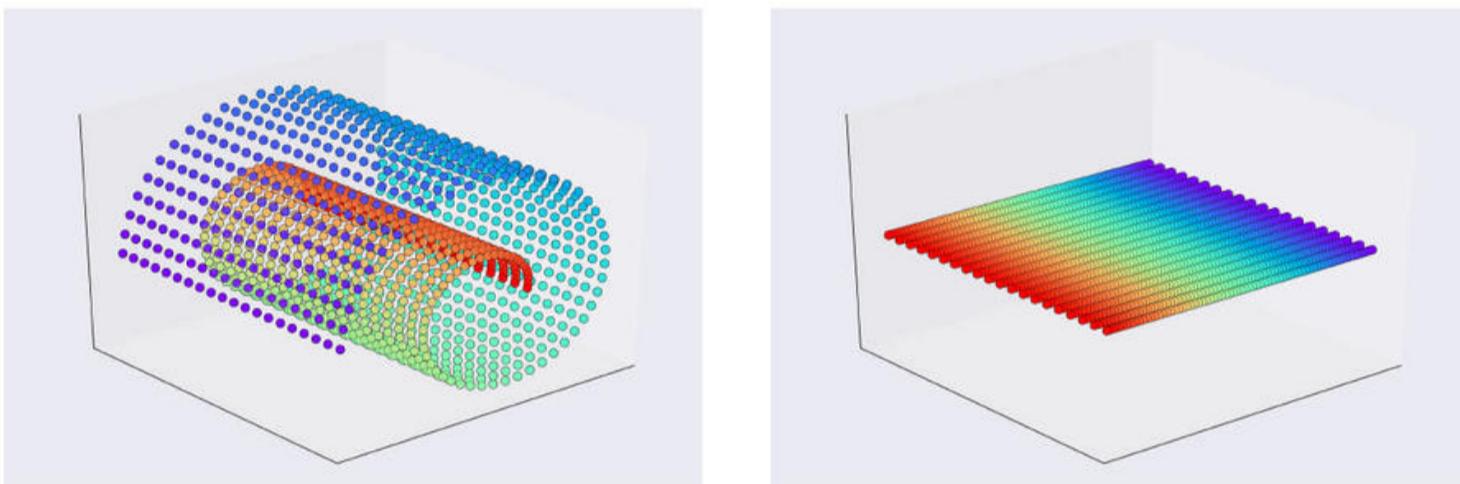


Figure 1.15: Measuring the infection of a tree trunk as it's turned and shaved. Left: The original data is in 3D. Right: We can make the problem easier by unrolling the data into 2D.

The spatial information here is important, but if we only want to follow the data along the spiral path we can make our lives easier by flattening it out, as in the right side of Figure 1.15. Now we need only 3 numbers for each data point: 2 for each point's location, and 1 for its value. This will make our processing faster without giving up anything that's important to this particular analysis.

Sometimes we can make our lives easier by simplifying the measurement of the data in the first place. When that's not possible, we might be able to simplify it after collection, by discarding information that's not relevant to the questions we want to ask. Then the computer doesn't need to process that information, saving us time and perhaps even increasing the quality of our results.

For example, let's suppose that we're monitoring traffic on a newly-constructed, curvy, one-way, one-lane road through the mountains. We want to make sure that the traffic is safe along one particular

segment that we're concerned about. So we set up some monitoring gear, and we periodically get back snapshots like Figure 1.16(a) that show us where each car is located, which way it's headed, and how fast it's going along the stretch we want to watch.

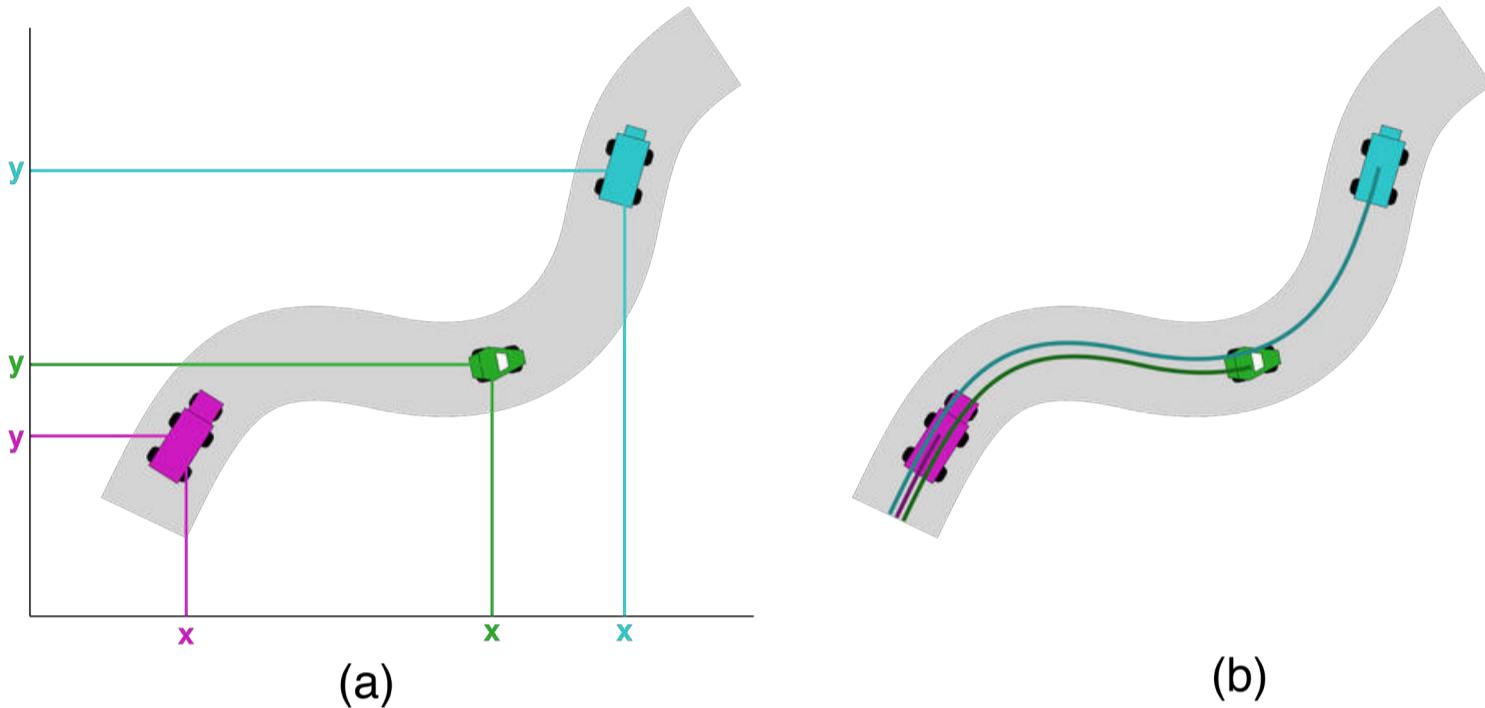


Figure 1.16: Where are the vehicles on this road? (a) We can measure each car's position using latitude and longitude, requiring two numbers (here marked x and y). (b) Alternatively, we can measure the distance that each vehicle has traveled along the road. This requires only one number, making our data simpler.

We might be tempted to describe the location of each car using latitude and longitude. But there's an easier way to describe where each car is. Since the road is only one car wide, we can describe each car's location by a single number that tells us how far it is from the start of this road section, as in Figure 1.16(b).

In all of these cases we can use unsupervised learning to analyze our samples and remove features that aren't contributing to understanding the underlying information. By making the data simpler, our learning system has to do less work to learn from it. This can make training faster, and perhaps even more accurate.

Sometimes we might be able to make our data simpler if we're willing to throw away a little bit of information. In some situations, such as when we're just running rough checks to make sure things are working properly, it's worth giving up some precision if it buys us a lot of simplicity, and thus training speed.

Let's see how to do that in our highway example. We found in Figure 1.16 how to locate vehicles with one number instead of two. In that case, we didn't lose any information. But in other situations we might be able to combine measurements so that we capture *most* of the information, though we lose some. For example, in a modified version of our highway we might have two lanes in each direction. In that case, we would really need the two numbers of latitude and longitude in Figure 1.16(a) to accurately locate each vehicle.

But if we wanted, we could use the single distance value anyway. As Figure 1.17 shows, this still gives us a close approximation of the car's location.

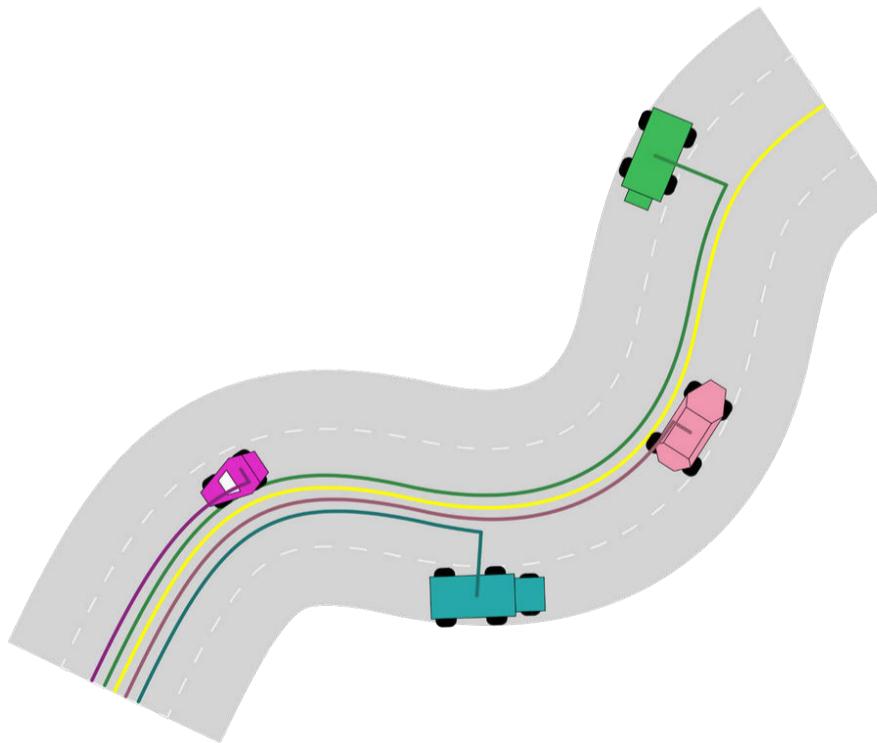


Figure 1.17: We can sometimes gain simplicity if we're willing to give up some accuracy. We can represent the positions of these vehicles in different ways, depending on our needs.

To find the positions of the vehicles in Figure 1.17, we can measure their distance from the start of the section or road in the lower left, as we did in Figure 1.16. But that assumes that each vehicle is located in the center of the road, which is no longer true. We can switch to a more elaborate and accurate representation. The tradeoff is that the simpler version will usually give us a system that is faster to train and run, but will contain inaccuracies. The more accurate system will be slower, but more precise. Both choices are valid, depending on how much accuracy and speed we need from our system.

In some situations, the approximate location we get from Figure 1.17 might be close enough.

In all of these examples, our goal was to simplify our data, either by removing uninformative features, combining redundant features, or trading off some accuracy for simplicity by combining or otherwise manipulating features.

In all of these cases, there are unsupervised learning algorithms available that can do the job.

Using unsupervised learning to find a way to reduce the number of values (or dimensions) that we need to describe our data is called **dimension reduction**. The name describes that we're reducing the number of features (also called dimensions) of each sample.

1.5 Generators

Suppose that we're shooting a movie where an important scene takes place inside a Persian carpet warehouse. It's important for us to see dozens, even hundreds of carpets, all over the warehouse. There are carpets on the floors, carpets on the walls, and carpets in great racks in the middle of the space. We want each carpet to look real, but be different from all the others.

Our budget is nowhere near big enough to buy, or even borrow, hundreds of unique carpets. So instead, we buy just a few carpets, and then we give them to our props department. We tell them to “make more like these, but all different.” They might just paint carpets on big flexible pieces of paper. However they do it, we want them to make lots of independent, unique things that look like real rugs that we can hang up all over the warehouse. Figure 1.18 shows our starting carpet.



Figure 1.18: A Persian carpet that we'd like to generalize.

Figure 1.19 shows some rugs that are based on Figure 1.18, but different.



Figure 1.19: Some new carpets based on the starting image of Figure 1.18.

We can do the same thing with machine learning algorithms. This process, called **data amplification** or **data generation**, is implemented by algorithms called **generators**. In this little example we

started with just one carpet as our example, but usually we train generators with large numbers of examples so that they can produce new versions with lots of variation.

We don't need labels to train generators, so we might call them unsupervised learning techniques. But we do give generators some feedback on the fly as they're learning, so they know if they're making good enough fakes for us or not. So maybe we should put our generators into the category of supervised learning.

A generator is in the middle ground. It doesn't have labels, but it is getting some feedback from us. We call this middle ground **semi-supervised learning**.

1.6 Reinforcement Learning

Suppose a bachelor with no children has to suddenly step in to take care of a friend's three-year old daughter. He has no idea what the young girl likes to eat.

Their first night together, he aims for simplicity, and makes pasta with butter. She likes it! But after pasta with butter every night for a week, the man and the child are both starting to get bored by the same old thing. So the man adds some cheese, and she likes that, too. Eventually alternating pasta with butter and cheese starts to get boring, so one night the man uses pesto sauce. But his friend's daughter refuses to take a bite. So he makes a new batch of pasta and tries marinara sauce instead, and she rejects that, too. Frustrated with variants on pasta, the man gives up and tries making her a baked potato with sour cream.

And so it goes, our new cook trying one recipe and one variation after another, trying to develop a menu that the child will enjoy. The only feedback he gets is that either his friend's daughter eats the meal, or she doesn't.

This approach to learning is called **reinforcement learning** [Champandardo2] [Sutton17].

We can re-phrase our cooking story above in more general terms by renaming the chef as the **agent**, and the child as the **environment**. The agent is the thing that is making decisions and taking actions (that's the chef), and the environment is everything else in the universe (that's the child). The environment gives the agent **feedback**, or a **reward signal**, after each action, characterizing how good that action turned out to be. Along with this evaluation of the action, the feedback also tells the agent if the environment has changed, and if so, what the environment looks like now. The value of this abstraction is that we can picture lots of combinations of agents and environments. Figure 1.20 shows the idea.

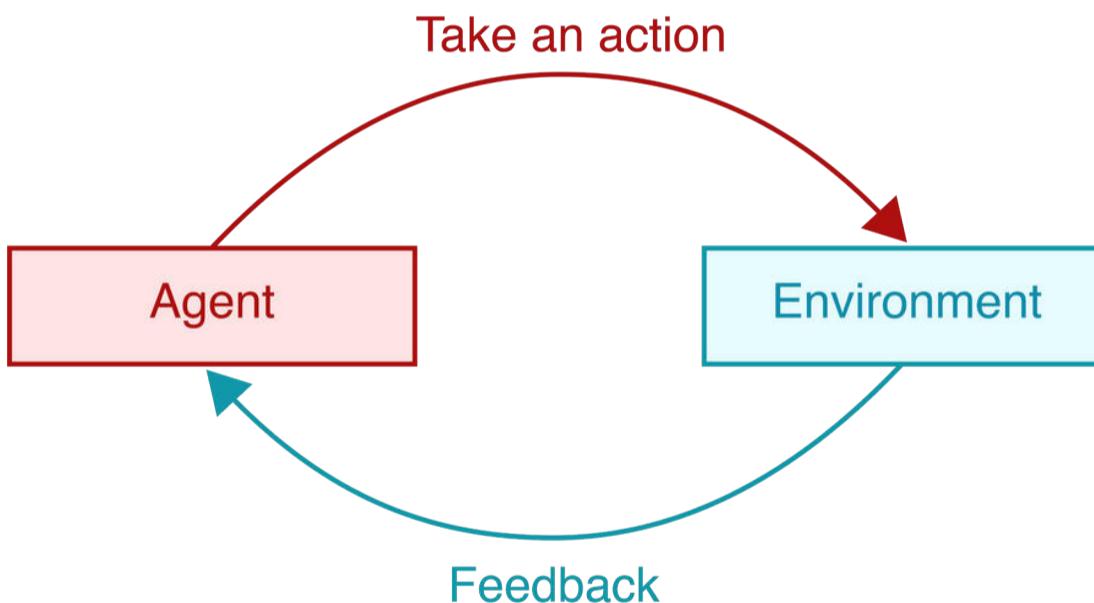


Figure 1.20: In reinforcement learning, we imagine an agent (who can take actions) and an environment (everything in the universe except the agent). The agent acts, and the environment responds by sending feedback in the form of a reward signal.

Reinforcement learning is different from supervised learning because our data is not labeled. The general plan of learning from mistakes is the same, but the mechanism is different.

By contrast, in supervised learning, our system produces a result (typically a category or a predicted value), and then we compare it to the correct result, which we provide. In reinforcement learning, *there is*

no correct result. The data has no label. There's just feedback that tells us how well we're doing. That feedback might just be a single bit of information, telling us that our action was "good" or "bad." Or it might be deeply nuanced with all kinds of descriptions about what changed in the environment as a result of our action, including any number of follow-on changes. Figuring out how to use whatever feedback we're given in the most effective way is at the heart of reinforcement learning algorithms.

For example, an autonomous car could be an agent, so the environment is composed of the other inhabitants of the street it's moving on. The car learns by watching what happens as it drives, preferring those actions that follow the laws and keep everybody safe. For another example, a DJ at a dance club is an agent, and the dancers who either like or dislike the music are the environment.

In essence, for any given agent, the environment is everything else in the universe that is not that agent. Therefore, the environment for every agent is unique. This means that in a multi-player game (or other activity), each player sees the world as made of their own self (the agent), and everyone and everything else (the environment). Thus every player is simultaneously both the agent to themselves, and part of the environment to everyone else.

In reinforcement learning, we create the agent in an environment, and then the agent learns what to do by trying out actions and getting feedback from the environment. There might be goal states (such as literally scoring a goal, or finding a hidden treasure), or the goal might merely be to reach a successful, steady state (like happy dancers at a club, or a car navigating city streets without problems).

The general idea is that the agent takes a single action, and the environment absorbs that action and usually changes in response. The environment then sends a reward signal back to the agent, telling them how good or how bad that action was (or if it made no difference).

The reward signal is often just a single number, where larger positive numbers mean the action was considered better, while more negative numbers can be seen as punishments.

In contrast to supervised learning algorithms, the reward signal is not a label or a pointer to a specific kind of “correct answer.” Rather, it is simply a measurement of how that action changed the environment and the agent itself with respect to the criteria we care about.

1.7 Deep Learning

The phrase **deep learning** refers to solving some machine learning problems using a specific type of approach.

The central idea is that we build our machine learning algorithm out of a series of discrete **layers**. If we stack these up vertically, we say that the result has **depth**, or that the algorithm is **deep**.

There are many ways to build a deep network, and many different types of layers to choose from when building it. Later in this book we’ll devote entire chapters to different types of deep networks.

A simple example is shown in Figure 1.21. This network has two layers, each shown as a box with a yellow tint. Inside each layer we have little pieces of computation called **artificial neurons**, or **perceptrons**, which we’ll discuss in detail in Chapter 10. The basic operation of these artificial neurons is to take in a bunch of numbers, combine them in a particular way, and output a new number. That output is then passed on to the next layer.

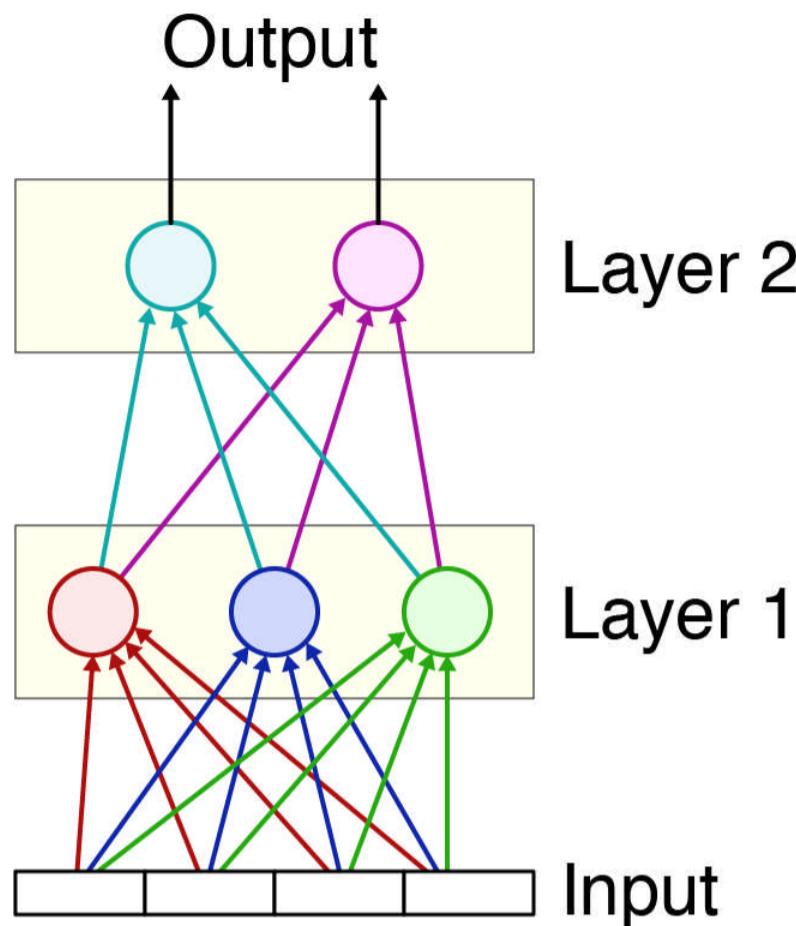


Figure 1.21: A deep network with 2 layers. Each layer is represented by a light-yellow box. Inside the box are artificial neurons. The outputs of one layer are connected to the inputs on the next layer. In this example, four numbers are provided as input, and 2 numbers make up the output.

Our example has only **2** layers, but networks can have layers numbering in the dozens or more. And instead of having only 2 or 3 neurons, each layer can contain hundreds or thousands, arranged in a variety of ways.

The appeal of a deep learning architecture is that we can create a full network by calling routines in a deep learning library, which take care of all the construction work. As we'll see in Chapters 23 and 24, we often need just one short line of code to instantiate a new layer and make it part of our network. The library handles all of the internal and external details for us.

Because of this structure and ease of construction, many people compare deep learning to building something out of Lego blocks. We just stack up the layers we want, each specified by a few arguments, name a few more options, and start training.

Of course it's not really that easy, because picking the right layers, picking their parameters, and handling the other options all need to be done with care. It's great that building a network is so easy, but that also means that every decision we make can have big implications. If we get something wrong, the network might not run. More commonly, it won't learn. We'll just feed it data and it will give us essentially useless results.

To minimize the chance of this frustrating situation, when we discuss deep learning we'll look at the key algorithms and techniques in sufficient detail that we'll be able to make all of those decisions in an informed way.

We've actually already seen an example of deep learning in Figure 1.10. We used a deep network with 16 layers to classify those photographs. Let's look more closely at what the system returns to us.

Figure 1.22 shows the four photos, along with the categories predicted by the network and their relative confidences. It recognized the hummingbird pretty perfectly. It wasn't very sure about the corkscrew ship. And since it had never seen a spoon or a set of headphones before, it guessed.

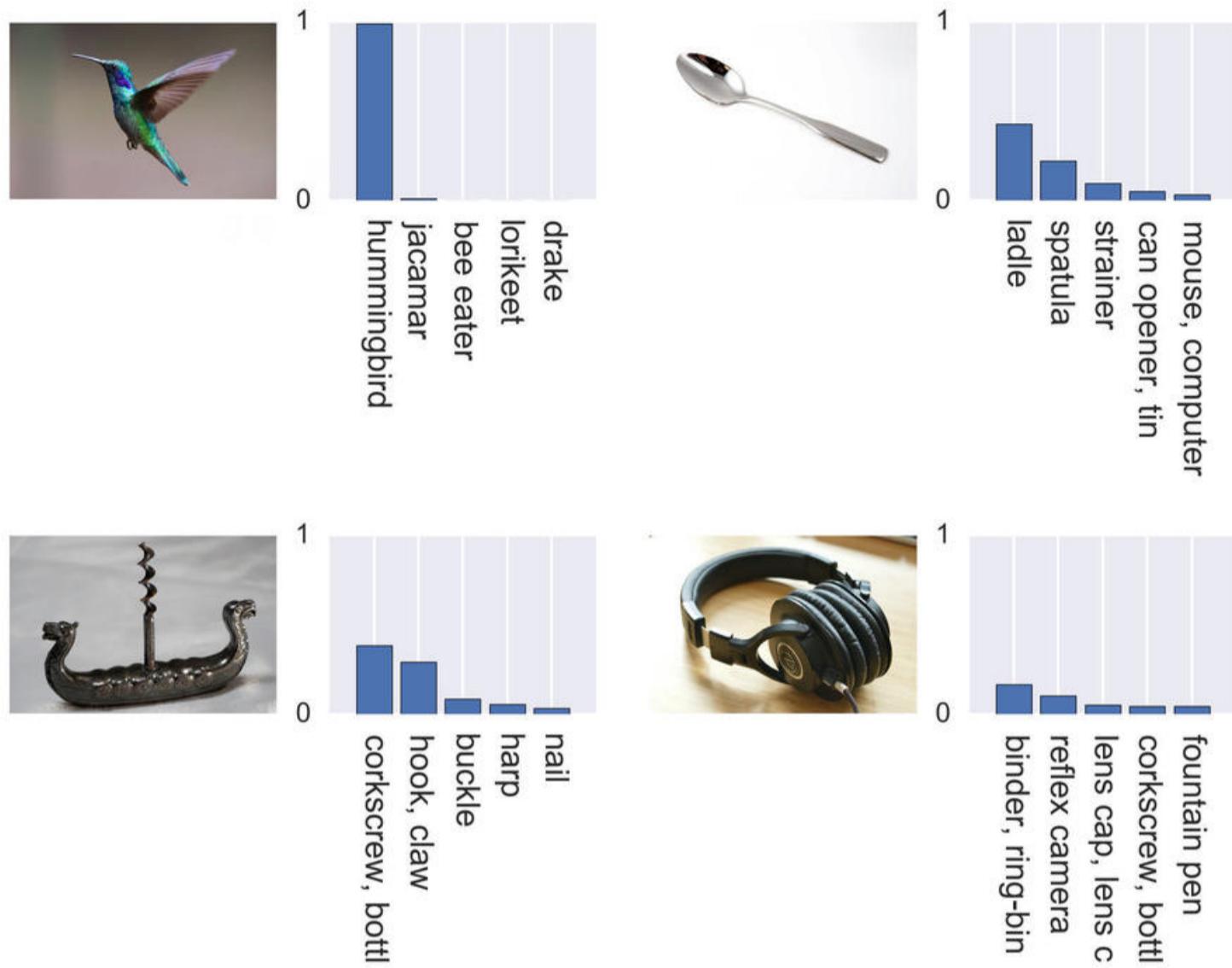


Figure 1.22: Four images and the predictions of our deep learning classifier. The system wasn't trained on spoons or headphones, so it's doing its best to match those images to things it knows about.

Let's try another image classification task. This time we'll train our system on hand-drawn digits from 0 to 9, and then give it some new images to classify. This is a famous data set called **MNIST**, and we'll return to it several times in this book. Figure 1.23 shows the system's predictions for a bunch of new digits it hadn't seen before.

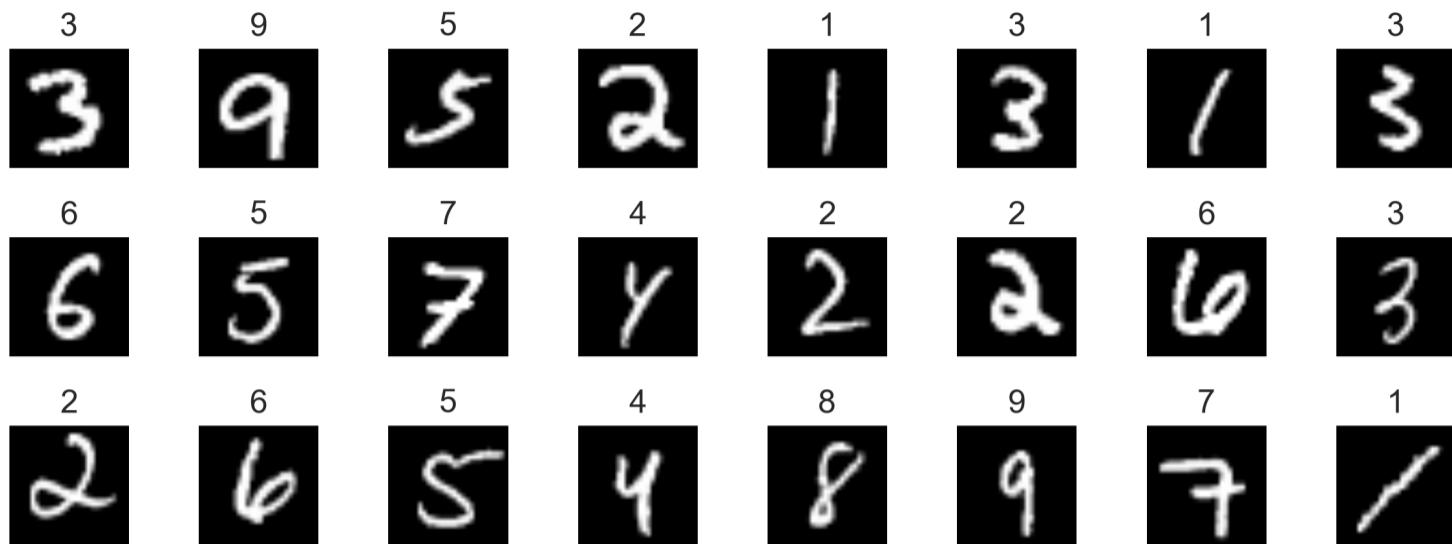


Figure 1.23: A deep learner classifying handwritten digits.

The classifier got every one of them correct. For this data, modern classifiers have become so good that they get over 99% accuracy. Even the little system we built just for this figure correctly classified 9,905 images out of 10,000, disagreeing with the label provided by the human expert only 95 times.

One of the beautiful things about deep learning systems is that they make good on **the promise of feature engineering**. Recall from above that this is the task of finding rules to let us decide whether, say, a digit is a 1 or a 7, or a picture is of a cat or dog, or a snippet of speech is the word “coffeeshop” or “movie.” We found that trying to come up with such rules by hand for a complex job is a nearly insurmountable task, because of all the variations and special cases we have to take into account.

But deep learning systems can implicitly learn to do feature engineering all on their own. They explore huge numbers of possible features, evaluating them, keeping some and discarding others, all on their own and with no guidance from us. The more representative power a system has, which usually boils down to more layers with more artificial neurons on each layer, the more it’s able to invent good features. **We call this **feature learning**.**

Deep networks of this type are being used every day to label faces in photos, understand commands spoken to cell phones, trade stocks, drive cars, analyze medical images, and an ever-expanding range of additional tasks.

As long as a system is built out of a sequence of computational layers of the types we'll see later, it can be called a "deep learning" system. So the phrase "deep learning" is more of an approach to building a machine learning program than a learning technique of its own.

When we want to solve big problems (like identifying which of 1000 different objects is in a photo), deep learning networks can become big, involving many thousands or even millions of parameters. Such systems require a mountain of data to train on, and typically a lot of computer time. The popularity of the **Graphics Processing Unit** (or **GPU**) has helped in this effort.

A GPU is a specialized chip that sits alongside the **Central Processing Unit** (or **CPU**), which carries out most of the tasks involved in running a computer, such as talking to peripherals and the network, managing memory, handling files, and so on. The GPU was originally designed to help relax the burden on the CPU by taking over the calculations involved in making complex graphics, such as rendered scenes in 3D games. The operations involved in training a deep learning system just happen to be a great match to the operations that GPUs were designed to do, so those training operations can be performed with great speed. And thanks to the design of GPUs, operations can also be carried out in parallel, so multiple calculations can be performed simultaneously. New chips designed specifically for accelerating deep learning calculations are starting to appear in the market.

Deep learning ideas are not applicable to all machine learning tasks, but where they are appropriate they can produce very good results. In many applications, deep learning approaches are so superior to other algorithms that they are revolutionizing the kinds of things that we use computers to figure out, from labeling photos to talking to our smart phones in plain language, and being understood.

1.8 What's Coming Next

We'll look into all of the ideas described above in much more detail in the coming chapters.

The first few chapters will deal with the basic principles that underlie all modern machine learning algorithms. We'll cover probability and statistics, and the basics of information theory. Then we'll look at basic learning networks and how to get them to work well.

All of that discussion will be applicable to any kind of machine learning language or library. But to make things specific, we'll see how to implement basic techniques using the popular (and free) scikit-learn library for the Python language.

Then we'll move back to general descriptions. We'll see how machine learning systems learn, and the different algorithms that people use to control that process. We'll follow that with a look at deep learning, and survey a variety of the most popular types of deep learning networks.

Near the end we'll take a second trip into the real world by showing how to use the Keras library (also a free library for Python) to build and run deep learning systems. We'll finish by looking at some recent powerful architectures for deep learning that are widely used today.

The book is cumulative, with many chapters building on previous chapters. Our goal is to present what you need to successfully use machine learning and deep learning to achieve your own goals, and to be able to work with libraries, understand the documentation, and understand the new advances that are coming every day.

If you're eager to learn something specific, feel free to jump to the most relevant chapter and dig in. Keep in mind that we'll be assuming you've read the previous chapters, so if we use terms or ideas that are unfamiliar, hop backwards as needed, or use the glossary, to fill in those gaps.

References

This section appears in every chapter. It contains references to all the documents that are referred to in the body of the chapter.

Whenever possible, we have preferred to use references that are available online, so you can immediately access them. The exceptions are usually books and other printed matter, but occasionally we'll include an important online reference even if it's behind a paywall.

[Bishop06] Christopher M. Bishop, “Pattern Recognition and Machine Learning”, Springer-Verlag, 2006.

[Bryson04] Bill Bryson, “A Short History of Nearly Everything”, Broadway Books, 2004.

[Champandardo2] Alex J. Champandard, “Reinforcement Learning”, Reinforcement Learning Warehouse, Artificial Intelligence Depot, 2002. <http://reinforcementlearning.ai-depot.com/Main.html>

[Darwin59] Charles Darwin, “On the Origin of Species by Means of Natural Selection”, November 1859. <https://www.gutenberg.org/files/1228/1228-h/1228-h.htm>

[Galton86] Francis Galton, “Regression Towards Mediocrity in Hereditary Stature”, The Journal of the Anthropological Institute of Great Britain and Ireland, Vol. 15, pgs 246–263, 1886. <http://galton.org/essays/1880-1889/galton-1886-jaigi-regression-stature.pdf>

[Goodfellow17] Ian Goodfellow, Yoshua Bengio, Aaron Courville, “Deep Learning”, MIT Press, 2017. <http://www.deeplearning-book.org/>

[Simonyan14] Karen Simonyan, Andrew Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition”, arXiv, 2014. <https://arxiv.org/abs/1409.1556>

[Sutton17] Richard S. Sutton and Andrew G. Barto, “Reinforcement Learning: An Introduction”, A Bradford Book, MIT Press, 2017. <http://incompleteideas.net/sutton/book/bookdraft-2017nov5.pdf>

[Towers15] Jared R. Towers, Graeme M. Ellis, John K.B. Ford, “Photo-identification catalogue and status of the northern resident killer whale population in 2014”, Canadian Technical Report of Fisheries and Aquatic Sciences 3139, 2015. <http://www.mersociety.org/Towersetal2015.pdf>

[VanderPlas16] Jake VanderPlas, “Python Data Science Handbook”, O’Reilly Media, 2016. <https://jakevdp.github.io/PythonDataScienceHandbook/05.04-feature-engineering.html>

Image credits

Figure 1.10, Figure 1.22, Hummingbird

<https://pixabay.com/en/hummingbird-2139279>

Figure 1.10, Figure 1.22, Spoon

<https://pixabay.com/en/teaspoon-coffee-spoon-metal-eat-554065>

Figure 1.10, Figure 1.22, Corkscrew

<https://pixabay.com/en/corkscrew-antique-old-wine-metal-1176167>

Figure 1.10, Figure 1.22, Headphones

<https://pixabay.com/en/audio-1840073>

Figure 1.14, Cow

<https://pixabay.com/en/cow-head-cow-head-animal-live-stock-1715829>

Figure 1.18, Persian Carpet

<https://pixabay.com/en/carpet-persians-red-retired-483855>



Chapter 2

Randomness and Basic Statistics

We'll see how to represent randomness in algorithms, and review some fundamental ideas from statistics that we'll use throughout the book.

Contents

2.1 Why This Chapter Is Here.....	48
2.2 Random Variables	49
2.2.1 Random Numbers in Practice.....	57
2.3 Some Common Distributions.....	59
2.3.1 The Uniform Distribution	60
2.3.2 The Normal Distribution	61
2.3.3 The Bernoulli Distribution	67
2.3.4 The Multinoulli Distribution.....	69
2.3.5 Expected Value	70
2.4 Dependence	70
2.4.1 i.i.d. Variables.....	71
2.5 Sampling and Replacement.....	71
2.5.1 Selection With Replacement	73
2.5.2 Selection Without Replacement	74
2.5.3 Making Selections	75
2.6 Bootstrapping	76
2.7 High-Dimensional Spaces	82
2.8 Covariance and Correlation.....	85
2.8.1 Covariance	86
2.8.2 Correlation.....	88
2.9 Anscombe's Quartet.....	93
References	95

2.1 Why This Chapter Is Here

We often want to discuss relationships between various pieces of data, without needing to talk about each piece of data individually. Are most of our pieces of data the “same” in some useful sense? Or do they span a wide range of “differences”? Are any of the pieces of data oddballs that don’t seem to fit? Does the data contain a pattern that connects some or all of the individual pieces?

These questions are important in machine learning, because the more we know about our data, the better we’ll be able to select and design the best tools for studying and manipulating that data.

As an analogy, suppose we’ve been asked to connect two wooden boards with a given bit of metal. If that bit of metal is a nail, we’d want to use a hammer. If it’s a screw, we’d want to use a screwdriver. By analyzing the data we’ve been given, we can pick the best tool for getting the most value from that data.

The tools that give us the language and concepts that let us discuss big collections of data are generally bundled together under the heading of **statistics**.

Let’s face it, you’re probably not reading a book on machine learning because you want to know about statistics. But these ideas are so important that we need to be at least familiar with them. Statistical ideas and language appear everywhere in machine learning, from papers and source code comments to library documentation. And understanding at least the basic statistics of a data set is often essential to selecting the right tools and algorithms for learning that data.

So we’ll do our best to keep this brief and focused. We’ll cover the core ideas, without delving into the math or details. Our goal here is to develop enough understanding and intuition to help us make good decisions when we do machine learning.

Closely connected to the ideas of statistics are those of **random numbers**. We'll see that there's more to the idea of random numbers than just the name of a library routine.

If you're already familiar with statistics and random values, or you really don't want to care about them yet, 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 on when these ideas get used.

2.2 Random Variables

Random numbers play an important role in lots of machine learning algorithms. Rather than simply picking some arbitrary number out of thin air, we use a variety of tools to control what kinds of random numbers we want to use and how we select them.

We deal with this idea when someone asks us, “Pick a number from 1 to 10.” They’re limiting our choices to the 10 integers in that range. When a magician asks us to “pick a card,” they usually offer us a deck of 52 cards, and we can select any one of them. In both of these examples our choices came from a finite number of options (10 and 52, respectively). But we can have an infinite number of options. Someone might ask us to “pick a real number from 1 to 10.” We could then choose 3.3724, or 6.9858302, or any other of the infinitely many real numbers between 1 and 10.

When we talk about ranges of numbers, we often also end up talking about their “average.” There are three different common types of averages, so let’s name them here to avoid confusion later.

As a running example, let’s make 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 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 it out sorted from the smallest to the largest value. 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. So for the list 1, 3, 4, 8, the median would be the average of 3 and 4, or 3.5.

Let's look more closely at these and related issues with a running analogy. Suppose we're photographers who have been assigned to illustrate an article on auto junkyards by taking lots of pictures of broken-down cars. Feeling adventurous, we go to a junkyard where there are hundreds of broken-down vehicles lying around.

We talk to the owner, and we agree on a price we'll pay for each car she brings us to photograph. To make it interesting, she has an old carnival wheel in her office, with one equally-sized slot for each car on the lot, as in Figure 2.1.

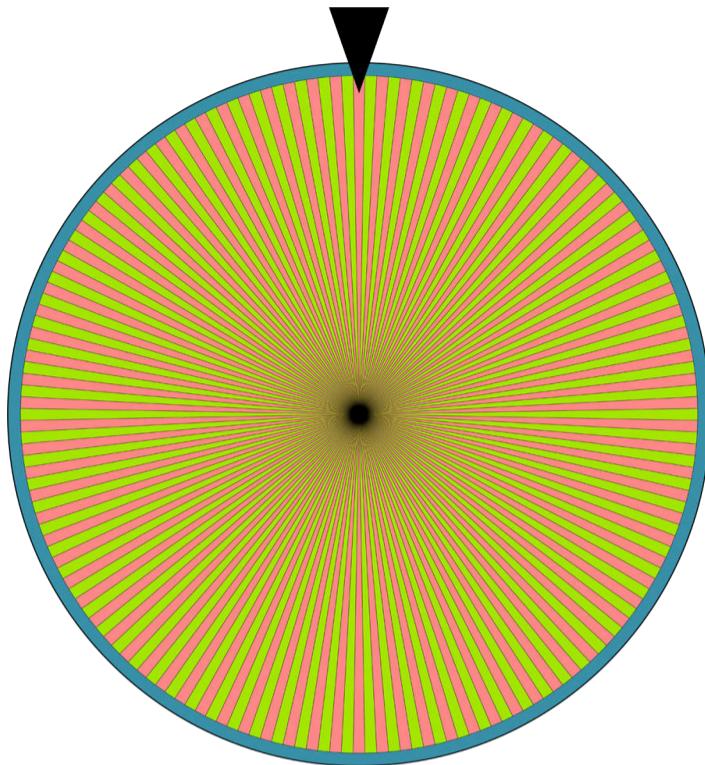


Figure 2.1: The junkyard owner's carnival wheel. There is one equally-sized sliver for every car on her lot. Each sliver has a number. When the wheel stops, she'll bring us the car with the number under the pointer.

Once we pay her, she spins the wheel. When the wheel stops, she notes the number at the top, and then goes out with her tow truck and drags the corresponding vehicle back to us.

We take some pictures, and she returns the vehicle to the lot. If we want to photograph another car, we pay again, she spins the wheel again, and the process repeats.

As far as we're concerned, each vehicle she brings us is **randomly chosen**, because we didn't know which one it would be. But it's not entirely arbitrary, because we knew a few things about it even before we started. For example, it couldn't be a car from the future. It couldn't be a vehicle that isn't on the lot. So the owner (via her carnival wheel) didn't choose from all possible vehicles that could ever exist. Instead, she chose one of the specific options that were available to her.

Suppose that our assignment calls for us to get pictures of 5 different types of cars: a sedan, a pickup, a minivan, an SUV, and a wagon. We would like to know the chance that each time she spins the wheel and gets back a random car from her lot, it's of each of these categories.

To work that out, let's suppose that we go into the lot and examine every vehicle, assigning it to one of these 5 categories. Our results are shown in Figure 2.2.

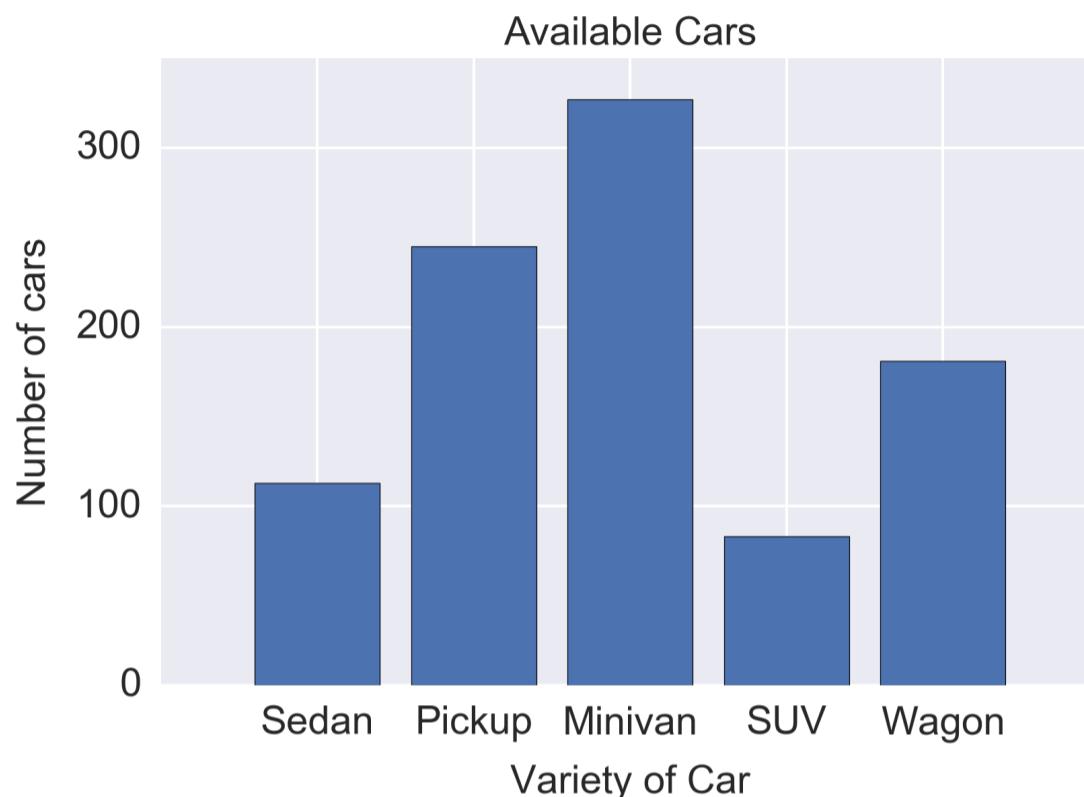


Figure 2.2: Our junkyard has 5 different types of cars in it. Each bar tells us how many cars we have of that type.

Of the almost 950 cars on her lot, there are more minivans than any other type, then 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 answer that, we can divide the height of each bar by the total number of cars. This relationship gives us the probability that we'll get that given type of car, as in Figure 2.3.

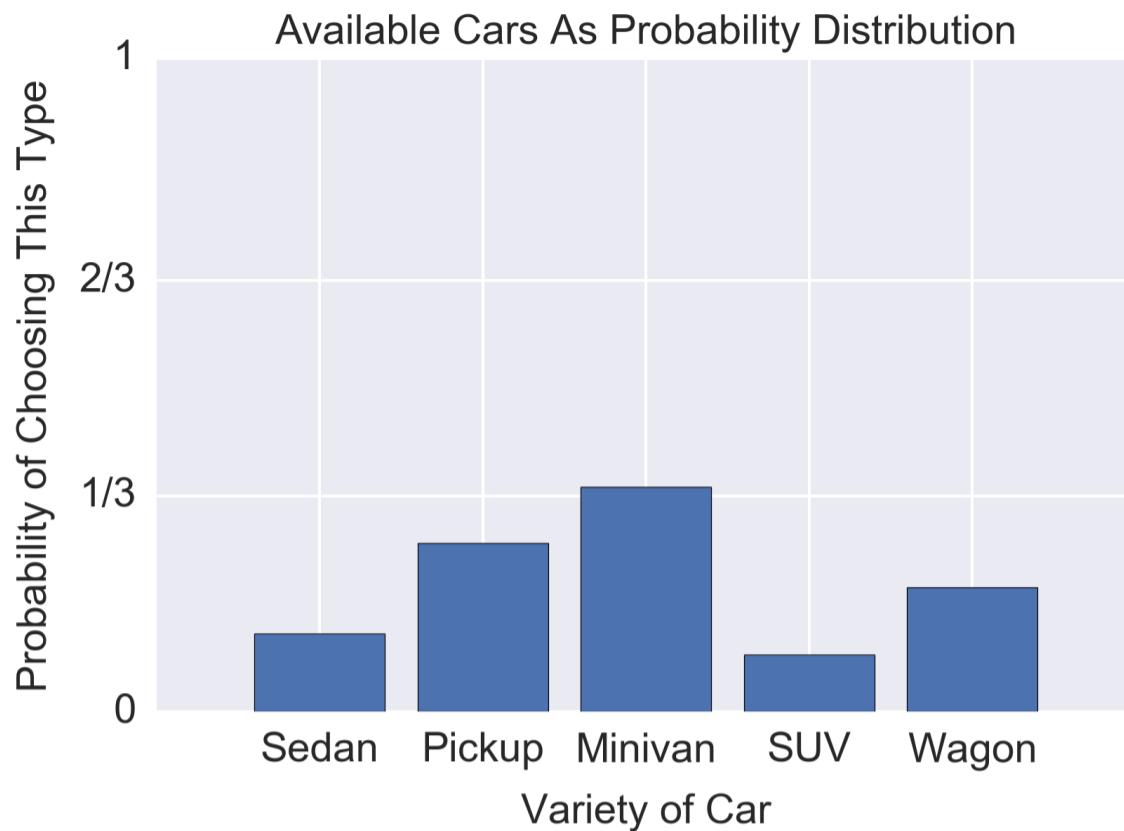


Figure 2.3: If we go into our junkyard and choose a car at random, the probability of getting each car depends on how many cars of that type are present, relative to the total number of cars in the yard. Since we know we're going to end up with *something*, the odds all add up to 1. When all the possibilities are covered and all the odds add up to 1, as they are here, we call it a probability distribution.

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% chance of getting a minivan.

If we add up the heights of all 5 bars, we'll find that the sum is 1.0. We say that the height of each bar is the **probability** that we'll get that kind of car. We can only use that term because the bars all up to 1. That sum tells us that the probability of getting *something* is 1 (or 100%), or a certainty. If the bars didn't add up to 1, then we could call the bars all sorts of casual names (like “expectation of getting this type of car”), but we shouldn't call them probabilities.

We call Figure 2.3 a **probability distribution** because it's “distributing” the 100% probability of getting a car among the 5 available options. We also sometimes say that Figure 2.3 is a **normalized**

version of Figure 2.2, where **normalized** here means that the values all add up to 1. The use of “normal” in this context comes to us from its meaning in mathematics.

We can represent our probability distribution as a simplified carnival wheel, as in Figure 2.4. 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 percentage as in Figure 2.3.

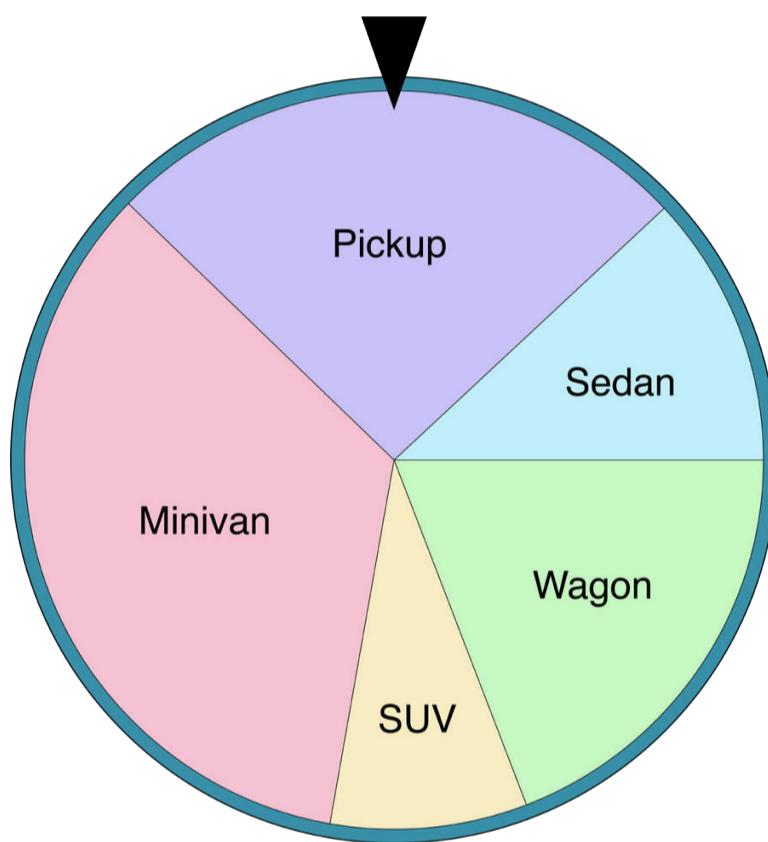


Figure 2.4: A simplified carnival wheel that tells us what kind of vehicle we’ll get if the owner spins the big wheel of Figure 2.1. The percentages of the circumference in each area follow the probability distribution in Figure 2.3.

Most of the time we don’t have carnival wheels around when generating random numbers on the computer. Instead, we rely on software to simulate the process. We give a routine a list of values, like the heights of the bars in Figure 2.3, and we ask it to return a value. We expect that we’ll get back “minivan” about 34% of the time, then “pickup” about 26% 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 be confusing to programmers, because programmers think of a “variable” as a named piece of storage that can hold data. But the term as used here comes from its usage in mathematics, where it has a different meaning.

Rather than a piece of storage, a random variable is a **function**, meaning it takes information as input and produces information as output [Wikipedia17c]. In this case, a random variable’s input is the distribution, or the probability of each possible value it can produce. Its output is a specific **value**, chosen from that collection according to those probabilities. The process of producing a value is called **drawing** a value from the random variable.

We called Figure 2.3 a probability distribution, but we can think of it as a function. That is, for each input value (in this case, the type of car) it returns an output value (the probability of getting that type of car). This idea leads us to the more complete and formal name of **probability distribution function** or simply a **pdf** (this acronym is usually written in lower-case). Sometimes people also refer to this kind of distribution with the more oblique name **probability mass function** or **pmf**.

Our probability distribution function has a finite number of options, each with a probability. So we can specialize its name even further and call it a **discrete probability distribution** (adding “function” at the end is optional). This name signals that the choices are discrete, like the integers, rather than continuous, like real numbers.

We can easily create probability distributions that are continuous. 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 (well, any real number up to the precision of our measuring device. Let's pretend that's so accurate that we can think of it as returning continuous values).

Figure 2.5 shows a continuous graph for our oil measurements. This graph shows us the probability of getting back not just a few specific values, but any real value at all, here between 0 (empty) and 1 (full).

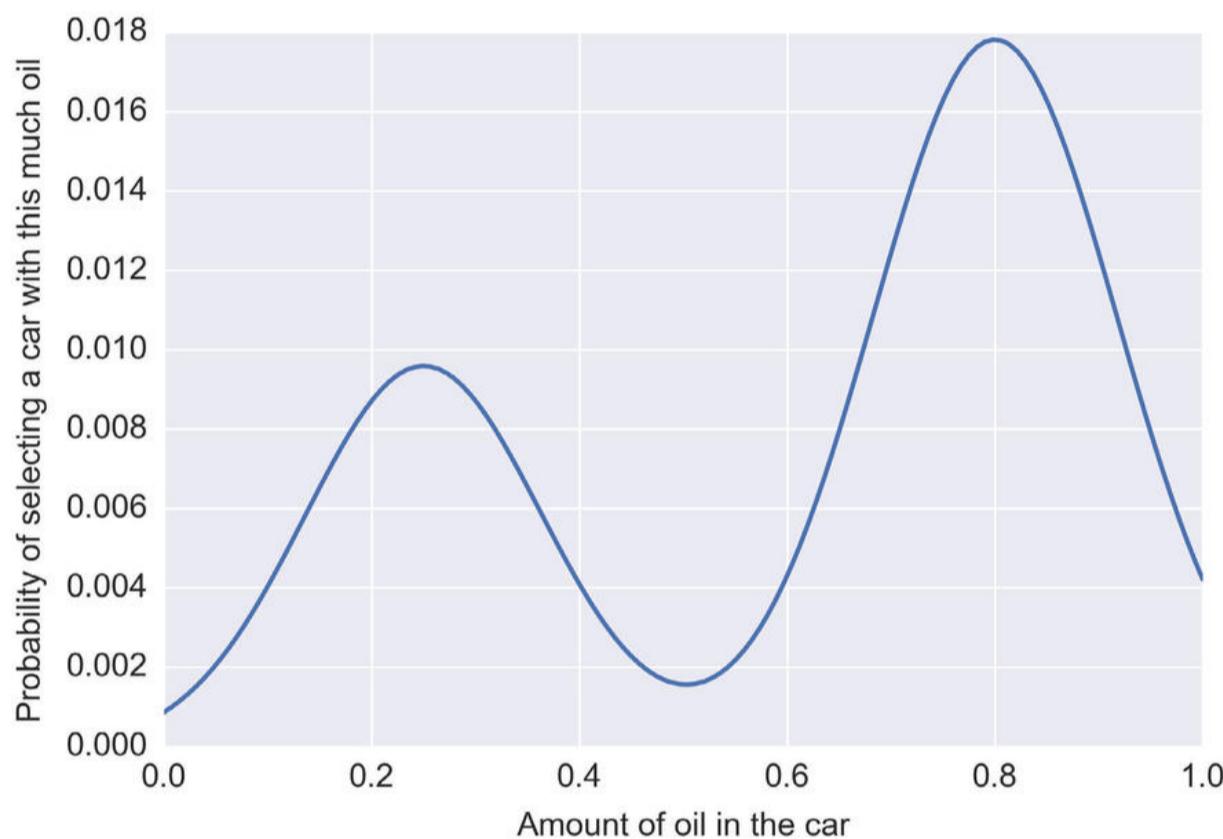


Figure 2.5: A distribution of probabilities for a continuous range of values. The sum of all probabilities still adds up to 1. Note that the vertical axis here runs from 0 to 0.018.

A distribution like Figure 2.5 is called a **continuous probability distribution** (or **cpd**), or **probability density function**. This latter term is unfortunately also abbreviated as **pdf**, but context usually makes it clear which abbreviation is intended.

Figure 2.5 shows input values from 0 to about 0.02, but the inputs can span any range. The only condition is that, like before, the graph must be normalized, meaning that all the values must add up to 1. When the curve is continuous, as in Figure 2.5, that means that the area under the curve is 1.

As we'll see, most of the time we use random numbers by selecting a distribution that fits what we want, and then calling a library function to produce a value from that distribution (that is, we draw a number from a random variable using 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 pre-built, pre-normalized distributions to control our random numbers. We'll see those distributions below.

2.2.1 Random Numbers in Practice

We've been talking about drawing a random variable from a distribution. It's worth pausing for just a moment to note the differences between that idea as a concept and how it shows up in practice.

By "random" we want to mean "unpredictable." It's not enough that the number would be hard to predict, like how long a light bulb would burn. Given enough information, we could work out when the bulb would expire. By contrast, a random number is fundamentally unpredictable (working out just when a number, or a sequence of numbers, veers from "difficult to predict" to "unpredictable" is a challenge that goes far beyond this book [Marsaglia02]).

Unpredictable random numbers are extremely hard to come by on the computer [Wikipedia17b]. The problem is that correctly functioning computer programs are **deterministic**, meaning that a given input will always produce the same output. So at a fundamental level, if we have access to the source code, a computer program can never completely surprise us. The library routines that give us "random numbers" are just programs like any other. If we look closely enough at what they're doing, we can predict what number they will give us each time we call them. In other words, there's nothing random

about them. To emphasize this, we sometimes call these algorithms **pseudo-random number generators**, and we say they produce **pseudo-random numbers**.

Suppose we're writing a program that uses pseudo-random numbers, and something goes wrong. To debug the problem, we want it to perform the very same actions again. In other words, we want it to get back the same stream of pseudo-random numbers it used when the error occurred.

We can make this happen because pseudo-random number generators produce their outputs in a sequence. Each new value builds on the computation of the previous value. To get the process started, we give the routine a number called the **seed**. Many library programs start with a seed that's derived from the current time, or the number of milliseconds since the mouse has moved, or some other value that we can't really predict beforehand. That makes the first output unpredictable, and sets the stage for the next output.

So by manually setting the seed before we start asking for outputs, we'll get the same sequence of pseudo-random numbers each time we run our program.

We don't have to settle for pseudo-random numbers if we don't want to. Random numbers can be created by using photographs of ever-changing and unpredictable scenes, like a wall of Lava Lamps [Schwab17]. Real random numbers also are available online through services that use real physical data, such as the static on an antenna resulting from atmospheric noise [Random17]. Typically, getting values back from such systems is much slower than using a built-in pseudo-random number generator, but the option is there when truly random values are required.

A possible middle road is to gather up a set of truly random numbers from physical data and then store them in a file. Then, like using a pseudo-random number generator with a seed, we'd always get the same values every time, but we'd know that they were truly random.

Of course, using the same sequence of numbers every time means that they’re no longer unpredictable, so this technique is probably most useful during a system’s design and debugging phases.

In practice, we rarely use the prefix “pseudo” when referring to random numbers generated by software, but it’s important to remember that it’s there, because every pseudo-random number generator runs the risk of creating numbers that have discernible patterns [Austin17]. These patterns could have consequences for our learning algorithms. Happily, the combination of an unpredictable seed and a well-designed pseudo-random algorithm gives us values that imitate truly random numbers well enough for the algorithms we’ll be using. It may be that as machine learning evolves, some algorithms will emerge that will be sensitive to the difference between numerical sequences that are “nearly” random and those that are really random, and that may force us to adopt a new strategy for generating these values. We don’t seem to be at that point right now.

2.3 Some Common Distributions

We’ve seen where random numbers come from, and how we can use a random variable to select a value from a distribution. Let’s now look at some popular distributions that are frequently used to generate the random numbers used in machine learning algorithms.

Most of these are offered as built-in routines by major libraries, so they’re easy to specify and use.

We’ll demonstrate these 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.

2.3.1 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. The values at exactly 0 and 1 vary from one definition to another. Here, we've set both 0 and 1 to have the value 1.

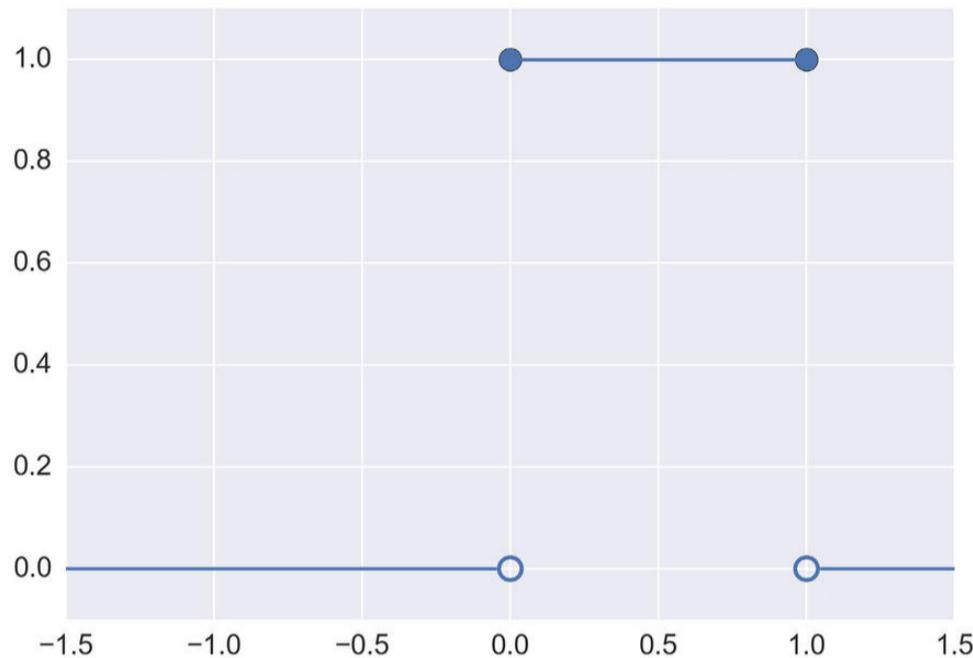


Figure 2.6: An example of a uniform distribution. In this version inputs from 0 to 1 (including 0 and 1) have an output of 1, while all other inputs produce 0. By convention, the unseen parts of the graph are assumed to have the values shown at the ends, so this graph is 0 everywhere to the right and to the left of the region shown.

In this graph, it appears 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 definition, but some implementations make either or both of those outputs 0. It always pays to check.

This distribution has two essential qualities. 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*. So 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 non-zero 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 the non-zero region to start and end where we like, rather than being fixed at 0 and 1. Probably the most popular choice, after the default of 0 to 1, is the range -1 to 1. The library will take care of details like adjusting the height of the function so that the area is always 1.0 (recall that this is the condition that turns any old graph into a probability distribution).

2.3.2 The Normal Distribution

After the uniform distribution, perhaps the next most popular choice 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.

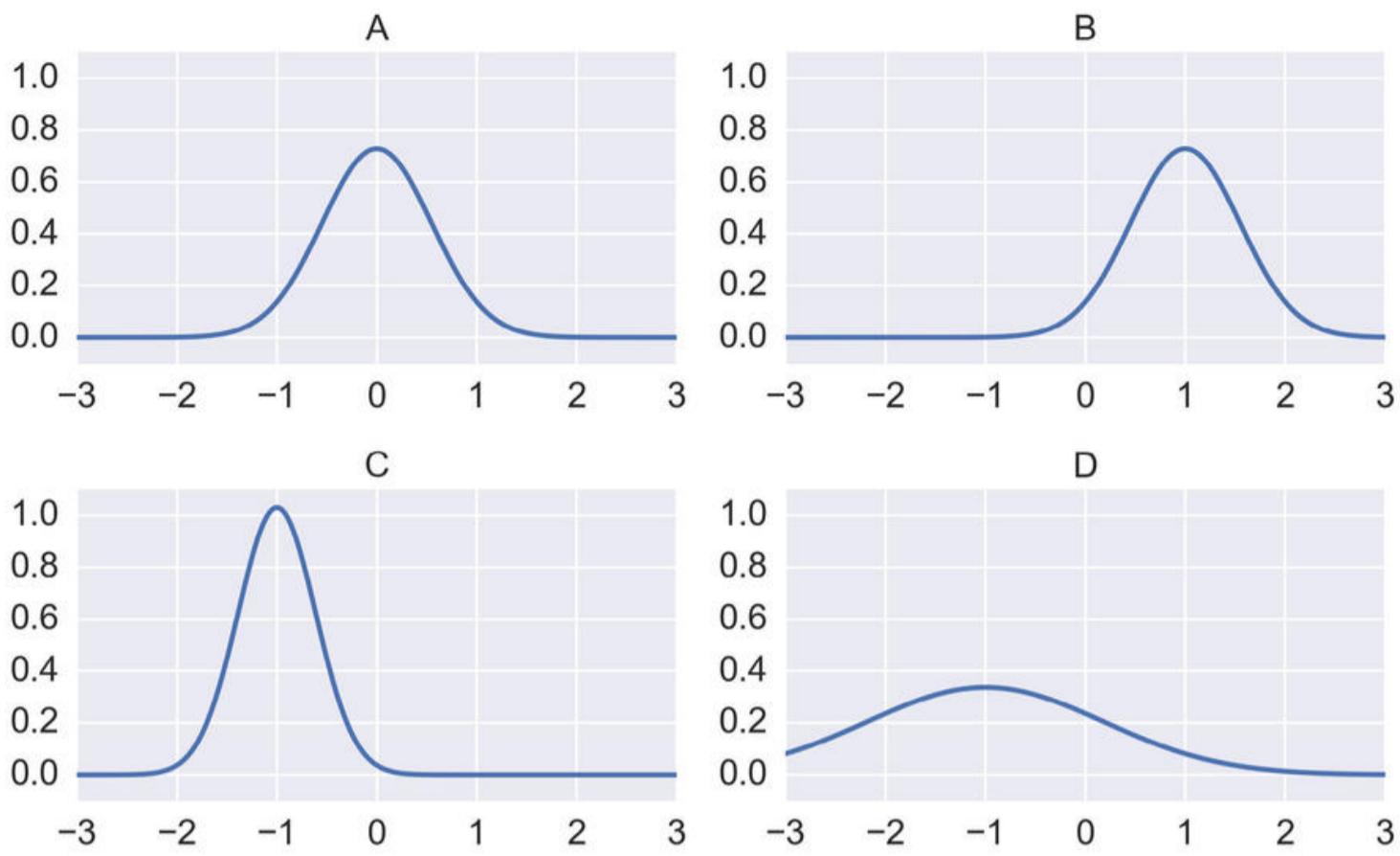


Figure 2.7: A few normal distributions. The basic blobby shape may be shifted left or right, scaled to be higher or lower, or stretched out or compressed. In all cases, it's still a normal distribution. (A) A canonical normal distribution. (B) The center is shifted to 1. (C) The center is shifted to -1, and the blobby shape is made more narrow. To keep the area under the curve at 1, a library will automatically increase the vertical scale of the blob as shown. (D) The center is shifted to -1, and the blob is made very broad. Again, the library automatically scales the height of the blob so the area under it is 1. Because the blob is wider, its height is reduced.

All four curves in Figure 2.7 have the same basic shape. The shapes only vary because the curve is moved horizontally, or was horizontally scaled (that is, stretched out or compressed). Such horizontal scaling causes the library to automatically scale the curve vertically, so that the area under the curve adds up to 1.

The vertical scaling is mostly unimportant to us, because we only care about the samples that come out. Figure 2.8 shows some representative samples that we'd get by drawing values from each distribution. It's easy to see that they bunch up where the distribution's value is high (that is, getting a sample at that value has a high probability), and there are fewer where the distribution's value is low (so getting back a

sample there has a low probability). The vertical locations of the red dots representing the samples are jittered to make the samples easier to see, and have no meaning.

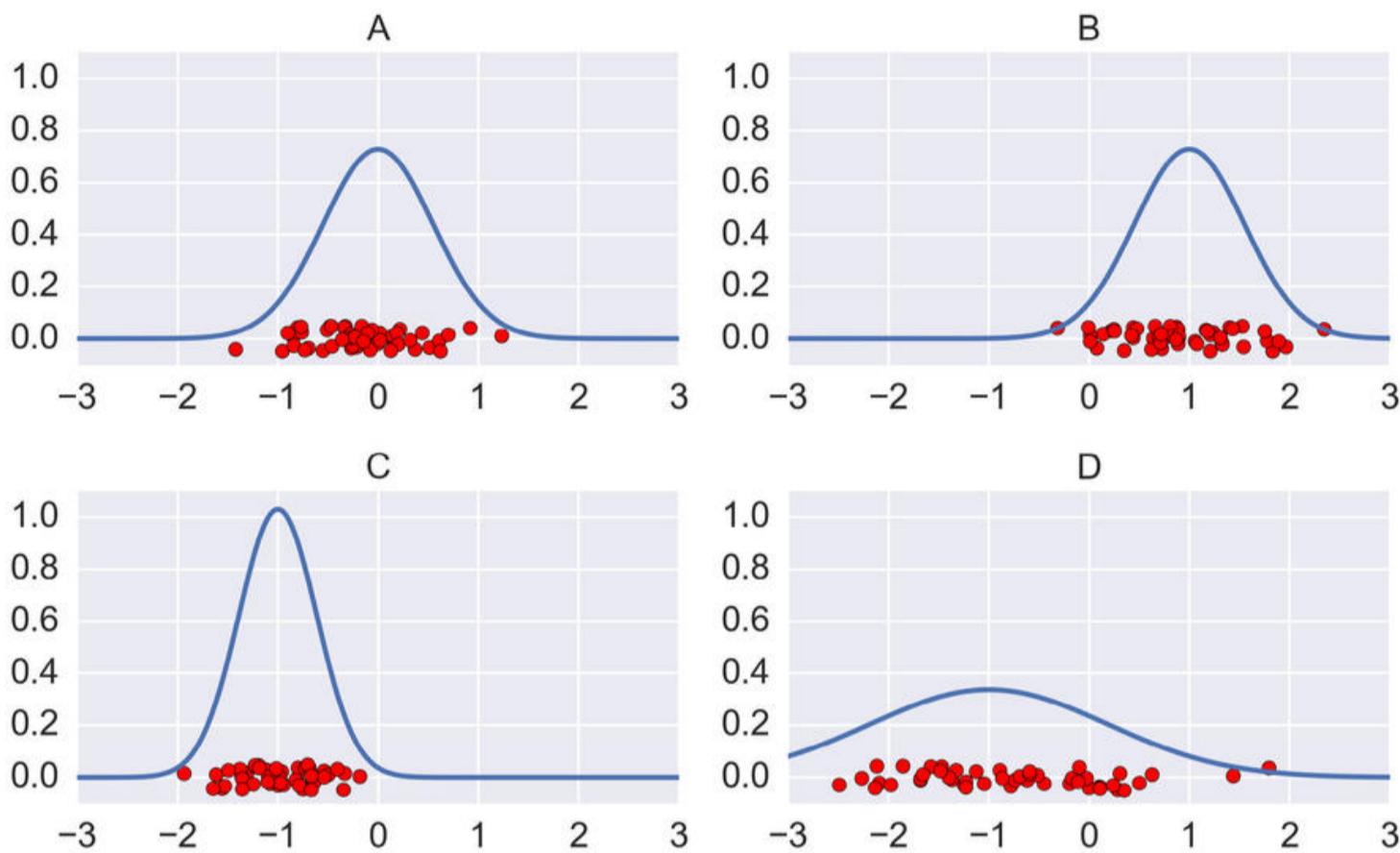


Figure 2.8: The horizontal location of each red circle shows the value of a sample resulting from drawing a value from each distribution. The vertical locations of the dots are irrelevant, added only to make the circles easier to distinguish.

The normal distribution is nearly zero almost everywhere, except for the region where it rises up in the shape of a smooth bump. Though the values drop off ever closer to 0 to the sides of the bump, they never quite reach 0 itself. So we say that the width of this distribution is **infinite**. In practice, we sometimes **clamp** the values more than a certain distance from the blob's center and pretend that they're 0 beyond that, giving us a finite distribution.

The normal distribution is popular in many fields, including machine learning, because an enormous variety of real-world measurements, from biology to the weather, return values that follow a normal distribution. A nice bonus is that its mathematics are particularly easy to work with in a wide range of settings.

The values produced by random variables that use a normal distribution are said to be **normally distributed**. They're sometimes also called **normal deviates**. We also say that they **fit**, or follow, a normal distribution.

Each normal distribution is defined by two numbers: the **mean** (the location of the middle of the bump), and the **standard deviation** (the horizontal stretching or compressing of the shape).

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.

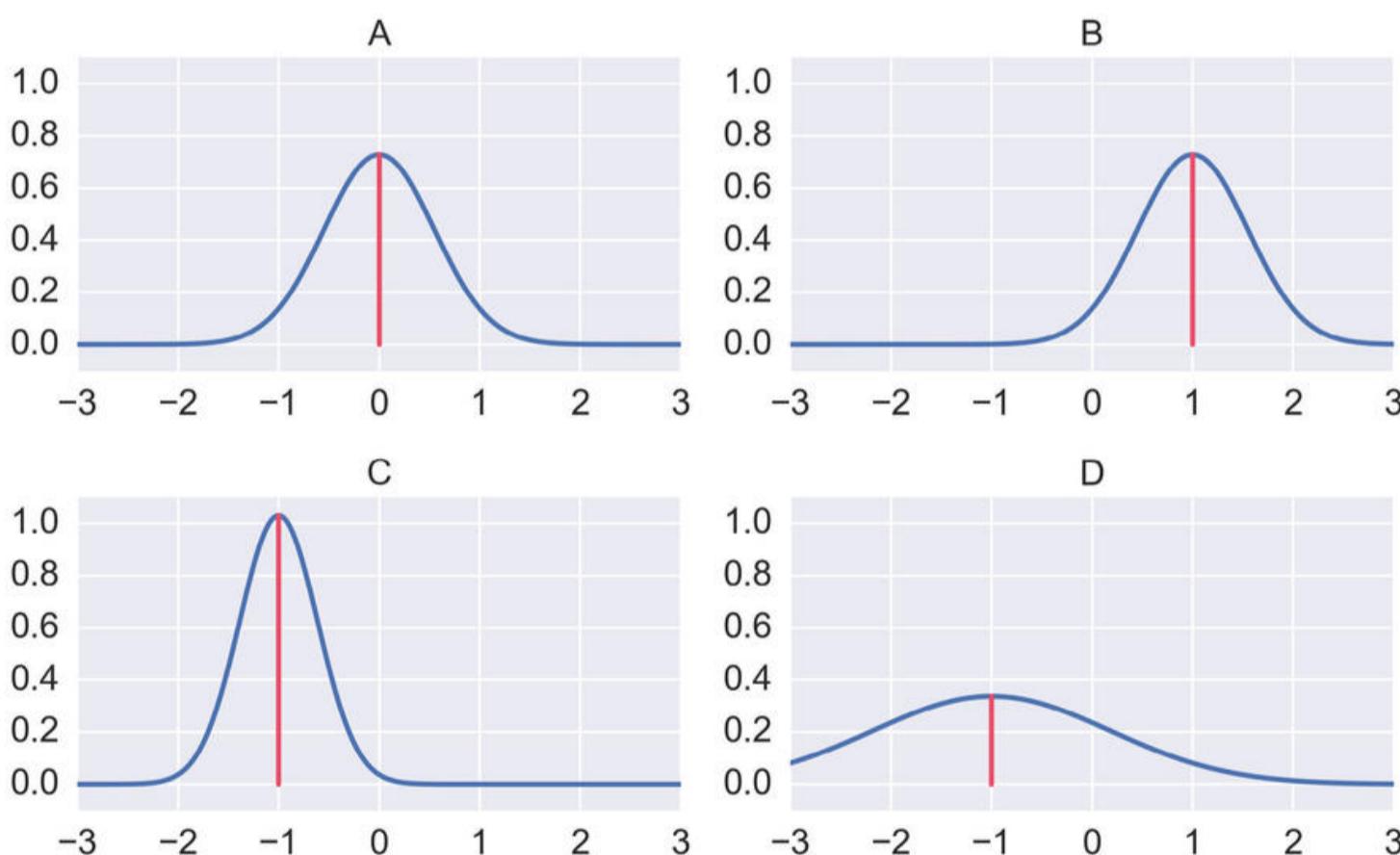


Figure 2.9: The mean of a normal distribution is the center of the bump, here shown with a red line.

The standard deviation is a number, often represented by the lower-case Greek letter σ (sigma). It tells us the width of the bump. Imagine starting at the center of the bump and moving symmetrically outwards until we're enclosing about 68% of the total area under the curve. The distance from the center of the bump to either of these ends

is the standard deviation. So the “standard deviation” is just a distance. Figure 2.10 shows our four Gaussians, with one standard deviation shaded in green.

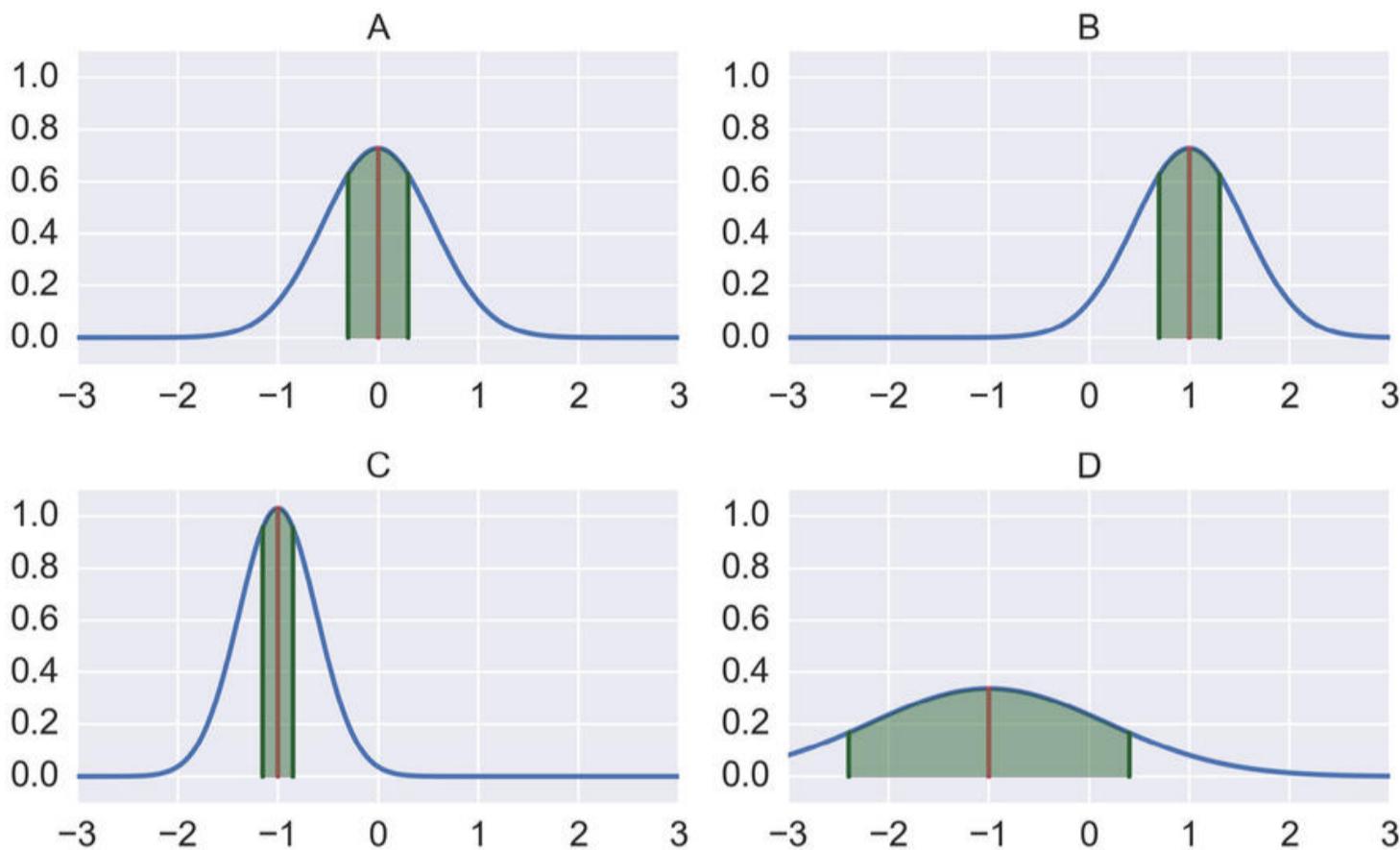


Figure 2.10: The standard deviation is a measure of how “stretched out” a normal distribution has become. Here the green shading shows the area under the curve that represents about 68% of the total area. The distance from the mean, or center, to either edge of the shaded region is the standard deviation of that Gaussian.

If we go out symmetrically from the center by another standard deviation, then we’ve enclosed about 95% 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% of the area under the curve, also shown in Figure 2.11. This property is sometimes called the **3-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**.

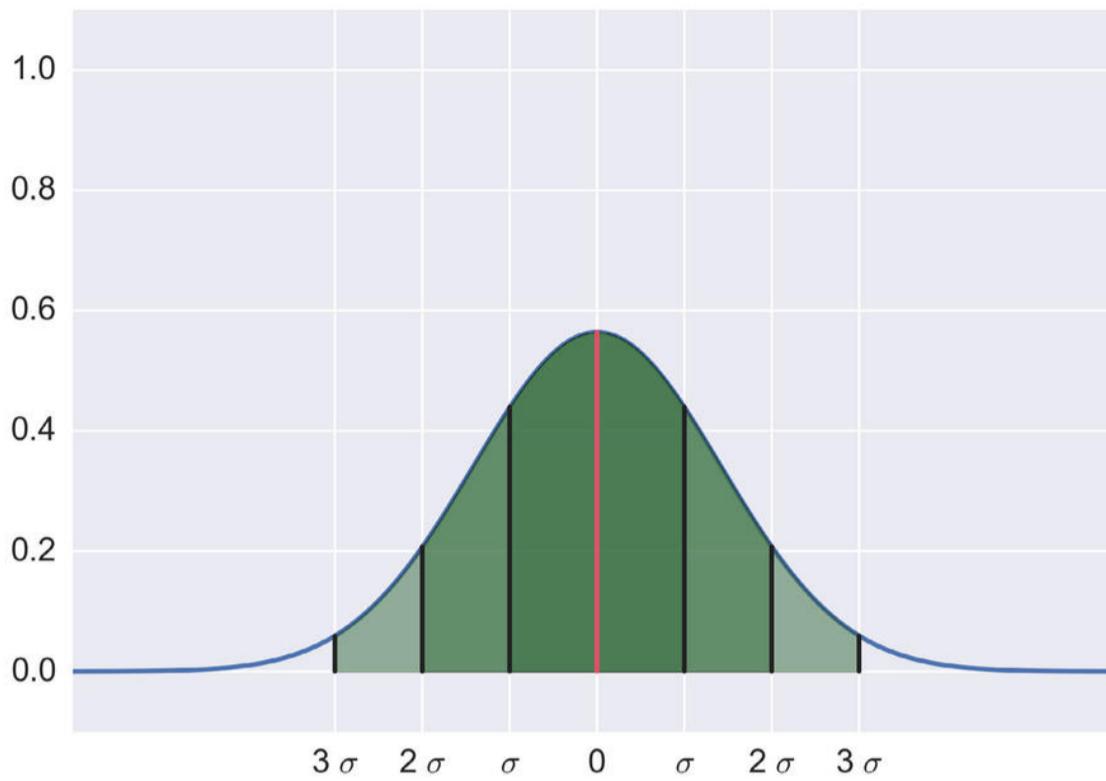


Figure 2.11: The standard deviation helps us find the probability of something happening. The values along the horizontal axis that are within one standard deviation of the mean (that is, in the range $(-\sigma, \sigma)$ when the mean is 0), make up about 68% of all the values that are selected from this distribution. Points within two standard deviations make up about 95% of all values, and points within three standard deviations make up about 99.7% of all values returned.

In other words, if we draw 1000 samples from any normal distribution, about 680 of them will be no more than one standard deviation from the mean, or in the range $-\sigma$ to σ , about 950 of them will be within two standard deviations, or in the range -2σ to 2σ , and about 997 of them will be 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% 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 normal distribution isn't just attractive because of its mathematics. It appears frequently because it naturally describes many real-world statistics. If we measure the height of adult males 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.

2.3.3 The Bernoulli Distribution

Another useful, but special, distribution is called the **Bernoulli distribution**. This discrete distribution returns just two possible values, 0 and 1. A common example of a Bernoulli distribution is the pattern of heads and tails that comes from flipping a coin.

We use the letter p to describe the probability of getting back a 1. Since the two probabilities must add up to 1 (after all, ignoring weird cases like landing sideways, the coin must land either heads or tails), this means that the probability of getting back a 0 is $1-p$.

Figure 2.12 shows this graphically for a fair coin and a weighted coin.

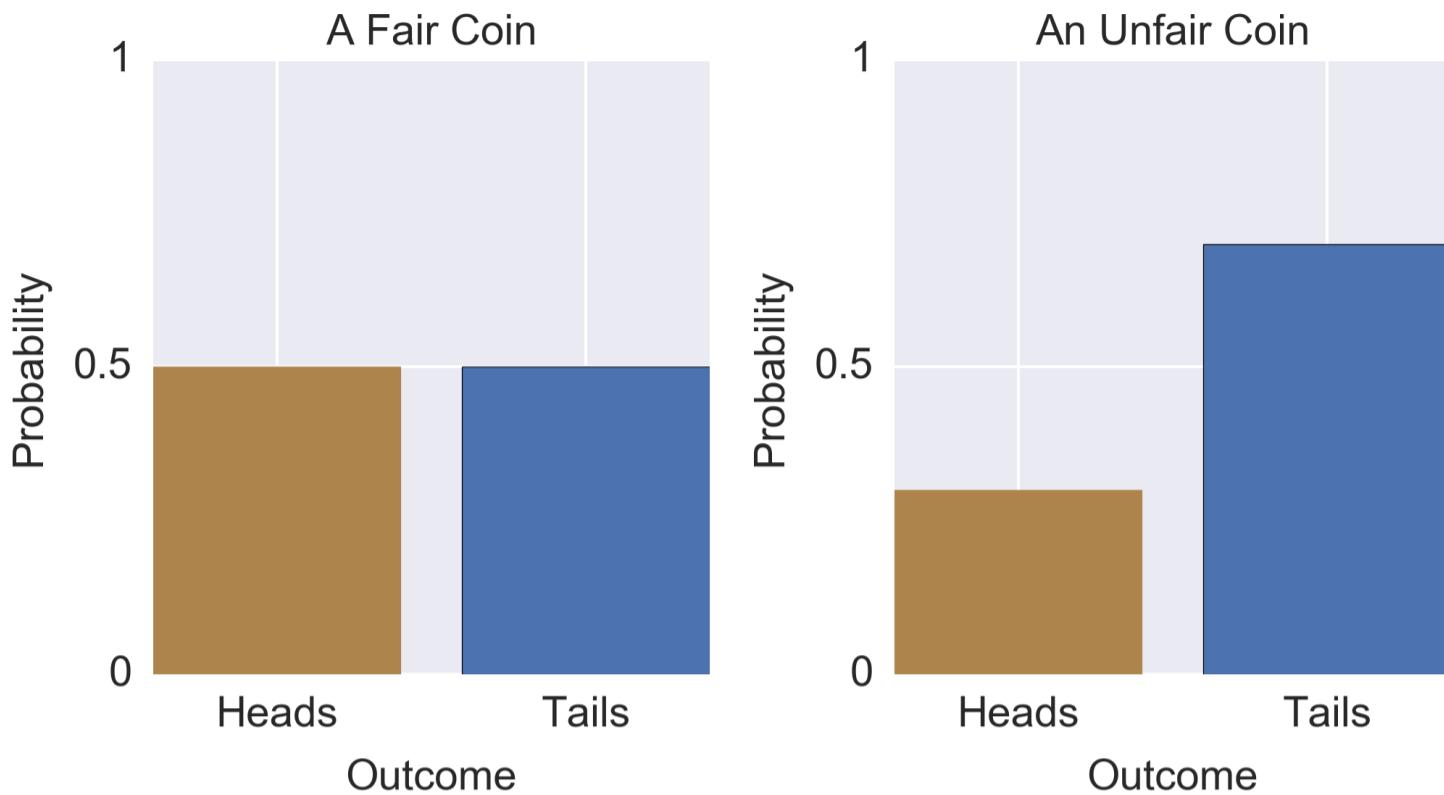


Figure 2.12: The Bernoulli distribution tells us the chance of drawing either a 0 or 1. Left: A fair coin has an equal chance of landing heads or tails on each flip. Right: An unfair coin might come up tails 70% of the time, so it will only come up heads 30% of the time.

We can label the two values something other than 0 and 1, or heads and tails. For example, if we're looking at photos they could be a-photo-of-a-cat and not-a-photo-of-a-cat.

The mean of a distribution is the mean value we'd get if we drew a huge number of values and found their mean. The mean of the Bernoulli distribution is p . The mode and median are a little messy to describe, so we won't get into them here.

The Bernoulli distribution may seem like overkill, because it describes such a simple situation. Its value is that it gives us a way to represent the odds of getting back either of two values in a distribution, so it uses the same formalism as the other distributions in this section. That means we can use the same equations and code that handle complicated distributions to handle this simpler situation as well.

2.3.4 The Multinoulli Distribution

The Bernoulli distribution only returns one of two possible values. But suppose we were running an experiment that could return just one of a larger number of possibilities? For instance, instead of flipping a coin that can come up either heads or tails, we 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 a number from **1** to **20**. But in other situations it's useful to have a list, where all the entries are **0** except for the one entry we drew, which is set to **1**. Using a list will be useful when we build machine learning systems to classify inputs into different categories, for example to describe which of 50 different animals appears in a photograph.

To demonstrate the idea, suppose that we want to choose from **5** values, which we'll label **1**, **2**, **3**, **4**, and **5**. To return a **4**, we'll return a list of **5** numbers. All will be **0** except for a **1** in the fourth position, making the list **(0,0,0,1,0)**.

So each time we draw a value from this random variable, we'd get back a list of four zeros and a single **1**. The probability of a **1** in each position is given by the probability of picking that position from the choices **1** through **5**.

The name of this distribution is a portmanteau, or a mash-up of two words. Because it's 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 and call it the **multinoulli distribution** (or sometimes the less colorful **categorical distribution**).

We might use a multinoulli distribution to come up with guesses for birthdays. Curiously, all birthdays are not equally likely, at least not in the US in the decade around the year **2000** [Stiles16]. We could represent the probability of each the **365** possible birthdays with a multinoulli distribution. If we draw a random variable from that

distribution, we'll get back a list of 365 values, all 0 except for a single 1. If we do this over and over, the 1's will appear more frequently in positions corresponding to dates where birthdays are more likely.

2.3.5 Expected Value

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

If these values are numbers, their mean is called the **expected value**. 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 all equally likely, then the expected value of the random variable that we use to draw a value from this list would be $(1+3+5+7)/4$, or 4, which we'd never get back from the distribution.

2.4 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 also random variables that *do* depend on each other. For example, suppose we had several distributions for the fur length of different animals: dogs, cats, hamsters, etc. We might first pick an animal at random from the 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 which animal we've chosen, so that's a **dependent** variable.

2.4.1 i.i.d. Variables

The math and algorithms of many machine-learning techniques are designed to work with multiple values drawn from random variables with the same distribution, and are independent of each other.

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 almost always written in lower case, with periods between the letters). We might see, for example, a technique described this way: "This algorithm will deliver the best results when the inputs are i.i.d."

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

2.5 Sampling and Replacement

In machine learning we often 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 think about two different ways to make a new dataset from an existing one. The key question is this: when we select an element from the dataset, do we remove it from the dataset, or do we make a copy of that element and use that?

As an analogy, suppose we go to the library to find a few short books to read. We want to make a little pile of them at our desk. To keep things interesting, we'll choose books from the stacks at random.

One approach starts by going into the stack and picking a book at random. We take it back to our table, drop it off, and return to the stacks. In this way we build up a pile of randomly-selected books. Note that because we've left each book we've picked at our table, we cannot possibly select the very same book twice (we can pick another copy of it, but not the same book).

An alternative approach is to go to the stacks and select a book. Ignoring legal and moral issues for the sake of the metaphor, we'll photocopy the entire book. We'll return the book to the stack where we found it, and then drop off the photocopy on our table. Then we'll return to the stacks, and again pick a random book, photocopy it, return it, and add it to our pile, over and over, building up a pile of photocopied books. Note that because we're returning each book to the stacks after we've copied it, but before we go to pick another book, it's possible that we could end up selecting and photocopying the same book more than once.

To build our new datasets in machine learning, we can follow either of these approaches. Each time we select a piece of data from our training set, we can either remove it from the set (so we can't pick it again), or we can make a copy of it and return it to the set (so we might pick it again).

These two approaches can give us very different types of results, both in obvious terms and in less obvious statistical terms. Some machine learning algorithms are designed to work only with selections made one way or the other. Let's look at these alternatives more closely. We'll say that we want to create a list of **selections**, which we choose from a starting **pool** of objects.

2.5.1 Selection With Replacement

Let's first look at the approach where we make a copy of each selected item, so the original stays in place, as in Figure 2.13. 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.

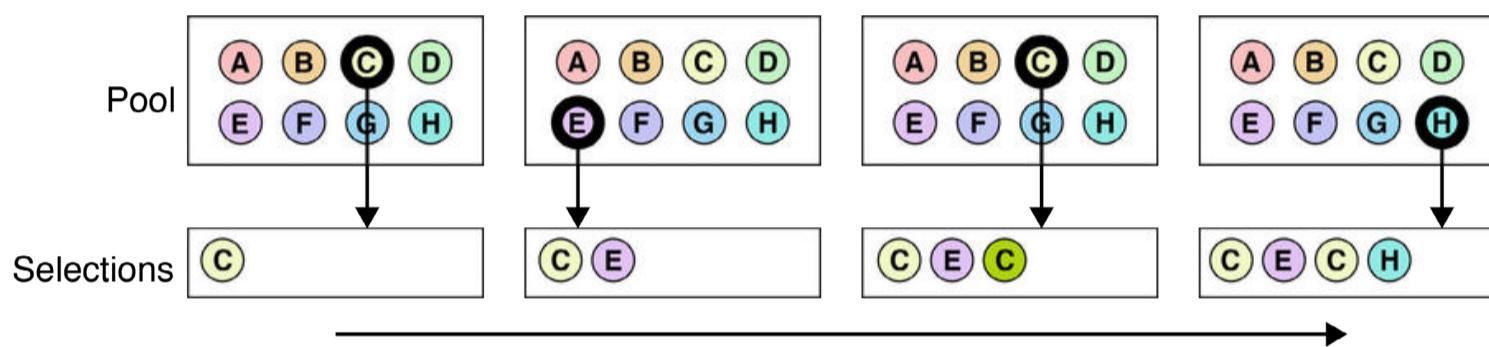


Figure 2.13: Selection with replacement. Each time we remove an element from the pool, we place a copy into our selections area, and then replace the original into the pool. With this technique, we build up our selections but the original pool never changes, so there's a chance we could pick the same item more than once. In this example, we get object C twice.

The most obvious 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, the pool in Figure 2.13 has 8 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 into the original set after selecting it. When we look again at the original dataset, all 8 items are still there, and if we choose again, each still has a 1 in 8 chance of being picked.

An everyday example of sampling with replacement is ordering a coffee drink 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.

2.5.2 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 2.14.

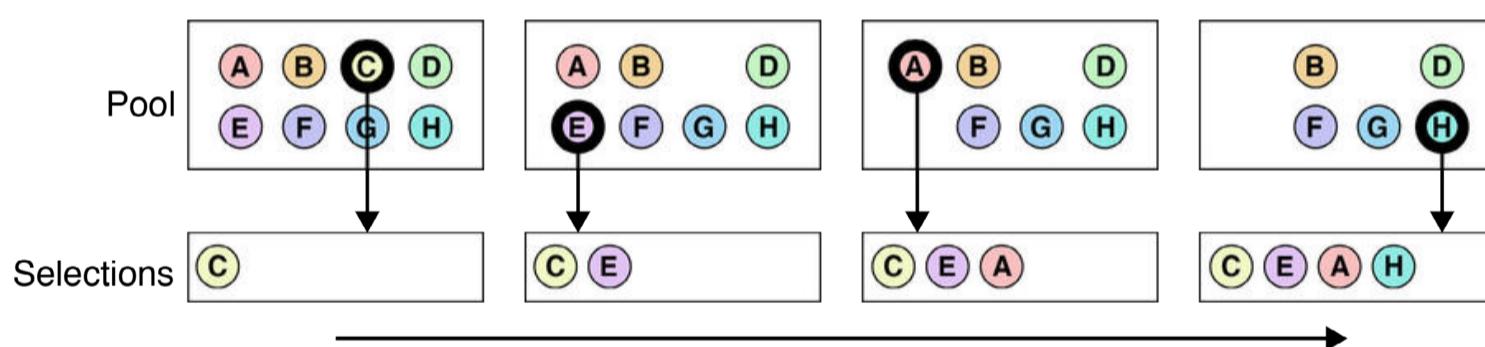


Figure 2.14: Selection without replacement. Each time we remove an element from the pool, we place it into our selections area. We do not replace it into the pool, so that element cannot be selected again.

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 database.

Second, our new database can be smaller than the original, or the same size, but it can never be larger.

Third, our choices are now **dependent**. In Figure 2.14 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. If we return to the

source data, there are only 7 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 7 items, the last one is 100% sure to be selected the next time.

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 re-collected and shuffled.

2.5.3 Making Selections

Suppose that we want to make a new database by selecting a pool that's smaller than the original set. We could build it with or without replacement.

But sampling with replacement can generate many more *possible* new databases than sampling without. To see this, suppose we had just three objects in our original database (let's call them A, B, and C), and we want a new database of two objects.

Sampling *without* replacement gives us only three possible new databases: (A,B), (A,C), and (B,C).

But sampling *with* replacement gives us those three, plus (A,A), (B,B), and (C,C).

Generally speaking, sampling with replacement will always give us a larger set of possible new databases. There are other interesting statistical differences, but we won't go into them here.

The important thing to keep in mind is that whether we replace elements or not will make a difference in the new datasets we build.

2.6 Bootstrapping

Sometimes we want to know some statistics about a data set 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, so we need an alternative approach.

Usually we try to answer this kind of question by extracting a piece of the dataset, and then measuring that. So 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**. The sample set is built without replacement, so each time we select a value from the population it's removed from the population and placed into the sample set, and cannot be chosen again.

We hope that by building our sample set carefully, it will be a reasonable proxy for the whole population with respect to the properties we want to measure. Figure 2.15 shows the idea for a population of 21 circled numbers. The sample set contains 12 elements from the population.

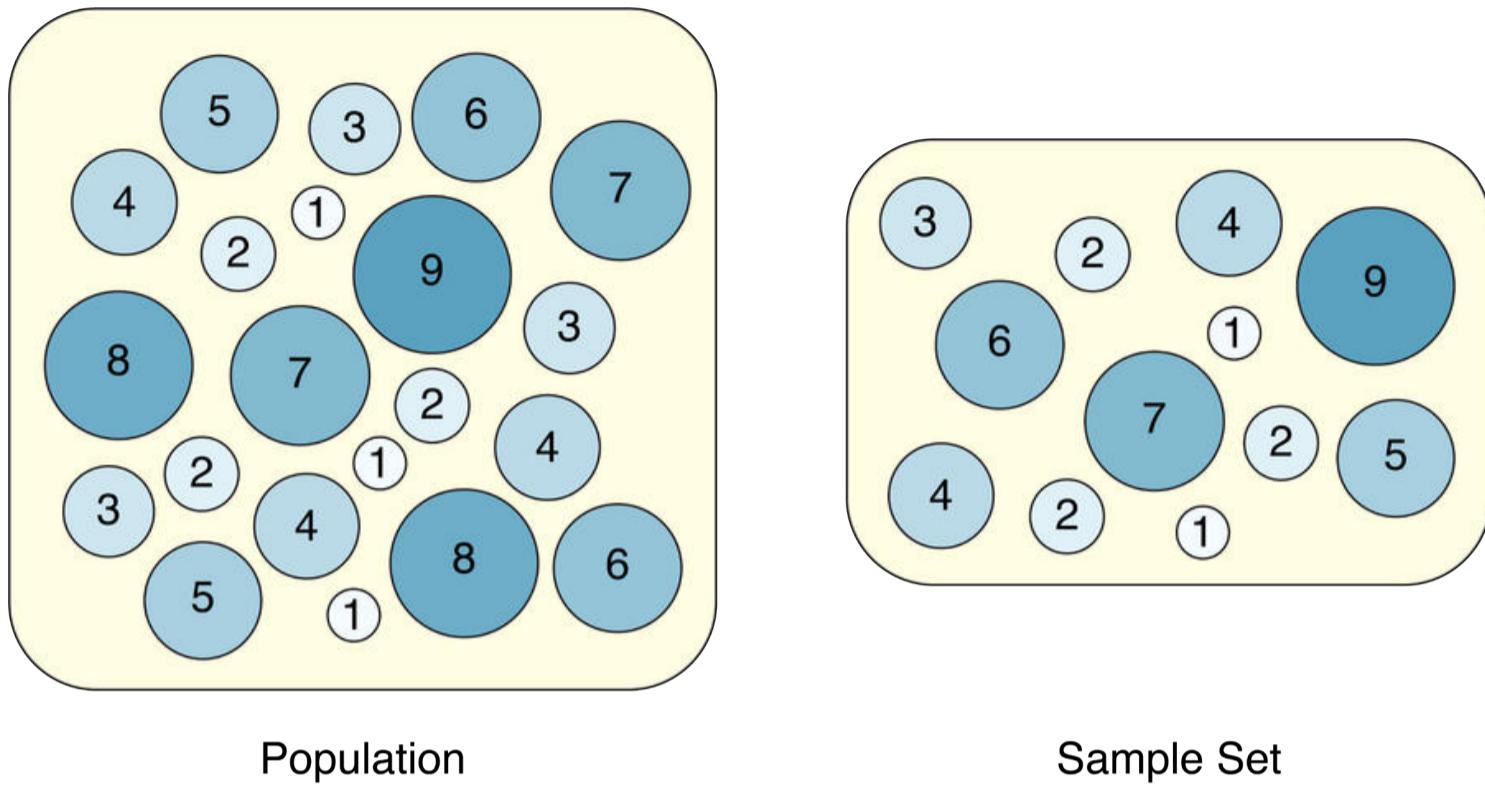


Figure 2.15: Creating a sample set from a population. Left: The population. In this example, the population contains 21 elements. Right: The sample set, containing only 12 elements.

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). So by finding the mean of the sample set, we've come up with an answer, 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**. Without getting too deep into this idea right now, this lets us make a statement of the form, “We are 98% 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 a measure of how confident we are that the value is in that range (here, 98%). 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.

The technique of **bootstrapping** is one way to find the values that let us express our confidence this way [Efron93] [Ong14]. We can talk about the mean, as in our example of people's heights, but we can also use bootstrapping to express confidence about the standard deviation, or any other statistical measure we're interested in.

The process involves two basic steps. The first is the step we saw in Figure 2.15, where we create a sample set from the original population. The second step involves **resampling** that sample set to make new sets. Each of these new sets is called a **bootstrap**.

To create a bootstrap, we 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. Then 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 2.16.

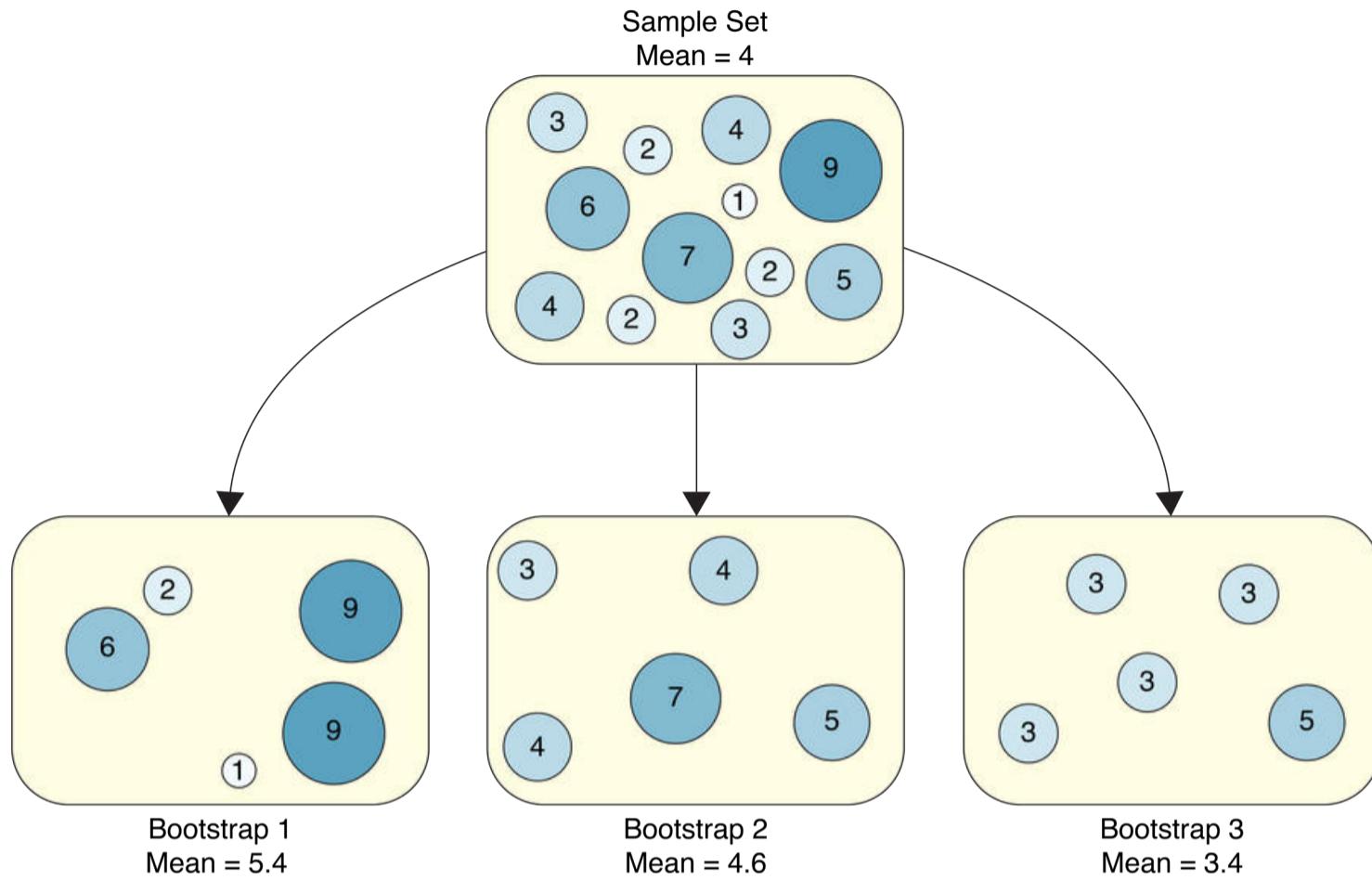


Figure 2.16: Creating bootstraps. Here our sample set holds 12 items, and we've decided each bootstrap should hold 5. Each bootstrap holds 5 elements selected randomly, with replacement, from the sample set.

We need to select with replacement 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, every bootstrap would be identical to the sample set.

Once we have these bootstraps, we can measure the mean of each one. If we plot these means on a histogram, as in Figure 2.17, we'll find that they tend to form a Gaussian distribution, as we discussed above. This result is a natural consequence of the process, and has nothing to do with the specific values we chose. For this figure, we started with a population of 5000 integers from 0 to 1000. We drew 500 values from that set at random to make a sample set, and then we created 1000 bootstraps, each with 20 elements.

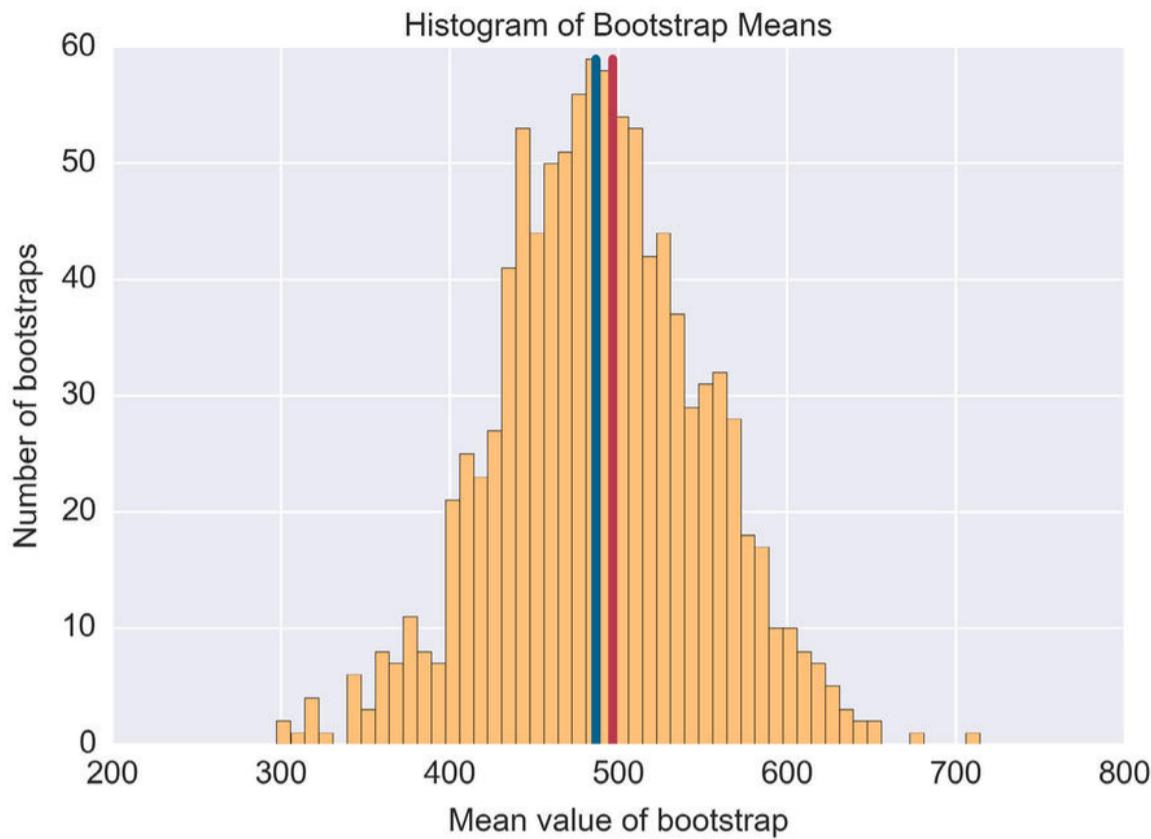


Figure 2.17: The histogram shows how many bootstraps had a mean of the given value. The blue bar at about 490 is the mean of the sample set. The red bar at about 500 is the mean of the population.

Since we know our population, we can compute its mean, which is about 500. The mean of our sample set is close to this, at about 490. The purpose of bootstrapping was 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% confident will bracket the mean of the population. Then we only need to slice off the lowest and highest 10% of the bootstrap values [Brownlee17]. Figure 2.18 shows a box that brackets the values that we are 80% confident contain the real value, which we know is 500. Reading from the graph, we could now say, “We are 80% confident that the mean of the original population is between about 410 and 560.”

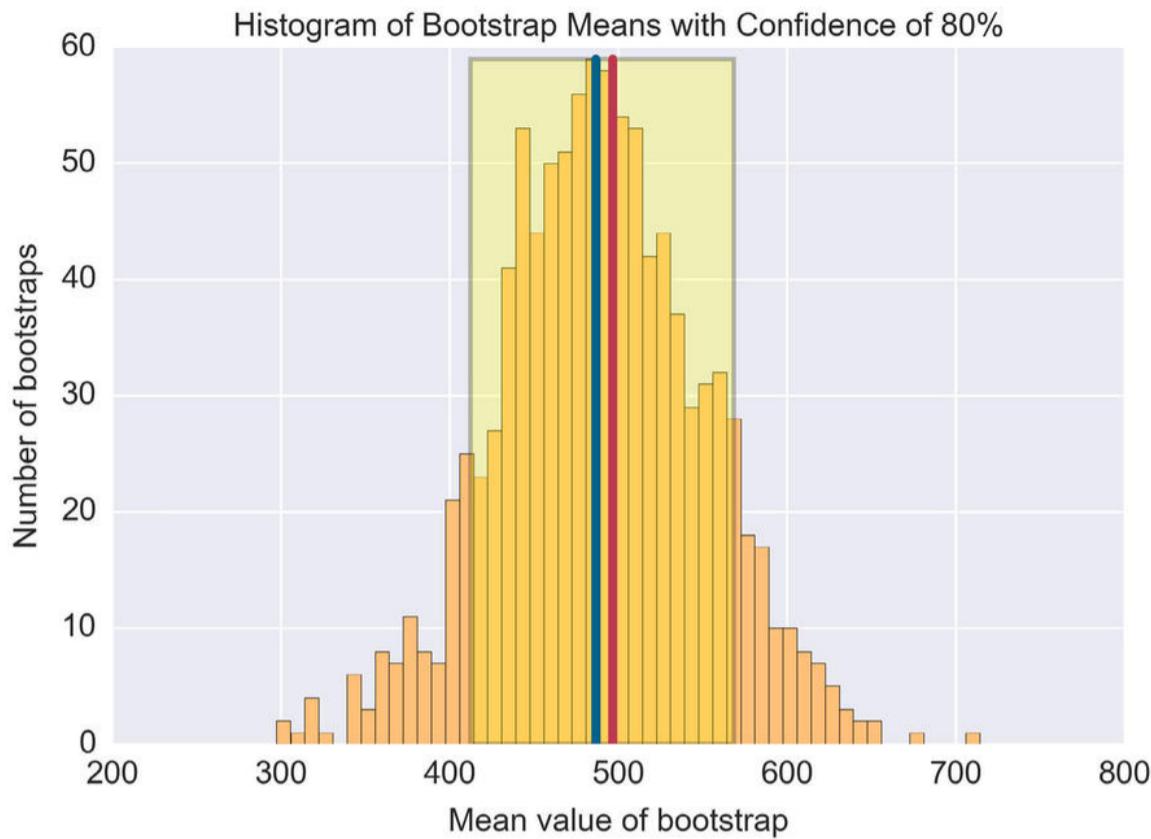


Figure 2.18: We are 80% confident that the box contains the population's mean.

To recap, we want to know some **property** describing a **population** of values (perhaps its mean, or variance). We suppose that for some reason we can't work with the whole population, so we can't measure that property directly. Instead, we pluck out some of its values to form a **sample set**. Then we take many **bootstraps**, which are each formed by choosing, with replacement, a fixed number of elements from the sample set. By looking at the statistics of the bootstraps, we're able to come up with a confidence interval that lets us say how sure we are that the value we're after lies within a particular range.

Bootstrapping is appealing because often we can use small bootstraps, perhaps only 10 or 20 elements each. This small size means that each bootstrap is 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 will look like a Gaussian bump, and the more precise we can be with our confidence intervals.

We'll use the bootstrapping idea again later in this book to help us build powerful machine-learning tools out of collections of simpler tools.

2.7 High-Dimensional Spaces

We'll often think of various objects as occupying, or "living in," one or another kind of imaginary, abstract space. Some of these spaces will have hundreds or thousands of dimensions, so there's no way we can draw them. But because we will talk about such "spaces" frequently, it's important to have a general feeling for what such a space means.

The general idea is that each dimension, or axis, of the space refers to a single measurement. If we have a piece of data that has just one measurement (say, a temperature), we can represent it with a list that is only 1 number long. Visually, we need only show the length of a line to show the size of that measurement, as in Figure 2.19. We call this line a **1-dimensional space**.



Figure 2.19: A piece of data with a single value requires only one axis, or dimension, to plot that value. Left: We usually refer to the horizontal axis with the letter X. Right: Three pieces of data, with their corresponding lines that show their size, as measured from the left end of the X axis.

If we have two pieces of information, say the temperature and wind speed, then we need two dimensions, or a list that is two items long, to hold that data. Graphically, we can use two axes, as in Figure 2.20. 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 **2-dimensional space**.

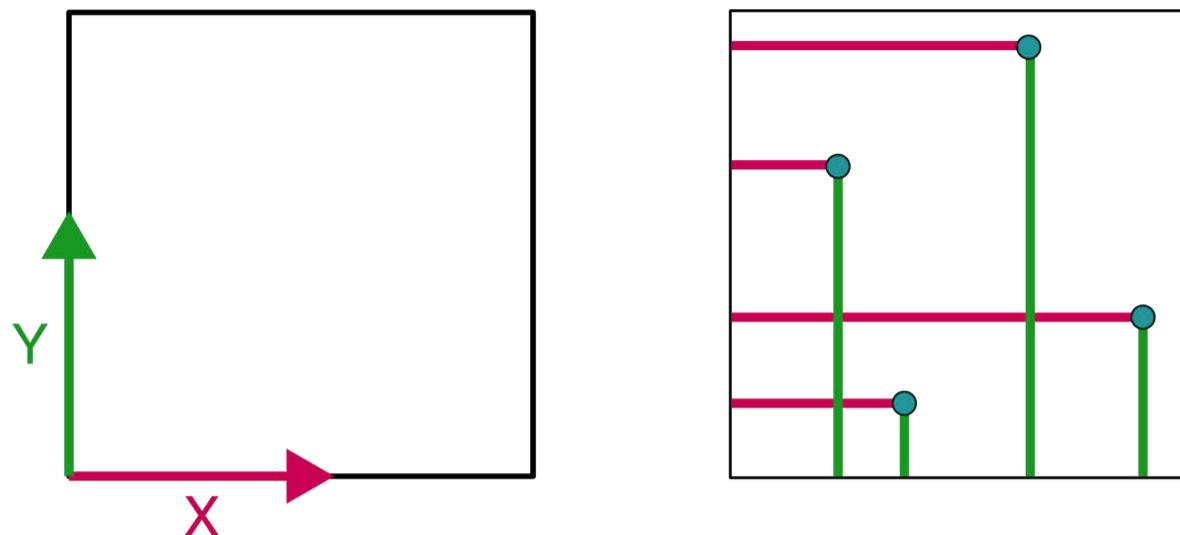


Figure 2.20: If our data has two values, we need two dimensions, or axes, to plot that data. Left: We usually refer to the vertical direction with the letter Y. Right: Four pieces of data. Each can be located in the two-dimensional space by providing two numbers, one each for X and Y.

If we have three values for every piece of data, then we need a list that can hold three values. These three dimensions can be represented using three axes, as in Figure 2.21. We call this a **3-dimensional space**.

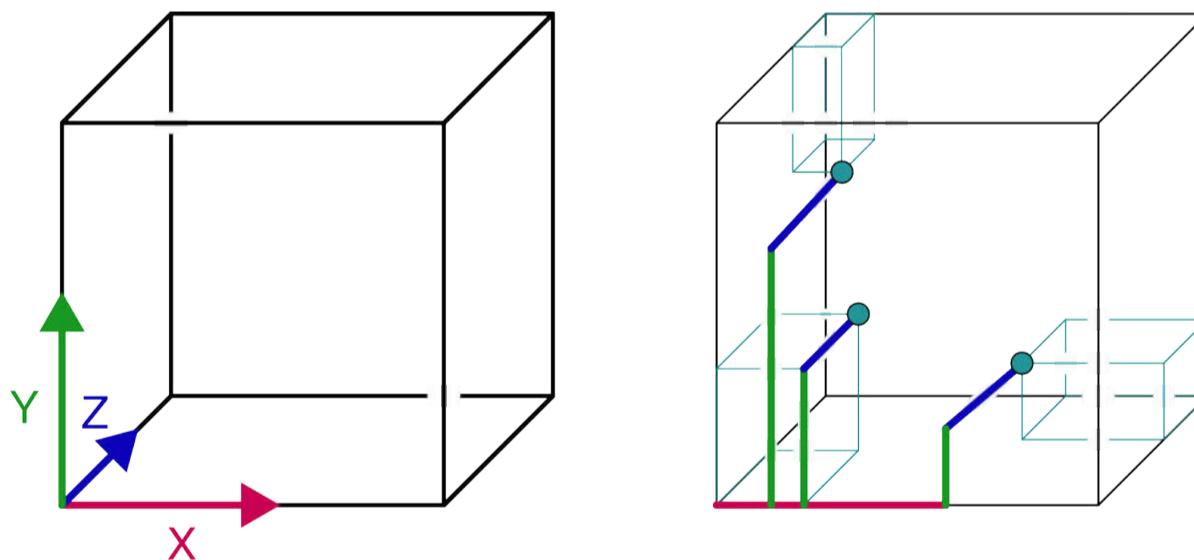


Figure 2.21: When each piece of data has three values, we need three dimensions, or axes, to draw it. Left: Conventionally, the third axis is drawn going into (or out of) the page, and is labeled Z. Right: Some points in the 3D space, and the lengths along the X, Y, and Z axes that we use to locate them. The lighter boxes are there just to help us see where the points are located in the cube.

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 [Banchoff90] [Norton14]. And once we start getting up to 5, 10, or 20 dimensions, it's pretty much a lost cause.

It might seem that these high-dimensional spaces are esoteric and rare, but in fact they're common and every day. To see that, consider what kind of space we'd need to represent a photograph.

Suppose that we have a grayscale image that is 500 by 500 pixels. How big a space would we need to represent this image? At 500 pixels on a side, the image has $500 \times 500 = 250,000$, or a quarter-million pixels. Each pixel has a value, or in the language of our discussion above, a measurement. Starting at one corner of the space, we'd move to the right (along the X axis) an amount given by the first pixel, then up (along the Y axis) by an amount given by the second pixel, then back (along the Z axis) by an amount given by the third pixel. Then we'd move along the fourth axis by an amount given by the fourth pixel, and similarly the fifth axis, the sixth axis, and so on, moving along a different axis for each pixel.

Since we have 250,000 pixels, we'd need 250,000 dimensions. That's a lot of dimensions! Each pixel would require three values, so we'd need $3 \times 250,000 = 750,000$ dimensions.

There's no way we can draw a picture with 750,000 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 is a *single point* in this vast space. Just as a 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 vast space. We can say that our image is represented by a single point in an enormous “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 3 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, even images with nearly a million of them. Things go slower when there's more data involved, but the algorithms work without any modification.

In this book we'll frequently use 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 un-visualizable) analogies of our line, square, and cube, where each piece of data is represented by a point.

2.8 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 identify how strongly two variables are related.

2.8.1 Covariance

Suppose that we have two variables, and we notice a specific pattern. 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 *co-vary*. We measure the strength of the connection between two variables with a number called the **covariance**.

If we find that when one value increases the other does as well, then the covariance is a positive number. The more they tend to remain in lock step, the larger the covariance.

The classic way to talk about covariance is to draw a plot where we show points on a 2D graph as in Figure 2.22. This kind of plot is called a *scatter diagram*. The axes are labeled x and y, but those are just stand-ins for whatever two variables we are interested in comparing.

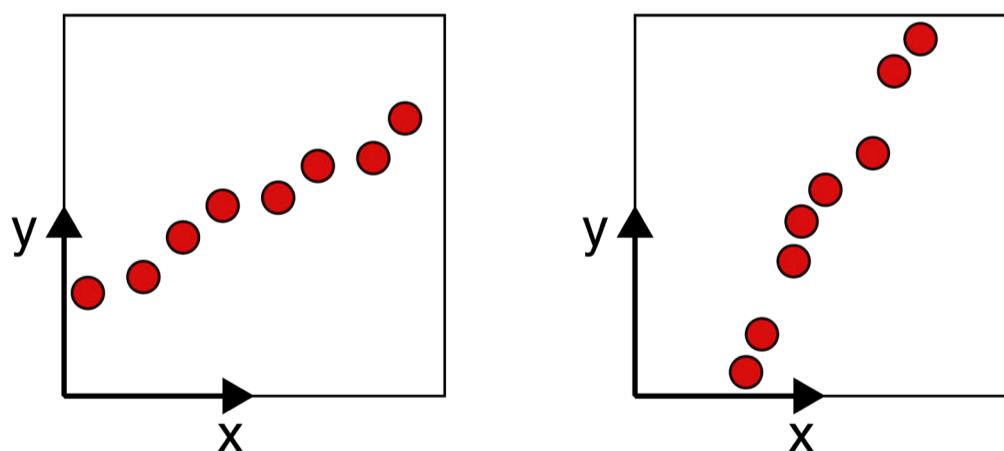


Figure 2.22: Illustrating covariance. (a) The change in y between each pair of points working left to right is about the same. This is positive covariance. (b) The change in x is a little more variable between each pair of points, indicating weaker positive covariance.

Let's say x is our first value, and y is our second. So if every time x increases (that is, we move one point to the right) we find the y also increases (that is, the point has moved upwards), we say that the two variables are demonstrating **positive covariance**. The more consistently the change in y tracks the change in x , the stronger the 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 will change by a different but consistent amount as well.

On the other hand, if one value *decreases* whenever the other increases, we say the variables have **negative covariance**. Figure 2.23 shows some examples.

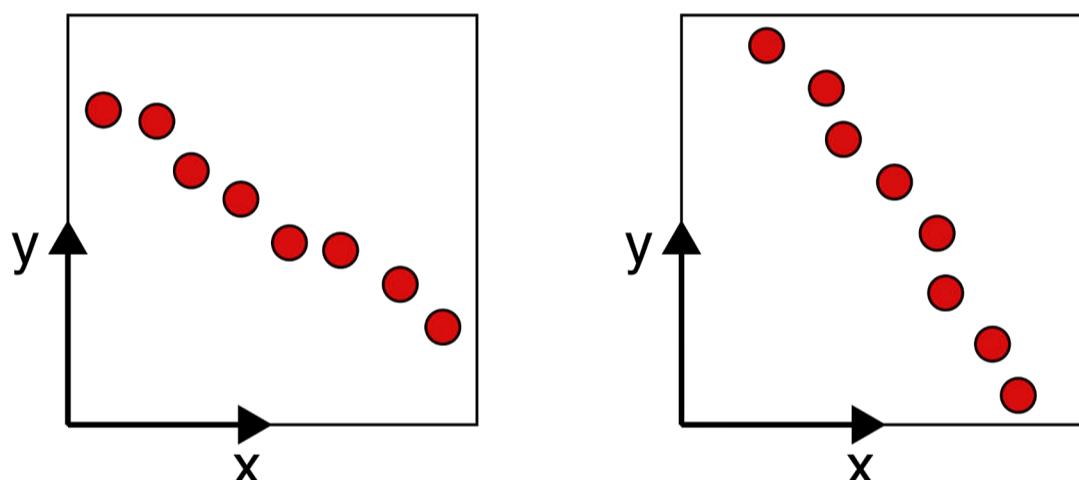


Figure 2.23: The change in y between two neighboring points in x is always about the same, but as x becomes more positive y becomes more negative. This type of connection is called negative covariance.

If the two variables have no such consistently matched motion, then the covariance is zero. Figure 2.24 shows some examples.

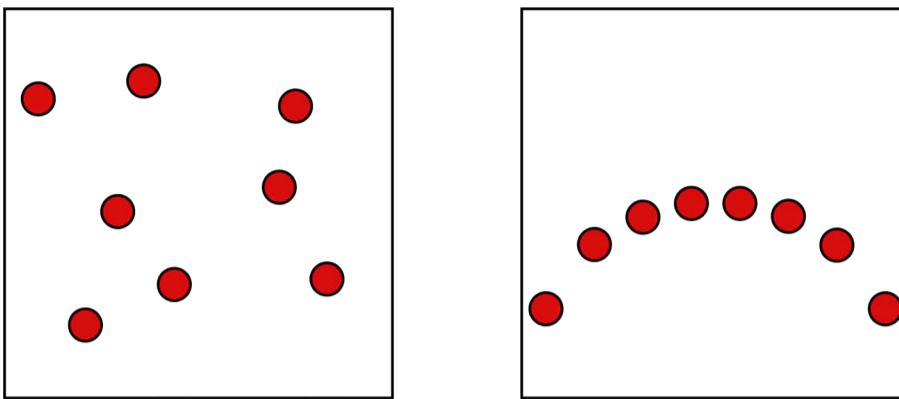


Figure 2.24: Each of these sets of data points has zero covariance. As we move from one point to the next one along the X axis, the change in the Y value is inconsistent in both magnitude and direction.

Our idea of covariance only captures relationships between variables when their changes are multiples of each other. The graph on the right of Figure 2.24 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.

2.8.2 Correlation

Covariance is a useful concept, but it has a problem. Because of the way it's defined, it doesn't take into account the units of the two variables, which makes it hard for us to decide how strong or weak a correlation is.

For example, suppose we measured a dozen variables on 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 wouldn't be able to compare them to find which pairs had the strongest and weakest relationships. The problem is that they're in different units: the thickness of the wood might be in millimeters, the time a string resonates is in seconds, and so on. We'd get a number for the covariance of each pair of measurements, but we wouldn't be able to compare them to one another.

The *sign* of the covariance is all that we really learn: a positive value means a positive relationship, a negative value means a negative relationship, and zero means no relationship.

Having only the sign of the covariance is a problem, because we'd really like to be able to compare different sets of variables. Then we could find out useful information such as which variables are the most and the least strongly positively and negatively correlated.

To get a measure that will let us make these comparisons, we can compute a slightly different number called the **correlation coefficient**, or just the **correlation**. This value is just the covariance but with 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 .

Because correlation avoids the problem of units, it's the right tool when we want to compare the strength of the relationship between different sets of variables.

Because the correlation can never be more than $+1$ or less than -1 , we can focus our attention just on those extremes and the values in between. 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 2.25.

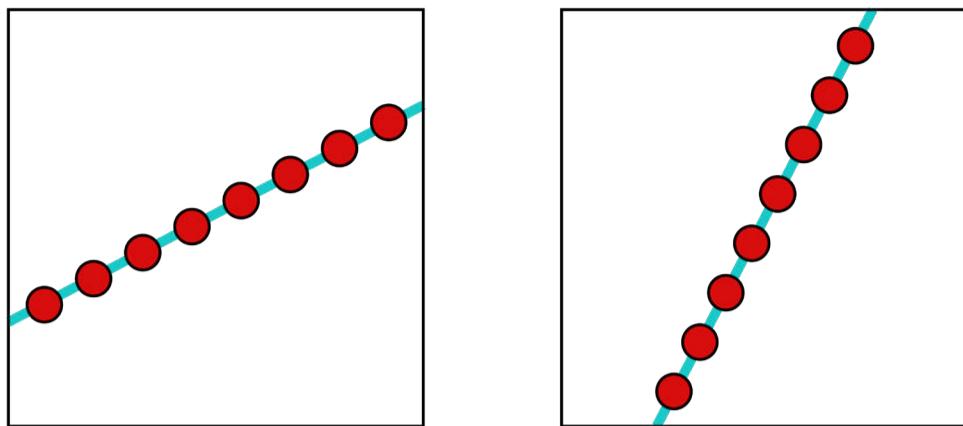


Figure 2.25: Every time we move right, we move up by the same amount. Both plots show perfect positive correlation, or a correlation of +1.

What kind of relationship between points would give us a positive correlation, but somewhere in the range between 0 and 1? The y value would continue to increase with x, but the proportion wouldn't be constant. We might not even be able to predict how much it will change, but we know that increases in x will cause increases in y, and decreases in x will cause decreases in y. Figure 2.26 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*).

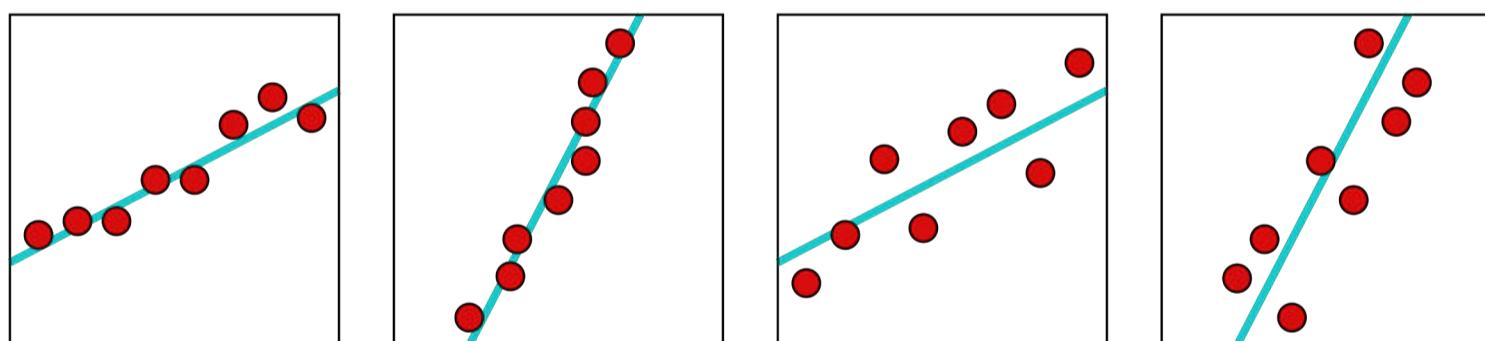


Figure 2.26: Examples of decreasing positive correlation, starting with a value of nearly 1 at the far left, then lower values successively as we move right. Generally speaking, the closer the points are to lying on a straight line, the higher the correlation.

Let's now look at a value of zero. A zero correlation means that there is no relationship between a change in one variable a change in the other. We just 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 2.27 shows some data with zero correlation.

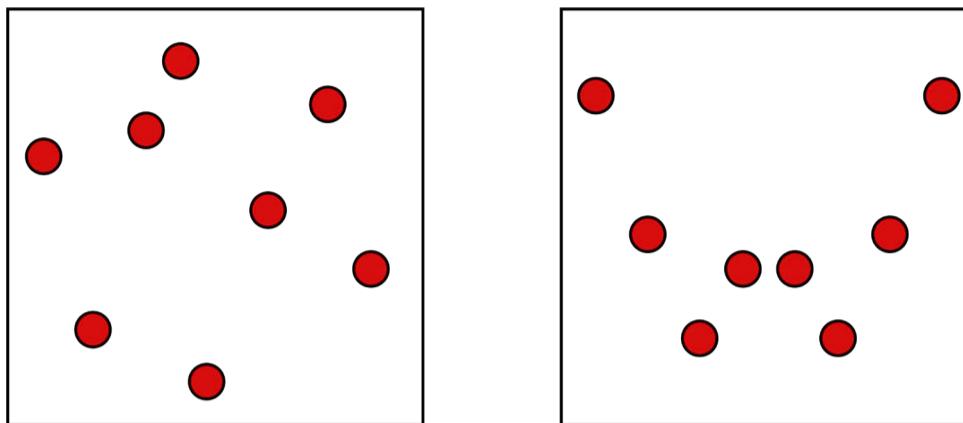


Figure 2.27: These patterns have zero correlation. Each time we take one step to the right, there is no consistent motion vertically compared to all the other single steps to the right.

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 Figure 2.28.

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*).

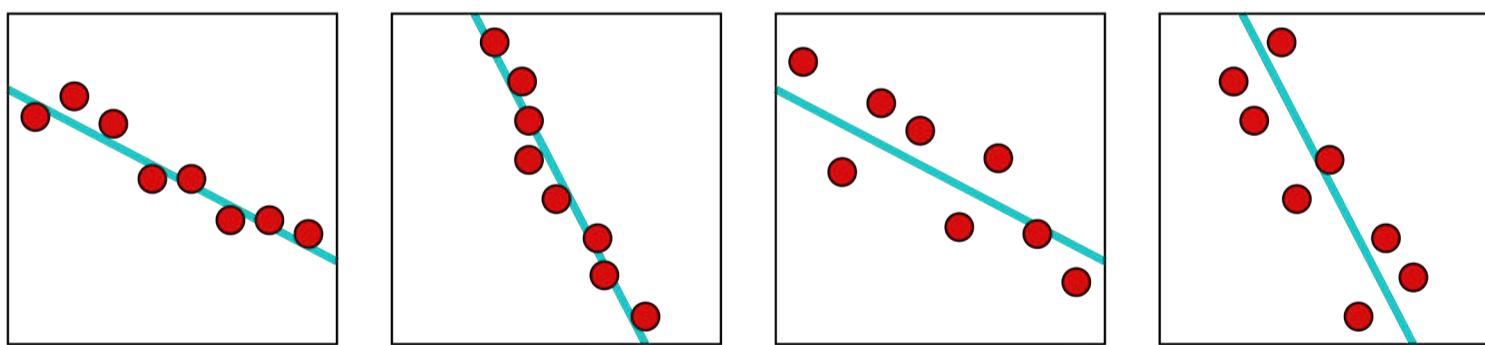


Figure 2.28: The pattern at the far left has a negative correlation of nearly -1 . As we move to the right, the amount of negative correlation moves closer to 0.

Finally, Figure 2.29 shows perfect negative correlation, or a correlation of -1 .

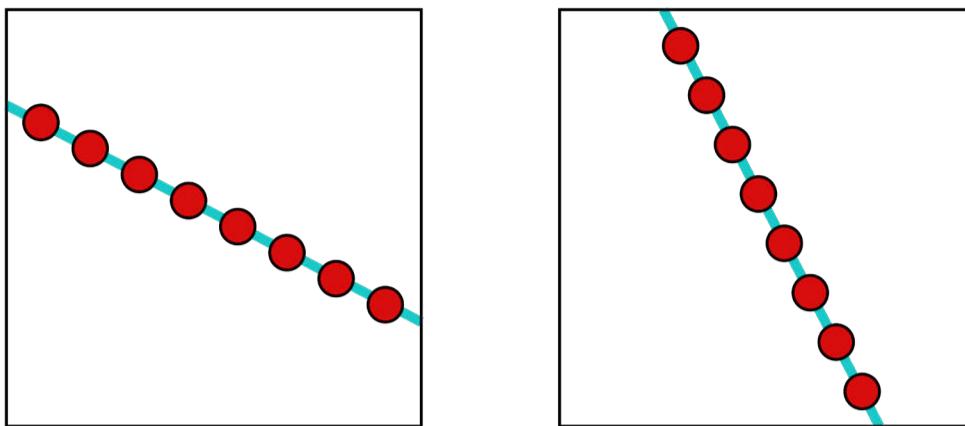


Figure 2.29: These patterns have a perfect negative correlation, or a correlation of -1 . Each time we move to the right, we move downward by the same amount.

A few other terms are worth mentioning, because they pop up in documentation and literature from time to time. Our discussion of two variables above is usually called **simple correlation**. We could find the relationship between more variables, 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 **non-linearly correlated**.

Figure 2.30 summarizes the meanings of different values of linear correlation.

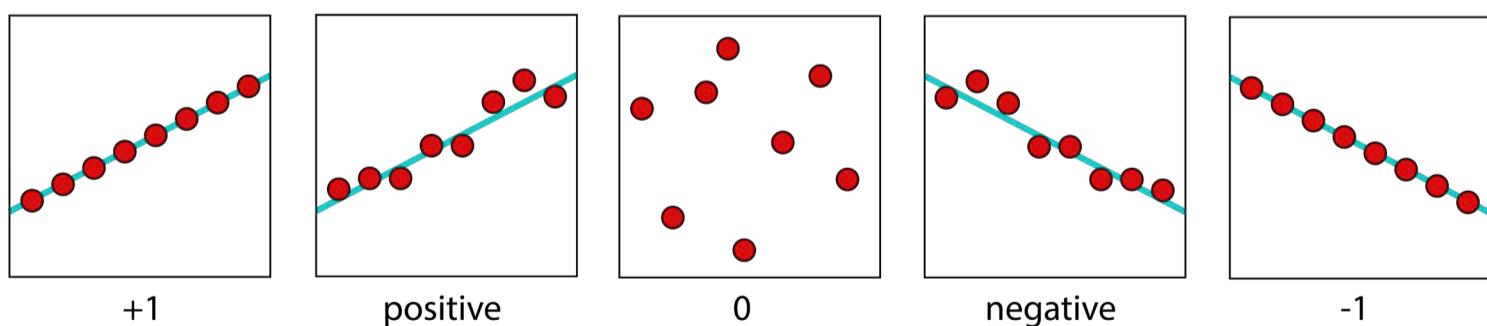


Figure 2.30: Summarizing the meanings of different values of linear correlation.

2.9 Anscombe's Quartet

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 [Anscombe73]. The values of the four data sets are widely available online [Wikipedia17a].

Figure 2.31 shows the four data sets in this quartet, along with the straight line that best fits each set.

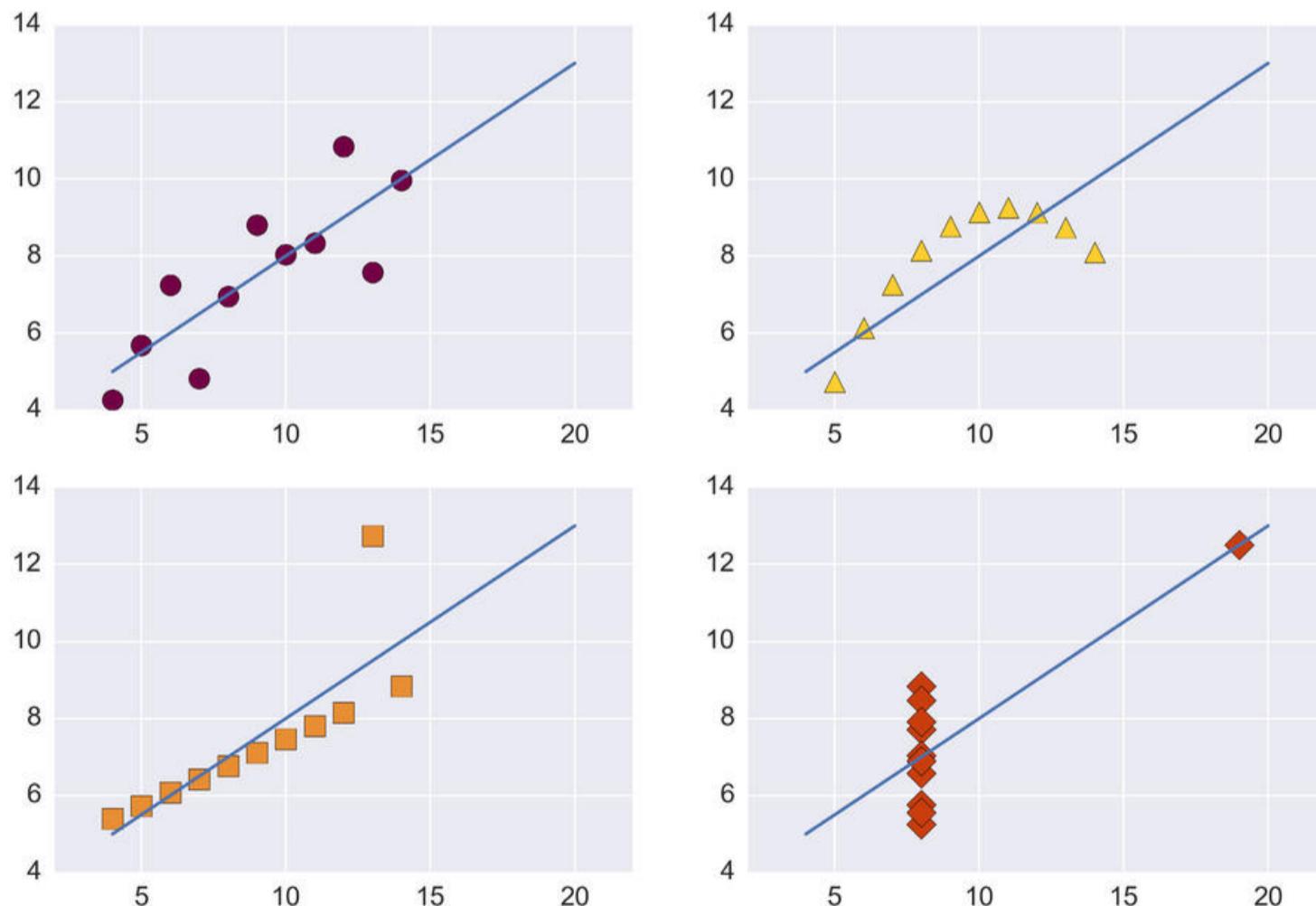


Figure 2.31: The four data sets in Anscombe's quartet, and the straight lines that fit them best.

The amazing thing about these four different sets of data is that the mean of the X values in each data set is 9.0. The mean of the Y values in each data set 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 data set is 0.82. And the best straight line through each data set has a Y-intercept of 3 and a slope of 0.5.

In other words, all seven of these statistical measures have the same value for all four sets of points. Actually, some of the statistics differ from one another when we look farther out into more digits, but they're so close that we can consider them nearly the same.

Figure 2.32 superimposes all four sets of points, and their best straight line approximations. All four lines are identical, so we only see one line in the plot.

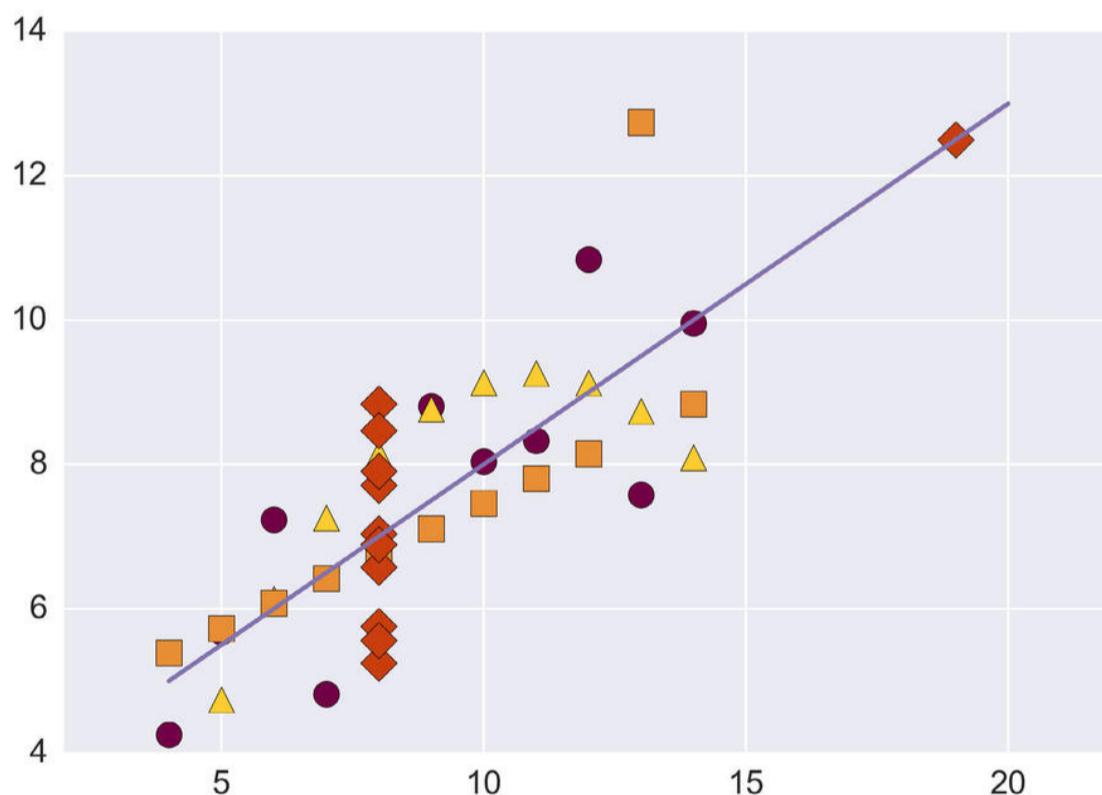


Figure 2.32: The four data sets of Anscombe's quartet and their best straight-line fits, superimposed.

The moral is that we shouldn't assume that statistics tell us the whole story about any set of data. They're a great place to start, but to use our data well we need to look at it closely and understand it deeply. Statistical measurements don't tell us everything we need to know.

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 [Matejka17].

References

- [Anscombe73] F.J. Anscombe, “Graphs in Statistical Analysis,” American Statistician, 27. 1973.
- [Austin17] David Austin, “Random Numbers: Nothing Left to Chance”, American Mathematical Society, 2017. <http://www.ams.org/samplings/feature-column/fcarc-random>
- [Banchoff90] Thomas F. Banchoff, “Beyond the Third Dimension: Geometry, Computer Graphics, and Higher Dimensions”, Scientific American Library, W H Freeman, 1990.
- [Brownlee17] Jason Brownlee, “How to Calculate Bootstrap Confidence Intervals for Machine Learning Results in Python”, Machine Learning Mastery blog, 2017. <https://machinelearningmastery.com/calculate-bootstrap-confidence-intervals-machine-learning-results-python/>
- [Efron93] Bradley Efron and Robert J. Tibshirani, “An Introduction to the Bootstrap”, Chapman and Hall/CRC Monographs on Statistics & Applied Probability (Book 57), 1993.
- [Marsaglia02] George Marsaglia and Wai Wan Tsang, “Some Difficult-to-pass Tests of Randomness”, Journal of Statistical Software, Volume 7, Issue 3, 2002. <http://www.jstatsoft.org/v07/i03/paper>

- [Matejka17] Justin Matejka, George Fitzmaurice, “Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics through Simulated Annealing”, Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, pgs 1290–1294, 2017. <https://www.autodeskresearch.com/publications/samestats>
- [Norton14] John D. Norton, “What is A Four Dimensional Space Like?”, Department of History and Philosophy of Science, University of Pittsburgh, 2014. http://www.pitt.edu/~jdnorton/teaching/HPS_0410/chapters_2017_Jan_1/four_dimensions/index.html
- [Ong14] Desmond C. Ong, “A Primer to Bootstrapping”, Department of Psychology, Stanford University, 2014. <https://web.stanford.edu/class/psych252/tutorials/doBootstrapPrimer.pdf>
- [Random17] Random.org, “Random Integer Generator”, 2017. <https://www.random.org/integers/>
- [Schwab17] Katharine Schwab, “The Hardest Working Office Design in America Encrypts Your Data - With Lava Lamps”, Co.Design blob, 2017. <https://www.fastcodesign.com/90137157/the-hardest-working-office-design-in-america-encrypts-your-data-with-lava-lamps>
- [Stiles16] Matt Stiles, “How Common is Your Birthday? This Visualization Might Surprise You”, The Daily Viz, 2016. <http://thedataviz.com/2016/09/17/how-common-is-your-birthday-dailyviz/>
- [Wikipedia17a] Wikipedia authors, “Anscombe’s Quartet”, 2017. https://en.wikipedia.org/wiki/Anscombe%27s_quartet
- [Wikipedia17b] Wikipedia authors, “Random Number Generation”, 2017. https://en.wikipedia.org/wiki/Random_number_generation
- [Wikipedia17c] Wikipedia authors, “Random Variable”, 2016. https://en.wikipedia.org/wiki/Random_variable

Chapter 3

Probability

Ways to describe how much confidence we have in our data, so we will know how much to trust the conclusions our systems produce.

Contents

3.1 Why This Chapter Is Here.....	99
3.2 Dart Throwing	100
3.3 Simple Probability	103
3.4 Conditional Probability.....	104
3.5 Joint Probability.....	109
3.6 Marginal Probability.....	114
3.7 Measuring Correctness	115
3.7.1 Classifying Samples.....	116
3.7.2 The Confusion Matrix	119
3.7.3 Interpreting the Confusion Matrix	121
3.7.4 When Misclassification Is Okay	126
3.7.5 Accuracy.....	129
3.7.6 Precision	130
3.7.7 Recall	132
3.7.8 About Precision and Recall	134
3.7.9 Other Measures.....	137
3.7.10 Using Precision and Recall Together	141
3.7.11 f1 Score	143
3.8 Applying the Confusion Matrix	144
References	151

3.1 Why This Chapter Is Here

Probability is a set of tools that let us decide how likely something is to happen. A probability of 0% means it cannot happen, while 100% means it's a sure thing. We can also use probability to express confidence, such as being 80% sure that a piece of fruit is ripe, or that a certain team will win a game.

Probability is one of the pillars on which machine learning is built. Many papers describe their techniques with the language of probability, and lots of documentation follows suit. For instance, some library functions require their input data to have some basic probabilistic properties. To understand how to use libraries properly to get good results, we need to know enough about probability to read and understand those descriptions.

Probability is an enormous subject, with many deep specialties. As with any deep subject, new skills in probability lead to new mastery, but the more we know the more we realize how much is left to learn!

Since our focus is on using machine learning tools sensibly and well, we only need command of a few basic terms and topics. This chapter covers those ideas. In other words, this chapter isn't about the tools we might use to calculate our own statistics, but about the concepts that let us confidently understand how to use tools that depend on our data conforming to their expectations.

In this chapter we'll focus on the basic ideas of probability, the measures that we often use, and a particular application called the **confusion matrix**. 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 books dedicated to probability [Jaynes03] [Walpole11].

3.2 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 we land in (where the background counts as a region as well). The idea is illustrated in Figure 3.1.

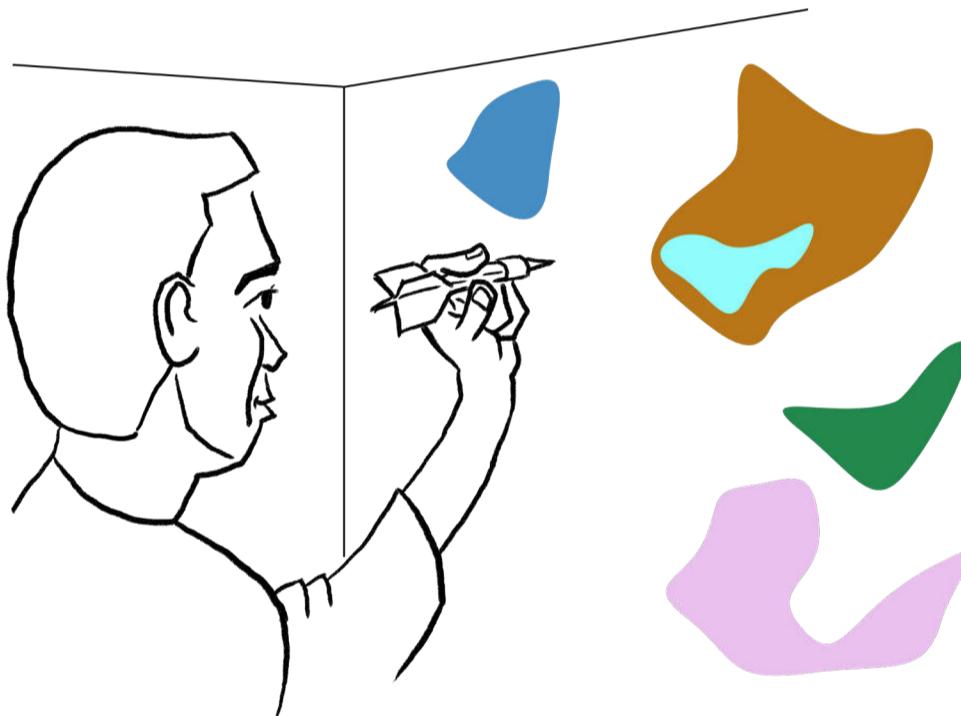


Figure 3.1: Throwing darts at a wall. The wall is covered in blobs of paint of different colors.

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%. It's a sure thing.

In this chapter, we'll use both real numbers and percentages for probabilities. So a probability of 1.0 would be a percentage of 100%. A probability of 0.75 would be a percentage of 75%, 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 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*. In other words, the probability of striking any given point is given by a uniform distribution, as we discussed in Chapter 2. This isn't terribly realistic, but we're just using dart throwing as a metaphor, so we don't need it to be physically exact. The restriction may be easier to get comfortable with if we think of throwing our darts at a very small wall from a very great distance.

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 circle on the wall. When we throw a dart, we know it will hit the wall with a probability of 1. What's the probability of hitting the circle?

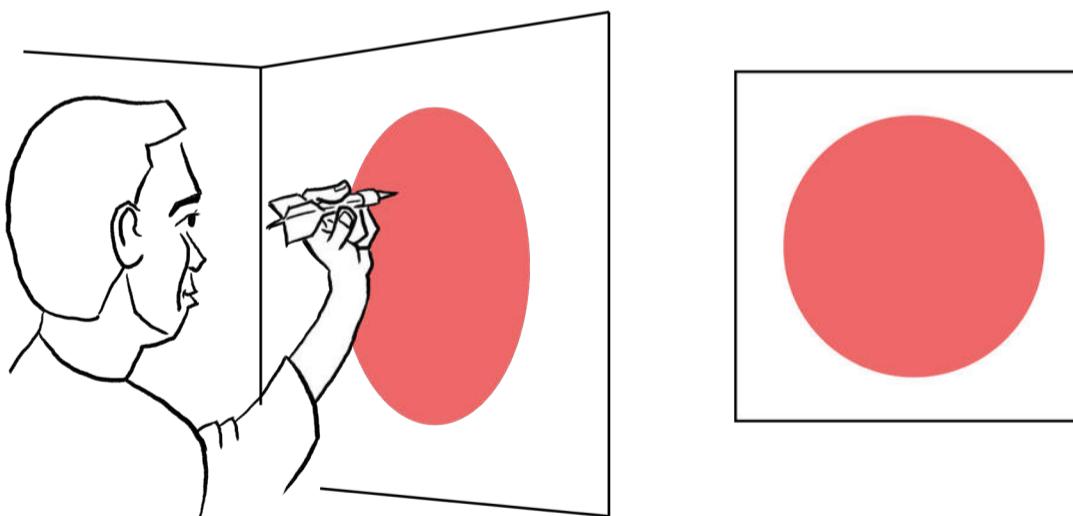


Figure 3.2: We're guaranteed to hit the wall. What's the probability that we'll hit the circle?

Let's say that the wall has an area of 2 square meters, and the circle has an area of 1 square meter. Then the circle covers half of the wall's 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% chance, or 0.5 probability, of the dart landing in the red circle. It's just the ratio of the areas. The larger our circle, the more points 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. We'll draw one shape above the other, and the chance of hitting the upper shape is the area of the circle relative to the area of the wall.

For example, Figure 3.3 shows the ratio for our circle with respect to the wall. The area of the circle is 1, and the area of the wall is 2, so the ratio is $1/2$ or 0.5. The exact numbers aren't what's important here. Instead, the drawings just give us a visual way to track which areas we're talking about, and get an intuitive feel for their relative sizes.

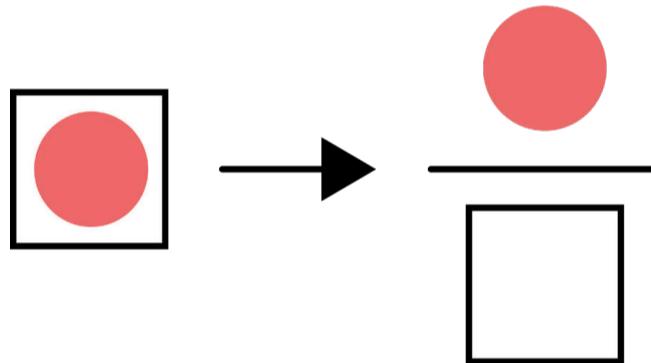


Figure 3.3: The probability of hitting the circle in Figure 3.2 is given by the ratio of the area of the circle to the area of the wall, here shown as a symbolic “fraction.”

Figure 3.3 shows the relative areas accurately, so the area of the red circle is really $1/2$ 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 is to illustrate the comparison of the area of one shape relative to the area of another.

3.3 Simple Probability

When we talk about probabilities, we often refer to abstract events named with capital letters, such as A, B, C, and so on. “The probability of A” simply means the probability of event A happening. We can represent this probability by drawing a circle on our wall. The area of the circle relative to the area of the wall is the chance that event A will happen.

Recall that when we throw a dart at the wall, it will always hit the wall *somewhere*, and it has an *equal chance* of hitting the wall anywhere. So if event A is that the dart landed in the circle, the phrase “the probability of A happening” is just a different way of stating the probability that when we throw a dart at the wall, it will land in the circle. Figure 3.4 shows this graphically.

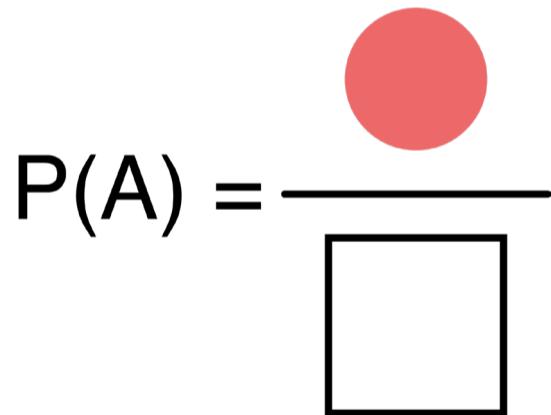


Figure 3.4: We'll say that hitting the circle with our dart is “event A.” The probability of event A occurring is given by the symbolic ratio of areas in Figure 3.3. We write this probability as $P(A)$.

To save some space, rather than write “the probability of event A happening” we usually just say “the probability of event A.” To save even more words, we usually write “the probability of event A” as $P(A)$ (some authors use a lower-case p, writing $p(A)$).

In our case, $P(A)$ is the area of the circle divided by the area of the wall: the probability that, when throwing a dart, we'll hit the circle rather than the rest of the wall.

We call $P(A)$ the **simple probability**. Sometimes it's also called the **marginal probability** of A. We'll later see where the word "margin" comes from.

3.4 Conditional Probability

Let's now talk about two events, not just one. Either of these events might happen, or both of them, or neither of them.

For example, we might ask for the probability that someone is thirsty, given that they are drinking water.

These events are related, but they don't have to happen together. Someone might drink water for many different reasons, such as to soothe a cough, or to swallow a pill. And of course people can be thirsty or not when they drink for these reasons. We say that two events that are not contingent on one another in any way **independent**.

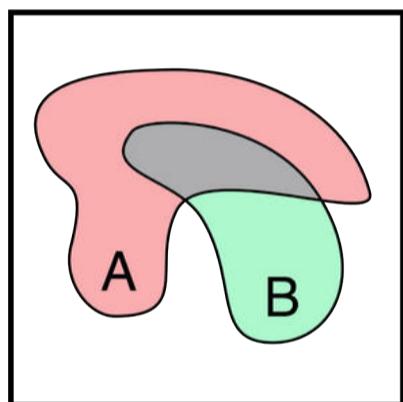
Our goal is to find the probability of one kind of event, given that another event has happened (or is happening). So we might ask, "What is the probability that someone is thirsty, *given* that they're drinking water?" or "What is the probability that someone is drinking water, *given* that they're thirsty?"

In other words, we know that one event has happened, and we want to know the probability of the other.

To keep things short, let's write the two events A and B as above. A will be "is thirsty," so if A is true, then the person is thirsty, and if A is false, they aren't. B is "drinking water," so if B is true, they're drinking water, and if B is false, they aren't. As before, we can talk about $P(A)$, which in this case is a shortcut for the probability that the person is thirsty, and $P(B)$, the shortcut for the probability that the person is drinking water.

When we want the probability that A is true given that we already know that B is true, we write this as $P(A|B)$. This is called the **conditional probability** of A given B. In words, “the probability that A is true, given that B is true,” or more simply, “the probability of A given B.” If we’re interested in the other situation, we can 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.



$$P(A|B) = \frac{\text{Area of } A \cap B}{\text{Area of } B}$$

Figure 3.5: The conditional probability tells us the chance that one thing will happen, given that another thing has already happened. In this case, we want to know the probability that our dart landed in blob A, given that we already know it landed in blob B. 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.

This picture tells us how to find the probability of the dart being in area A, given that we know it landed in area B: it’s the ratio of the area of the overlap of A and B, divided by the area of B.

$P(A|B)$ is a positive number that we can estimate by using our darts. The thinking is that each dart will land in one of the regions of the wall (in this case, there are four regions: A, B, the overlap of A and B, and the wall outside of them). We can get a rough value for $P(A|B)$ by counting all the darts that land in the overlap of A and B, and all those

that land in any part of B. The ratio of the darts in the overlap area to the darts that land in any part of B tells us roughly the likelihood that a dart landing in B will have also landed in A.

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 so small that we can consider them to be nothing but a point, but that makes them hard to see. So we'll show the location of each dart's impact by a black circle, where the center of the circle shows where the dart struck.

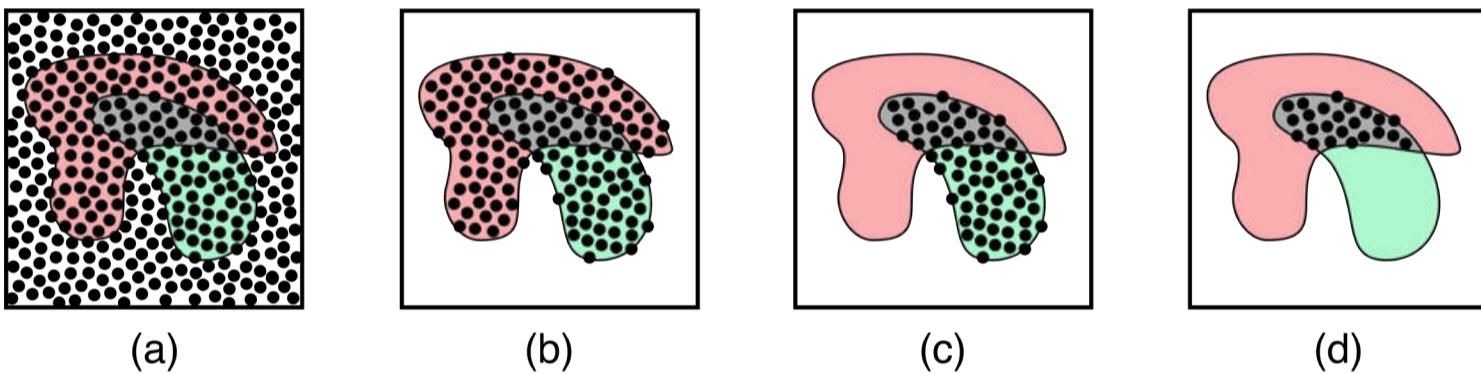


Figure 3.6: Throwing darts at the wall to find $P(A|B)$. (a) Darts striking the wall. (b) All the darts in either A or B. (c) The darts only in B. (d) The darts that are in the overlap of A and B.

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 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.

Now we can see the reason why we earlier said that every point on the wall has the same likelihood of being hit: it lets us use the number of darts in any area as an estimate of the relative size of that area.

Note that this doesn't tell us 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 was 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, so the probability of being in both is 100%, 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. Thus the area of the overlap of A and B is the same as the area of 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%, or $P(A|B)=0$.

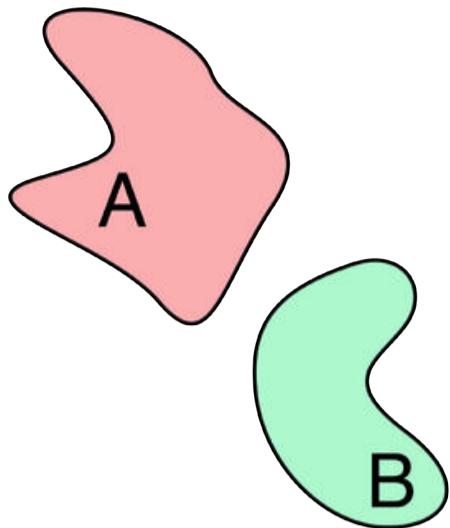


Figure 3.8: Left: Another two new blobs on the wall. Right: The probability of landing in A given that we're in B is 0, because there's no overlap between A and B. The symbolic ratio shows that the area of overlap is 0, and 0 divided by anything is still 0.

For fun, let's flip this around the other way, and ask about $P(B|A)$, 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.

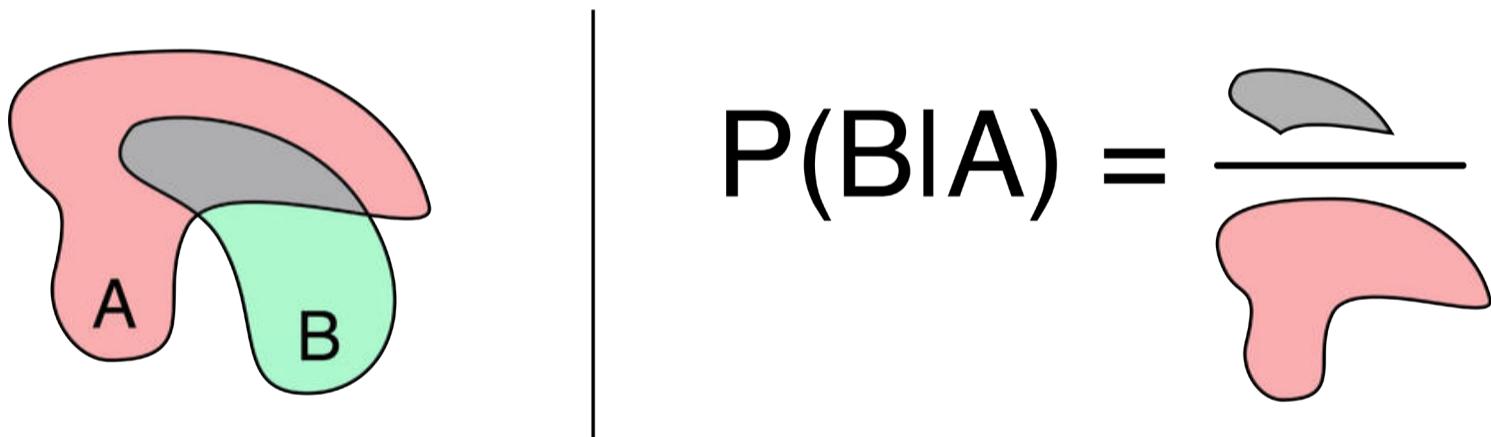


Figure 3.9: The conditional probability $P(B|A)$ is the probability we landed in B, given that we landed in A. So we find the area of overlap and divide it by the area of A.

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.

Note that the order in which we name the regions 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.

We can use an expression like $P(A|B)$ without knowing what its value is. We might not even know what the shapes of A and B look like, much less what their areas are. When we say that something like $P(A|B)$ “tells us” or “gives us” the probability of A given B, without explicitly providing A and B at the same time, we mean that somewhere down the road, when we get around to figuring out what A and B are, we know that we can calculate $P(A|B)$ from their areas.

In other words, the expression $P(A|B)$ is a kind of shorthand, or specification, for an algorithm that takes A and B as inputs, and returns a probability.

The same thing is true for $P(A)$ and $P(B)$ and all the other expressions like them that we’ll see in this chapter. They’re references to algorithms that produce values, which we can compute when we get their inputs.

3.5 Joint Probability

In the last section we saw a way to find the probability of one event happening given that another event had already occurred. What if we don’t know if either event has happened yet?

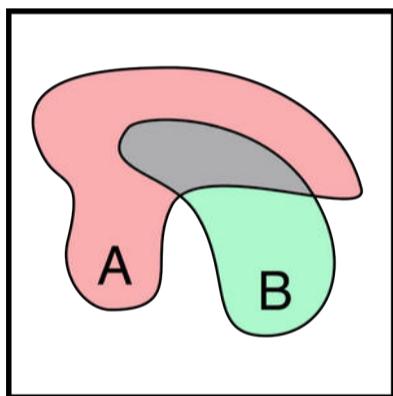
It would be helpful to know the probability of both things happening at once, without already knowing about either one.

Returning to our water-drinking example, what’s the chance that someone is both thirsty *and* drinking water? 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 should think of the comma as meaning the word “and.” That is, $P(A,B)$ is the bit of notation that we write down to represent the probability that *both* A and B are true. We read it out loud as “the probability of A and B.”

We call this the **joint probability** of A and B.

Using our blobs, we can find this joint probability $P(A,B)$ by finding the area of the overlap of blobs A and B relative 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 else. Figure 3.10 shows this idea.



$$P(A,B) = \frac{\text{Area of Overlap}}{\text{Area of Wall}}$$

Figure 3.10: The probability that both A and B will occur is called their joint probability, written $P(A,B)$. We can find it from the area of the overlap of regions A and B, divided by the area of the entire wall. So $P(A,B)$ is just the name for the chance that any randomly-thrown dart will land in both A and B at the same time.

There’s another way to look at this that’s a little more subtle, but interesting and useful. It’s based on our conditional probability from the last section.

Let’s take the conditional probability $P(A|B)$. That’s the probability of hitting A, knowing that we hit B. Let’s flip this around conceptually, observing that if we *know* we hit B, then this tells us the probability that we hit A as well. And we know the odds that a random dart hit B is just $P(B)$.

Let's think this through. Suppose that blob B covers half of the wall (that is, $P(B) = 1/2$), and that blob A covers a third of blob B (that is, $P(A|B) = 1/3$). Then half of our darts thrown at the wall will land in B, and a third of those will fall in A. So suppose we throw 60 darts. Half, or 30, fall in blob B. Then a third of those, or just 10, also fall in A. So our probability of A *and* B, or $P(A,B)$, is 10/60 or 1/6.

This example shows us the general rule: to find $P(A,B)$ we find $P(A|B)$ and multiply that with $P(B)$. This is really quite remarkable: we just found $P(A,B)$ using only $P(A|B)$ and $P(B)$! We write this as $P(A,B) = P(A|B) \times P(B)$. In practice, we usually leave off the explicit multiplication sign, writing just $P(A,B) = P(A|B) P(B)$. In an expression like this, multiplication is implied.

Figure 3.11 shows what we just did using our little area diagrams.

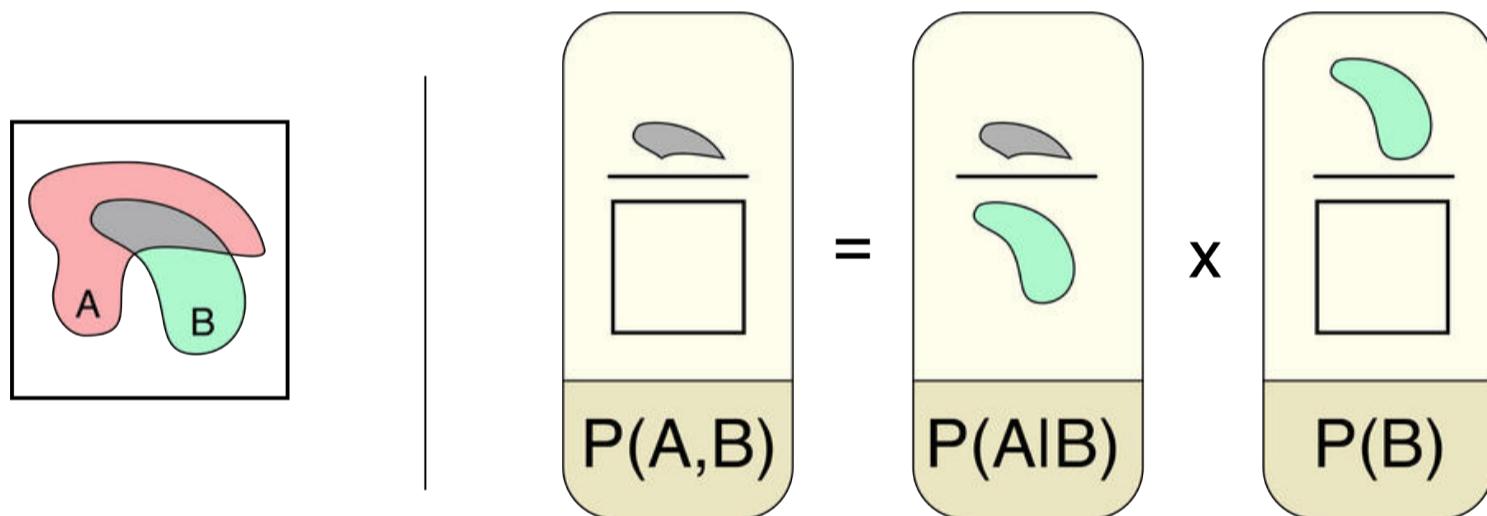


Figure 3.11: Another way to think about the joint probability $P(A,B)$, which is the probability that our dart will land in both A and B, or the probability that both A and B occur. in Figure 3.10.

The trick in Figure 3.11 is to treat the little symbolic ratios as actual fractions, which lets us cancel the green blobs of area B. To see this explicitly, in the third line of Figure 3.12 we've re-arranged the terms on the right hand side of the equals sign. The rightmost fraction in that line is the area of B divided by the area of B, which is 1, so that term can be ignored. That leaves the left and right sides equal.

$$P(A,B) = P(A|B) \times P(B)$$

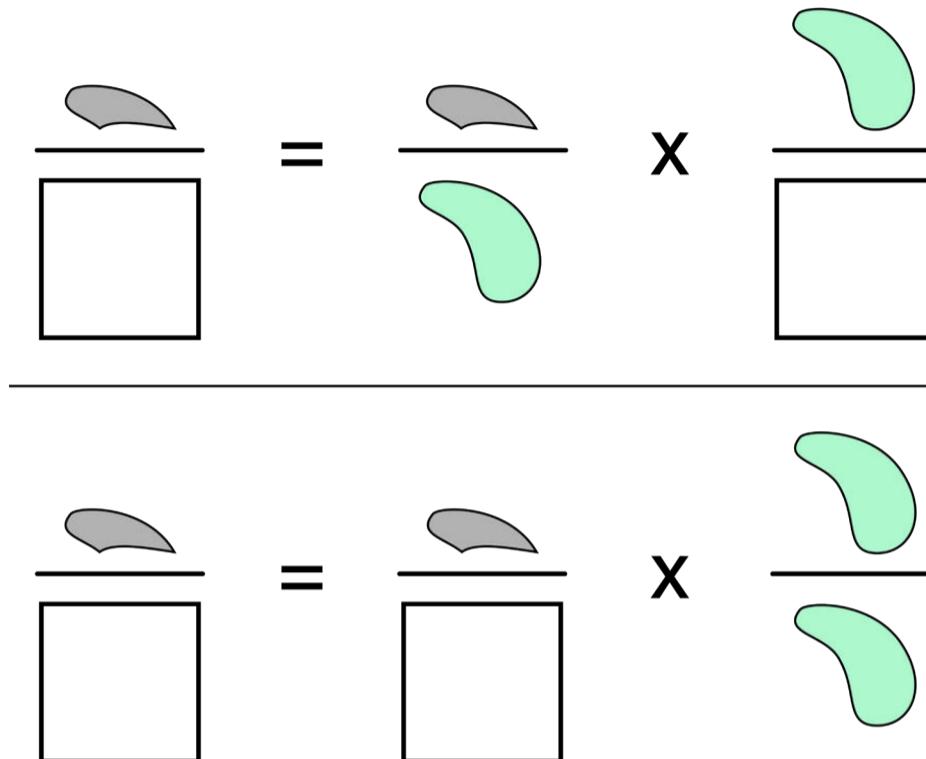


Figure 3.12: We can rearrange Figure 3.11 to better see how the areas of B cancel each other. Top: Our original formula. Middle: The little symbolic version we saw in Figure 3.11. Bottom: We've re-arranged the two symbolic fractions on the right, as though they were actual fractions. The term on the far right is just 1, showing that the left and right sides are the same.

We can do this the other way around, too. We can start with $P(B|A)$ to learn the probability of landing in B given that we landed in A, and then multiply that by the probability of landing in A, or $P(A)$. The result is $P(A,B) = P(B|A) P(A)$. Figure 3.13 shows this visually. The result is the same as before, which is what we'd expect. In symbols, $P(B,A) = P(A,B)$, since both refer to the probability of landing in A *and* B simultaneously. In this expression, that the order of A and B doesn't matter.

$$P(A,B) = P(B|A) \times P(A)$$

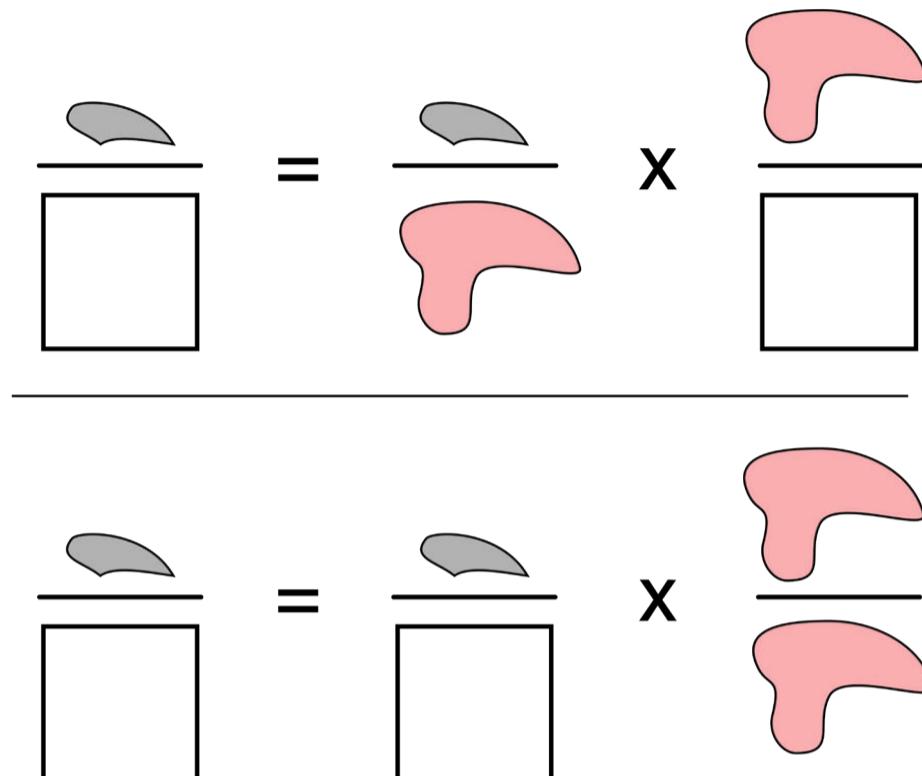


Figure 3.13: The joint probability of $P(A,B)$ found just like we did in Figure 3.12, but this time starting with the probability of our dart landing in A, and then multiplying that by the probability that we landed in B as well, given that we landed in A. Top: Our starting formula. Middle: The symbolic representation of the formula. Bottom: The same type of rearrangement as in Figure 3.12, showing that the formula is correct.

These ideas can be a little challenging to get used to. It often pays off 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, let's imagine an ice-cream shop where people can buy different flavors of ice cream, in either a cone or cup. We might say V is true if someone orders vanilla ice cream, and C is true if a person orders their ice cream in a cone. Then $P(V)$ is how likely a random customer will order vanilla, and $P(C)$ is how likely an independently chosen customer asked for a cone. $P(V|C)$ tells us how likely it is that someone who got a cone ordered vanilla, and $P(C|V)$ tell us how

likely it is that someone who ordered vanilla got it in a cone. And $P(V,C)$ tells us how likely it is that a randomly chosen customer got a vanilla cone.

3.6 Marginal Probability

We met the term **marginal probability** above. It may seem like an odd choice: 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 pre-computed probabilities. The idea is that we (or the printer) would sum up the totals in each row of a table of probabilities, and write those totals in the margin of the page [Andale17].

Let’s illustrate this idea by returning to our ice cream shop. In Figure 3.14 we show some recent purchases made by our customers. Our shop is brand new and serves only vanilla and chocolate, in either a 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 vs. a cone, or vanilla vs. 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.

	Vanilla	Chocolate	
Cone	40	60	$P(\text{Cone}) = 100/150 \approx 0.66$
Cup	20	30	$P(\text{Cup}) = 50/150 \approx 0.33$
	$P(\text{Vanilla}) = 60/150 = 0.4$	$P(\text{Chocolate}) = 90/150 = 0.6$	

Figure 3.14: Finding marginal probabilities for 150 recent visitors at an ice cream shop.

Note that the probabilities of someone buying a cup *or* 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% sure that *one* of those choices will be selected.

3.7 Measuring Correctness

In later chapters we'll frequently want to assign data to one of several categories. We'll focus here on the simplest case, where we ask if a piece of data is or is not in some specific category.

For instance, we might ask if a photograph contains a dog, a hurricane is likely to hit land, a stock appears to be headed for a big change, or if our high-tech enclosures are strong enough to hold our genetically engineered super-dinosaurs (hint: they're not).

Naturally, we'd like to make the most accurate decisions we can. 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. If we want to use our mistakes to improve our performance, then we need to identify the different ways our predictions can be wrong.

This discussion applies far beyond just machine learning. The following ideas can help diagnose and solve all kinds of problems, whether they arise from specialized tasks or everyday life, where we're making decisions on the basis of labels we've assigned.

Before we dig in, we'll note that two of the terms we'll be using here, **precision** and **accuracy**, have different meanings in different fields. 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

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.

3.7.1 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 is not, in a given category. Think of this in question form: Is this sample in the category? There are no "maybe" answers allowed, so the only options are yes or no.

If the answer is "yes," we call the sample **true** or **positive**. If the answer is "no," we call the sample **false** or **negative**.

We'll discuss accuracy by comparing the results we get from our classifier against the real, or correct, results that we've observed. The value of positive or negative that we've manually assigned to the sample is called its **ground truth** or **actual value**. We'll call the value that comes back from our categorizer, whether human or computer, 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 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 for the other.

Our samples will each belong to one of two **categories**, or **classes**. Let's call them *category 0* and *category 1* (or *class 0* and *class 1*). To identify a sample's category, we'll use color and shape cues. Figure 3.15 shows an example of 2D data in two categories.

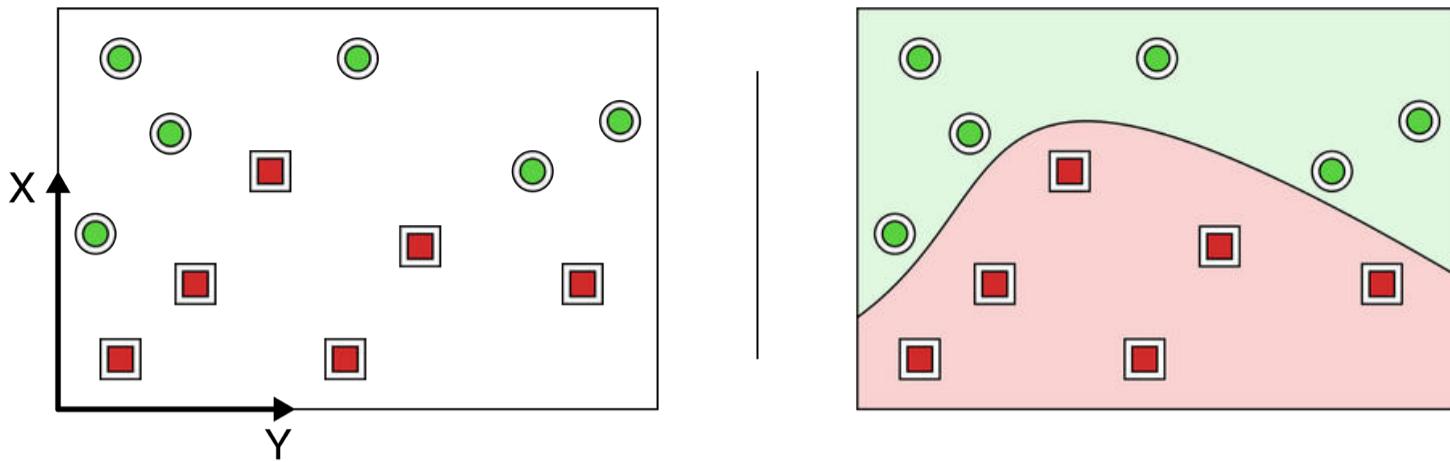


Figure 3.15: Two-dimensional data from two different categories. Left: Each sample has an X and Y value, allowing us to place it on the page. We show the two different categories of samples with different colors and shapes. Right: We can draw a boundary between the two groups to separate them. This is just one of an infinite number of possible boundaries.

We'll show the results of our categorization, whether done by human or computer, by drawing a **boundary**, or curve, through the collection of points. The boundary may be smooth or twisty, but its purpose is to represent our classifier. 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 category, while all those on the other side will be predicted to be in the other category.

We sometimes say that the boundary has a “positive” side and a “negative” side. This matches up to our categories if we think of the classifier as answering the question, “Does this sample belong to a given category?” If the answer is “true,” or positive, then the prediction is “yes”, otherwise the prediction is “no”.

The nature of the boundary can be represented by drawing an arrow pointing into the positive side, as in the diagram on the left of Figure 3.16. Since this can clutter up the diagram when we include the samples as well, we can instead use a notation borrowed from weather maps [MetService16]. On a weather map, an *isobar line* shows the edge of a hot or cold front, with little symbols pointing in the direction of the temperature region of interest. In the right diagram of Figure 3.16 we've replaced the arrow with a warm-front isobar line. The little triangles point into the positive side of the line.

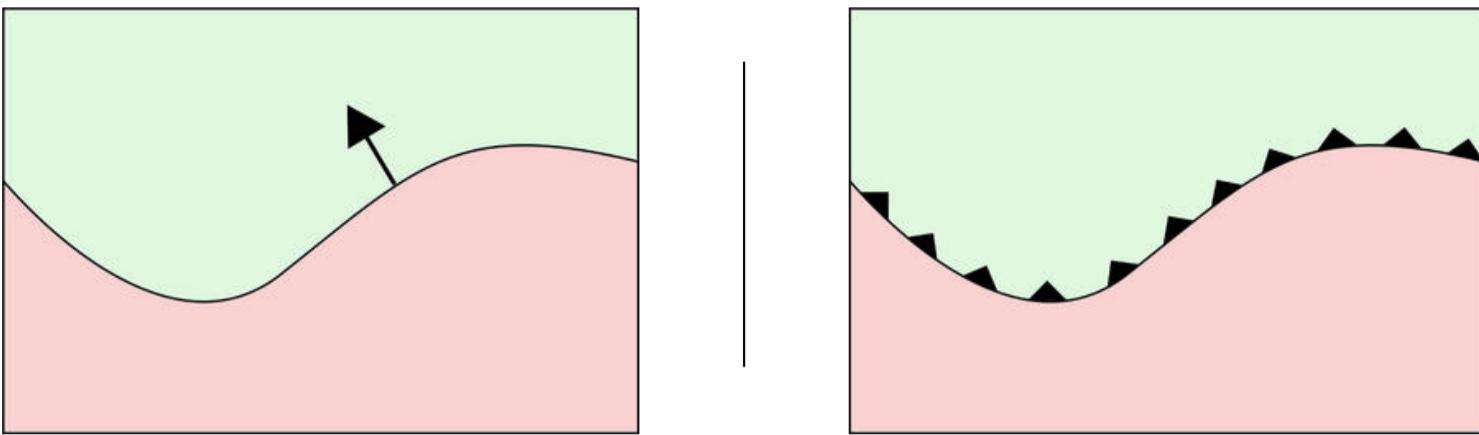


Figure 3.16: This curve separates the space into two regions. Left: We can point to the positive side of the line with an arrow. Right: We can also use isobar markers.

For our dataset, we'll use a set of 20 samples, shown in Figure 3.17. Samples that we have manually determined to one category are marked with a green circle, while those we've assigned to the other category are shown as a red box. So the color of each sample corresponds to its ground truth, not the value assigned by the categorizer.

The categorizer's boundary splits our diagram into two regions. We've used light green to show the positive region, and light red for negative. Every point in the light-green region is classified as positive, and every point in the light-red region is classified as negative.

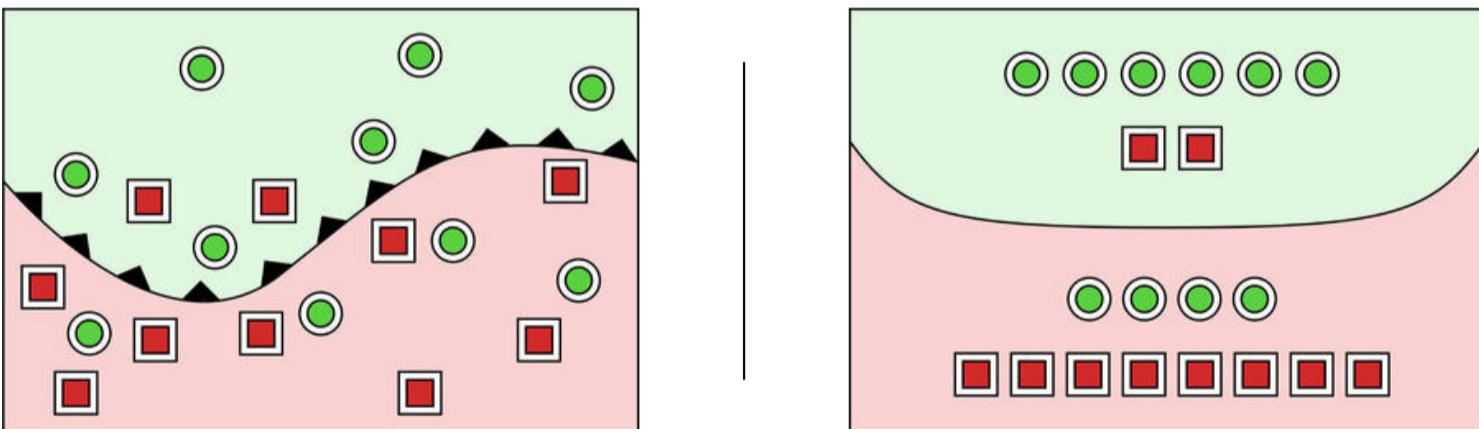


Figure 3.17: Our starting set of 20 samples. The isobar line points in the direction of "positive." We've given green circles a label of "positive," and red squares a label of "negative." Left: The categorizer's curve does an okay job of separating the classes, but it makes some mistakes. Right: A schematic version of the same diagram.

In a perfect world, all the green circles would be on the green side of the curve, 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 3.17 we show the data as we might have measured it. But we don't really care in this discussion about the 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.

In the right diagram of Figure 3.17, 6 out of 10 green circles have been correctly identified as “positive,” and 8 out of 10 red squares have been correctly identified as “negative.” Two red squares and four green circles are on the wrong sides of the boundary.

This diagram represents what typically happens when we run a classifier on a real data set. Some data is classified correctly, and some isn't. If our classifier isn't classifying points well enough for us, we'll need to take some sort of action, perhaps 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.

3.7.2 The Confusion Matrix

We'd like to characterize the errors in Figure 3.17 in a way that tells us something about the nature of the classifier's performance, or how well its predictions matched our given labels.

We can make a little table that has two columns, one for each predicted value, and two rows, one for each actual value. That gives us a 2 by 2 grid, usually referred to with the somewhat weird name of a **confusion matrix**. The name refers to how the matrix shows us the ways in

which our classifier was mistaken, or confused, about some of the data. The classifier's output is repeated in Figure 3.18, along with its confusion matrix.

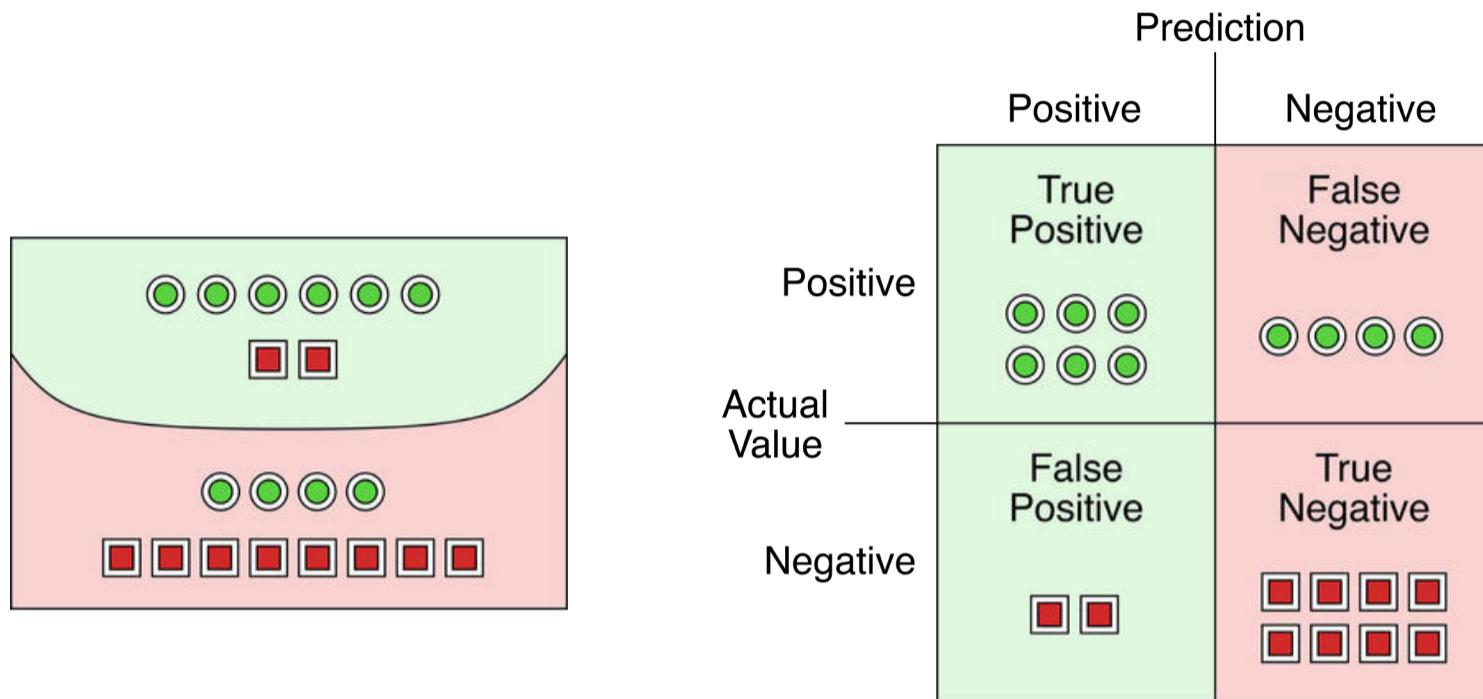


Figure 3.18: We can summarize what went where in Figure 3.17 (repeated here on the left) into a confusion matrix, which tells us how many samples landed in each of the four categories.

As Figure 3.18 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 labeled (or 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. In other words, they were incorrectly, or falsely, assigned the label “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, labeled as “positive.”

We can write this more concisely using two-letter abbreviations for the four categories, and a number describing how many samples fell into each category. Figure 3.19 shows the form that the confusion matrix is usually shown in. Note that, unfortunately, there is no universal agreement on where the various labels go. Some authors put predictions on the left and actual values on top, or place positive and negative in the opposite locations than we show here. When we encounter a confusion matrix, it is important to look at the labels and make sure we know what each box represents.

		Prediction	
		Positive	Negative
Actual Value	Positive	TP 6	FN 4
	Negative	FP 2	TN 8

Figure 3.19: The confusion matrix of Figure 3.18 in its conventional form. Note that not all authors place the labels in these positions, so it's always important to check the labels.

3.7.3 Interpreting the Confusion Matrix

The confusion matrix can be, well, confusing. So let's walk through its various categories with a new, specific example.

The classical way to discuss the confusion matrix is to present it in terms of a medical diagnosis, where “positive” means someone has a particular condition, and “negative” means they’re healthy. Let’s follow that scenario here.

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*, or MP. Anyone who has MP needs to have their thumbs surgically removed right away, or the disease will kill them within hours.

It's critical that we correctly diagnose who has MP and who doesn't. Thumbs are very important to people. If their only way to stay alive is to lose their thumbs, most people would probably go for it. But we definitely do not want to make any incorrect diagnoses that lead to removing anyone's thumbs if their life is *not* in danger.

Let's imagine that we have a test for detecting MP. It's slow and expensive, but we know that it's perfectly reliable. A positive diagnosis means the person has MP, and 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 reliable test was slow, expensive, and required a specialized lab that we can't easily move around. We're worried about future outbreaks, so based on what we've just learned, we develop a new, fast, cheap, and portable blood test that will predict immediately if someone does or does not have MP.

Our goal will be to use our cheap and fast blood test whenever possible, using the expensive and slow test only when necessary. Ideally, if our blood test returns True, then the person really does have the disease, and we need to amputate right away. If the blood test returns False, then we'll send that person home, taking no action. So making sure our blood test does a very good job is very important!

But we're still refining the blood test and it's not yet completely reliable. This cheap and fast version of the test has two problems. First, sometimes it misses the disease, and comes up negative even when the person has MP. Second, sometimes it comes up positive when it shouldn't, because the person ate or drank something recently that fooled the test into thinking they had MP.

We know our blood test is flawed, but when we're on-site it's the only tool we have. So until we find a better test, we want to characterize how often the test is right, how often it's wrong, and when it's wrong, in what ways it's wrong.

So we'll go to a town where a few people have reported MP, and we'll build a lab for our expensive, slow, but accurate test. We'll check everyone with both tests. Then we'll compare the results by putting everything into the four categories of a confusion matrix. This way we can compare the results of our quick blood test against the real facts, as revealed by the slower and more expensive test.

True Positive means that the person has MP, and our quick test correctly says that they have it. We'll operate and save their lives.

True Negative means the person does *not* have MP, and our quick test agrees. That person isn't in danger and doesn't need surgery.

False Positive means that the person does *not* have MP, but our quick test says that they do. This would lead us to amputate the thumbs of a healthy person, which would be a terrible mistake.

False Negative means that the person has MP, but our quick test says they don't. So we wouldn't amputate, and that person would die. That's even more terrible.

Both true positive and true negative are "correct" answers, while false negative and false positive are "incorrect." A false positive means we operate without cause, and a false negative leaves someone at risk of dying. The more accurate our quick test is, the more it will return true positives and negatives, and not the alternatives.

Although we'd love to use the slow and expensive test, that's not practical, so we need to understand the quick blood test that we've got.

Figure 3.20 shows a confusion matrix in terms of MP.

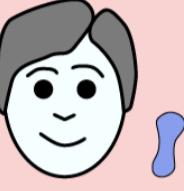
		Prediction	
		Positive	Negative
Actual Value	Positive	TP  	FN  
	Negative	FP  	TN  

Figure 3.20: A qualitative confusion matrix for our MP test. The faces show the response of a person upon given the result of the test. The little blue blob shows whether the person has really got MP or not. The true positives and true negatives are correct diagnoses. But a false negative means we've failed to detect MP in a sick person, who will die. A false positive means we'll operate on a healthy person. We will assign numbers to these categories below.

We can attach numbers to each cell in this confusion matrix, and then use those values to answer questions about our population and our test. The result will be numerical measures that tell us how we're doing. We're going to do just that in a later section.

But before we dig into numbers and scores, it's important that we look at an issue associated with confusion matrices. This issue is often overlooked, with potentially terrible, real-life consequences: we have to pay close attention to the question we're asking.

Questions that appear to be perfectly reasonable, and *are* perfectly reasonable, can lead us to some very incorrect conclusions if we're not careful.

Suppose for the moment we ask this question: “How many people who have MP did our test correctly classify?” This sounds, on the face of it, to be a useful measure of accuracy, but it can be very misleading.

Let’s pretend that our test always returned True. Then the answer to our question is that the test correctly classified 100% of the people with MP. In other words, the test performed *perfectly* with respect to this question, even though we might have produced tons of wrong answers, since every person who doesn’t have MP was still diagnosed as positive. These wrong answers would have led to unnecessary operations.

We hardly needed a test at all. Our “test” might simply have been a piece of paper with the word “True” printed on it. It only sounds like we have a great test because the question excluded all the other possible diagnoses. If we just want a perfect record for one specific diagnosis, then this discussion shows us that we can just assign that diagnosis to everyone who walks in the door. This would be an awful process.

Alternatively, we might ask, “How many people did we correctly identify as healthy?” Once again, we’re only looking at one diagnosis. So we could simply give everyone a diagnosis of False, and our test is again 100% accurate with respect to this question. After all, every healthy person got a clear bill of health. Unfortunately, the sick people would all die.

Mistakes like asking the wrong type of questions are very easy to make, particularly when the situations become complex. So people have developed another set of terms that make it harder to accidentally ask the wrong question.

3.7.4 When Misclassification Is Okay

Before we get into proper measuring of our system's performance, note that sometimes we *want* false positives or false negatives.

For example, suppose we work for a company that makes toy figurines in the likeness of dozens of popular TV characters. Our toys are a hit right now, so our production line is running at full capacity to manufacture as many figurines as possible. We're in charge of packaging the figurines and shipping them to retailers. The figurines arrive at our facility all jumbled together in the shipping box, so we first sort them, find the right packaging for each group, put each figurine into its box, and then route that package into the right shipping box to go out 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 prevent any of them from leaving 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. So our goal is to make sure that not a single Glasses McGlassface goes out the door.

Figure 3.21 shows the situation.

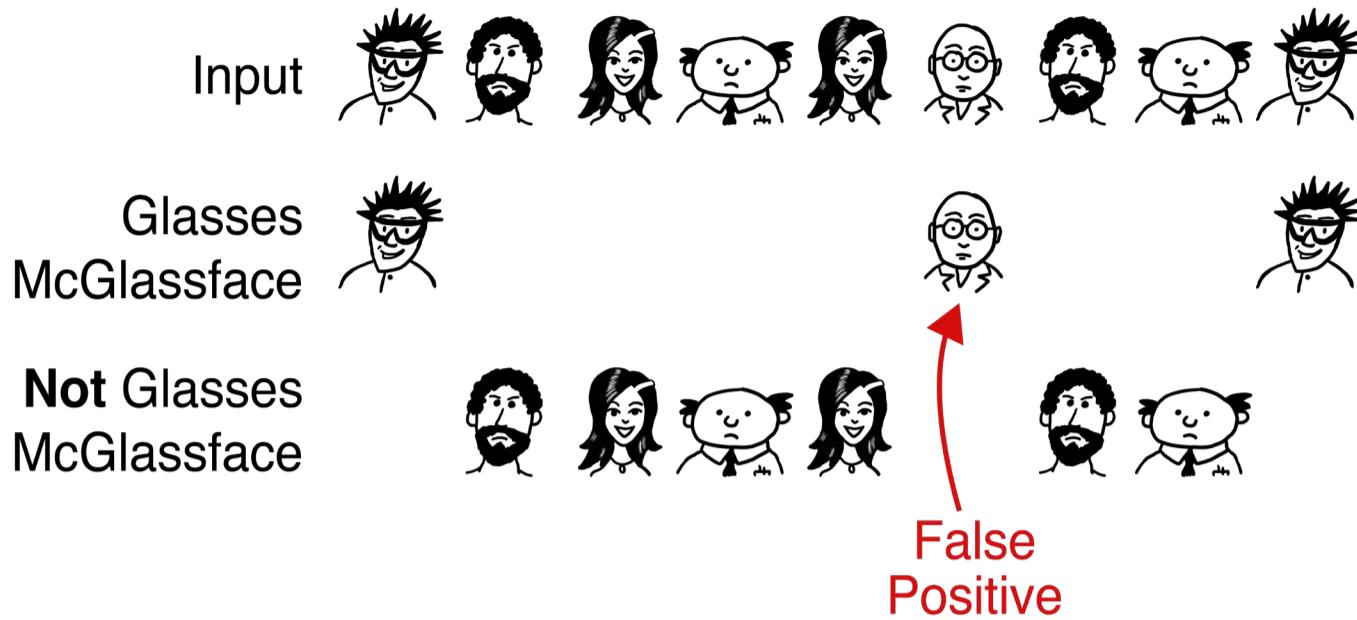


Figure 3.21: Glasses McGlassface is the first character on the top row. We want to remove any doll that could be that character. Top row: All dolls. Middle row: All dolls we've removed from the line because they could be Glasses McGlassface. Bottom row: The dolls we're shipping. In the middle row we have a false positive, or a character who isn't Glasses McGlassface, but was removed anyway.

To make the process of detecting these figures efficient, we set up a camera that looks at every figurine that comes along the conveyor belt. Each time it sees a forbidden doll, a robot arm picks it up and places it in the bin.

How should we program this camera? If we let even one figurine get out the door we'll get sued for a fortune, so it's better for us to err on the side of caution. If the camera thinks that a figurine might be the forbidden doll, it should be removed from the line. We can always later box up and sell the incorrectly removed figurines.

In this situation, false positives (toys that we incorrectly classify as forbidden) are just an annoyance, but false negatives (forbidden toys that we accidentally ship) have a huge cost. So in this situation, we can adopt the policy that it's okay to have some false positives (as long as there aren't too many).

Let's say the next step in the line is a quality-control step, where we make sure that each figurine has been properly painted. We've just received a memo from the factory that the eye-painting robot may be

messing up, so we need to pay particular care to the eyes, and only pass through dolls that have their eyes painted on correctly. If their eyes are *not* present, we want to pull that doll off the assembly line. So we'll look for the eyes, and if we don't find them, another robot arm will pull that figurine off the line and place it into another bin.

Figure 3.22 shows the idea.

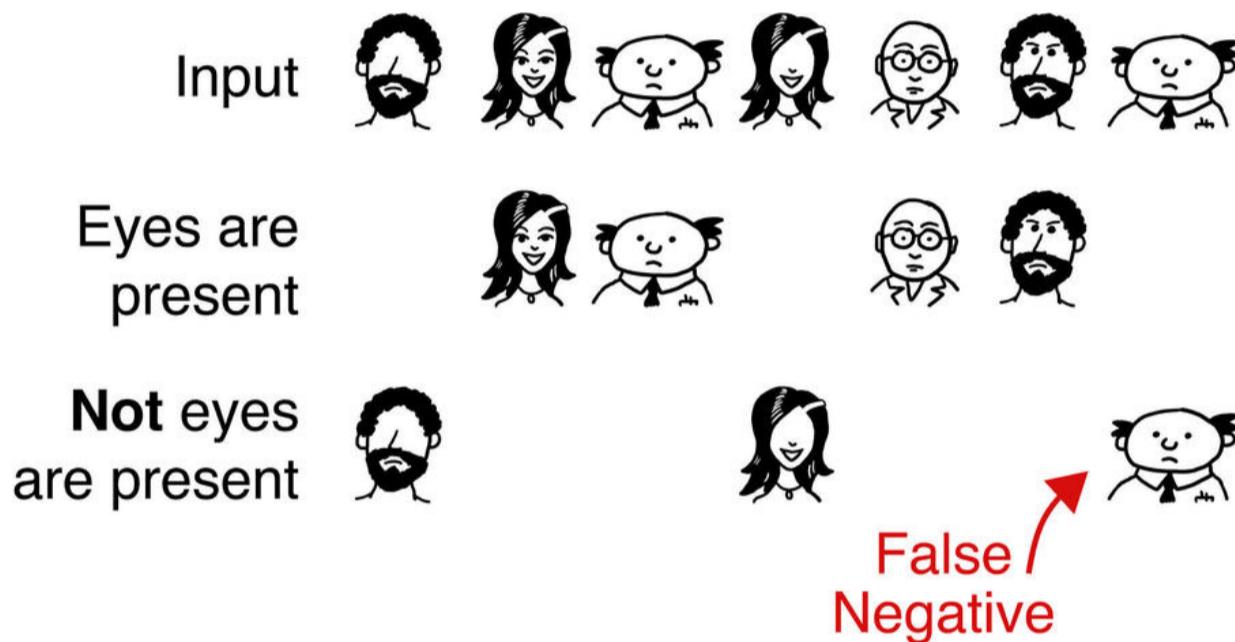


Figure 3.22: A new group of toys. Now we're looking for any with mispainted eyes. Shipping a doll missing its eyes would be a big mistake we want to avoid. We're looking for eyes, so a positive result is that toy has eyes. Top row: The figurines we're scanning. Middle row: All the figurines with eyes. Bottom row: Figurines we've classified as having missing eyes. One character with his eyes was classified as missing them, so he's a false negative.

In this case, we're looking for dolls that *definitely* have their eyes. Those are our positives. This means that we'll remove any figurine that doesn't have its eyes, since shipping a negative, or a character without eyes, would be awful.

So we'll only pass through the certain positives. We'll be extremely careful to never pass through a "false positive," or a doll that doesn't have its eyes. In this example, we've incorrectly said that one doll with eyes is missing them. That's a false negative. But a false negative is okay, because we can always put it back onto the line and ship it.

So in this situation, our policy would be that false positives are terrible, but false negatives are only a minor annoyance.

This is the opposite policy from the previous step.

To sum up, true positives and true negatives are the easy cases. 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 misclassifications.

Now let's get back to the terms we promised earlier that help us do a good job of describing the performance of our classifier.

3.7.5 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 called the **accuracy**. The accuracy is a number from 0 to 1. It's a general measure of how often the prediction is correct. So it's just the sum of the two "correct" values, TP and TN, divided by the total number of samples measured. Figure 3.23 shows the idea graphically. In this figure, as in the ones to come, the points we're counting in any given step will be shown, and the points we're not counting will be omitted.

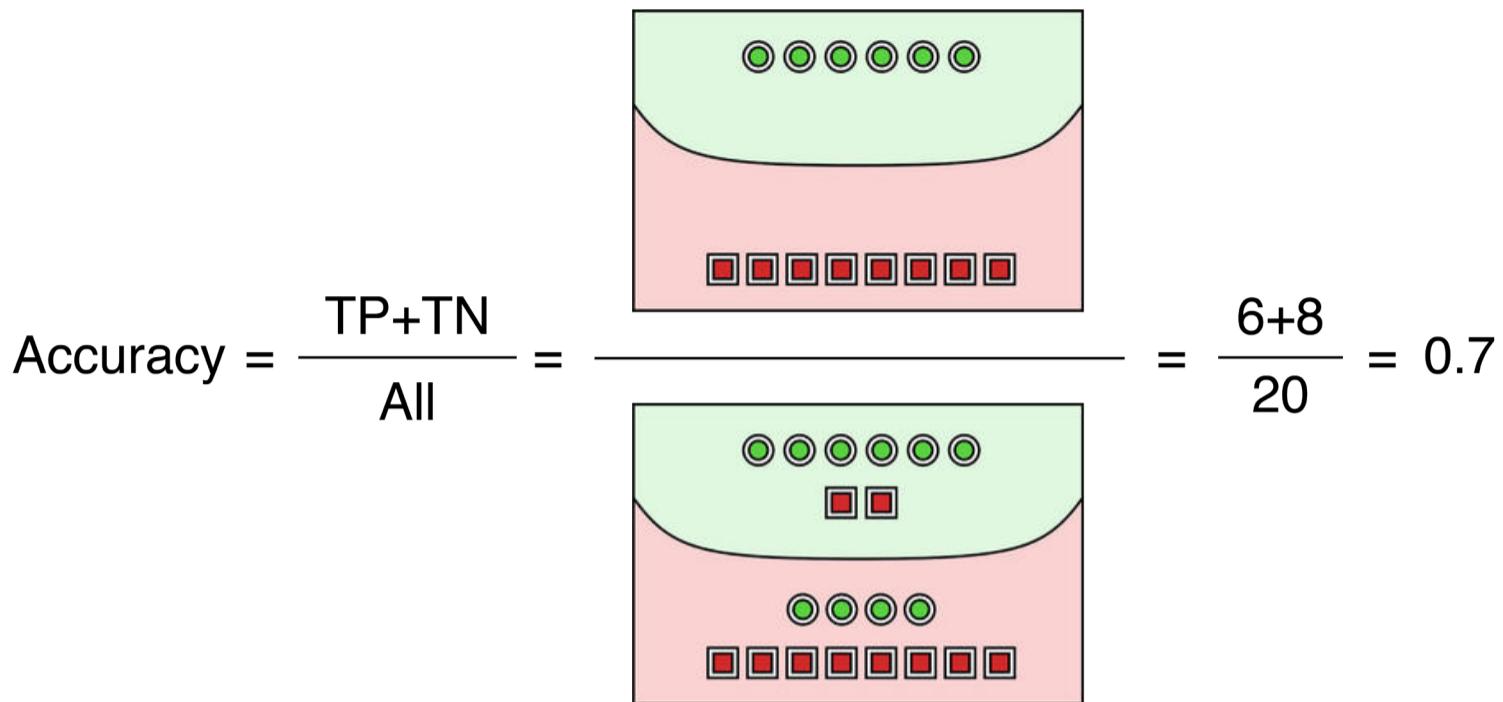


Figure 3.23: Accuracy is a number from 0 to 1 that tells us how often our prediction is correct. We find it by creating a fraction. On top is the number of points that are correctly classified. On the bottom is the total number of points. Here, 6 red circles and 8 red squares were correctly categorized, out of a total of 20 points, giving us an accuracy of 14/20 or 0.7.

We want the accuracy to be 1.0, but usually it will be less than that. In Figure 3.23, we have an accuracy of 0.7, or 70%, which isn't really 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.

To get a better handle on the quality of our predictions, we'll look at two other measures.

3.7.6 Precision

Precision (also called **positive predictive value**, or **PPV**) 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 how many of the “positive” predictions were really positive.*

If the precision is 1.0, then every sample we labeled as “positive” was correctly classified as positive. As the percentage falls, it carries with it our confidence in the predictions. For example, if the precision is 0.8, then we can only be 80% sure that any given sample that’s labeled “positive” has the correct label. Figure 3.24 shows the idea visually.

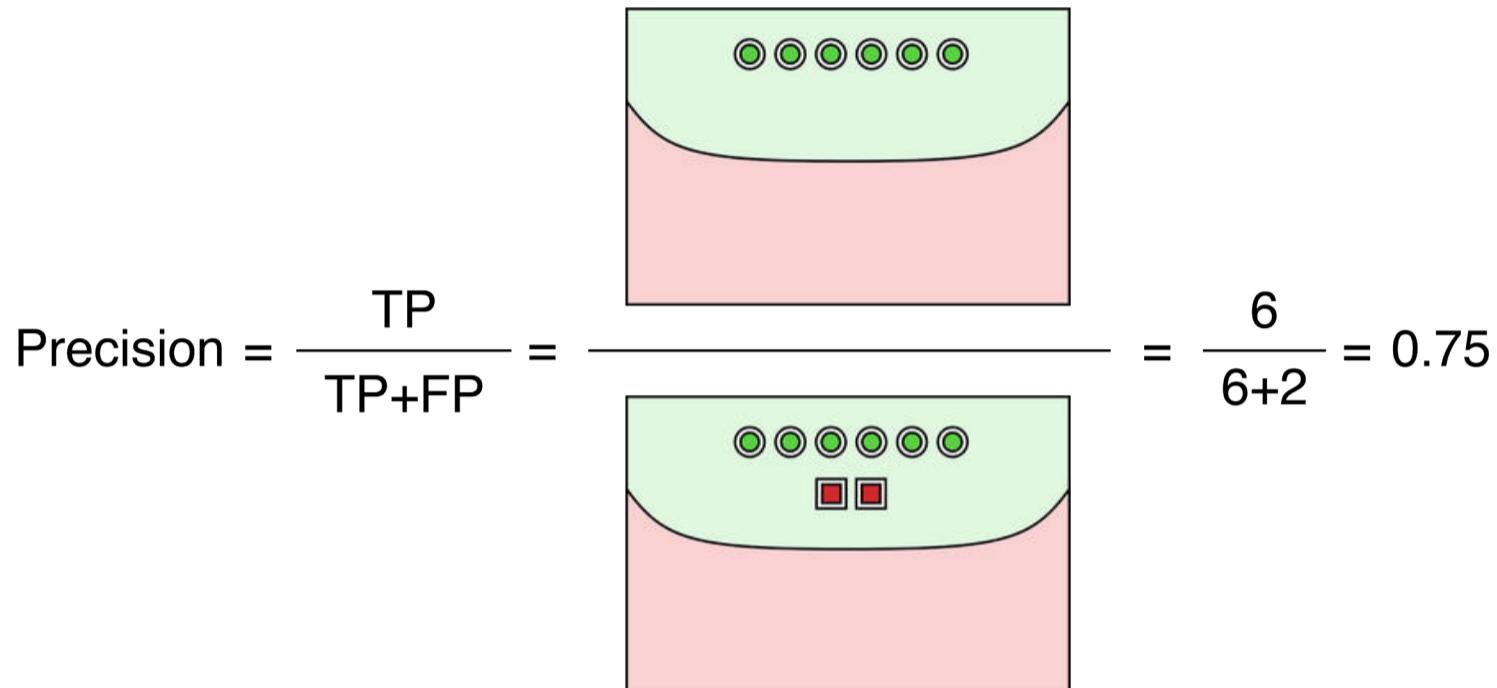


Figure 3.24: The value of precision is the total number of positive samples that really are positive, divided by the total number of samples that we labeled as positive. Here, we have 6 green circles correctly labeled as positive, and 2 red squares incorrectly labeled as positive, giving us a precision of 6/8 or 0.75.

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 the positive points that were labeled as negative.

We can speak in more general terms of class A and class B. Then values of precision less than 1.0 tell us the percent of samples that we reported as type A, but are really type B, as shown in Figure 3.25. Here

the two blobs labeled A and B are made up of samples that really are in those classes. The diagonal dividing line shows where we predict samples are in class A or B.

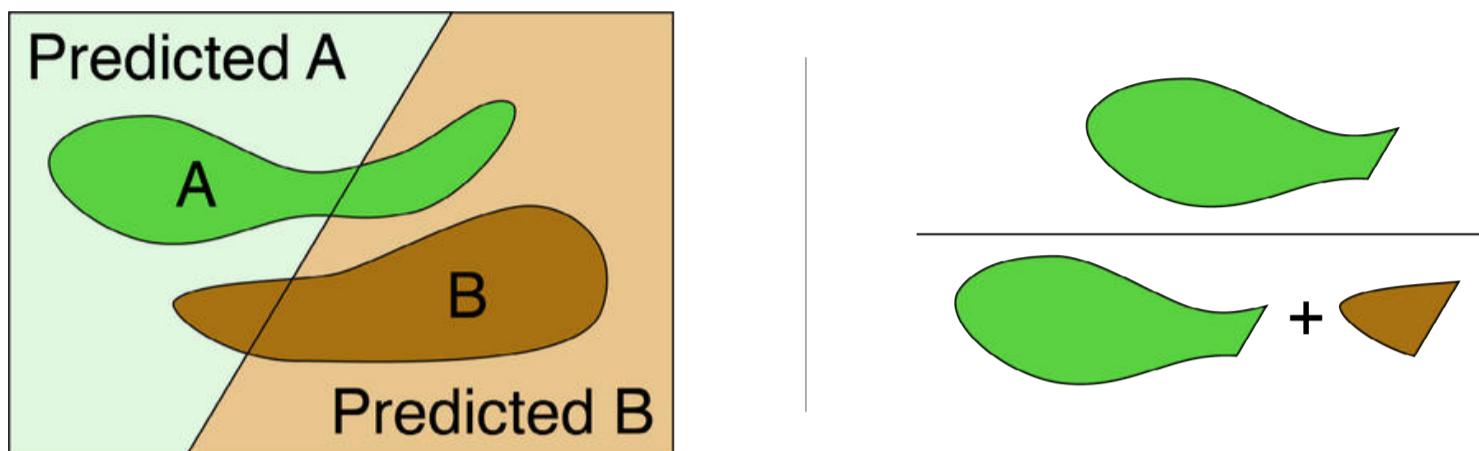


Figure 3.25: The precision is the ratio of the number of elements that we correctly labeled as belonging to class A, relative to the total number of elements that we labeled as class A.

3.7.7 Recall

Our third measure is **recall**, (also called **sensitivity**, **hit rate**, or **true positive rate**). This tells us the percentage of the positive samples that we correctly labeled.

When recall is 1.0, then we caught every positive event. The more that recall drops below that number, the more positive events we missed. Figure 3.26 shows this idea visually.

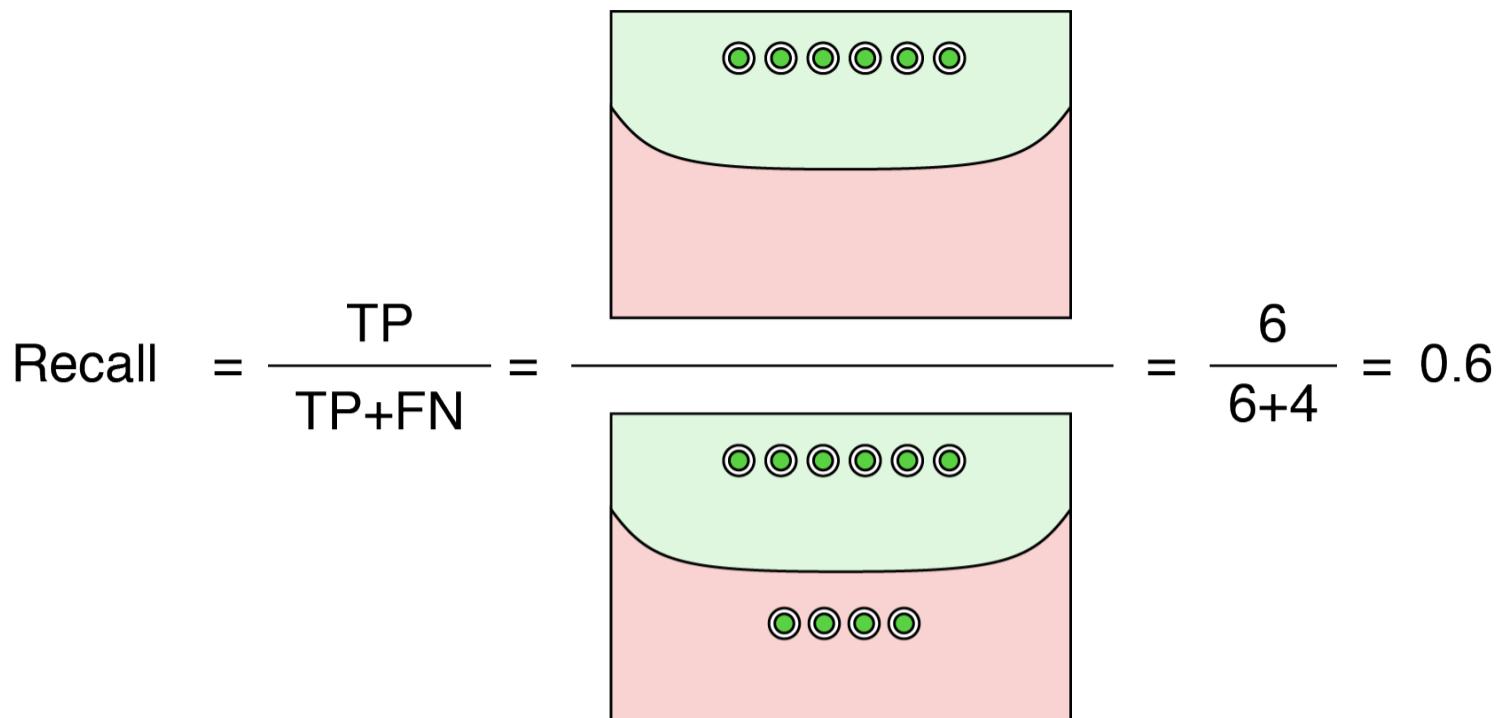


Figure 3.26: The value of recall is the total number of correctly-labeled positive samples, divided by the total number of samples that should have been labeled as positive. Here we have 6 correctly-labeled green circles, but we mislabeled 4 of them as negative, giving us a recall of 6/10 or 0.6.

We can think of this in terms of our A/B categories as well, as shown in Figure 3.27.

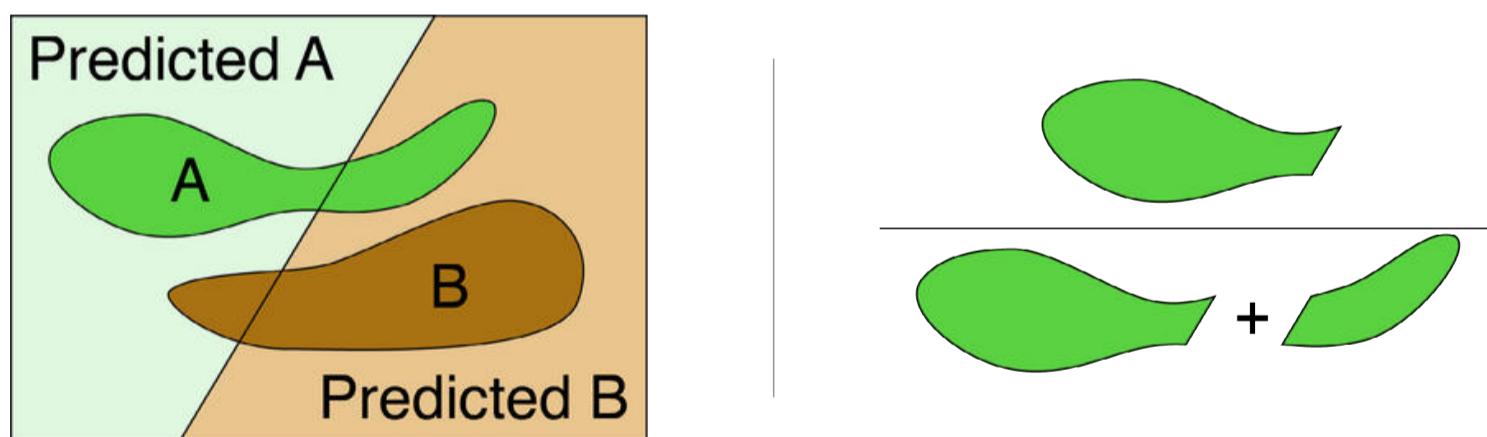


Figure 3.27: Recall tells us how well we did at finding samples in class A. It's the ratio of the number of elements that we labeled as class A divided by the total number of elements that actually were in class A.

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.

3.7.8 About Precision and Recall

Let's look at precision and recall with a concrete example.

Suppose we have an in-house wiki that contains 500 pages, and we do a search on the phrase “dog training.”

Let's say that each page that our search engine returns to us that really is about dog training is a *positive*, in the sense that it is something we want to see. Each page that shouldn't have been returned is a *negative*, in the sense that we don't want to see it. Figure 3.28 shows our four categories of results in terms of page relevancy, and our four categories of true/false and positive/negative.

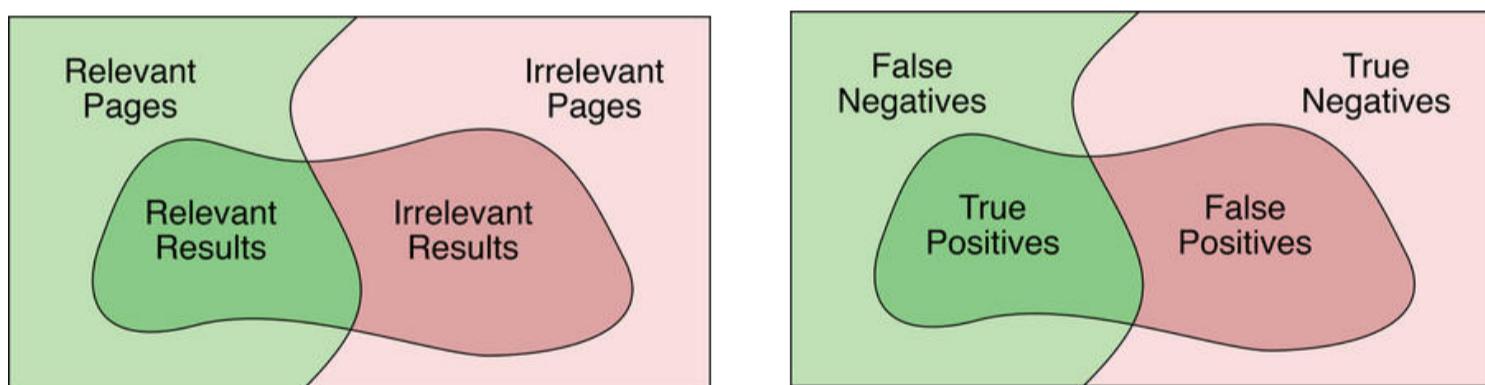


Figure 3.28: Picturing our four categories in terms of web pages returned from a search. The blob in the middle contains the pages that are returned to us from our search. Left: We see that there are relevant and irrelevant pages in our wiki, and we get back some of each in our search, while others are missed. Right: We see how these four categories correspond to our categories of true and false positives and negatives.

Starting with accuracy, this is the ratio of correctly-labeled pages relative to the total number of pages in our wiki. The closer the system gets to correctly identifying each page, the closer the accuracy comes to 1.0. Figure 3.29 shows the idea graphically.

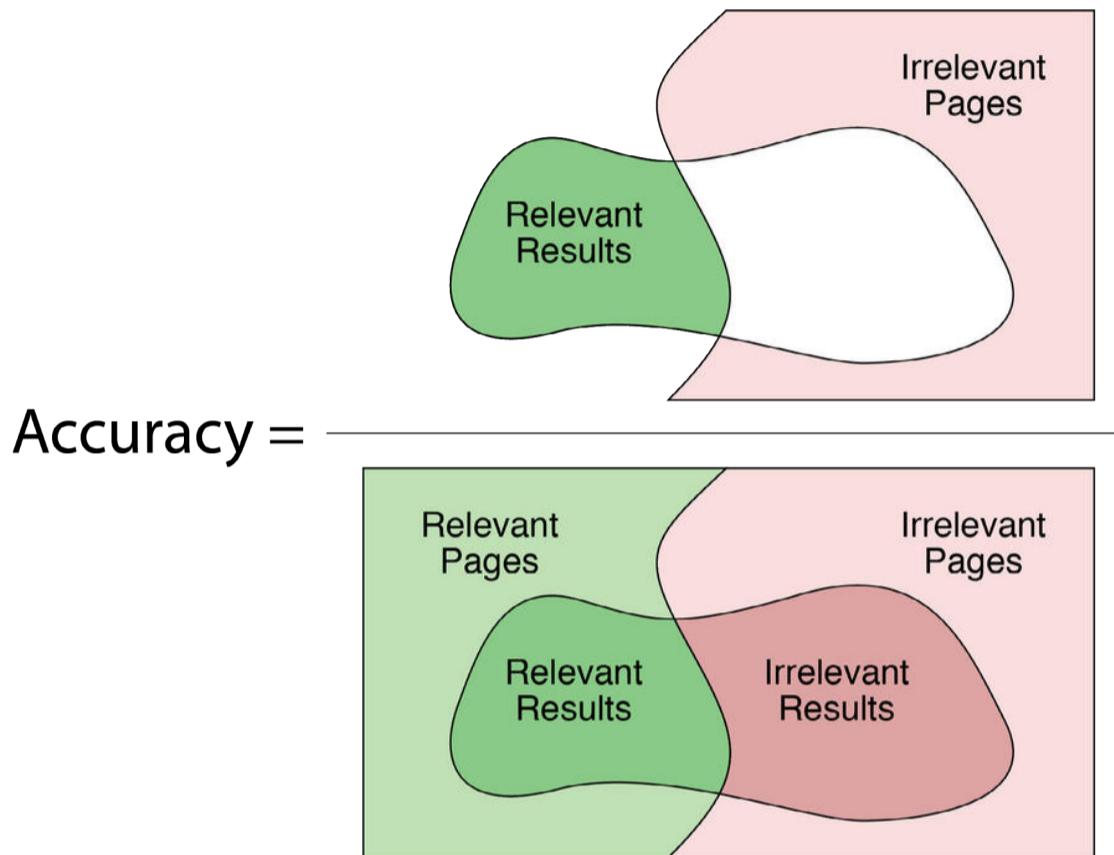


Figure 3.29: The accuracy is the ratio of the number of pages correctly identified as being relevant or irrelevant to our search, compared to the total number of pages in the wiki.

Let's move on to precision. In this situation, precision is the ratio of how many results have something to do with training a dog, relative to the total number of hits. The more precise our search engine is, the more relevant hits (true positives) we get back. If the search engine is *imprecise*, we get back hits that are irrelevant (false positives). Figure 3.30 shows this idea.

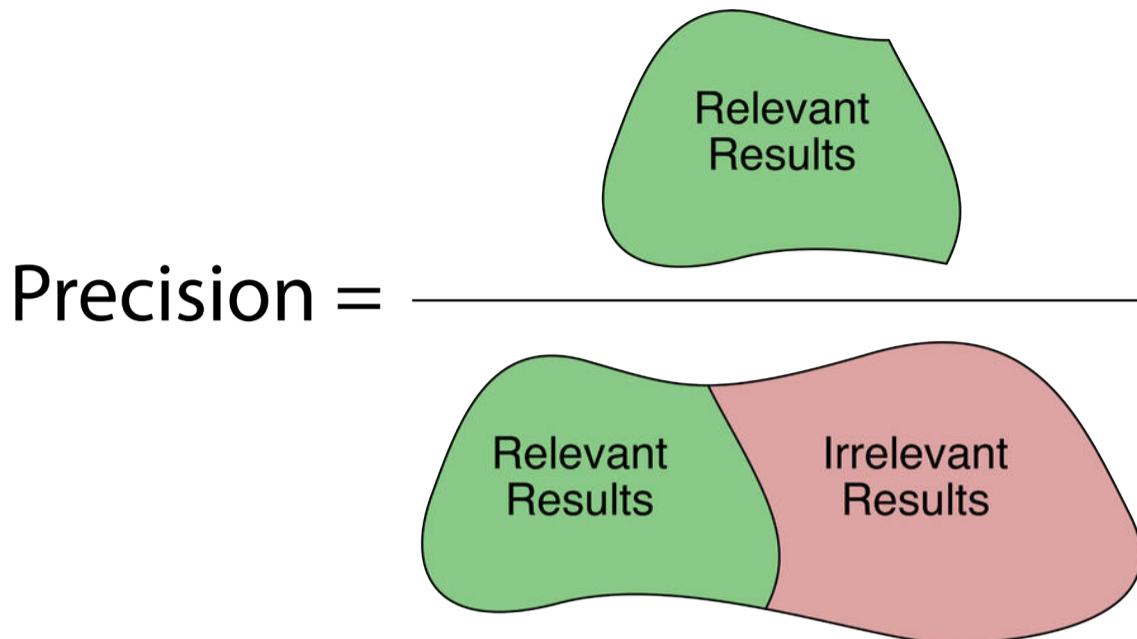


Figure 3.30: Precision is the number of relevant results we got back compared to all the results we got back.

Let's move on to recall, and we'll finally see why this seemingly strange name makes sense. In this context, recall is the ratio of how many relevant results we got back, relative to how many pages in the database that *should* have come back. That is, if the recall is 1.0, the system *recalled* (or retrieved) 100% of the pages it should have. If the recall is 0.8, then the system missed 20% of the pages it should have. Those missed pages are false negatives: the system marked them as “negative” and didn't recall (or retrieve) them, but it should have, as shown in Figure 3.31.

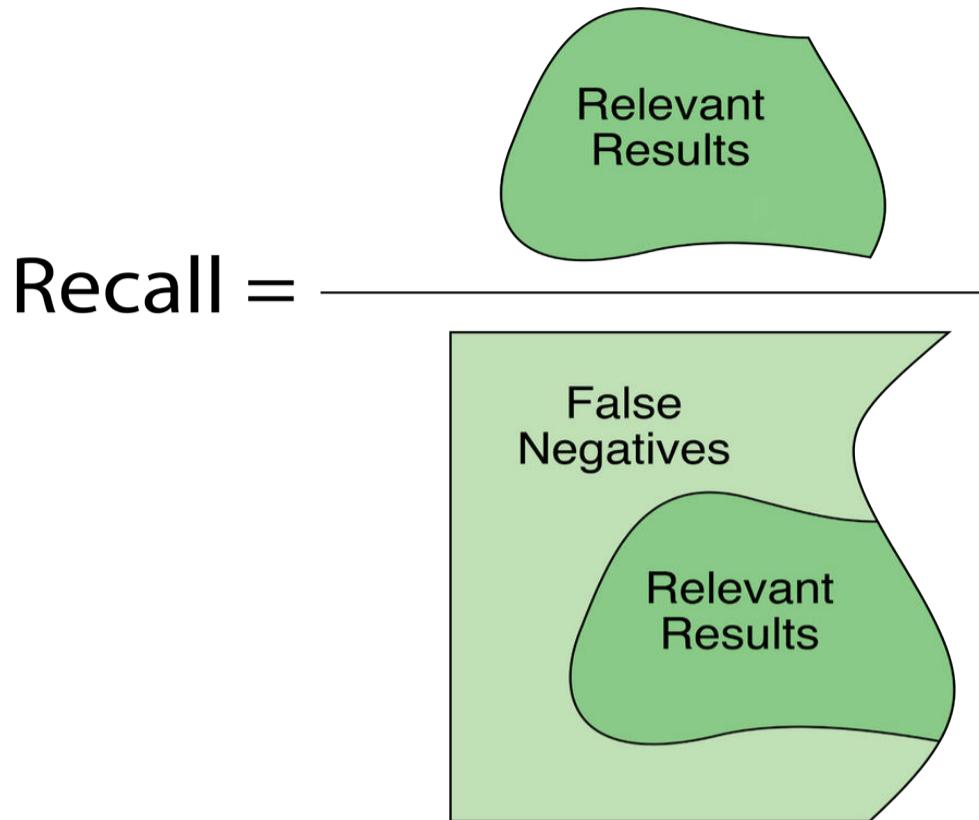


Figure 3.31: Recall is the ratio of how many pages the system recalled, or brought back to us, compared to the number of pages that it should have recalled.

3.7.9 Other Measures

We've seen the measures of accuracy, recall, and precision. There are lots of other terms that are sometimes used in discussions of probability and machine learning [Wikipedia16]. With the exception of the **f1 score**, discussed below, we won't encounter most of these terms in this book, but we'll summarize them to provide a one-stop reference that gathers all the definitions in one place.

Figure 3.32 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.

Chapter 3: Probability

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	Sensitivity, 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 3.32: Common confidence terms derived from the confusion matrix. Don't try to memorize this! This chart is here so you can refer back to it when you encounter an unfamiliar term.

This table is a lot to take in. We provide an alternative that presents the terms graphically, using our distribution of samples from Figure 3.17, repeated here as Figure 3.33.

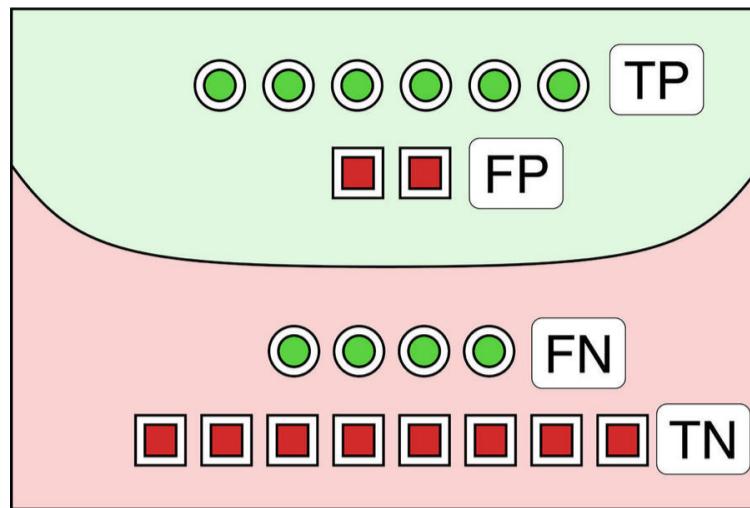


Figure 3.33: The data of Figure 3.17 labeled with the four categories of True Positive, False Positive, False Negative, and True Negative.

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

With these points, we can illustrate the measures of Figure 3.32 by combining these four numbers, or their pictures, in different ways. Figure 3.34 shows how we'd compute the measures using just the relevant pieces of the data.

Chapter 3: Probability

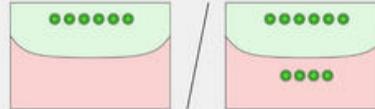
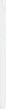
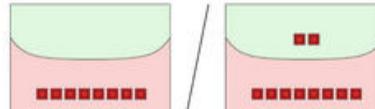
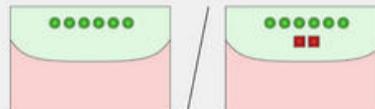
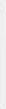
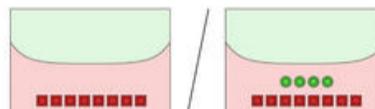
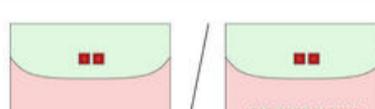
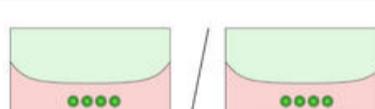
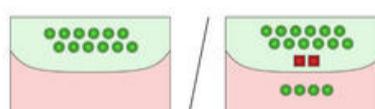
Recall	TPR	$\frac{TP}{TP+FN}$	 / 	$= \frac{6}{6+4} = 6/10 = 0.6$
Specificity	TNR	$\frac{TN}{TN+FP}$	 / 	$= \frac{8}{8+2} = 8/10 = 0.8$
Precision	PPV	$\frac{TP}{TP+FP}$	 / 	$= \frac{6}{6+2} = 6/8 = 0.75$
Negative Predictive Value	NPV	$\frac{TN}{TN+FN}$	 / 	$= \frac{8}{8+4} = 8/12 \approx 0.66$
<hr/>				
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	$\frac{FN}{TN+FN}$	 / 	$= \frac{4}{4+8} = 4/12 \approx 0.33$
<hr/>				
Accuracy	ACC	$\frac{TP+TN}{TP+FP+TN+FN}$	 / 	$= \frac{6+8}{6+2+4+8} = 14/20 = 0.7$
F1 Score	F1	$\frac{2 \cdot TP}{2 \cdot TP+FP+FN}$	 / 	$= \frac{2 \cdot 6}{(2 \cdot 6)+2+4} = 12/18 \approx 0.66$

Figure 3.34: Our statistical measures of Figure 3.32 in visual form, using the data of Figure 3.33.

3.7.10 Using Precision and Recall Together

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 perfect value for either measure, while overall performance is lousy.

To see this, let's revisit the two extremes of **perfect precision** and **perfect 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 3.35 shows the idea.

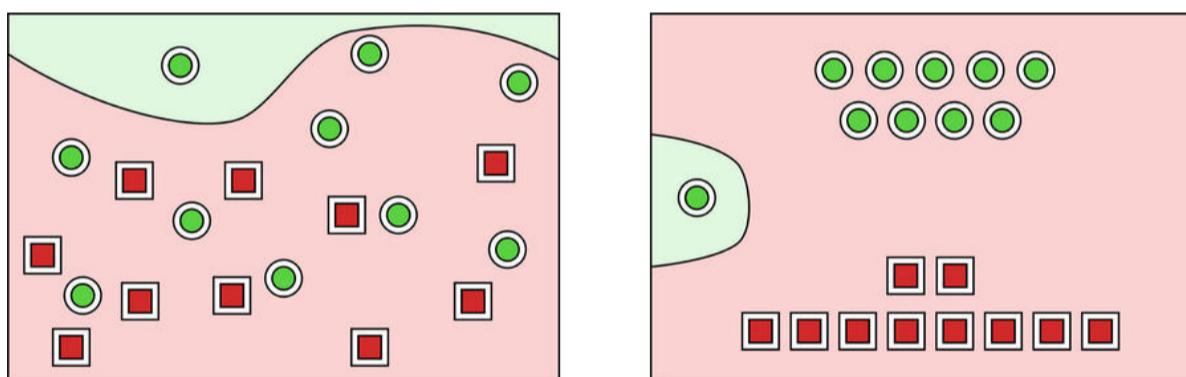


Figure 3.35: Left: This boundary curve gives us a perfect score for precision. Note that all of the red squares are correctly classified as negative, while only 1 green circle is correctly classified as True. The other 9 green circles are incorrectly classified as False. Right: A schematic version of 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 this 1). So we get the fraction $1/1$, or 1, which is a perfect score. But the accuracy and recall are both pretty awful, as shown in Figure 3.36.

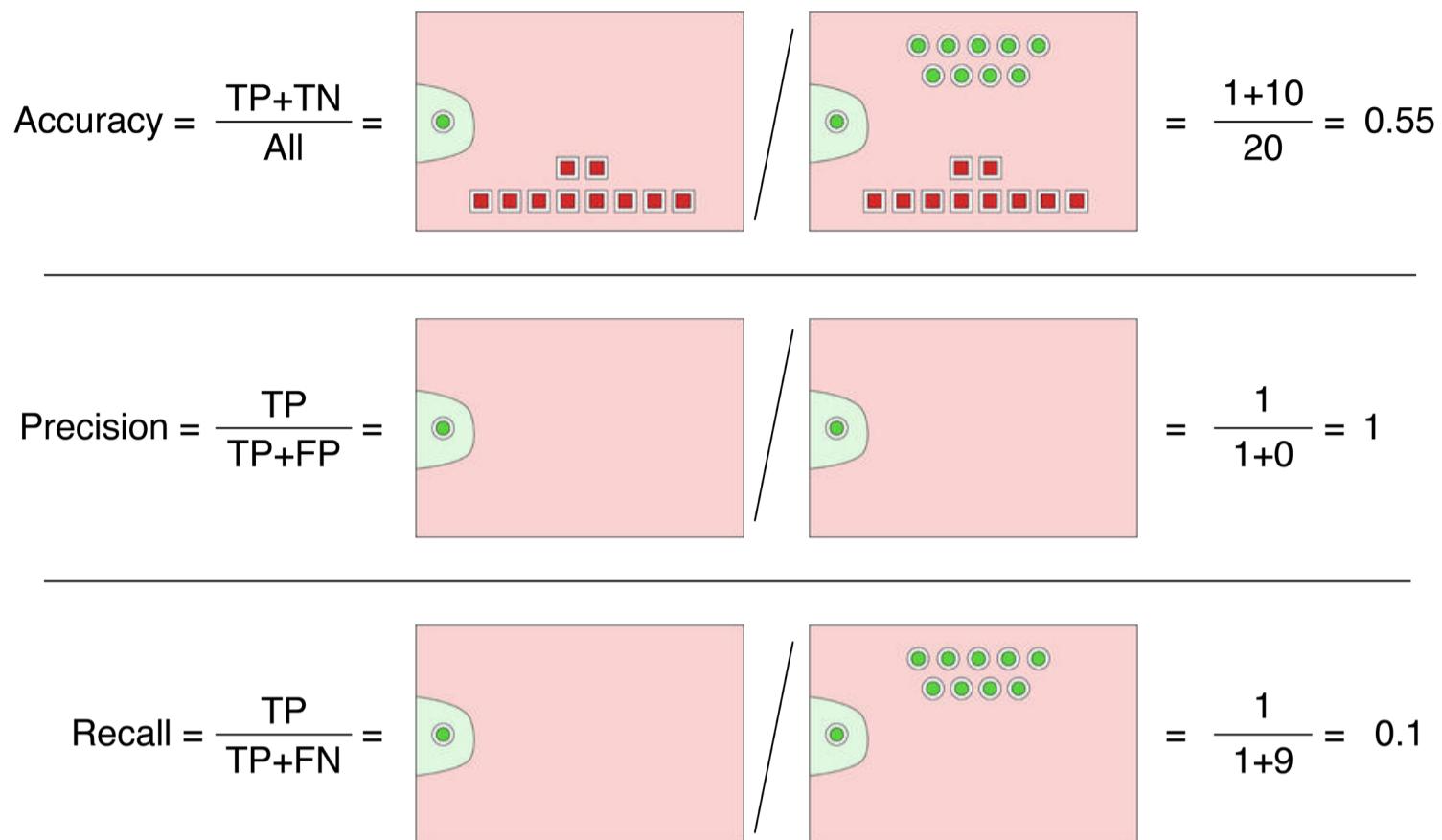


Figure 3.36: These figures all share the same boundary curve, which has labeled exactly one green circle as positive, and all the others as negative. Because of how precision is defined, this gives us a perfect score of 1. But the accuracy is 0.55, and the recall is 0.1, both of which are terrible scores.

Let's do the same 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 3.37 shows the idea.

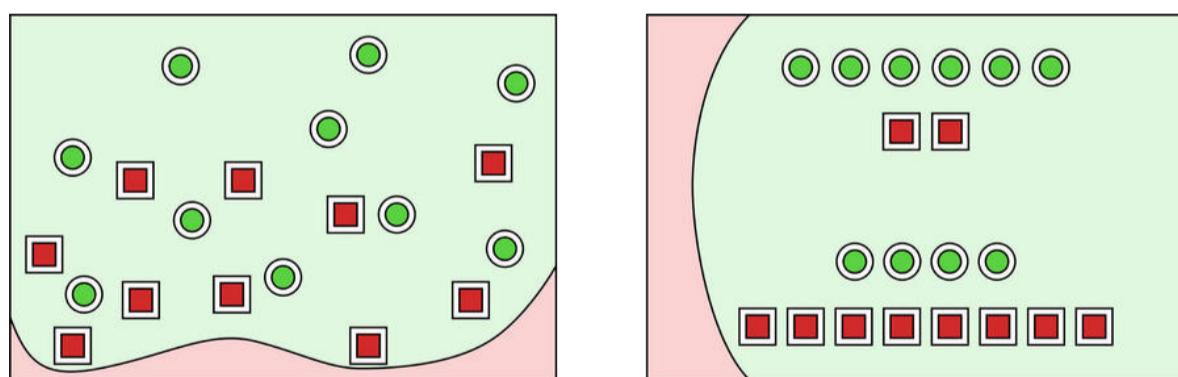


Figure 3.37: Left: This boundary curve gives us a perfect recall score. Note that all 10 green circles are correctly classified as positive, but all 10 red squares are also misclassified as positive. Right: A schematic version of the figure on the left.

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, as shown in Figure 3.38.

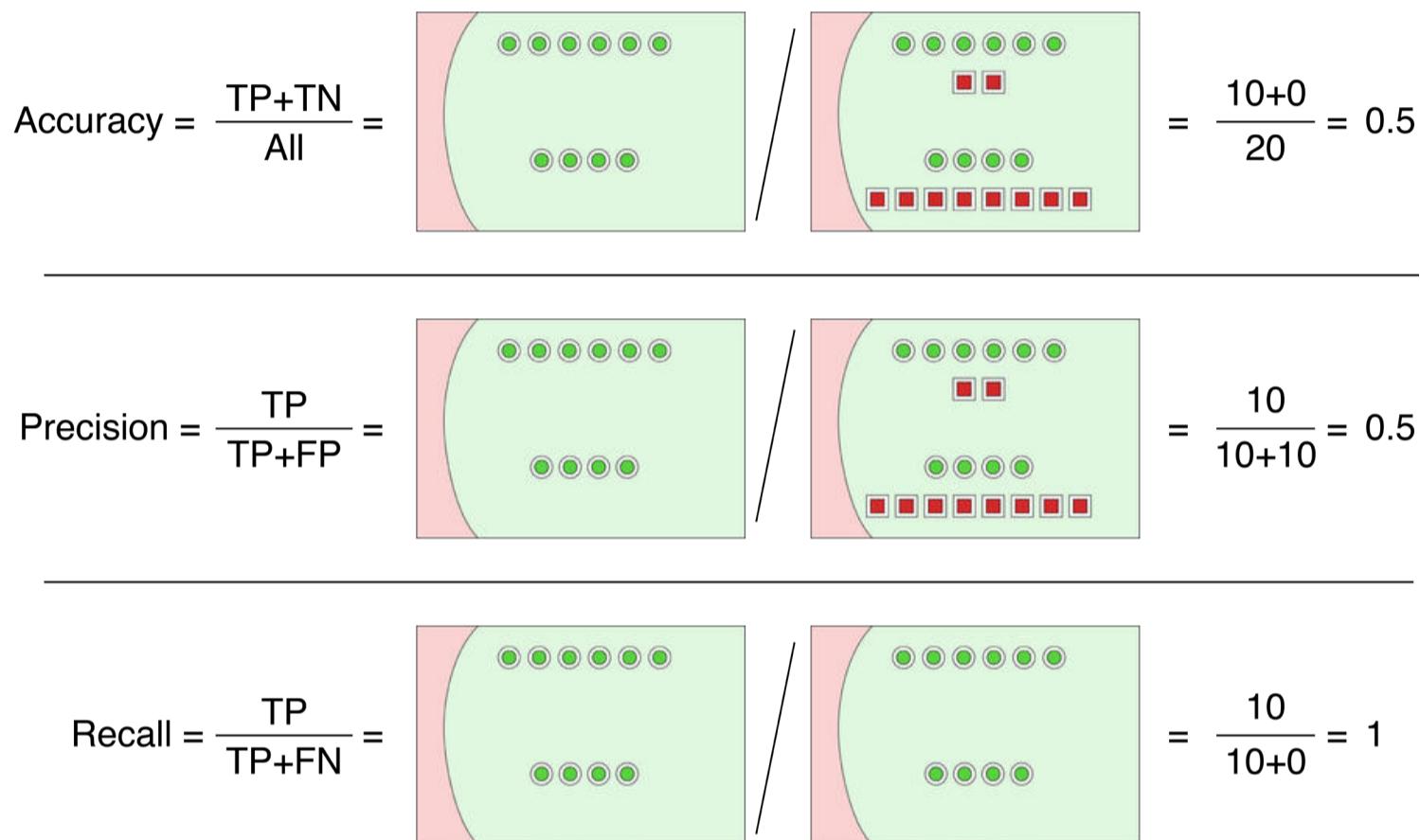


Figure 3.38: All of these figures share the same boundary curve. With this curve, every point is predicted to be positive. We get a perfect recall, because every positive point is correctly labeled. Unfortunately, accuracy and precision both have very low scores.

3.7.11 f1 Score

Looking at both precision and recall together 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 in the last lines of Figure 3.32 and Figure 3.34).

Generally speaking, the f_1 score will be low when either precision or recall is low, and will approach 1 when both measures also approach 1. Figure 3.39 shows the f_1 score visually.

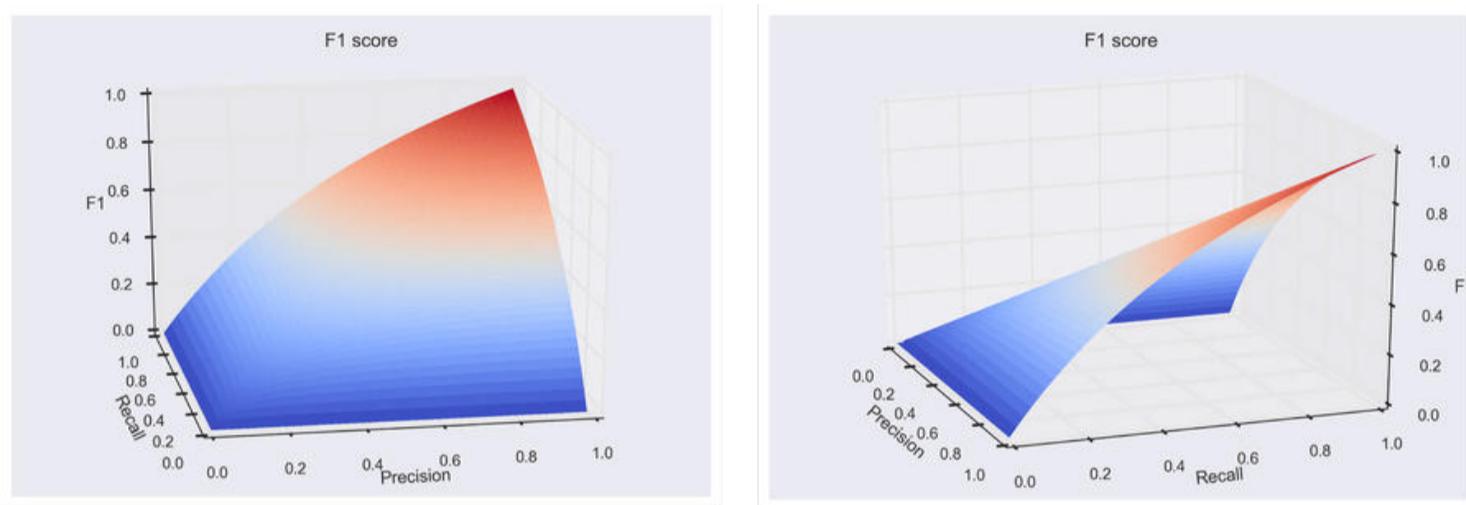


Figure 3.39: The f_1 score is 0 when either precision or recall is also 0, and 1 when both are 1. In between it slowly rises as both measures increase.

3.8 Applying the Confusion Matrix

Let's 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 blood test. Recall that we earlier said that we'd measure everyone in a town with our slow and expensive test (giving us the *ground truth*), as well as our faster, cheaper blood test.

Our measurements show that the blood test seems to have a high true positive rate. We found that 99% 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 0.01.

The test does a bit worse for people who *don't* have MP. The true negative (TN) rate is 0.98, so 98 times out of 100 when we say someone is not infected, they really aren't. This means that the false positive (FP) rate is 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 a suspected 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% of the population is already infected.

This is essential information. We're not testing people blindly. We *already know* that most people do *not* have MP. There's only a 1 in 100 chance that someone does have it. But even one infected person is one person too many, so we head into town at top speed.

Arriving in town, we get everyone to come down to city hall to get tested. Suppose someone comes up positive. What should they think? How likely is it that they have MP? Suppose instead the test is negative. 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 3.40. But this matrix is *incomplete* and will lead us to the *wrong answers* to our questions.

		Prediction	
		Positive	Negative
Actual Value	Positive	TP 0.99	FN 0.01
	Negative	FP 0.02	TN 0.98

Figure 3.40: This is *not* the confusion matrix we're looking for. This matrix assumes that the chances of having or not having MP are 50-50. We know that's not the case, but this chart ignores that information.

The problem is that we're ignoring the critical piece of information from above: only 1% of the people in town will have MP right now. The chart in Figure 3.40 assumes that the chance of having or not having MP are equal, and they're not.

Let's work out the proper chart 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.

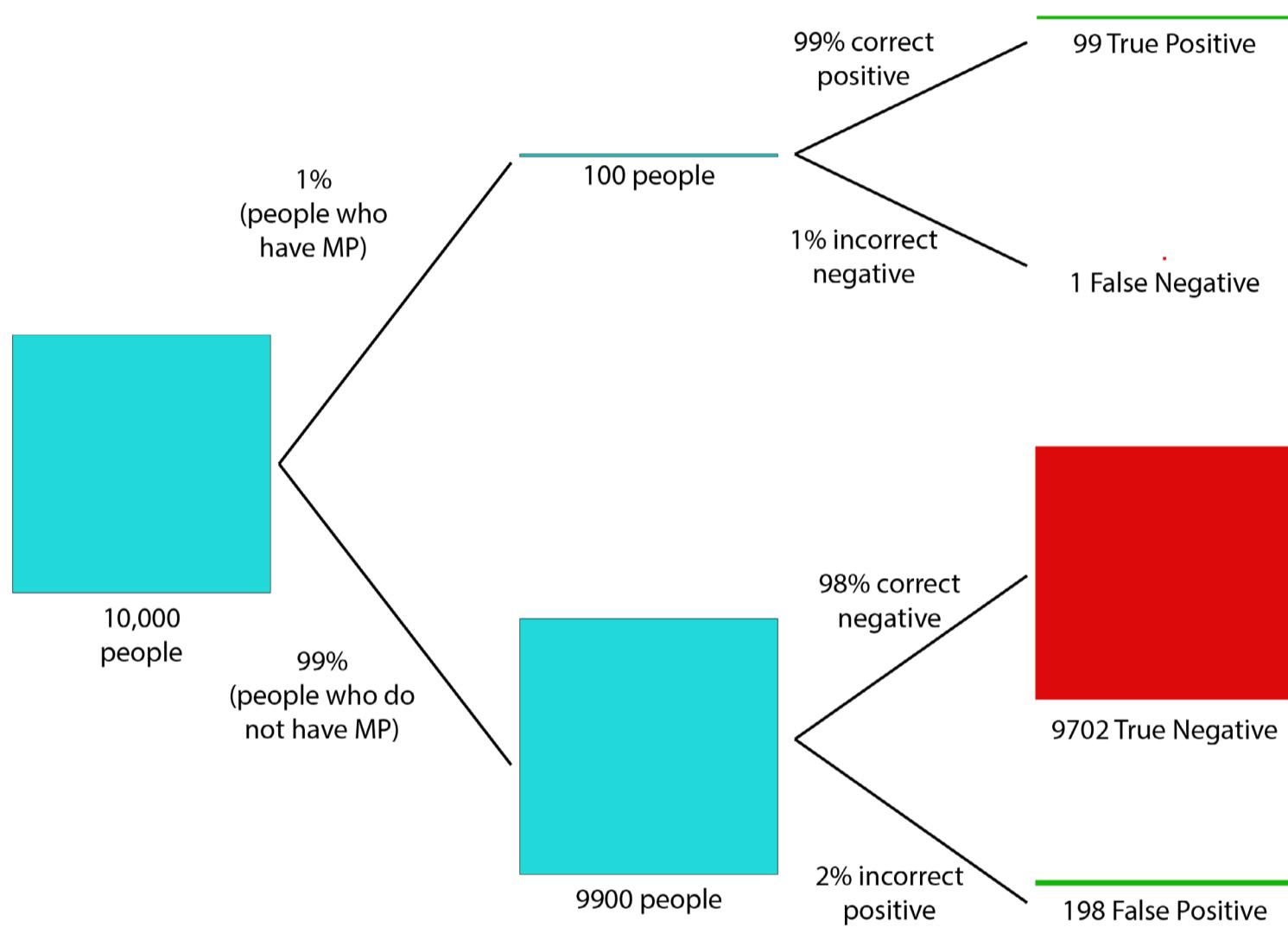


Figure 3.41: Working out the populations we expect from our infection rate and our test.

Let's walk through Figure 3.41. 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% of the population, will be infected with MP. That's shown in the upper path, where we get 1% 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% of people who are *not* infected. Our test will correctly identify 98% of them, or 9,702 people, as being negative. 2% of those 9900, or 198 people, will get an incorrect positive result.

Figure 3.41 tells us the values that we should use to populate our confusion matrix, because they incorporate our assumed 1% infection rate. 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 3.42.

		Prediction	
		Positive	Negative
Actual Value	Positive	TP 99	FN 1
	Negative	FP 198	TN 9702

Figure 3.42: The proper confusion matrix for our MP test, incorporating our knowledge of the 1% infection rate. The values are derived from Figure 3.41.

Now that we have the right chart, we're ready to answer our questions.

Suppose someone gets a positive test result. What's the chance that they really do have MP? That is, what's the conditional probability that someone has MP, given that the test says they do?

In other words, how many of the positive results we get back are true positives? That's the ratio of $TP/(TP+FP)$, which we saw above is the definition of the precision.

In this case, it's $99/(99+198)$, or 0.33, or 33%.

Wait a second. What does this mean? Our test has a 99% probability of correctly diagnosing MP, yet $\frac{2}{3}$ of the times when it gives us a positive result, that person does *not* have the disease.

In other words, *most of our positive diagnoses are wrong*.

This surprising result comes about because there's a huge number of healthy people, and a small chance that each one will get an incorrect diagnosis of positive. Those incorrect positive diagnoses add up fast.

The result is that if someone gets a positive result, we shouldn't 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 order to make them big enough to see, so we'll label them in Figure 3.43.

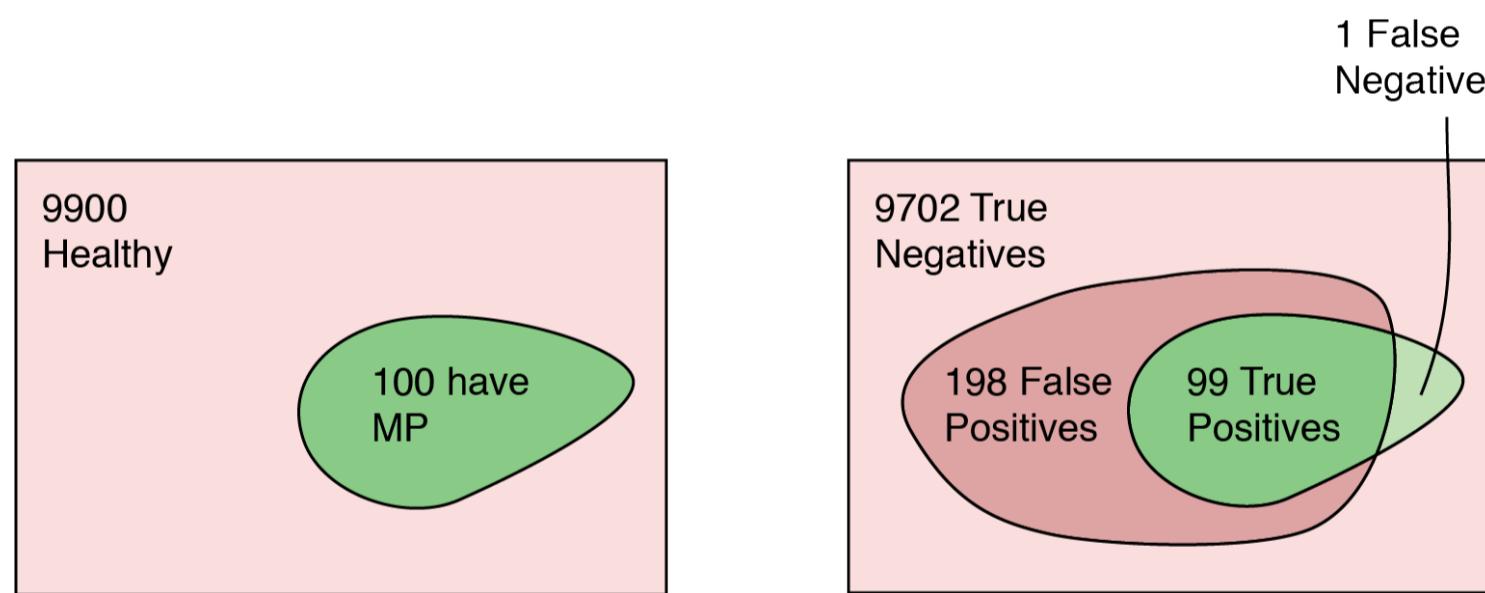


Figure 3.43: Left: The population contains 100 people with MP, and 9900 without. Right: The results of our test. It correctly diagnoses almost all the people with MP as positive, but it also incorrectly labels as positive 198 people without the disease. The areas are not to scale with respect to one another.

We saw above 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 3.44. We can see that the precision incorrectly labels

198 people without MP as positive. This is only a small fraction of the 10,000 people in the town, but compared to the 100 with MP, it's enough for us to look closer at any positive diagnosis.

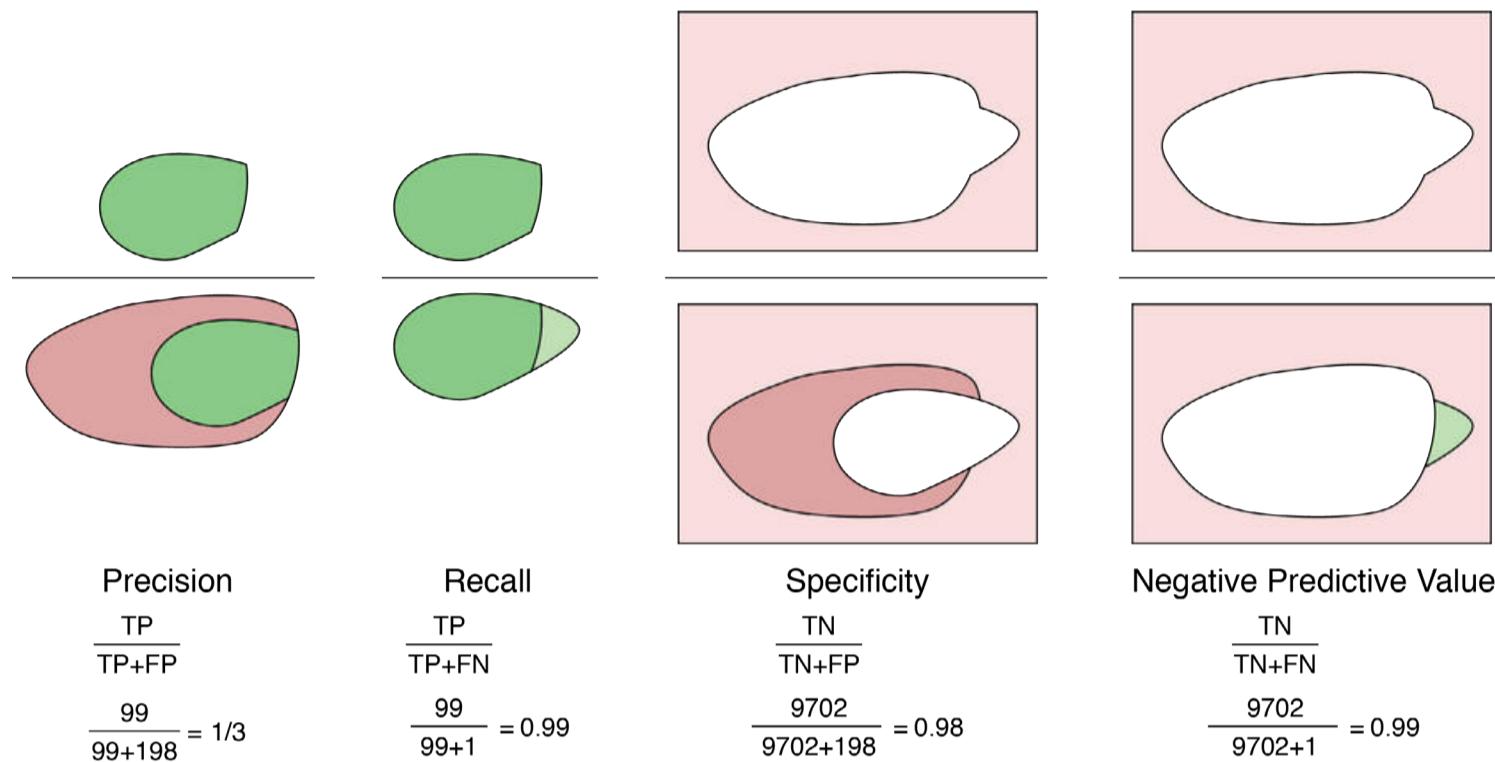


Figure 3.44: Four statistics describing our test for MP, based on the results of Figure 3.43.

What if someone gets a negative result? Are they really clear? That's the ratio of $\text{TN}/(\text{TN}+\text{FN})$, which we saw is called the negative predictive value. In this case it's $9702/(9702+1)$. That's well over 0.999, or 99.9%. 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.

To summarize, the chance that a positive result means that someone actually does have MP is only about 33%. On the other hand, a negative result is 99.9% sure to be really negative.

Figure 3.44 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%. The specificity tells us the percentage of people that are properly diagnosed as negative. Since we only gave 1 person an incorrect negative diagnosis, that result is also very nearly 1.

Thus, out of 10,000 people, we'll 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 another, slower test.

Our example of MP was imaginary, but the real world is full of situations where people are making decisions based on incorrect confusion matrices or bad questions. And a lot of those decisions are related to real and very serious health issues. For example, many women had needless mastectomies because their surgeons misunderstood the probabilities from a breast exam, and gave their patients bad counseling [Levitin16]. Men had a similar problem, because many were given bad advice based on misunderstanding the statistics of using elevated PSA levels as evidence for prostate cancer [Kirby11]. Probability and statistics can be subtle, and we need to always make sure we're working with good data and interpreting it correctly.

Now we know that we shouldn't be fooled by hearing that some test is "99% accurate," or even that it "correctly identifies 99% of the positive cases." In this town where only 1% of the people are infected, 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. As we saw above, our test could find just one person who obviously has MP and declare them positive, and then declare everyone else to be negative, and we could honestly claim that our test is 100% precise. Figure 45 shows the idea.

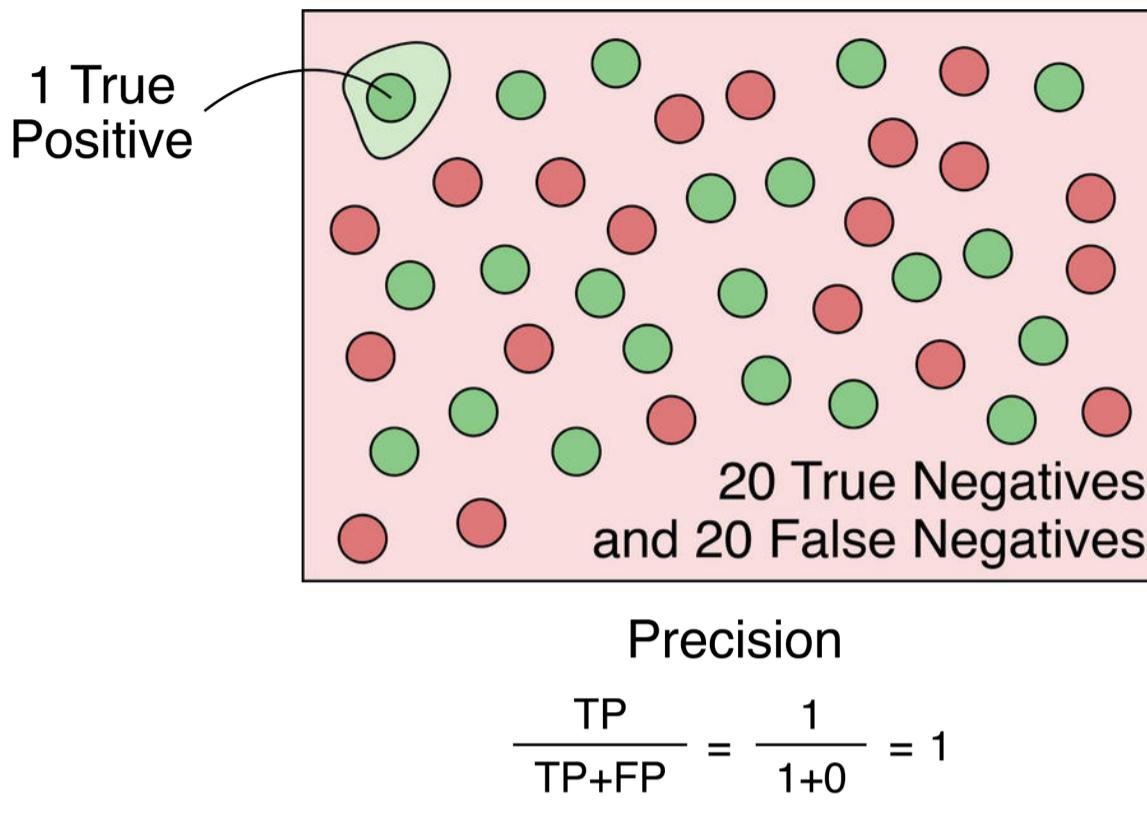


Figure 3.45: Our test identifies just one obvious case of MP (the green blob in the upper left) as positive. Everyone else is classified as negative, whether they have MP (in green) or not (in red). We can still honestly claim that our test has 100% precision.

References

- [Andale17] Andale, “Marginal Distribution”, Statistics How To Blog, 2017. <http://www.statisticshowto.com/marginal-distribution/>
- [Jaynes03] E. T. Jaynes, “Probability Theory: The Logic of Science”, Cambridge University Press, 2003.
- [Kirby11] Roger Kirby, “Small Gland, Big Problem,” Health Press, 2011
- [Levitin16] Daniel J. Levitin, “A Field Guide to Lies: Critical Thinking in the Information Age”, Dutton, 2016.
- [MetService16] Meteorological Service of New Zealand Ltd., “How to Read Weather Maps”, 2016. <http://about.metservice.com/our-company/learning-centre/how-to-read-weather-maps/>

[Walpole11] Ronald E. Walpole, Raymond H. Myers, Sharon L. Myers, Keying E. Ye, “Probability and Statistics for Engineers and Scientists (9th Edition)”, Pearson, 2011.

[Wikipedia16] Wikipedia, “Sensitivity and Specificity”, 2016. https://en.wikipedia.org/wiki/Sensitivity_and_specificity

Chapter 4

Bayes' Rule

A way to think about probability that lets us easily and explicitly combine our expectations of events with the events that really happen.

Contents

4.1 Why This Chapter Is Here	155
4.2 Frequentist and Bayesian Probability	156
4.2.1 The Frequentist Approach	156
4.2.2 The Bayesian Approach	157
4.2.3 Discussion	158
4.3 Coin Flipping	159
4.4 Is This a Fair Coin?.....	161
4.4.1 Bayes' Rule.....	173
4.4.2 Notes on Bayes' Rule	175
4.5 Finding Life Out There	178
4.6 Repeating Bayes' Rule.....	183
4.6.1 The Posterior-Prior Loop	184
4.6.2 Example: Which Coin Do We Have?	186
4.7 Multiple Hypotheses.....	194
References	203

4.1 Why This Chapter Is Here

In Chapter 3 we covered some useful ideas related to probability. As we start to dig a little deeper into probability, 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, and has a lot of common sense behind it, is called the **frequentist** method. The other approach is called the **Bayesian** method. The Bayesian method, while less well known, is popular in machine learning. There are many reasons for this, but one of the most important is that it lets us explicitly identify how we're thinking about the situation we're studying, and it gives us a way to explicitly identify our expectations. This means that if we have good reason to believe something about the process or the results, we can use that information to our advantage, giving us answers faster, or of better quality, or both, than we'd otherwise get.

In this chapter we'll cover the basics of Bayesian probability well enough to allow us to make sense of machine learning papers and documentation that refer to Bayesian ideas. We'll come back to comparing Bayesian and frequentist approaches at the end of the chapter.

The name of the field is drawn from the importance of a formula called **Bayes' Rule**, also called **Bayes' Theorem**, originally presented by Thomas Bayes in the 1700's. It's a substantial topic that we'll address only in its broadest terms [Kruschke14].

Bayesian probability, or the use of Bayes' Rule, is often applicable in everyday life. Any time we want to know how sure we can be about some question of interest, based on a few pieces of evidence, Bayes' Rule is exactly the right tool for the job.

4.2 Frequentist and Bayesian Probability

Many mathematical systems have just one usual approach. For instance, there's really just one way to add a pair of numbers.

Probability is a bit different in that there are at least two different philosophical approaches, each with its strengths and weaknesses.

Many people are familiar with the **frequentist** approach, because it's widely taught, and often used in everyday discussions. In this chapter we'll cover the **Bayesian** approach, which is widely used in machine learning, but generally not as well known.

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 [VanderPlas14]. So it's difficult to summarize the differences without getting into a wealth of detail. Despite being so different, the challenge of describing the distinctions between these two approaches to probability has been called "especially slippery" [Genoveseo4].

Let's look at some of the main differences between these philosophies, without diving into the details. This will help set the stage conceptually for our discussion of Bayes' Rule, which is the cornerstone of the Bayesian approach.

4.2.1 The Frequentist Approach

Generally speaking, someone who takes a frequentist point of view, who we call a **frequentist**, distrusts any specific measurement or observation, since it's considered to be only an approximation of a true, underlying value. For instance, if we want to know the height of a mountain, the frequentist assumes that each measurement is likely to

be at least a little too high or too low. At the heart of this attitude is the belief that a true answer already exists, and our job is 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, a frequentist combines a large number of observations. Though a frequentist considers each measurement to be incorrect, they also expect each measurement to be a noisy approximation of the real value, and generally close to that value. So if we take a large number of measurements, the value that comes up most *frequently* is the one that is 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.

4.2.2 The Bayesian Approach

Using the same broad brush, the Bayesian approach doesn't distrust the measurements. Instead, someone who follows this philosophy, who we call a **Bayesian**, trusts each observation as an accurate measure of *something*, though it might be a slightly different something each time. The Bayesian attitude is that there is no "true" value waiting to be found at the end of a process. For instance, each 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'll never be the identical two points every time. So even though each of these measurements is different from the others, each one is accurate in its own way. The concept of a single "true" height of the mountain is meaningless.

Instead, there's only a range of possibilities, each described by a probability. As we take more observations, those ranges of possibilities will generally become more narrow, but they never go away. The "truth" is considered a fuzzy idea that we can only describe with probabilistic language. One feature of this approach is that since there is no single

“true” value being sought, if whatever we’re measuring happens to change over time, the results of the Bayesian approach would naturally shift to accommodate that change.

4.2.3 Discussion

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.

Let’s compare the two approaches in a loose way with an example.

Suppose we want to know the length of a dull, unsharpened pencil with a somewhat used eraser. So using great care, we put our pencil up against a precise ruler and write down its length. Just to be sure, we’ll do that a few times. We’ll probably end up with several different, but very similar, measurements.

To a frequentist, there is *one true value* for the length of the pencil, and our job is to find that value. Each measurement is considered unreliable and suspicious, since it probably has some error due to us not reading the ruler properly, or the presence of a shadow that disguised where the tip of the pencil was, or perhaps using different criteria to determine the location of the end of the blunt point. By combining these measurements, we can chip away at those errors and zero in on that correct answer. Ultimately, by combining the measurements carefully, the value that emerges most frequently is likely to be the right answer.

To a Bayesian, we don’t imagine that there’s a true value for the pencil’s length. There are just different possible values, each with a probability. We start by trusting the measurements. We interpret their differences

as telling us that there's inherent ambiguity in the idea of the length of the pencil. The blunt tip of the point and the flaky eraser mean that there isn't any "real" answer for the pencil's length, but just a range of answers, each with its own confidence. Our job is to use our measurements to gradually refine the probabilities of the multiple possible values for the pencil's length.

The differences between the frequentist and Bayesian approaches are often subtle, but they lead to different styles of working with data and deriving information. 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 [Stark16].

4.3 Coin Flipping

People love to use flipping (or tossing) coins as an example when discussing probability [Cthaeh16a] [Cthaeh16b]. It's a popular approach because it's familiar to everyone, and each flip has only two possible outcomes: heads or tails. This makes the math so simple 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'll use that as our running example here.

We say that a **fair** coin is one that will, on average, come up heads half the time, and tails the other half. A coin that isn't fair (even if it looks and feels as if it is), we call a **rigged**, **unfair**, or **cheater's** 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 would come up heads about 10% of the time, and a bias of 0.8 tells us to expect heads about 80% of the time. If our coin has a bias of 0.5, it will come up heads and tails equally often, and it would be indistinguishable from a fair coin. We actually might say that a coin with a bias of 0.5 is the definition of a fair coin.

Figure 4.1 illustrates the idea with three different coins, using one row per coin. On the left of each row we show 100 consecutive flips of the coin. On the right, we show our estimate of that coin's bias after each flip, found by dividing the number of heads we've found up until then by the total number of flips.

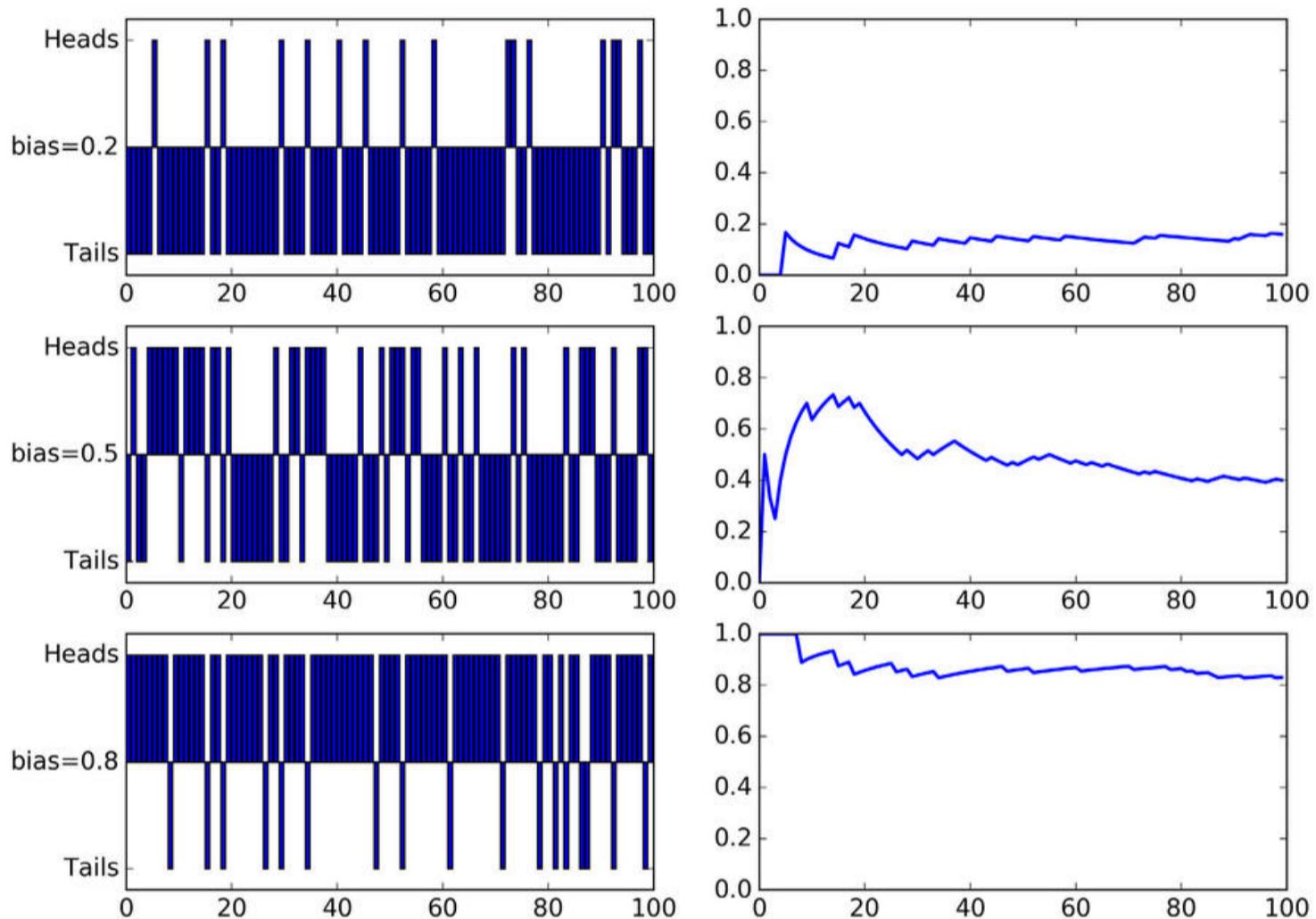


Figure 4.1: The bias of a coin tells us how likely it is to come up heads. Each row shows the results for one coin. The left plot shows 100 consecutive flips, and the right shows our running estimate of the bias. Top row: A coin with a bias of 0.2. Middle row: A coin with a bias of 0.5. Bottom row: A coin with a bias of 0.8.

The plots on the right of Figure 4.1 show a frequentist's approach to finding the bias. Let's consider how we might estimate a coin's bias using a Bayesian point of view.

4.4 Is This a Fair Coin?

Let's suppose that we have a friend who's a deep-sea marine archaeologist. Her latest discovery was an ancient wreck that had, among its treasures, a box containing a marked board and a bag of two identical-looking coins. She thinks it's a 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 $2/3$ of the time (that is, it has a bias of $2/3$). The difference between a bias of $1/2$ and one 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 picking them up and casually feeling them.

The game involves players trying to figure out which coin is which, and various forms of bluffing and betting along the way. When the game is over, they 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 will drop 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's asked us to help sort it out. She's given us two envelopes, marked "Fair" and "Rigged," and our job is to put the matching coin in each envelope.

We could use the spinning test above to work out which coin is which, but let's do it with probabilities, so we can get some experience with thinking that way.

Let's continue to use the dart-throwing metaphor of Chapter 3. This lets us talk about estimating the ratios of areas by throwing darts at a wall that's been painted with regions of different colors.

The easiest way to get started is to pick a coin, flip it once, see if it comes up heads or tails, and then see what we can do with that information.

So let's first select a coin. Since we can't tell the two coins apart by looking at them, we have a 50% chance of picking up the fair coin, and the same 50% chance for selecting the rigged coin.

This choice sets up our big question: which coin do we have? 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 re-phrase 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 sure that we don't, we'll know everything we need.

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 we can already make a valid, quantified result about which coin we have.

Let's represent what we've done so far with colored wall regions. Our first step was to choose a coin, where the choice was 50/50, so we can imagine painting the wall so it's covered in two regions of equal size (the two regions cover the wall completely because the dart always hits a colored region, corresponding to the fact that we always pick one coin or the other). We'll paint the fair region with flax (a kind of beige), and the rigged region with red, as in Figure 4.2. So picking one of the two coins at random corresponds to throwing a dart at the wall and landing in one of the two equal areas.

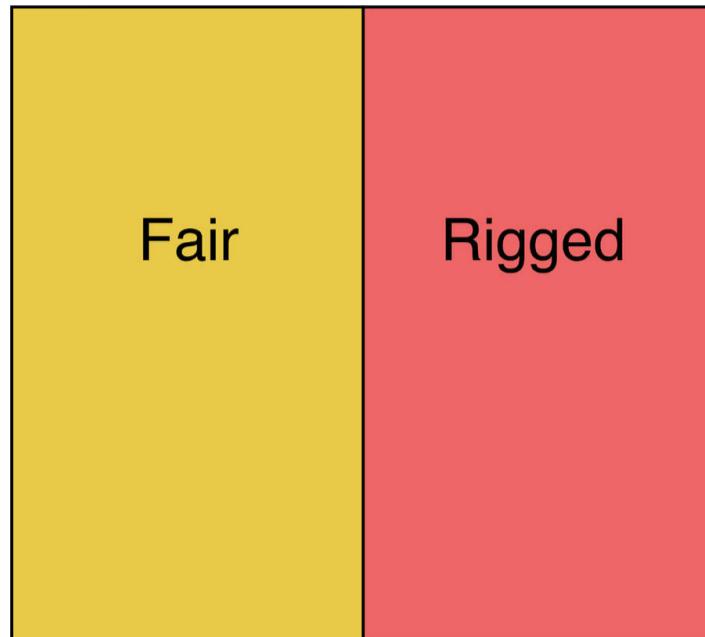


Figure 4.2: If we pick one of the two coins at random, it's the same as throwing a dart at a wall that's been painted with two equal areas, one for the fair coin and one for the rigged coin.

We can paint the wall in a more revealing way. 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.

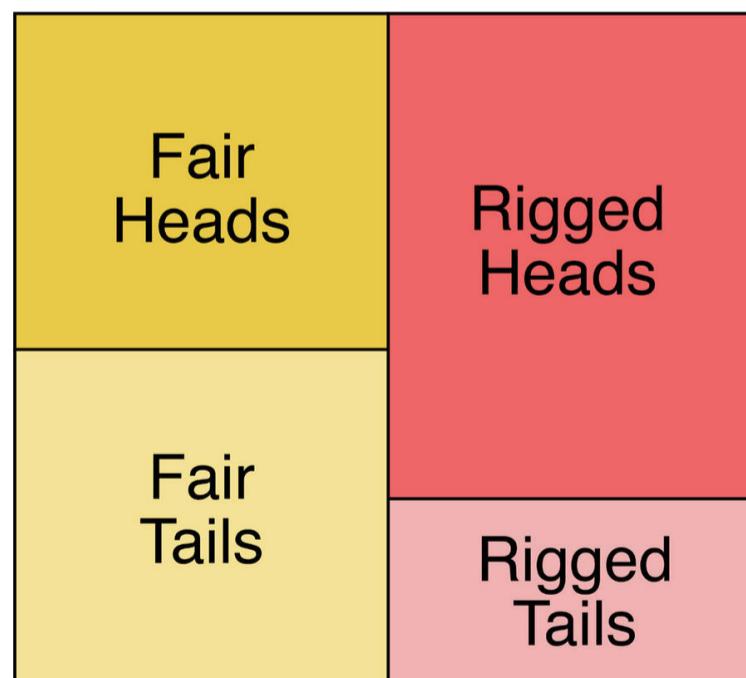


Figure 4.3: We can split up the fair and rigged regions of the wall into "heads" and "tails" for each coin, using the information we already know about how they are likely to come up when flipped.

In Figure 4.3 we also split the rigged side. Since we know from our friend that the rigged coin has a $2/3$ chance of coming up heads, we assigned $2/3$ of its area to heads and $1/3$ 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).

If we throw a dart at a wall painted like Figure 4.3, as we did in Chapter 3, our dart will strike 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), where the comma means that both conditions are true.

Remember our question: What is the probability that we picked the fair coin? We can tighten that up by including 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 terms of our 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.

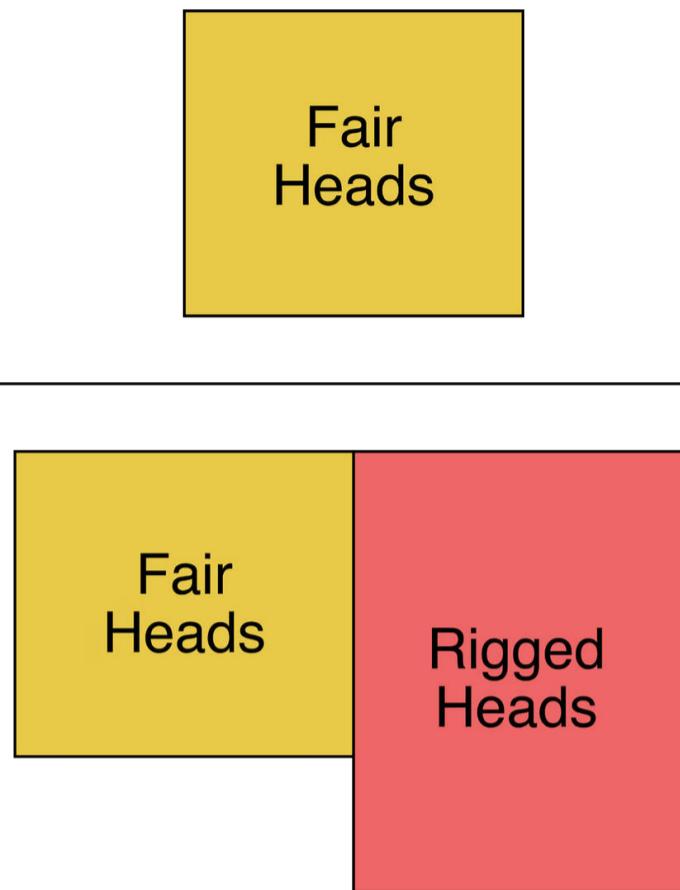


Figure 4.4: If the coin comes up heads, how likely is it that we were flipping the fair coin? It's the size of the region where a fair coin gives us heads, divided by all the areas combined that would give us heads.

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 our coin came up heads, it's a little more likely that it's the rigged coin, just as a dart thrown at the wall is more likely to land in the (rigged,heads) region than the (fair,heads) region.

Below 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 need to account for all of the possible events that would give us that result. In this case, the bottom half of the ratio is the sum of all the ways we could get heads. In other words, we could have received heads from the fair coin, or from the rigged coin, so "all the ways to get heads" means combining these two possibilities.

Let's re-phrase this picture using our 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 is showing us the chance that our coin, which we know came up heads, was 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's the answer to our question, so this probability is exactly what we want to know.

We can put this all together into Figure 4.5.

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

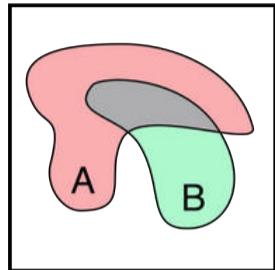
Figure 4.5: Translating Figure 4.4 into the language of probability. The ratio we've created is $P(F|H)$, the probability that we chose the fair coin, given that the coin came up heads. We've scaled the shapes down a bit to save space.

Can we plug in numbers into this diagram and come up with an actual probability?

Sure, in this case we can, because the situation is 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. As we saw in Chapter 3, we can write any joint probability in two different and equivalent ways. The terms in those other versions will usually be much easier for us to put numbers to. Those two approaches are repeated here as Figure 4.6 and Figure 4.7.

$$P(A,B) = P(A|B) \times P(B)$$

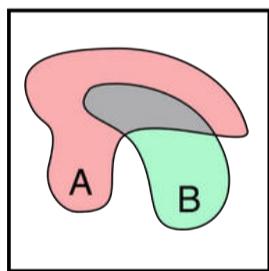


$$\frac{\text{shaded area}}{\text{square}} = \frac{\text{shaded area}}{\text{green area}} \times \frac{\text{green area}}{\text{square}}$$

$$\frac{\text{shaded area}}{\text{square}} = \frac{\text{shaded area}}{\text{square}} \times \frac{\text{green area}}{\text{shaded area}}$$

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 $P(B)$. This is a version of a figure from Chapter 3.

$$P(A,B) = P(B|A) \times P(A)$$



$$\frac{\text{shaded area}}{\text{square}} = \frac{\text{shaded area}}{\text{red area}} \times \frac{\text{red area}}{\text{square}}$$

$$\frac{\text{shaded area}}{\text{square}} = \frac{\text{shaded area}}{\text{square}} \times \frac{\text{red area}}{\text{shaded area}}$$

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 $P(A)$. This is a version of a figure from Chapter 3.

Let's write our fraction in Figure 4.5 without the colored boxes, and then replace $P(H,F)$ with the version in Figure 4.7. It 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, or $P(H|F)$, by the chance of having the fair coin in the first place, or $P(F)$. This change is shown Figure 4.8.

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

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

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. Because we don't know the value of $P(H,F)$, we can use Figure 4.7 to replace $P(H,F)$ with $P(H|F)$ times $P(F)$. These are values that we usually know (or can find out).

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.

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

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

Figure 4.9: We can replace the other two joint probabilities in Figure 4.8 with their expanded versions as well. The expansion of $P(H,F)$ is just the same as it was before, and $P(H,F)$ follows the same pattern.

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 find a numerical value for each term in Figure 4.9.

$P(F)$ is the probability that we picked the fair coin when we started. Since we chose between the two coins at random, $P(F)=1/2$. Since $P(R)$ is the probability that we picked the rigged coin when we started, $P(R)$ is also $1/2$.

$P(H|F)$ is the chance of getting heads from the fair coin, and $P(H|R)$ is the chance of getting heads from the rigged coin. In other words, these values are the bias of each coin. $P(H|F)$ 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.

$$\begin{aligned}
 P(F|H) &= \frac{P(H|F) \times P(F)}{P(H|F) \times P(F) + P(H|R) \times P(R)} \\
 &= \frac{\frac{1}{2} \times \frac{1}{2}}{\left(\frac{1}{2} \times \frac{1}{2}\right) + \left(\frac{2}{3} \times \frac{1}{2}\right)} \\
 &= \frac{\frac{1}{4}}{\frac{1}{4} + \frac{1}{3}} = \frac{\frac{3}{12}}{\frac{3}{12} + \frac{4}{12}} = \frac{3}{7} \approx 0.43
 \end{aligned}$$

Figure 4.10: Finding the probability that we picked the fair coin, given that we just saw heads. The chance of picking the fair coin, or $P(F)$, is $1/2$, and the chance of picking the rigged coin, $P(R)$, is also $1/2$. The probability of getting heads from the fair coin, $P(H|F)$, is its bias, $1/2$. And we know from our friend that the probability of the rigged coin coming up heads, $P(H|R)$, is its bias, $2/3$. So we put those numbers into the last line of Figure 4.9, and we find that the chance that we're using the fair coin is about 43%.

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% chance we have the fair coin, and therefore a 57% chance that we have the rigged coin. That's a 14% difference, from one flip!

Just for fun, let's suppose we'd received tails on our first flip. 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.

The picture is just the ratio of the area of the wall where we'd land in the (fair,tails) area compared to the total area of all the regions that could have given us tails, (fair,tails)+(rigged,tails). Figure 4.11 shows these areas.

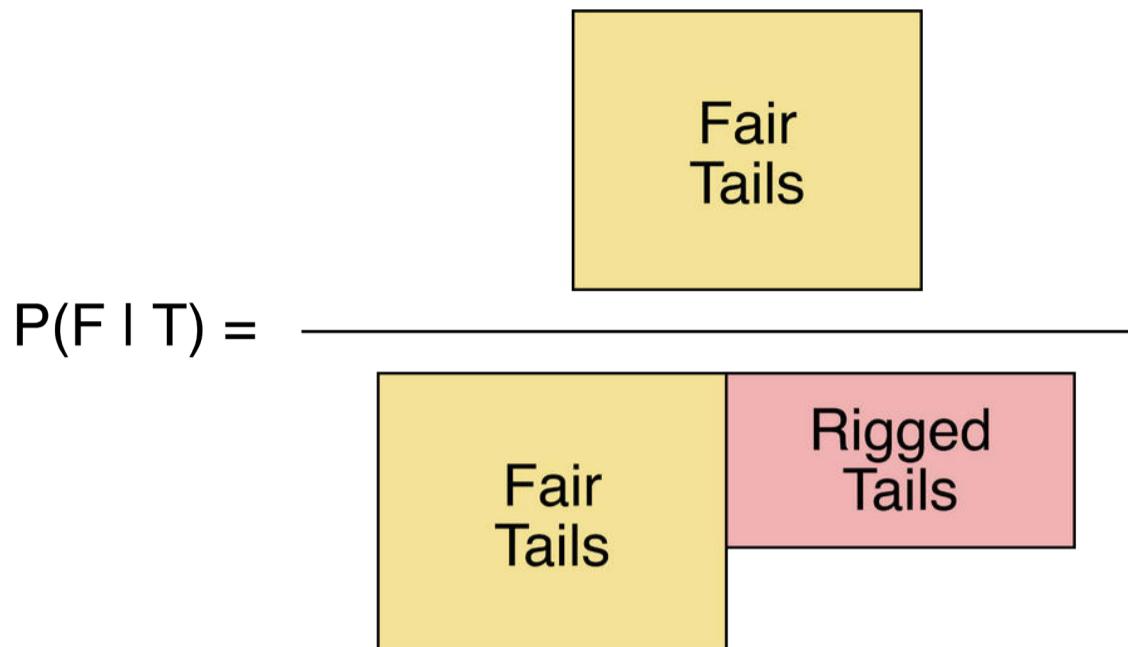


Figure 4.11: If our first flip came up tails, what's the chance that we were using the fair coin? It's the area of the (fair,tails) region divided by the total area that could have given us tails, or (fair,tails)+(rigged,tails). Since it's more likely we'd get tails if we had the fair coin than the rigged coin, we'd expect the probability that we have the fair coin to be greater than 50%.

Since the bias is the chance of coming up heads, $1 - \text{bias}$ for each coin is the chance of that coin coming up tails (since the coin must land on either heads or tails).

For the fair coin, the chance of 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. So 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 4.12 shows the steps.

$$\begin{aligned}
 P(F|T) &= \frac{P(T|F) \times P(F)}{P(T|R) \times P(R)} \\
 &= \frac{\frac{1}{2} \times \frac{1}{2}}{\left(\frac{1}{2} \times \frac{1}{2}\right) + \left(\frac{1}{3} \times \frac{1}{2}\right)} \\
 &= \frac{\frac{1}{4}}{\frac{1}{4} + \frac{1}{6}} = \frac{\frac{3}{12}}{\frac{3}{12} + \frac{2}{12}} = \frac{3}{5} = 0.6
 \end{aligned}$$

Figure 4.12: What if we got tails on our coin flip? We can use the last line of Figure 4.9 again, if we replace $P(H|F)$ and $P(H|R)$ with their “tail” versions. $P(T|F)$ tells us the probability of getting tails from the fair coin, which we know is $1/2$. $P(T|R)$ tells us the probability of getting tails from the rigged coin, which we know from our friend is $1/3$. The result is that $P(F|T)$, the probability that we’re using the fair coin given that we just got back tails, is 60%.

This is an even more dramatic answer, telling us it’s 60% likely that this result of tails means that we are flipping the fair coin (and thus it’s 40% 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% probability of a fair coin, but if we get tails, we have a 60% probability of a fair coin.

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

4.4.1 Bayes' Rule

Let's go back to our first version where we got heads back from our first flip.

In Figure 4.5 through Figure 4.10 we saw several different ways to write expressions for $P(F|H)$, the probability that we picked a fair coin given that we saw heads.

Going back to the version in Figure 4.8 (repeated on the first line of Figure 4.13), 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 had to come from either the fair or rigged coin). If we were dealing with, say, 15 coins, then we'd have to write a sum of 15 joint probabilities, which would make for a very messy expression. So we usually use a shortcut, and write these combined probabilities as $P(H)$, or “the probability of getting heads.” This implicitly means the sum of all the ways we could have gotten heads. So if we had 20 different coins, this would be the sum of the probability of each of those coins giving us heads. Figure 4.13 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 4.13: The last line of Figure 4.8, but we've replaced the bottom part of the ratio with the symbol $P(H)$. This stands for the probability we got heads from any kind of coin, and it's the sum of every probability of getting heads. This is a more compact to write this expression when there are many different possibilities.

Figure 4.14 shows this most recent version all by itself. This is the famous **Bayes' Rule** or **Bayes' Theorem** that we mentioned at the start of this section.

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

Figure 4.14: Bayes' Rule, or Bayes' Theorem, as it's often written. This is just the last line of Figure 4.13. Following convention, we've dropped the explicit multiplication sign in the top half of the ratio, leaving it implied.

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 did indeed have a fair coin, it would come up heads. We multiply that by $P(F)$, the probability that we have a fair coin. As we've seen, this 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. We're showing here the normal way to write Bayes' Rule, where we leave out the explicit multiplication sign, and rely on the mathematician's convention that two values placed side by side should be multiplied.

Now we divide that by $P(H)$, or the probability that our coin would come up heads, *taking into account both the fair and rigged coins*. This is how likely we'd be to get heads using *either* of these coins.

Bayes' Theorem is usually written in the form of Figure 4.14, because it breaks things down into pieces that we can conveniently measure (it's often written with the more generic letters A and B rather than F and H). All we need are the probabilities of the choices (here, $P(F)$ and $P(R)$), and the conditional probabilities of the result we saw for each choice ($P(H|F)$ and $P(H|R)$). We just plug in the numbers 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 4.13.

We can see now why we said at the start of our discussion that 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 Bayes' Rule calculates the value of a conditional probability, and that's how we phrase a conditional probability in words. If we can't express our problem in that form, then Bayes' Rule isn't the right tool for answering it.

4.4.2 Notes on Bayes' Rule

Bayes' Rule can seem 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 just did above gives us the second line of Figure 4.15.

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

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

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

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

Figure 4.15: How to re-discover Bayes' Rule if we forget, or a quick demonstration of why it's true. We write the joint probability of H and F on the first line in each version, and then expand each one to get the second line. On the third line we divide each side by $P(H)$, leaving us with Bayes' Rule on the bottom line.

To get Bayes' Rule, just divide each side by $P(H)$. This can be a handy way to come back up with the rule if we need it and have forgotten it.

Each of the four terms in Bayes' Rule has a conventional name, summarized in Figure 4.16.

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

Figure 4.16: The four terms in Bayes' Rule, with their names. Here we used the traditional, generic letters A and B to label the various probabilities.

In Figure 4.16 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 it and observing the result, 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 the word "evidence" 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. Quite reasonably then, 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**.

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** [Genovese04].

Earlier in this chapter we said that a virtue of the Bayesian approach is that it lets us explicitly identify our preconceptions and expectations. Those ideas are all captured in how we assign a value to the prior.

4.5 Finding Life Out There

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: Is there life on this planet?

Your experience is that about 10% of these planets have some kind of life. Usually it's 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."

Now we 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, any non-Earth life forms.

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.

The probes they sent us out with have the performance shown in Figure 4.17. To get these results, they sent their probes down onto 1000 known planets, 101 of which contained life. The probe correctly reported that it found life (that is, a true positive) 100 times out 1000.

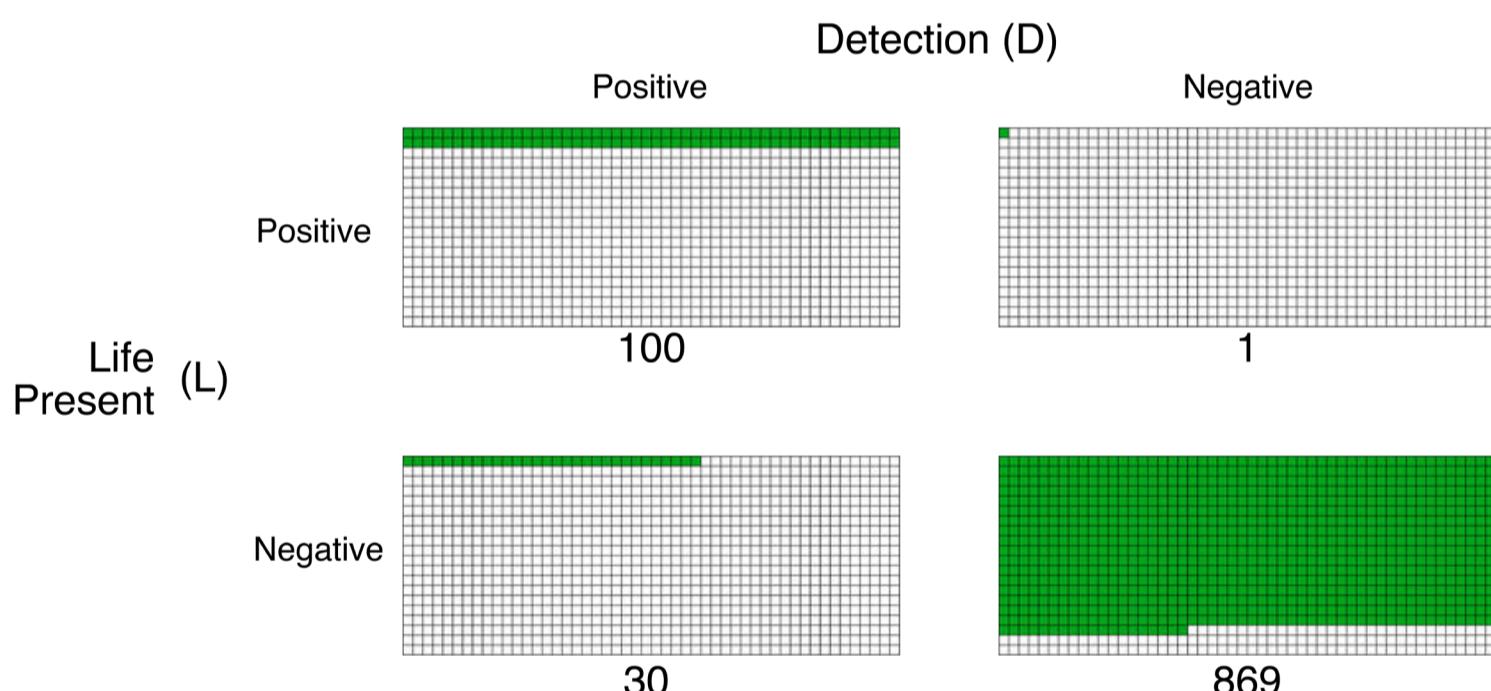


Figure 4.17: The performance of our life-seeking probe on 1000 test planets, 101 of which harbor life. The probe correctly reported life on 100 planets, but missed existing life once. Of the 899 barren planets, it correctly reported that 869 of them had no life, but incorrectly reported life was found on 30 planets.

Of the 100 planets with life, the probe missed life signs (the 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 (the false positive) 30 times. All told, these aren't bad numbers.

Writing 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 4.18. 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).

		Detection (D)		
		Positive	Negative	
Life Present (L)	Positive	TP 100	FN 1	$P(L) = 101/1000$
	Negative	FP 30	TN 869	$P(\text{not-}L) = 899/1000$
		$P(D) = 130/1000$	$P(\text{not-}D) = 870/1000$	

Figure 4.18: The confusion matrix that summarizes Figure 4.17, demonstrating the performance of our life-detecting probe. The four marginal probabilities are shown in the right and bottom margins.

Figure 4.19 gathers up the four marginal probabilities, plus two conditional probabilities that we’ll be using.

$$P(D) = 130/1000 \quad P(\text{not-}D) = 870/1000$$

$$P(L) = 101/1000 \quad P(\text{not-}L) = 899/1000$$

$$P(D | L) = 100/101 \quad P(\text{not-}D | L) = 1/101$$

Figure 4.19: Summaries of the four marginal probabilities, and two conditional probabilities, that we’ll be using based on the data in Figure 4.18.

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 to be 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 $P(\text{not-}D|L)$, we carried out the calculation the other way. It missed finding life once out of 101 planets. So 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 fun, we can use the definitions in Chapter 3 to also find the probe's **accuracy** as $969/1000$, which is 0.969. We can also find its **precision** as $100/130$, which is about 0.77.

Now we can answer our question. The probability that there actually *is* life, given that our probe says there isn't, is $P(L|\text{not-}D)$. We can use Bayes' Rule, and plug in the numbers from above, or the fractions in Figure 4.19. We do just that and grind through the numbers in Figure 4.20.

$$\begin{aligned}
 P(L|\text{not-}D) &= \frac{P(\text{not-}D|L) \times P(L)}{P(\text{not-}D)} \\
 &= \frac{\frac{1}{101} \times \frac{101}{1000}}{\frac{870}{1000}} \\
 &= \frac{\frac{1}{1000}}{\frac{870}{1000}} = \frac{1}{870} \approx 0.001
 \end{aligned}$$

Figure 4.20: Working out the probability that a planet has life, given that the probe reported that no life was detected. We just plug in the values from Figure 4.19 into Bayes' Rule.

This is reassuring. The probability that there's life on that planet, given that our probe said there wasn't, is about 1 in 1000. 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 suppose that the probe 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 we work out $P(L|D)$, the probability of life given that the probe detected life. The numbers are worked through in Figure 4.21.

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

$$= \frac{\frac{100}{101} \times \frac{101}{1000}}{\frac{130}{1000}}$$

$$= \frac{\frac{100}{1000}}{\frac{130}{1000}} = \frac{100}{130} \approx 0.77$$

Figure 4.21: Working out the probability that a planet has life, given that the probe reported it found signs of life. As before, we just plug in the value from Figure 4.19 into Bayes' Rule.

Wow. We can be about 77% 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 has a greater chance of reporting a false positive than a false negative.

As we mentioned, we can send more probes to increase our confidence in either result. But we'll never get to absolute certainty either way. We might get very close to a confidence of 1.0 or 0.0, but we'll never get all the way there. So 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 see how sending more probes can help us increase our confidence.

4.6 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've approached it as a one-time event, plugging in what we know about the system and getting back a probability.

But one event is not much. Let's return to our two-coin example, where one coin is fair, and one is rigged to come up heads more than half the time. We flipped the coin, 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 we'll put Bayes' Rule in the heart of a loop, where each new piece of data gives us a new posterior, which we then use as the prior for the next observation. Over time, if the data is consistent, the prior should hone in on the underlying probabilities we're looking for.

4.6.1 The Posterior-Prior Loop

In Figure 4.16 we gave names to the terms of Bayes' Rule. These aren't the only names that are used. We also refer to the elements of Bayes' Rule in terms of a **hypothesis** and an **observation** (sometimes abbreviated **Hyp** and **Obs**). Our hypothesis is that something is true (for example, “we have the fair coin”). The observation is what we've seen about our system (for example, “we got heads”). Figure 4.22 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 4.22: Writing Bayes' Rule with more informative labels.

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 probability with the likelihood of the observation given that hypothesis, and get the joint probability of both the observation and hypothesis being true. We then scale that by the evidence, or the probability that the observation could have come about by any means. The result is the posterior, which tells us the probability that our hypothesis is true, given the observation.

Here's the key insight to using Bayes' Rule repeatedly. We know that the posterior is the probability of the hypothesis being true, if we know that the observation happened. But we *do* know that the observation happened, because that's why we computed the posterior. So the posterior is the probability of the hypothesis. Compare this to the prior, which is our estimate of the probability of the hypothesis.

In short, *the posterior is an improved version of the prior*, based on the observation we just saw.

Suppose that another observation arrives (for example, we flip the coin again). Instead of using our original prior as our estimate of the probability of the hypothesis, we use the posterior we found from the previous observation.

Let's summarize the basic loop, shown in Figure 4.23.

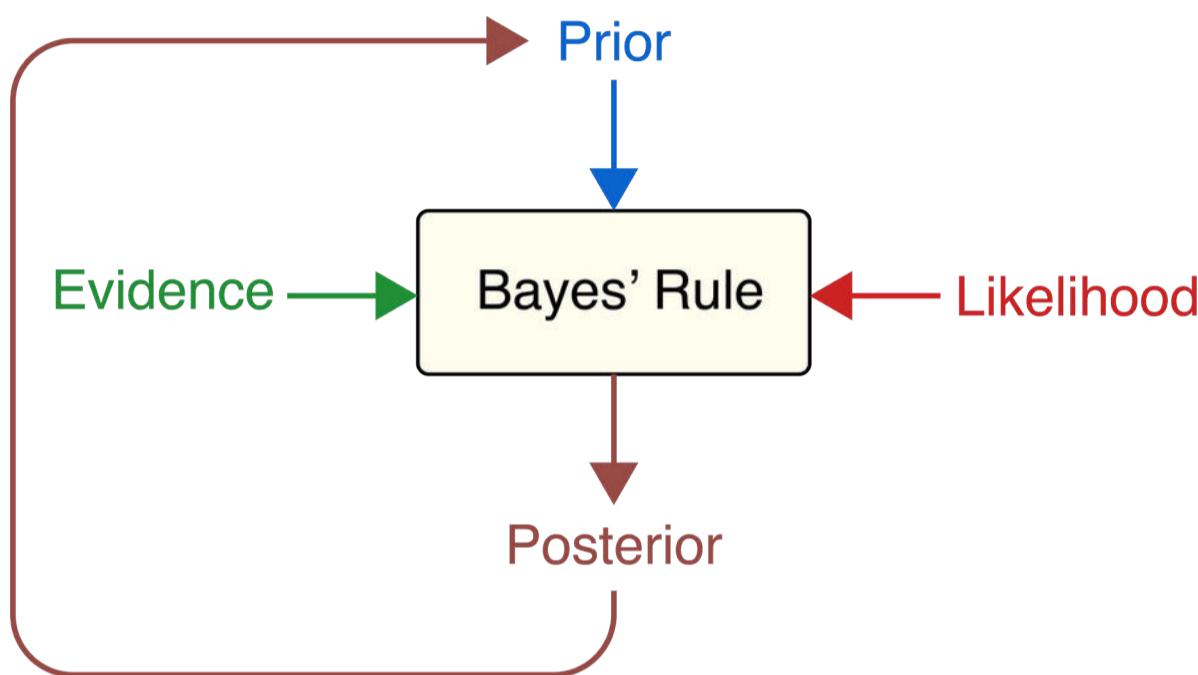


Figure 4.23: Each time we have a new observation, we combine the evidence, the likelihood of that observation, and the prior to compute a posterior. That posterior is then used as the prior when a new observation is evaluated.

We start with a prior. This comes from analysis, experience, data, an algorithm, or just guesswork. Then we make an observation and start the loop. We combine the evidence, the likelihood of that observation, and the prior using Bayes' Rule and compute a posterior. That posterior becomes our new prior. Now when another observation arrives, we repeat the process, using our previous posterior as the new prior.

The idea is that each time through the loop, the observation helps us compute a posterior that, because the observation really did happen, is an improved version of the prior. It's improved because it incorporates this latest observation in addition to all the previous observations.

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

4.6.2 Example: Which Coin Do We Have?

Recall our archaeologist friend and her two-coin problem. Let's generalize it so we can try out a number of variations.

We'll suppose we have a *fair* coin, which comes up heads and tails equally frequently. Rather than having just one rigged coin, we'll suppose we also have a set of *multiple* rigged coins. Each rigged coin will come up heads with a probability given by its **bias**, often written with the lower-case Greek θ (theta). On each coin there's a little sticker with that coin's value of bias (which someone else figured out for us), so there's no mystery.

A coin with a bias near 0 will almost never come up heads, and a coin with a bias near 1 will come up heads almost every time.

Our process will be to pick one of the rigged coins, note the value on its sticker (that is, we'll remember its bias), and then we'll take off the sticker. We'll put the coin into a bag and drop in another coin that looks identical but is known to be fair. Then we'll pull out a random coin from the bag, being equally likely to get either one.

Now we're back to our two-coin problem, but we can select whatever value of bias we like for the rigged coin just by selecting the appropriate coin to put into the bag.

Now we've got one of the two coins, and we want to know if we picked the fair coin. We'll use the repeated form of Bayes' Rule by flipping the coin 30 times, record the heads and tails we get, and watch what Bayes' Rule does with the observation, or result, of each flip.

Note that with only 30 flips, we can see unusual events. For instance, we might have the fair coin and get 25 heads and 5 tails. It's very unlikely, but possible. But we could get the same results from a rigged coin with a high bias. We'll see how Bayes' Rule handles observations like this.

Let's start by choosing between a fair coin and a rigged coin with a bias of 0.2, so we'd expect 2 heads out of every 10 flips. And let's suppose that out of our 30 flips, only 20% (that is, 6) came up heads, so the other 24 came 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 that we have the rigged coin.

Figure 4.24 shows the result of Bayes' Rule after each flip. The probability that we have the fair coin is shown in flax (or beige) as before, while the probability of the rigged coin is shown in red. The two probabilities always add up to 1.

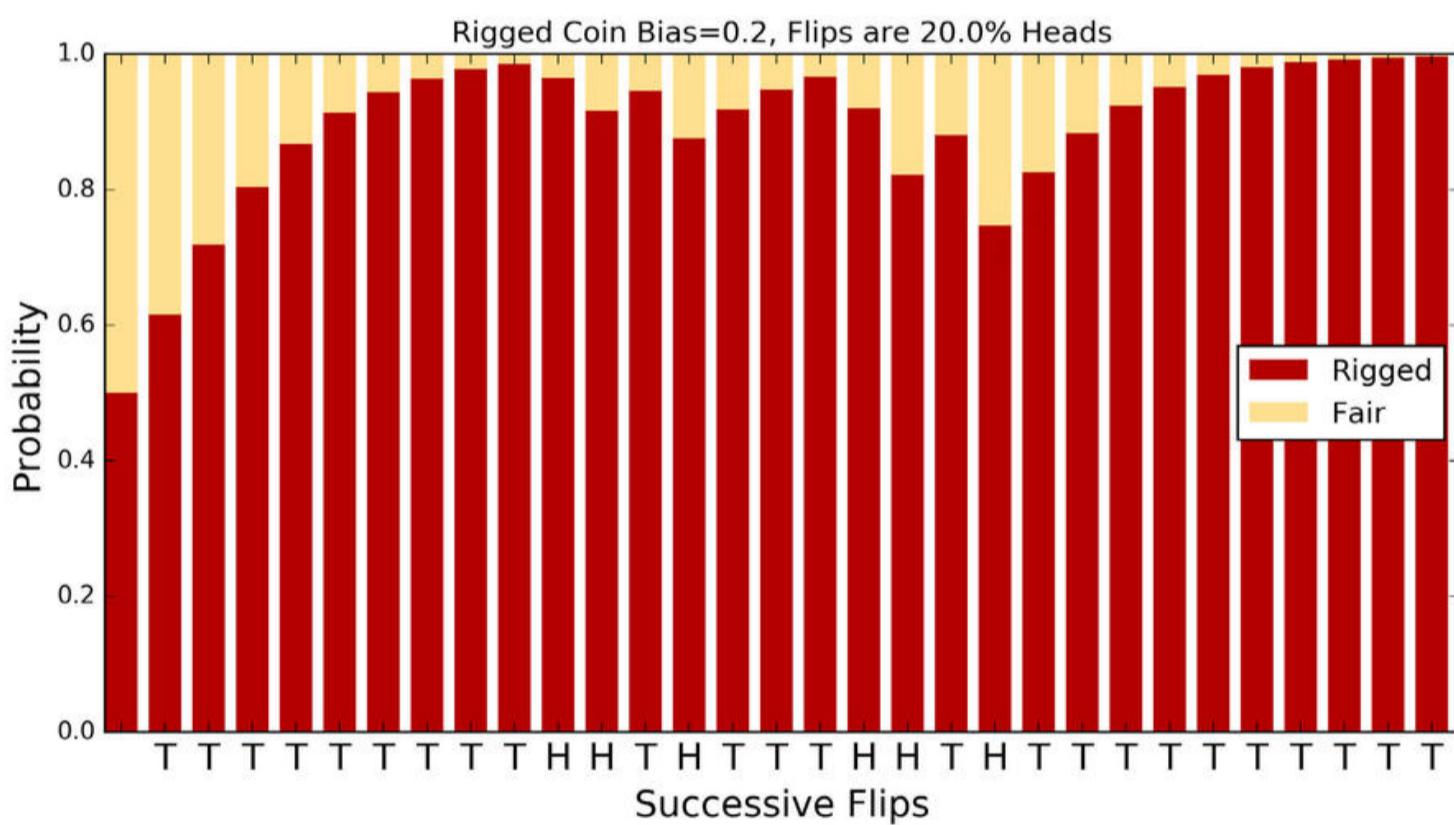


Figure 4.24: What is the probability that we have the fair coin? That's shown in flax (beige), while the complement of that probability (the probability that we have the rigged coin) is shown in red. The 30 flips are labeled along the bottom.

To understand what this chart is telling us, look first at the labels 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 here and there.

Now consider the bars, starting at the left. This column shows which coin we think we have before we've flipped the coin at all, and the probabilities are both 0.5. After all, the chances are equal that we picked either coin, and we haven't flipped it yet to gain any data.

The bar to the right shows the result 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% of the flips are tails. That's what we'd expect from the rigged coin, so its probability quickly approaches 1. Note that it dips about 2/3 of the way through the run when we get a bunch of heads close to one another, but then picks back up with each new tail.

In this example the data we got back was a close match to the rigged coin. Let's keep this coin, but suppose we happen to get fewer heads on the next run, perhaps only 3 in total. Running this through the loop produces the results in Figure 4.25.

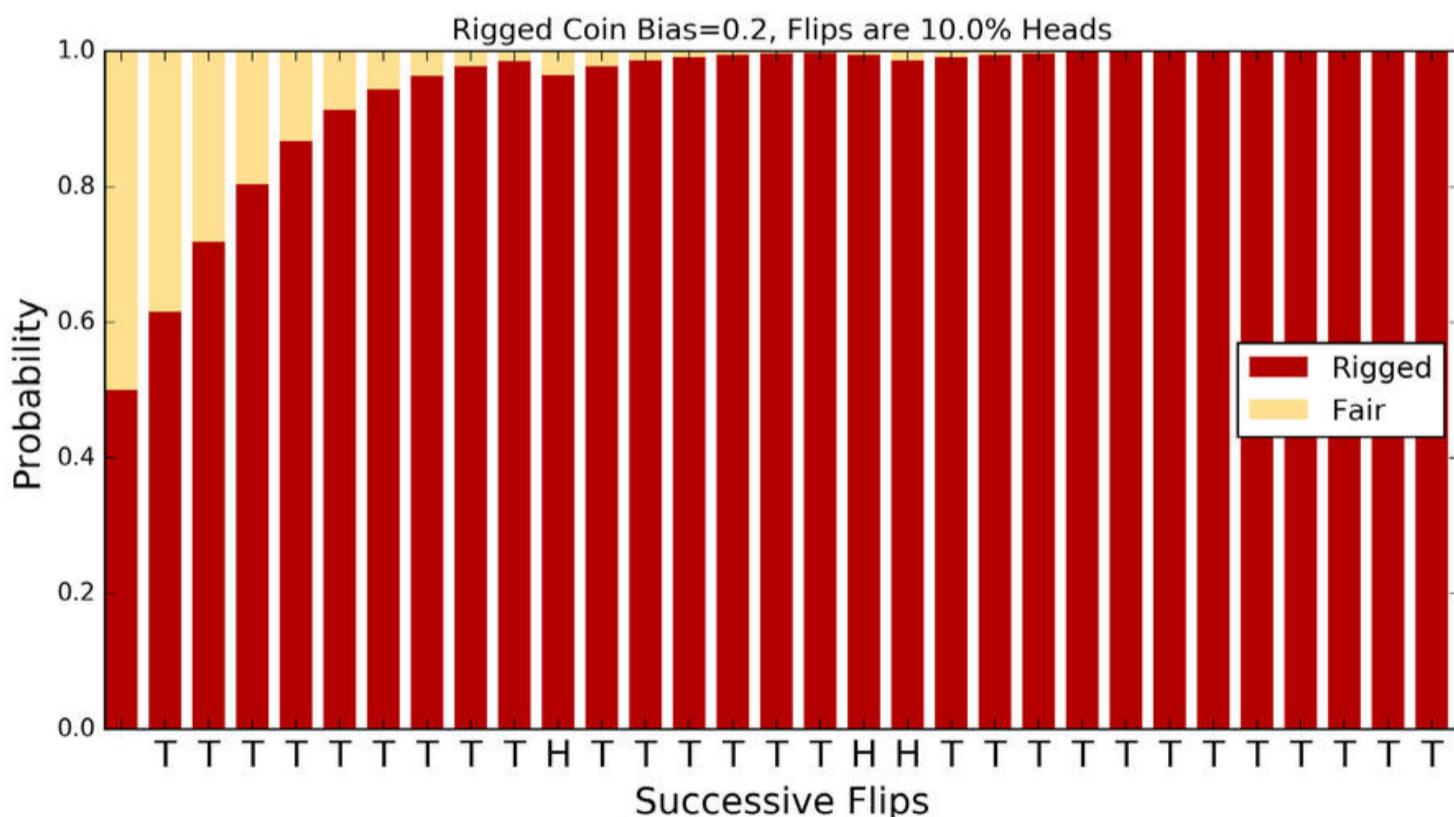


Figure 4.25: Using the same coin with a bias of 0.2, but this time we happened to get only 3 heads in our 30 flips.

We get to about 90% confidence that we've got the rigged coin after just four flips.

Suppose we do another run of 30 flips, and 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. So the fair coin seems more likely. Figure 4.26 shows our results from Bayes' Rule.

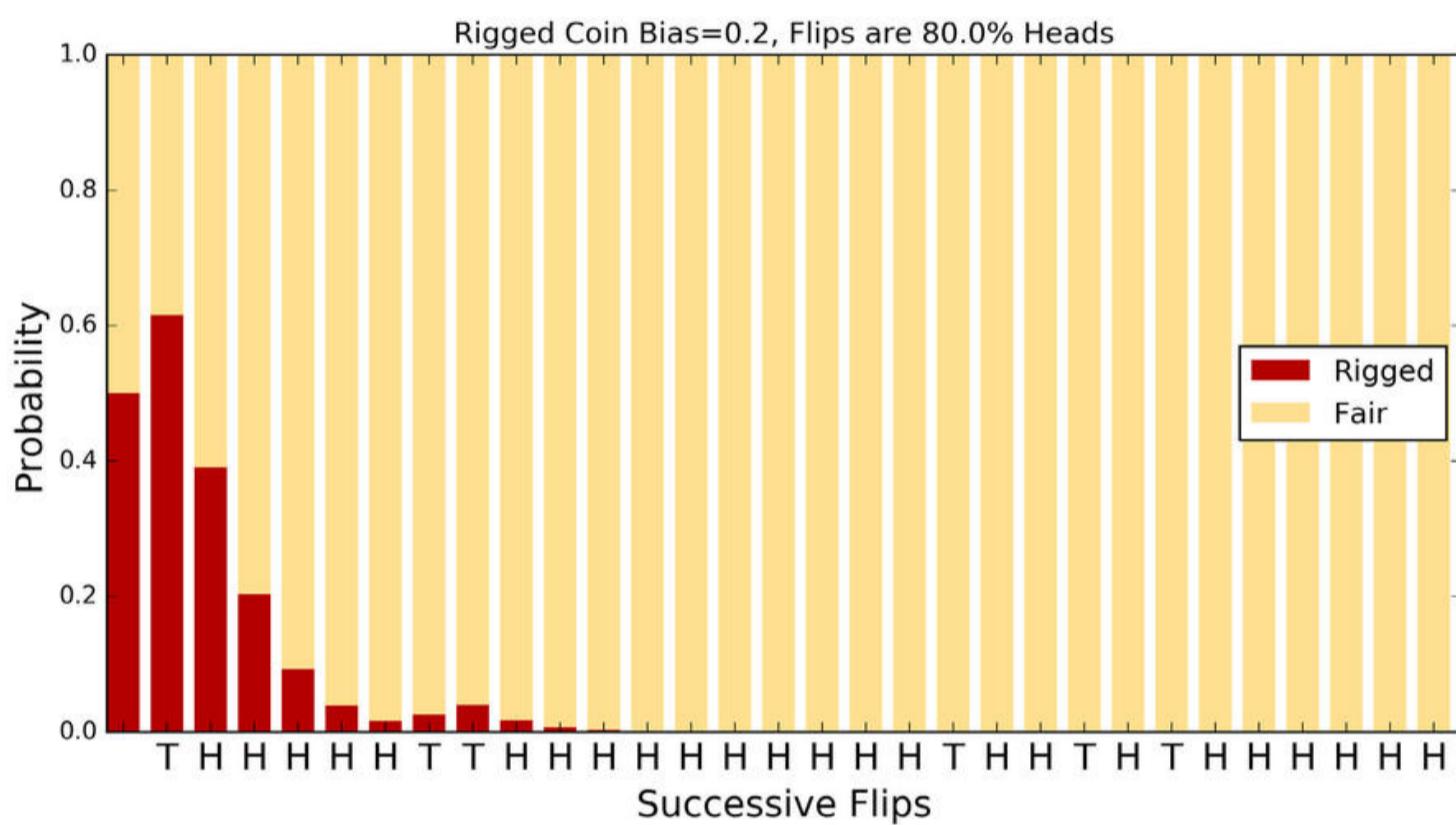


Figure 4.26: Using the same coin with a bias of 0.2 but this time we happened to get 24 heads in our 30 flips.

Even though the fair coin only comes up heads half the time, the rigged coin comes up heads only 20% of the time. So 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 heads-tails patterns for our coin with a bias of 0.2, from a pattern of almost all tails to a pattern of almost all heads. Let's do that again, but we'll try 10 patterns for 10 coins with different biases.

The results are in Figure 4.27. 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 above, 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've invented 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.

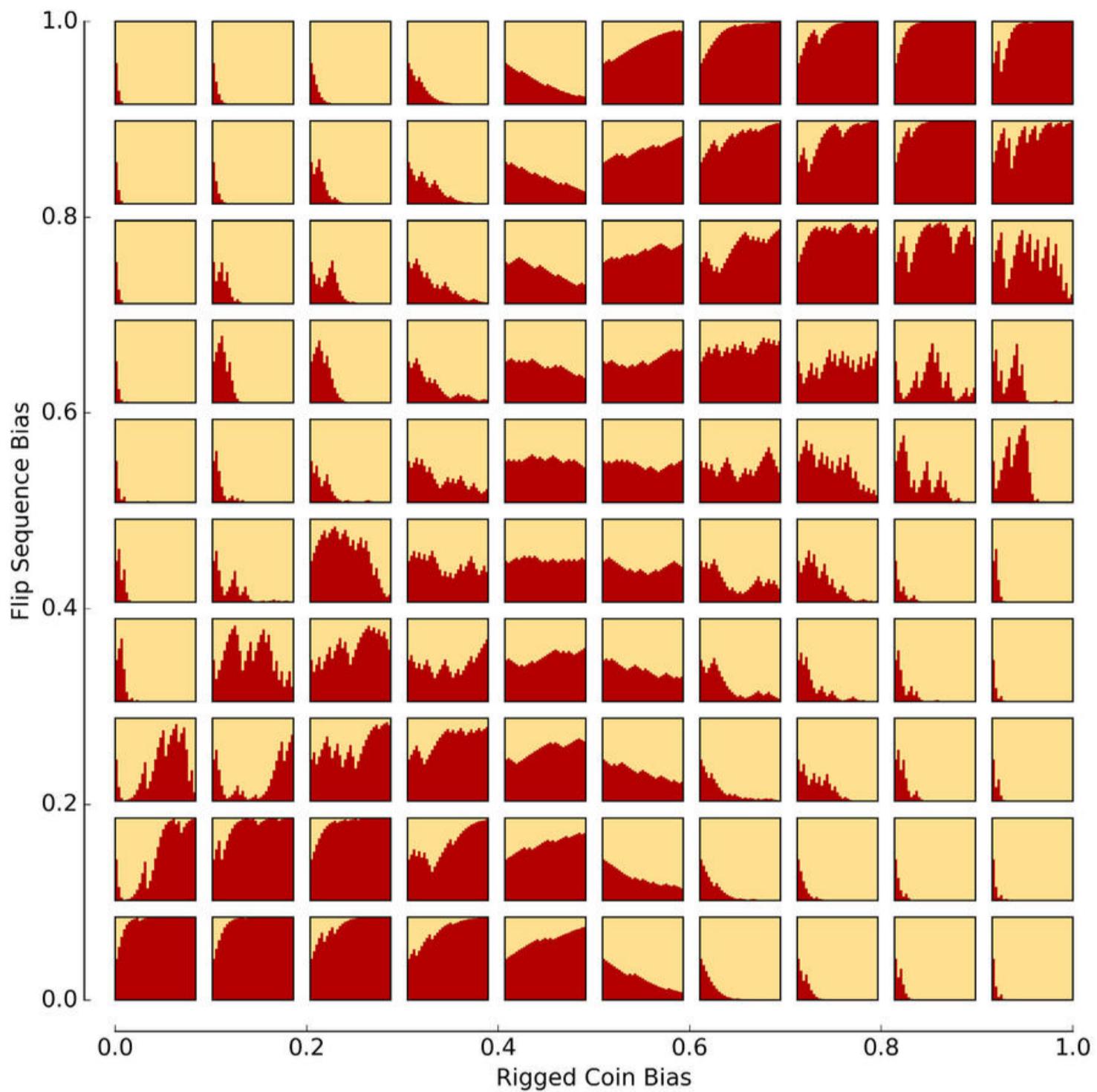


Figure 4.27: Flipping a coin 30 times, and using repeated Bayes' Rule applications to decide which coin we're flipping. Each square reports our results for just one run of 30 random flips. The probability we're using the fair coin is in flax, the probability for the rigged coin is in red. The inherent bias in the rigged coin increases from about 0 to 1 moving left to right. The percentage of heads in our 30 flips increases from about 0% to about 100% from bottom to top.

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 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. So our confidence that the coin is fair grows stronger as the number of flips increases.

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 graph) 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 1000 in each plot, as in Figure 4.28, such unusual sequences become more rare, and the patterns become clearer.

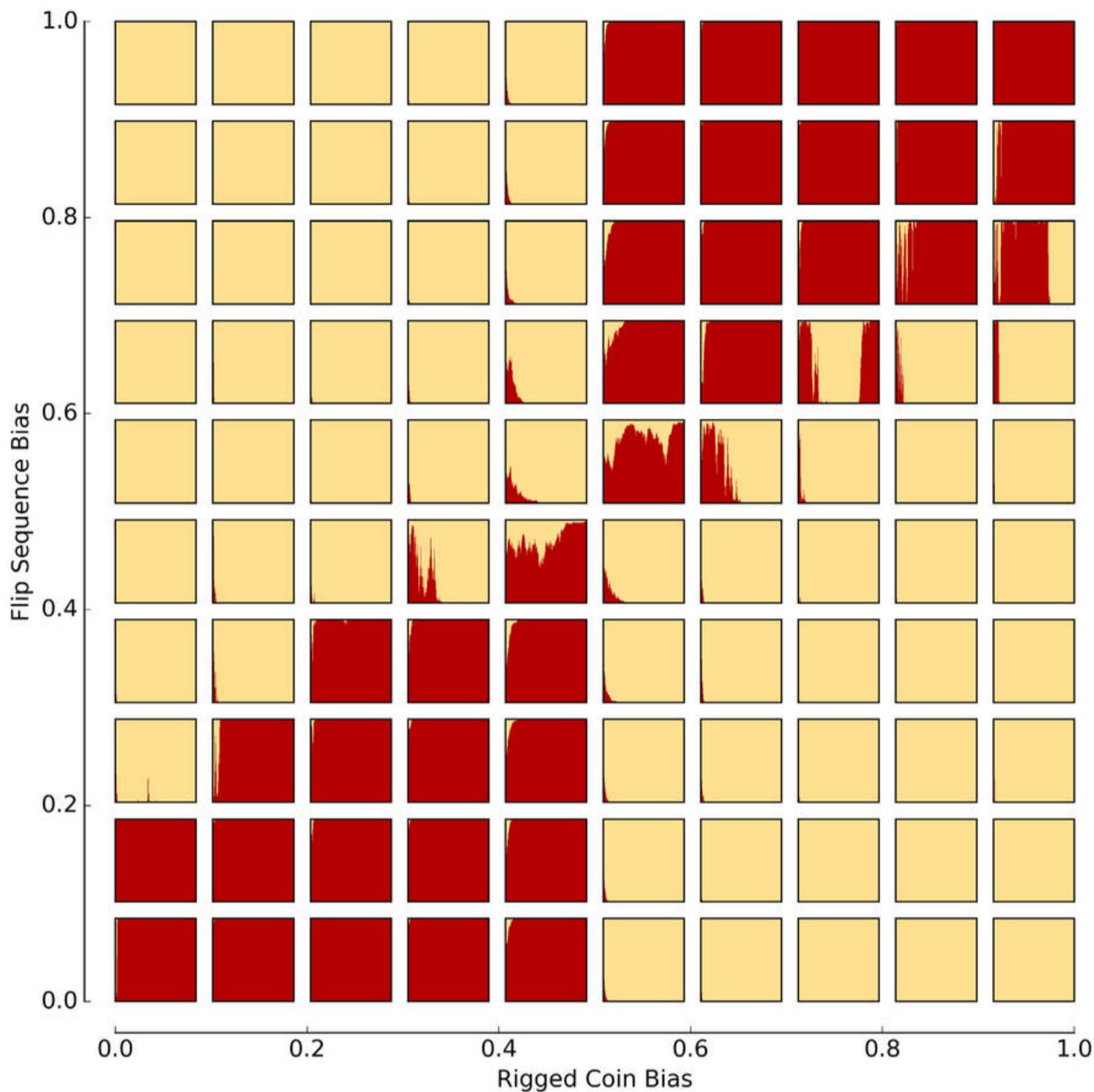


Figure 4.28: The same setup as Figure 4.27, but now we flip each coin 1000 times.

The lesson of Figure 4.27 and Figure 4.28 is that the more observations we make, the more certain we can become of our hypothesis. 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. 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, resulting naturally from Bayes' Rule when our observations are, or are not, consistent with our hypothesis.

4.7 Multiple Hypotheses

We've seen how to use Bayes' Rule to improve a hypothesis by combining it with an observation. But there's nothing limiting us to making 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 of them revealed the other.

But we could explicitly calculate both probabilities using Bayes' Rule if we wanted. So we'd have a prior for the probability of the condition "this is the fair coin," and a prior for the probability of the condition "this is the rigged coin." Each time we get an observation telling us heads or tails, we first calculate the posterior for "this is the fair coin," and make that our new prior for that condition. Then we evaluate the posterior for "this is the rigged coin," and use that as the new prior for that condition. This is shown in Figure 4.29.

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

$$(b) \quad P(R|H) = \frac{P(H|R) \cdot P(R)}{P(H)}$$

$$(c) \quad P(H) = P(H|F) \cdot P(F) + P(H|R) \cdot P(R)$$

Figure 4.29: Calculating probabilities for two hypotheses. (a) The probability that we have the fair coin, given that we observed heads. (b) The probability that we have the rigged coin, given that we observed heads. (c) The term $P(H)$, or the probability of seeing heads from any source, is the sum of the probabilities for seeing heads from each of the two possible coins.

We can see from Figure 4.29 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.

With this approach, we can run multiple hypotheses at once, updating them all with each new piece of information.

We can use this to help out our archaeologist friend again. She's just found another chest filled with game boards and pairs of coins. She knows that one of the two coins in each bag is fair and the other is rigged, but she suspects that this box holds coins that are rigged by different amounts.

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 strongly biased, but as they got more skilled they'd switch to rigged coins whose bias was closer and closer to 0.5.

She's sent us all the coins she's found, rigged and not, and has asked us to find the bias of each coin.

Just for the moment, let's pretend that there are only 5 possible bias values: 0, 0.25, 0.5, 0.75, and 1.

So we'll create five hypotheses. We'll number them 0 through 4, corresponding to the different bias values. So 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 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 which coin we're picking, let's say the chance of having picked each one is the same. So all five prior values are $1/5$ or 0.2.

When we get back our posterior, we're going to use that as our new prior. The best way to keep these values from becoming extremely large or small is to make sure that they always form a probability mass function or pmf, as discussed in Chapter 3. For our purposes, it just means that all of our priors have to add up to 1. Thanks to how Bayes' Rule works, if our priors are all pmfs then our posteriors will also come out as pmfs, so once we get this started properly that condition will sustain itself. Since our 5 priors each have a value of $1/5$, and $5 \times 1/5 = 1$, we're already set.

The only thing left is to pick 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 $1 - 0.2 = 0.8$.

So 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%), and a likelihood of tails of 0.8 (or 80%). Our likelihoods are plotted in Figure 4.30. Note the likelihoods are not pmfs, so their values don’t have to add up to 1.

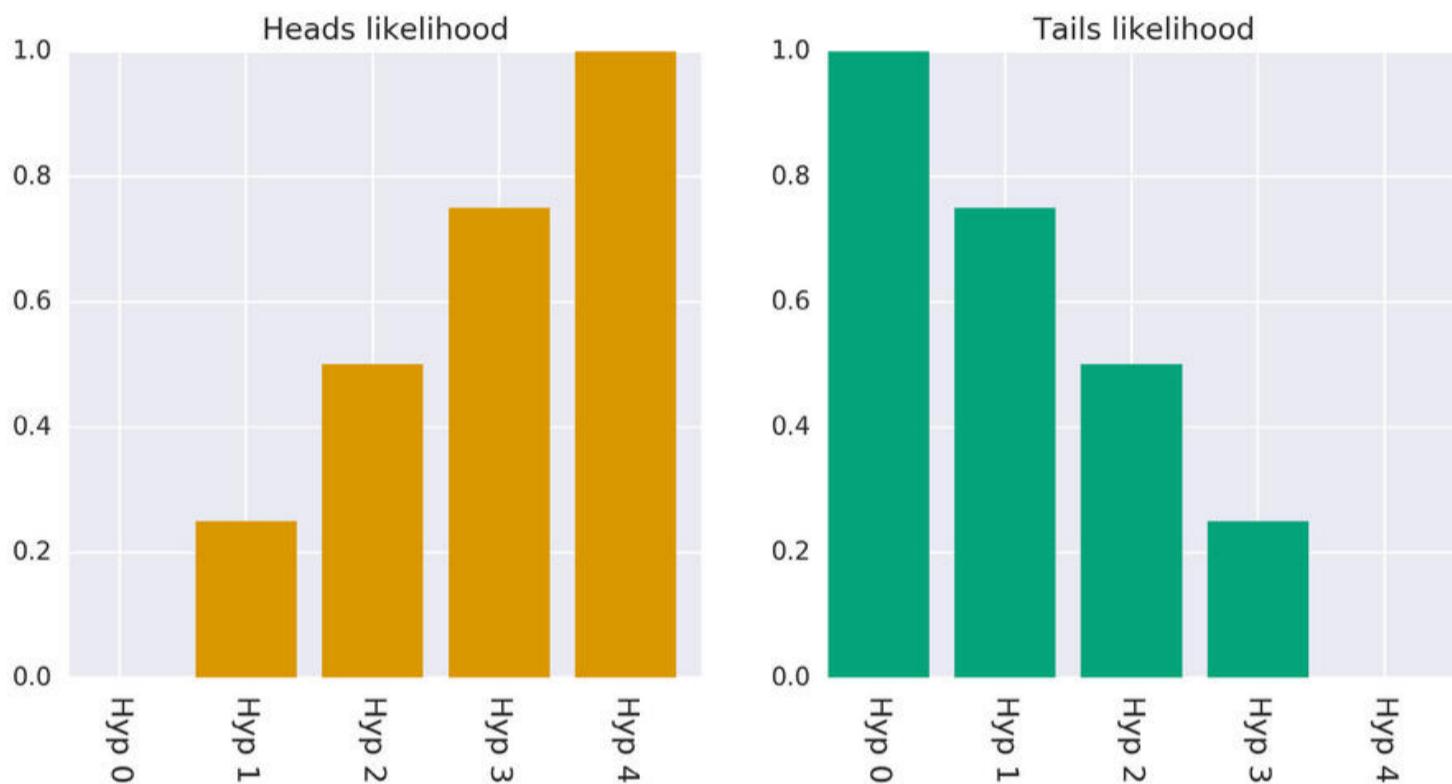


Figure 4.30: The likelihoods for getting heads or tails for each of our 5 hypotheses. The chance of getting back heads runs from 0 to 1, and the chance of getting tails correspondingly runs from 1 to 0.

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

Our goal will be to flip our coin over and over and watch what happens to our priors as they evolve. To show what’s happening at each flip, we’ll draw the 5 prior values in red, and the 5 posterior values in blue.

In Figure 4.31 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 4.30, which gives us our likelihoods. After dividing by the sum of all 5 probabilities for getting heads, we get the posterior, or the output of Bayes' Rule, shown in blue.

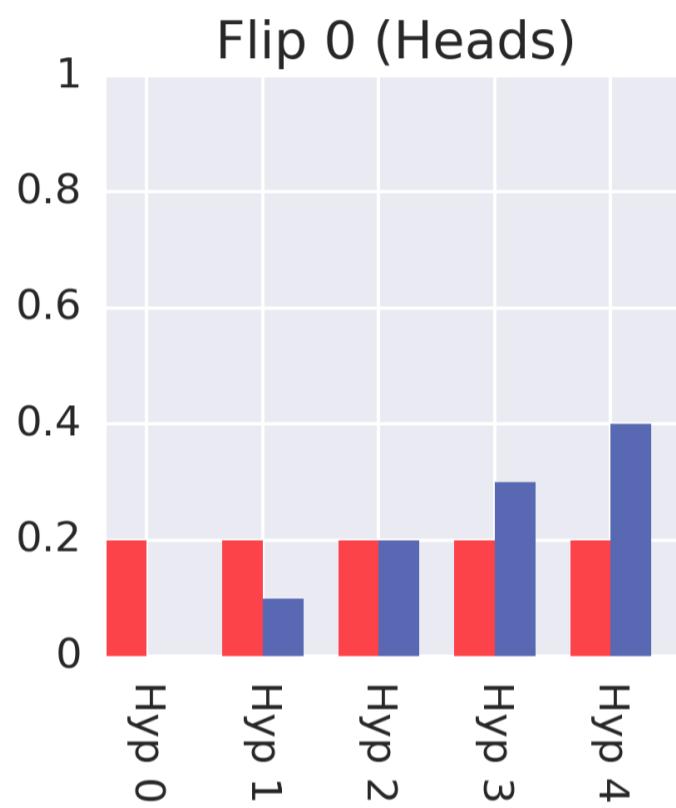


Figure 4.31: We're testing five hypotheses, which assert that our coin has a bias of 0, 0.25, 0.5, 0.75, and 1.0. We start with a prior of 0.2 (in red) for each hypothesis. After one flip of the coin, which came up heads, we compute the posterior (in blue).

In Figure 4.31, 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.

Let's now make a bunch of flips. We'll generate a series of heads and tails that contain 30% heads. That is, the flips correspond to a coin that has 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 4.32 shows the results for the first 10 flips in the top two rows, and then takes bigger jumps in the bottom row.

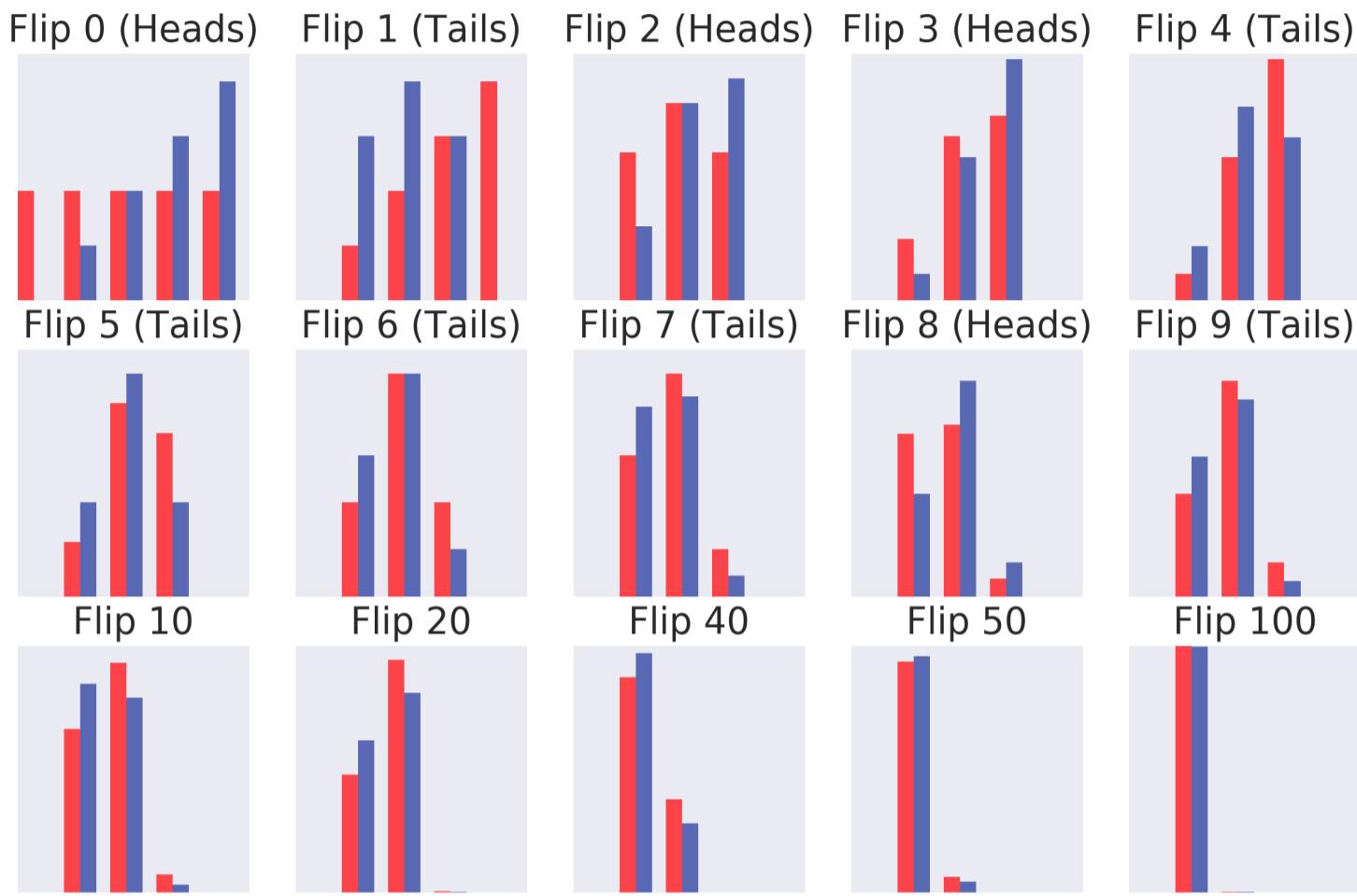


Figure 4.32: The evolution of our priors (red) and posteriors (blue) in response to a series of flips generated by a coin with bias 0.3.

As we can see, after each flip the old posterior (in blue) becomes the new prior (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%, and Hypothesis 1 dominates.

When we've reached 100 flips, the system has pretty much decided that Hypothesis 1 is the right one, 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 4.33 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.

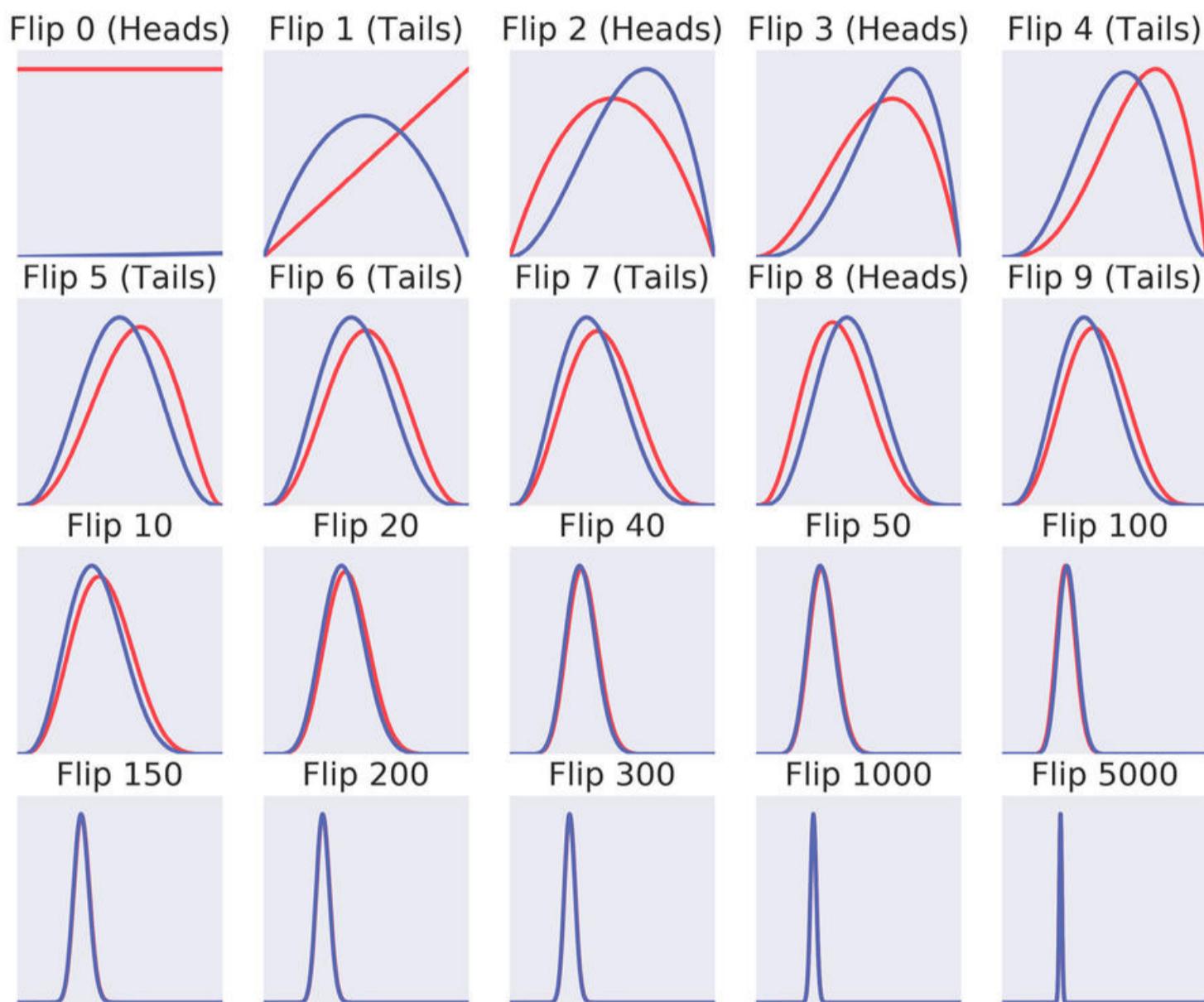


Figure 4.33: The same situation as Figure 4.32, but now we're evaluating 500 simultaneous hypotheses, each based on a coin with a slightly different bias. The flips are the same sequence as in Figure 4.32.

In this figure (as in the figures to come), we are re-using the identical series of flips that we used in Figure 4.32.

As we'd expect, the winning hypothesis is the one predicting a bias of 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 is flat except for a symmetrical bump.

What would be the result if we *started* with priors that formed a symmetrical bump? Let's make it even harder for the system by putting the height of the bump out at around 0.8. That says we expect that the coin we're testing is most likely to have a bias of 0.8, and the farther we are from 0.8, the less likely that coin is. The values away from the main bump get very small, but they never go quite to zero. The probability at 0.3 (which describes our coin) starts at about 0.004, so we're asserting, through our prior, that the chance of this coin having a bias of 0.3 is 0.4%, or 4 out of 1000. How will the system respond?

Figure 4.34 shows the result.

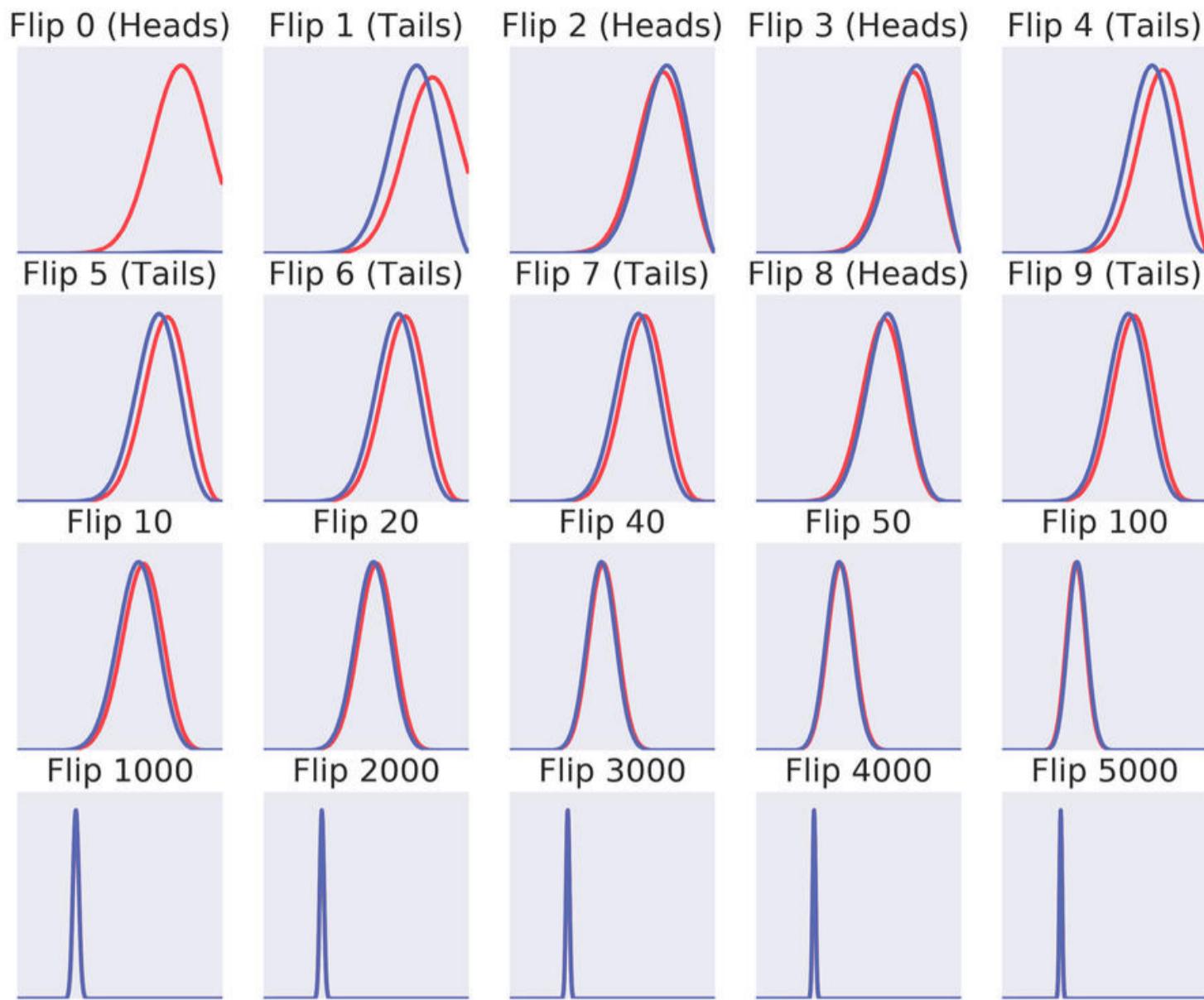


Figure 4.34: The same setup as Figure 4.33, only now we're starting with set of priors formed in a Gaussian bump centered at 0.8.

Nice. Even with our misleading priors, the system honed in on the proper bias of 0.3. It took a while, but it got there. Figure 4.35 shows a set of curves from this process overlaid on one another.

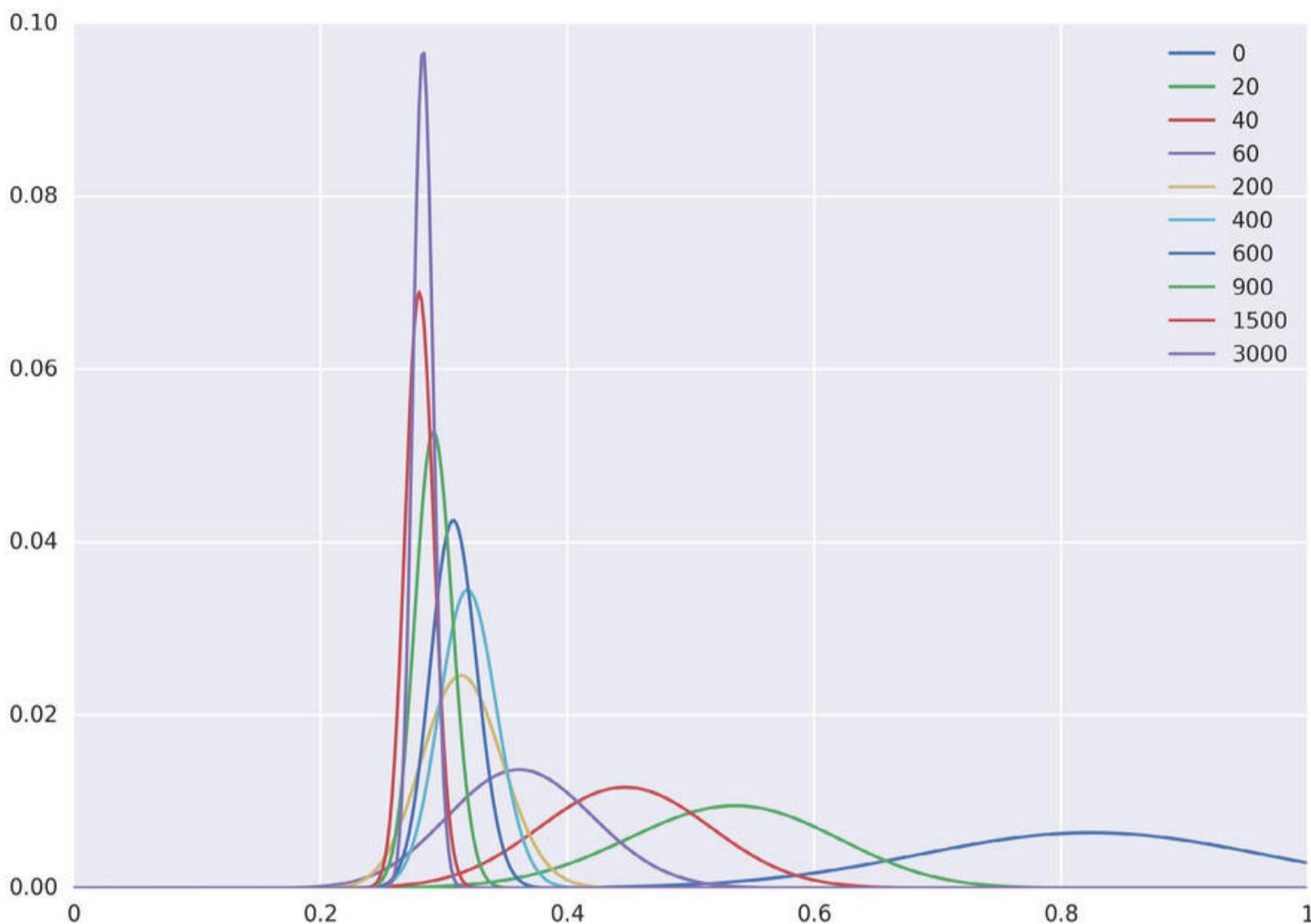


Figure 4.35: Some snapshots of the posteriors from the first 3000 flips from Figure 4.34, overlaid on one another. We can see the system giving more and more weight to the priors near 0.3, while reducing the probabilities elsewhere.

We won't get into the details, but with some math we can carry our increasing density to its logical extreme, replacing our lists of values with continuous curves, like those suggested by Figure 4.35. 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 our list.

References

- [Cthaeh16a] The Cthaeh, “The Anatomy of Bayes’ Theorem”, Probabilistic World, 2016. <http://www.probabilisticworld.com/anatomy-bayes-theorem/>

- [Cthaeh16b] The Cthaeh, “Calculating Coin Bias With Bayes’ Theorem”, Probabilistic World, 2016. <http://www.probabilisticworld.com/calculating-coin-bias-bayes-theorem/>
- [Genovese04] Christopher R. Genovese, “Tutorial on Bayesian Analysis (in Neuroimaging)”, Institute for Pure & Applied Mathematics Conference, 2004. <http://www.stat.cmu.edu/~genovese/talks/ipam-04.pdf>
- [Kruschke14] John K. Kruschke, “Doing Bayesian Data Analysis, Second Edition: A Tutorial with R, JAGS, and Stan 2nd Edition”, Academic Press, 2014.
- [Stark16] P. B. Stark and D. A. Freedman, “What Is the Chance of an Earthquake?”, UC Berkeley Department of Statistics, Technical Report 611, 2016. <https://www.stat.berkeley.edu/~stark/Preprints/611.pdf>
- [VanderPlas14] Jake Vanderplas, “Frequentism and Bayesianism: A Python-driven Primer”, 2014. <https://arxiv.org/abs/1411.5018>

Chapter 5

Curves and Surfaces

How to describe the properties of the curves and surfaces that we'll be using to describe and control many of our algorithms.

Contents

5.1 Why This Chapter Is Here.....	207
5.2 Introduction	207
5.3 The Derivative.....	210
5.4 The Gradient	222
References	229

5.1 Why This Chapter Is Here

This chapter describes important properties of curves and surfaces that get used all the time in machine learning.

Two of the most important of these ideas are called the **derivative** and the **gradient**. They tell us about the shape of the curve or surface, letting us know in which directions to move to climb uphill or slide downhill.

These ideas are at the heart of how deep-learning systems learn. That process is carried out by an algorithm called **back propagation**, which we will discuss in detail in Chapter 18. Knowing about the derivative and gradient is key to understanding backpropagation, and thus knowing how to build and train successful networks.

In this chapter we will as usual skip the equations, and instead focus on getting an intuition for what these two terms describe. Mathematical depth on everything we touch on here can be found in most books on modern calculus [Apostol91] [Berkey92].

5.2 Introduction

In machine learning we often deal with various kinds of curves. Most often, these are plots of **mathematical functions**.

We usually think of these in terms of an **input** and an **output**. When we're dealing with a curve, 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. So we provide one number as input, and get back one number as output.

When 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 case, 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 take in multiple values, also called **arguments**, and can provide multiple values as outputs, sometimes called **returns**. At their heart, we can think of a function as a look-up table of potentially infinite size. One or more numbers go in, and one or more numbers come out. As long as we don't deliberately introduce randomness, every time we give a particular function the same inputs, we'll 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, will be to determine how to move along them, starting at a particular point, to get back larger or smaller outputs. The mathematics that let us carry out that process require that our functions satisfy a few conditions. We'll illustrate those conditions below in terms of 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 points (called **cusps**). Figure 5.1 shows a curve that has both of these forbidden features.

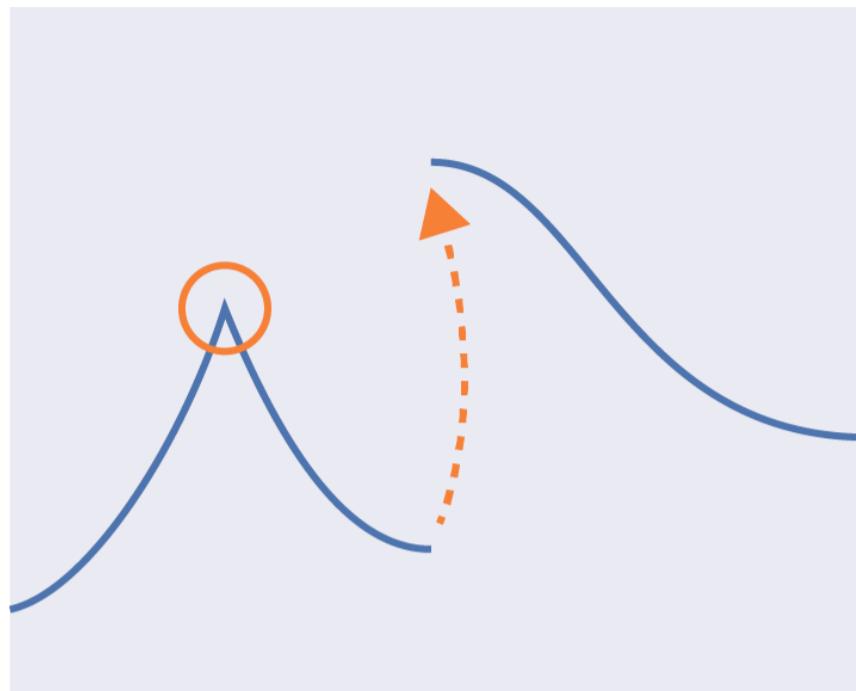


Figure 5.1: The circle encloses a cusp, or point where the curve suddenly changes direction. The dashed arrow shows a discontinuity, or jump, where we would need to lift our pen from the page to draw the curve.

We also want our curves to be **single-valued**. This means that for each horizontal position on the page, if we draw a vertical line at that point, the line will cross the curve only once, so there is only a single value that 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.

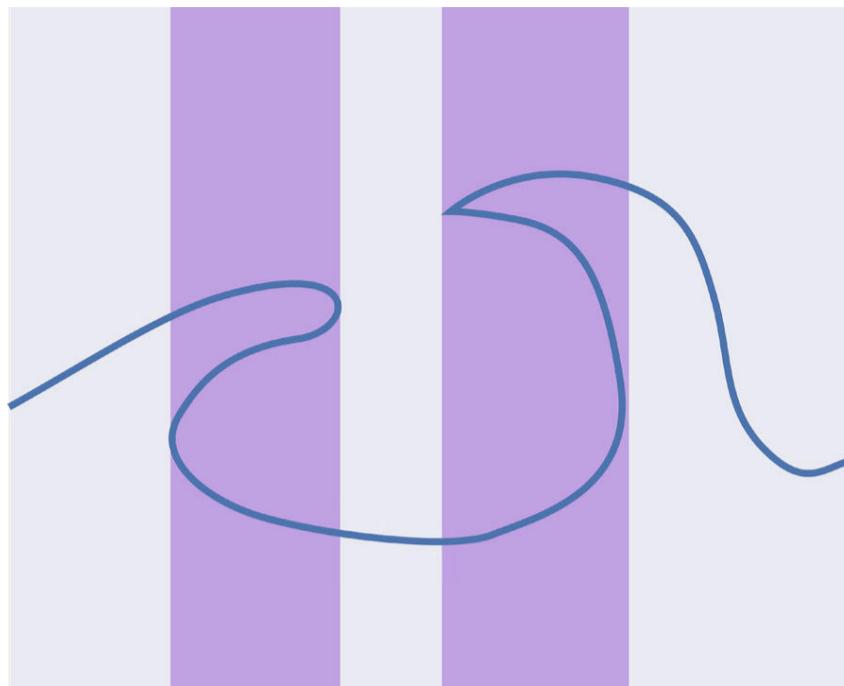


Figure 5.2: Inside the purple zones the curve has multiple values in the vertical direction.

From now on, we'll 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.

5.3 The Derivative

In machine learning we're going to frequently want to find the **minimum** or **maximum** value of a curve.

Sometimes we want to find the smallest or largest value of the curve *anywhere* along its entire length, as illustrated in Figure 5.3. Thinking of the whole curve as the “world” of choices, we call these points the **global minimum** and **global maximum**.



Figure 5.3: The **global maximum** (red circle), and the **global minimum** (orange square) are those points with the largest and smallest values along the entire curve.

Sometimes we just want the largest and smallest values of the curve, but other times we want to know *where* on the curve those points are located.

Sometimes it's hard to find those values. For example, if the curve goes on forever in both directions, how can we be sure we have the very smallest or largest values? Or if the curve repeats, as in Figure 5.4, which of the high (or low) points should we pick as the location of *the* maximum or minimum?

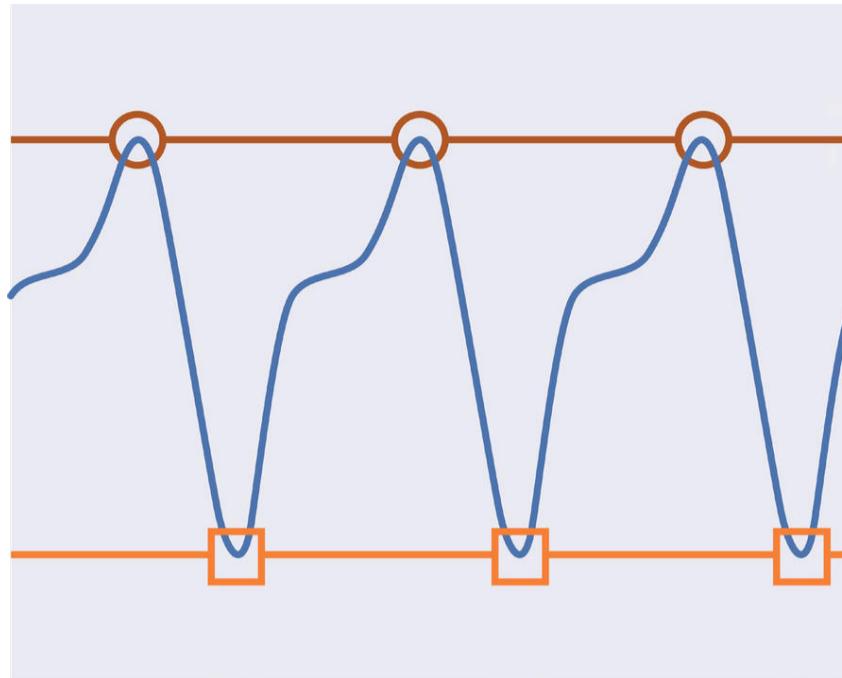


Figure 5.4: When a curve repeats forever, we can have infinitely many points that we could use as the location of “the” global maximum (red circles) or minimum (orange squares).

To get around these problems, we can imagine a little thought experiment. Starting from some point, we travel to the left until the curve changes direction. If the values start increasing as we move to left, we'll continue as long as they increase, but as soon as they start to decrease we'll stop. We'll follow the same logic if the values are decreasing as we move to the left, stopping when they start to increase. We'll do the same thought experiment again, starting at the same point, but moving to the right. This gives us three interesting points: our starting point, and the two points where we stopped 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.

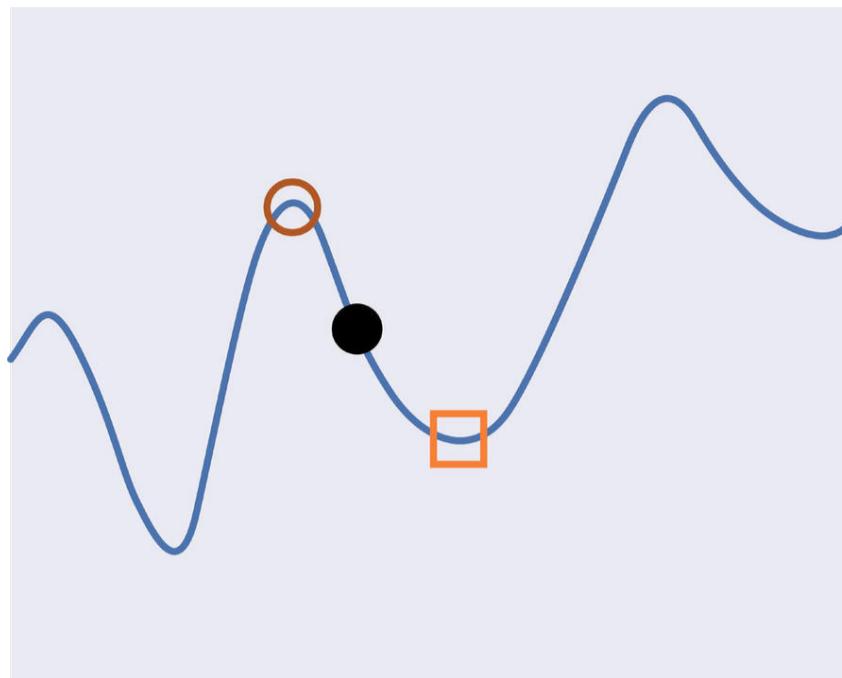


Figure 5.5: The **local maximum** and **local minimum** give us the smallest and largest values in a neighborhood around a point. For the point in black, the red circle and orange box show that point's local maximum and minimum. Note that these are not the global maximum and minimum.

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 then marked with a square. We consider the trio of points consisting of the center of the circle, our starting point, and the center of the square. The local maximum is given by the largest value for these three points, which in this case is the center of the circle. The local minimum is given by the smallest value of these three points, in this case the center of the square.

If the curve zooms off to positive or negative infinity, things get more complicated. We'll never be finding the minima or maxima of such curves in this book, so we will 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, but there can be many local maxima and minima, since they can vary for every point we consider. Figure 5.6 shows this idea visually.

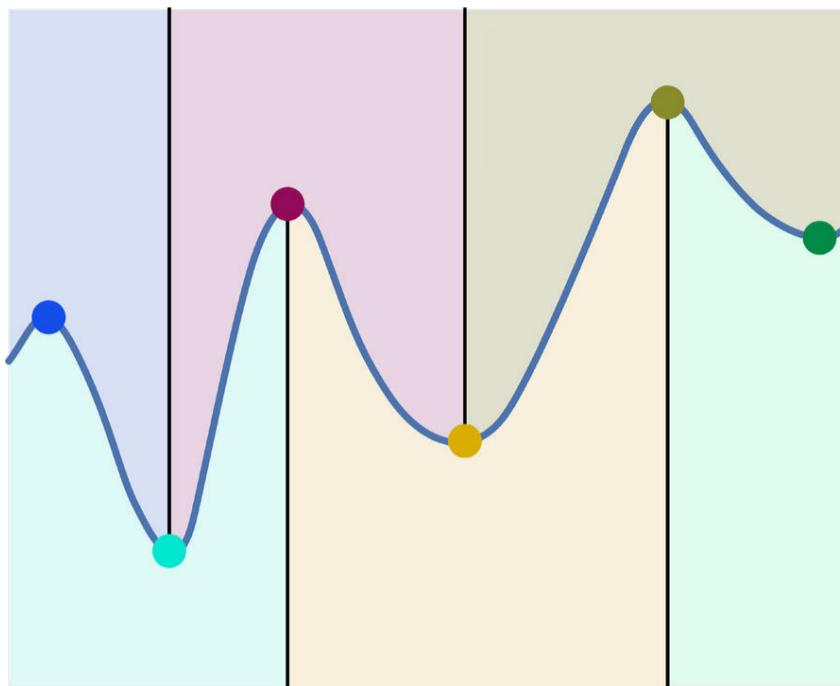


Figure 5.6: The influences of these local minima and maxima are shown by their corresponding colored region. For example, the second maximum from the left, in dark red, is the local maximum for all points in the red region. The dark yellow point is the local minimum for all points in the yellow zone.

We can find local minimum and maximum values by eye, but the computer needs an algorithm. We'll get there by creating a little helper object called a **tangent line**.

To illustrate the idea, we've marked up a two-dimensional curve in Figure 5.7.

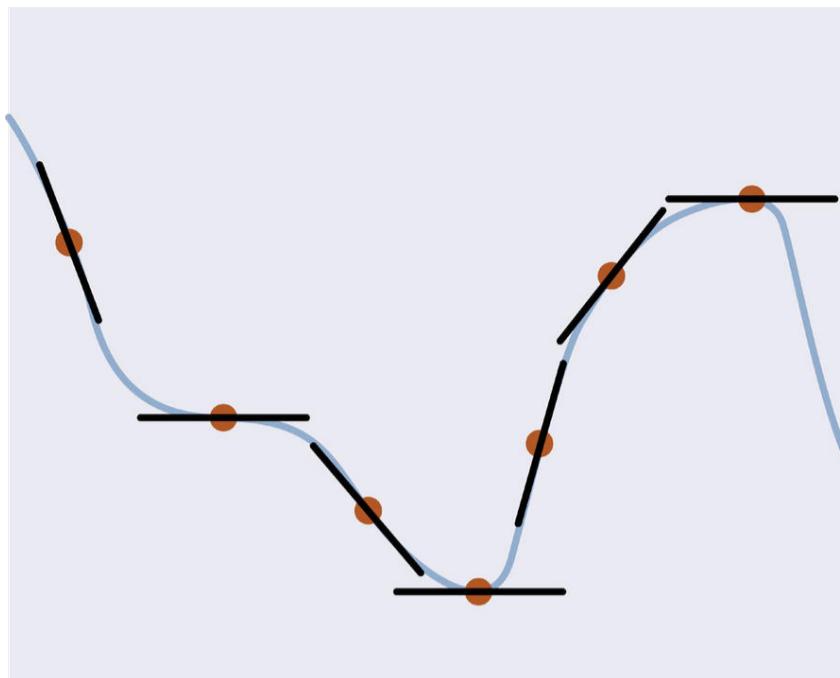


Figure 5.7: Some points on this curve are marked with dots. The tangent line at each of those points is drawn in black, showing the slope of the curve at that location.

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.

Here's one way to find that 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.

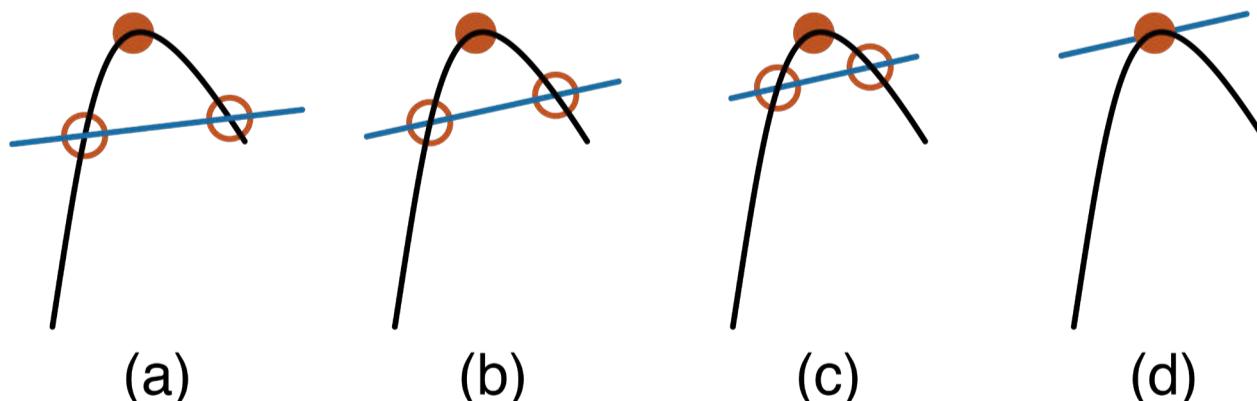


Figure 5.8: To find the tangent line at a given point, we can look at a pair of points at equal distances along the line around that point, and draw a line between them. As we pull those two points closer to the target point, we find the tangent line.

Now we start pulling the two dots in towards the target point at the same speed, each staying on the curve. At the very last instant before they merge, the line that passes through them is the one we want.

We say that this line is **tangent** to the curve, meaning that it just touches it. The tangent line at a given point gives us a sense of how the curve is turning at that point.

We can measure the **slope** of the tangent line we constructed in Figure 5.8. The slope just a single number that's 0 when the line is horizontal, and becomes larger and larger as the line tilts towards vertical. Mathematicians call this number the **derivative**. There's a potentially different value for the derivative at every point on a curve, because every point can potentially have a different 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, or the derivative that describes its slope, at every point on the curve.

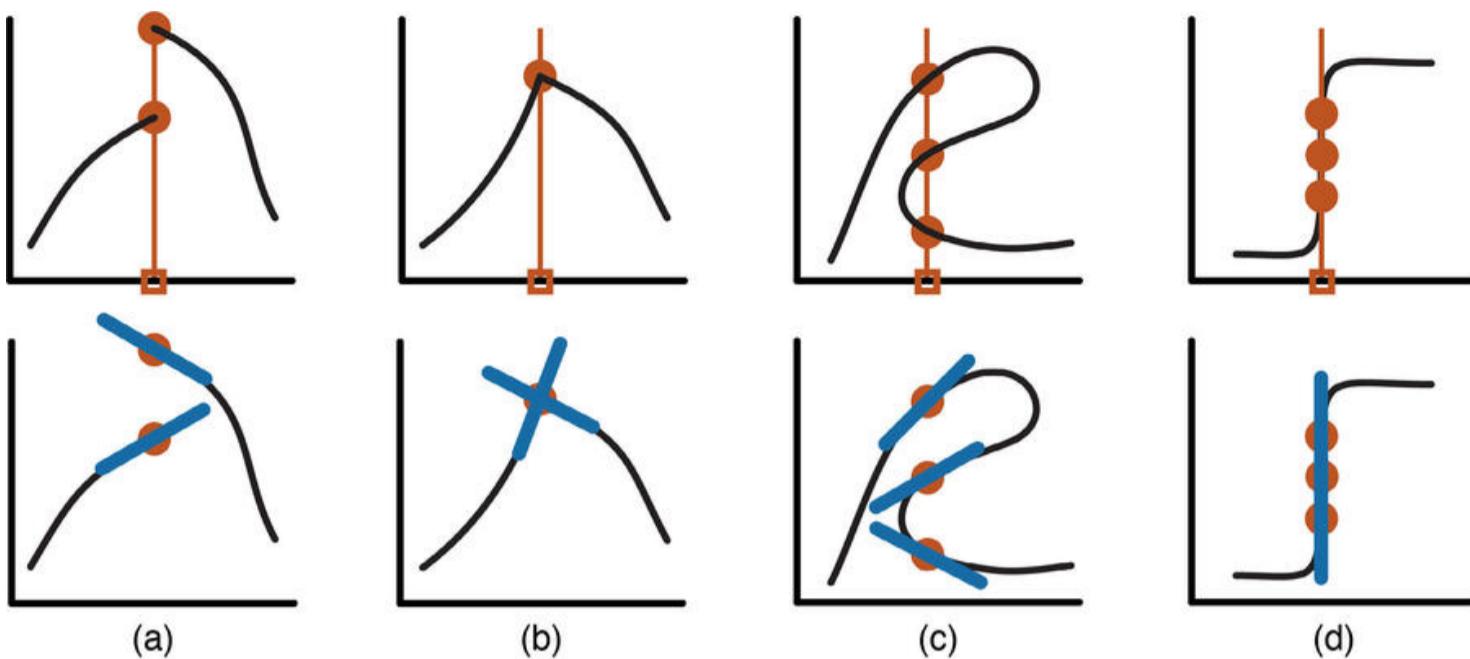


Figure 5.9: Why we prefer continuous, smooth, single-valued curves. Here we're picking a point on the horizontal axis, shown by the small square, and we want to find the derivative of the curve above that point. Top row: Curves with issues. Bottom row: Problems finding the derivative.

In Figure 5.9(a), the curve isn't continuous, so we can have different derivatives above our chosen point. We don't know which derivative to pick. In Figure 5.9(b), the curve isn't smooth, so the slope can be very different as we arrive at the cusp from the left and right. Again, we don't know which one to pick. Using our gradual approximation of Figure 5.8 doesn't help, because that gives us a derivative that doesn't come close to what the curve is actually doing. 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 would also violate our single-value rule. Worse, the tangent line is perfectly vertical, which means it has an infinite slope, or an infinite derivative. Handling infinite values can make simple algorithms messy and complicated. So we just sidestep the problem by saying curves can never become vertical, and thus we will never need to worry about infinite derivatives.

Let's think of a curve as a way to turn a value of X into a value of Y. The convention that we'll use is that X becomes more positive running to the right, and Y becomes more positive going upwards, as in Figure 5.10.

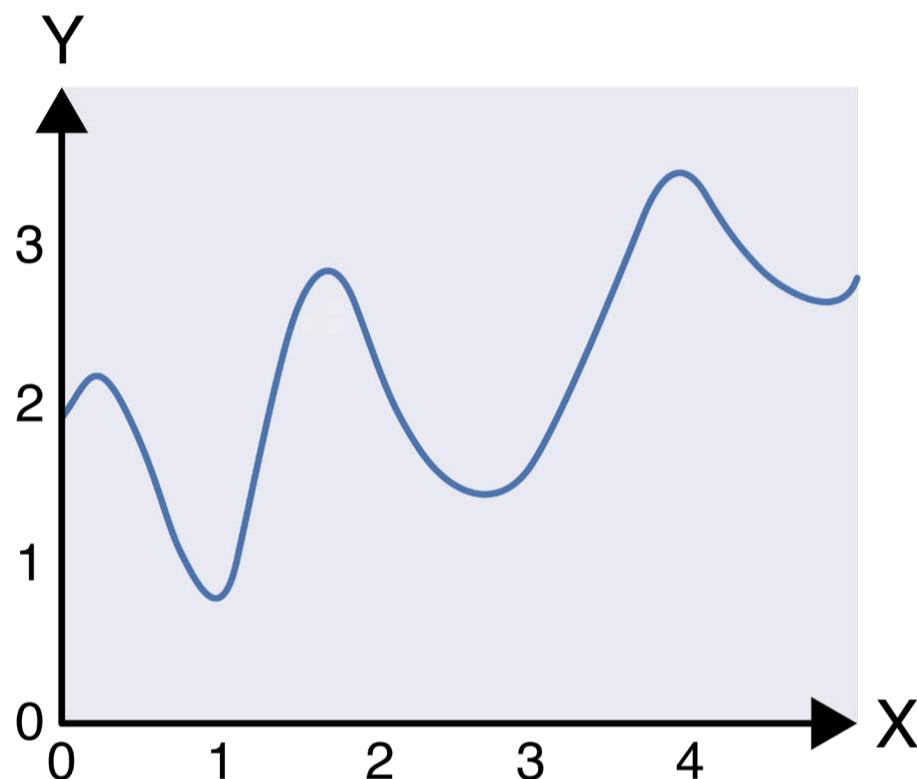


Figure 5.10: A curve in two dimensions. Values of X increase as we move right, and values of Y increase as we move up.

As we move to the right from some point (that is, increasing X), we can ask if the curve is giving us values of Y that are also increasing, or 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. Figure 5.11 shows the idea.

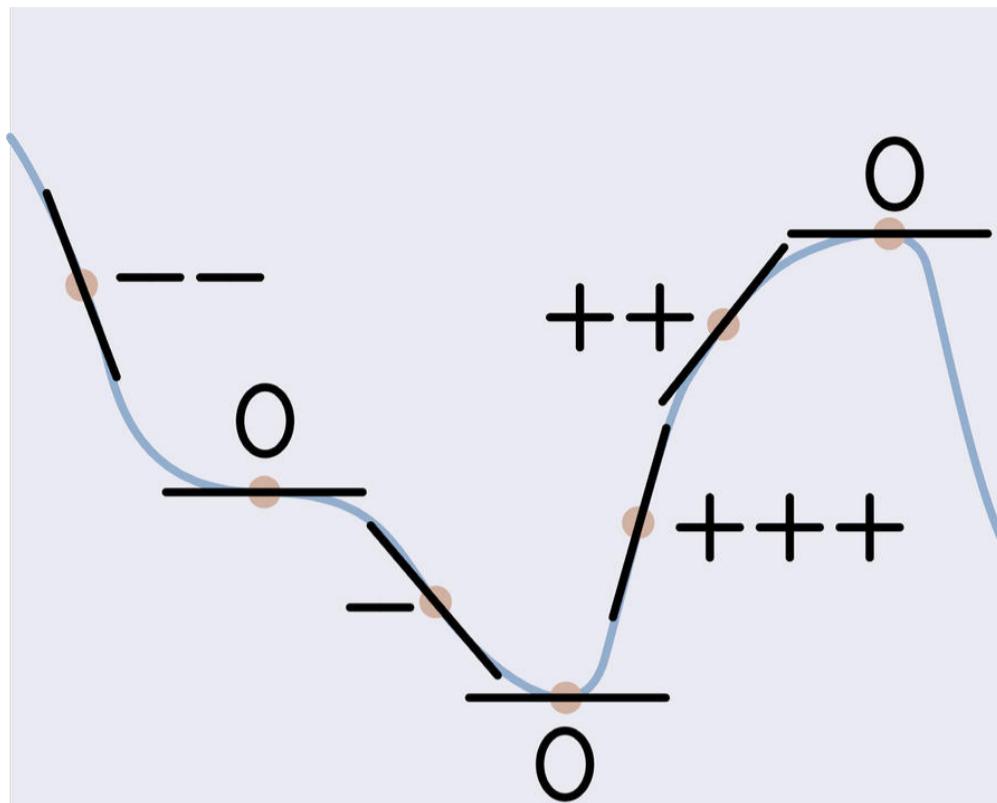


Figure 5.11: Marking the tangent lines from Figure 5.7 by whether they have a positive slope (+), negative slope (-), or are flat (0). To indicate the range of slopes shown, the more positive or negative each slope, the more + or - signs we've drawn.

Notice that there's a point near the left side that isn't a hill or a valley, but still has a slope of 0. We only find slopes of 0 at the tops of hills, the bottoms of valleys, and plateaus like this one.

We say that the **sign** of a number tells us if it's positive, negative, or zero. The sign of any positive number is 1, the sign of any negative number is -1 , and the sign of 0 is 0.

It turns out that it's usually not too hard to calculate the value for a derivative on a curve. We can use the derivative as an engine to drive an algorithm that finds a local minimum or maximum at a point.

Given a point on a curve, we'll find its derivative. If we want to move along the curve so that the Y values increase, we'll move in the direction of the sign of the derivative. That is, if the derivative is positive, then moving in the positive direction along X will take us to larger values. In the same way, to find smaller values of Y we'll move in the opposite direction. Figure 5.12 shows the idea.

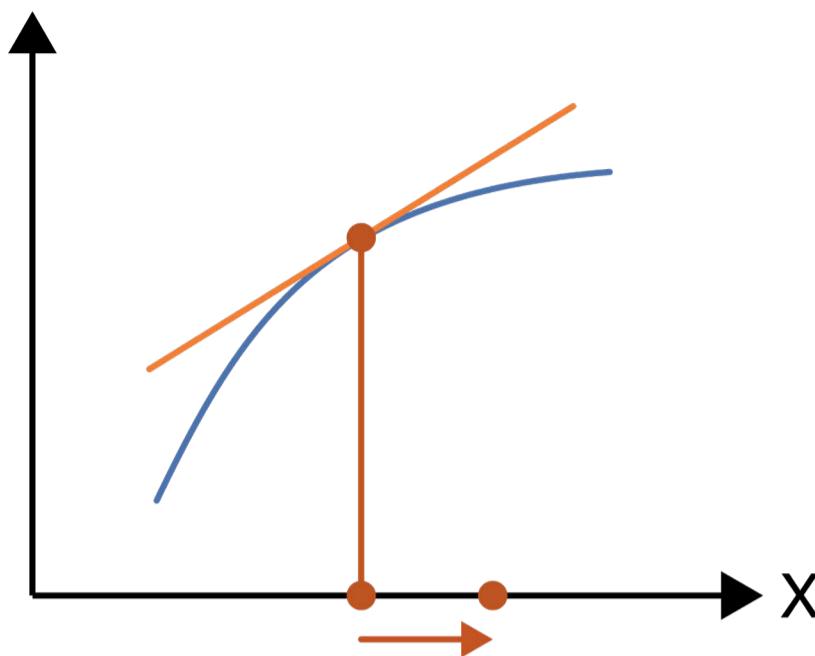


Figure 5.12: The derivative at a point tells us which way to move to find larger or smaller values of the curve. At the marked point, the derivative is positive, so if we take a step to the right (along positive X), we'll find a larger value of the curve.

To find the local maximum near some point, we'll find the derivative at that point and take a small step along X in the direction of the sign of the derivative. Then we'll find the derivative there, and take another small step. We'll repeat this process over and over until the derivative hits zero. Figure 5.13 shows this in action.

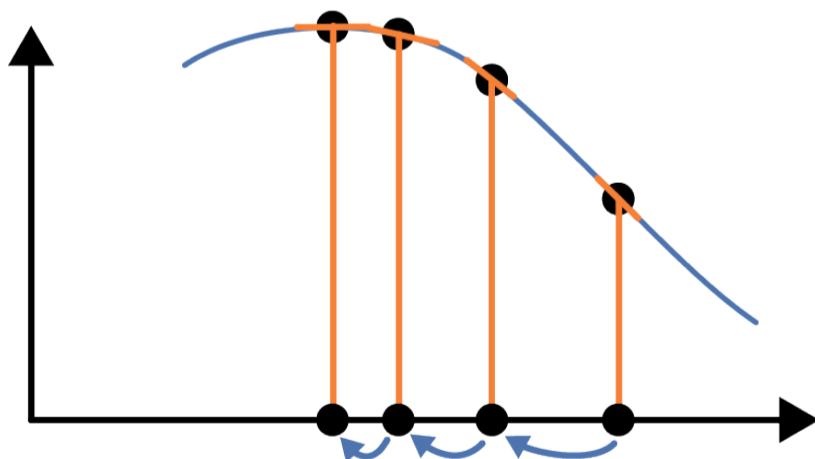


Figure 5.13: We can use the derivative to find the local maximum at a point. Here we start at the rightmost point. The derivative is negative, and somewhat large, so we take a big step to the left. The derivative is again negative but a bit smaller, so we'll take a smaller step to the left. A third, smaller yet step takes us to the local maximum, where the derivative is zero.

To make this 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 move along X in direction opposite to the derivative's sign, as in Figure 5.14.

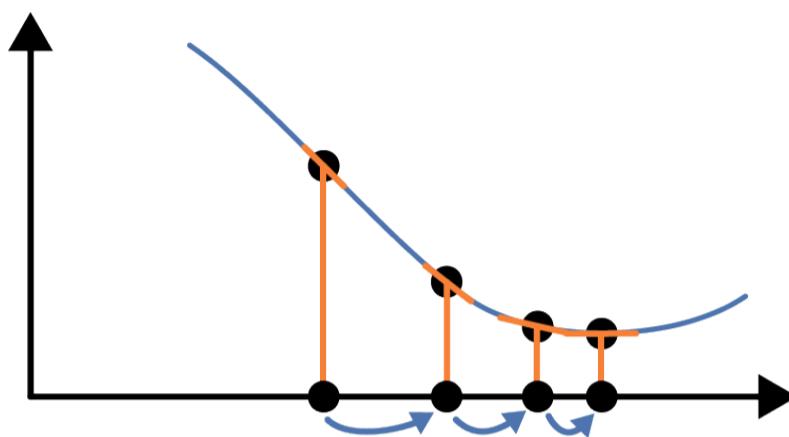


Figure 5.14: To find a local minimum we follow the same procedure as in Figure 5.13, but because the derivative is negative, we move to the right. Here we start at the far left point. The derivative is negative and large, so we move to the right by a big step. We'll find the derivative increases (that is, becomes closer to 0) with each step until it lands us at the local minimum, where the derivative is zero.

Finding local minima and maxima is a core numerical technique that is used throughout machine learning, and it depends 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 minima and maxima.

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 often (though not always) let us finesse the problem and carry on.

5.4 The Gradient

The **gradient** is the generalization of the derivative into three dimensions, or four dimensions, or *any* number of dimensions beyond that. Let's see how this works.

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.

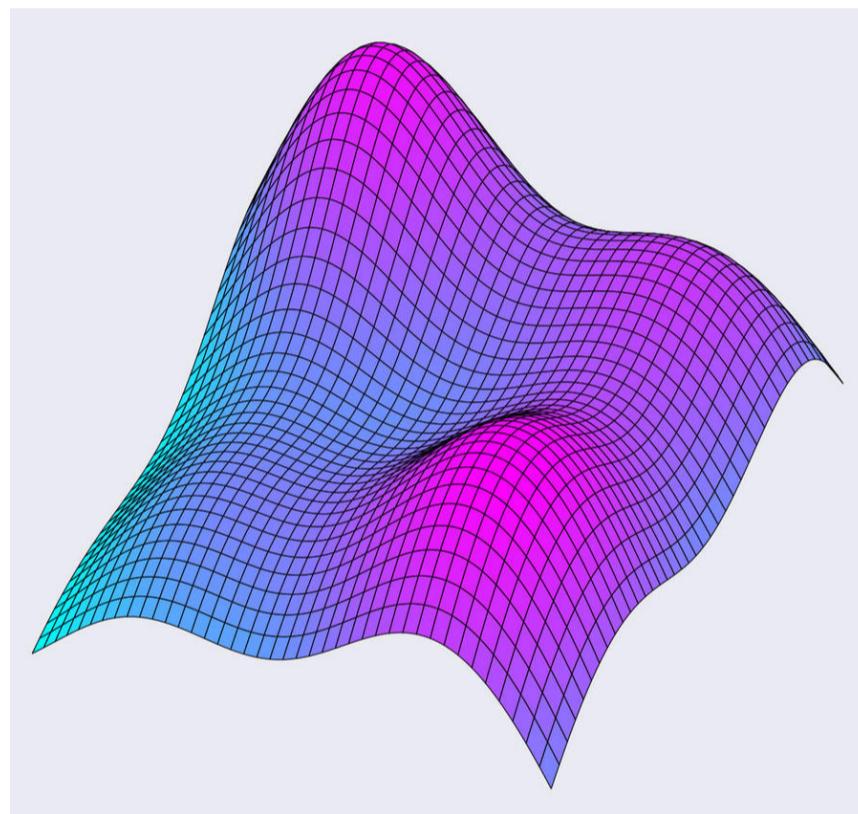


Figure 5.15: A sheet of smooth fabric without creases or tears.

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 that height.

Now let's imagine that we can freeze the fabric at a particular moment. If we move up to the fabric and walk around it, or even on it, it will feel like a *landscape* of mountains, plateaus, and valleys.

Suppose that the fabric is dense enough that water doesn't pass through it. At any given point, we can pour some water onto the fabric at our feet. The water, naturally, will flow downhill.

In fact, the water will follow the path that takes it downhill *in the fastest possible way*.

The water we pour out is being pulled downward by gravity. At every point, it effectively searches the local neighborhood and finds all the points that are downhill from where it is now, and it moves towards the point that takes it downhill the fastest, as shown in Figure 5.16.

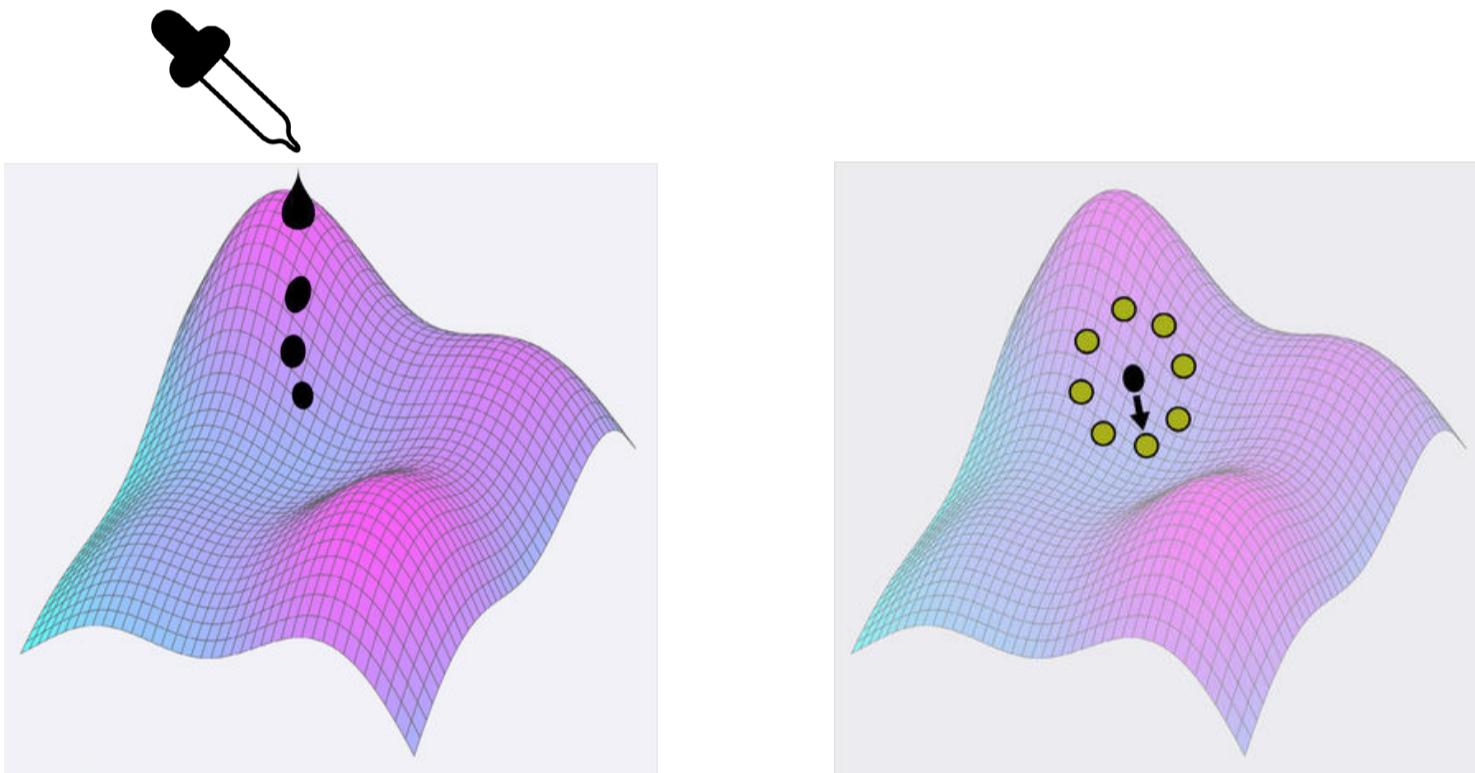


Figure 5.16: If we drip some water onto our fabric, it will flow downhill as quickly as possible. We say it follows the path of maximum descent. Left: Dripping water onto the surface. Right: A drop of water exploring multiple points in its local neighborhood to find the one that is the most downhill. That's the direction the water will move towards.

Out of all the ways to move, the water will always follow the steepest route downhill. The direction followed by the water is called the direction of **maximum descent**.

Suppose that we instead wanted to climb upwards as quickly as possible, and find the direction of **maximum ascent**? In the very local neighborhood of a point on a surface, the direction of maximum ascent

is in exactly the opposite direction as that of maximum descent. These two paths are always on the same line, pointing in opposite directions. Figure 5.17 shows the idea.

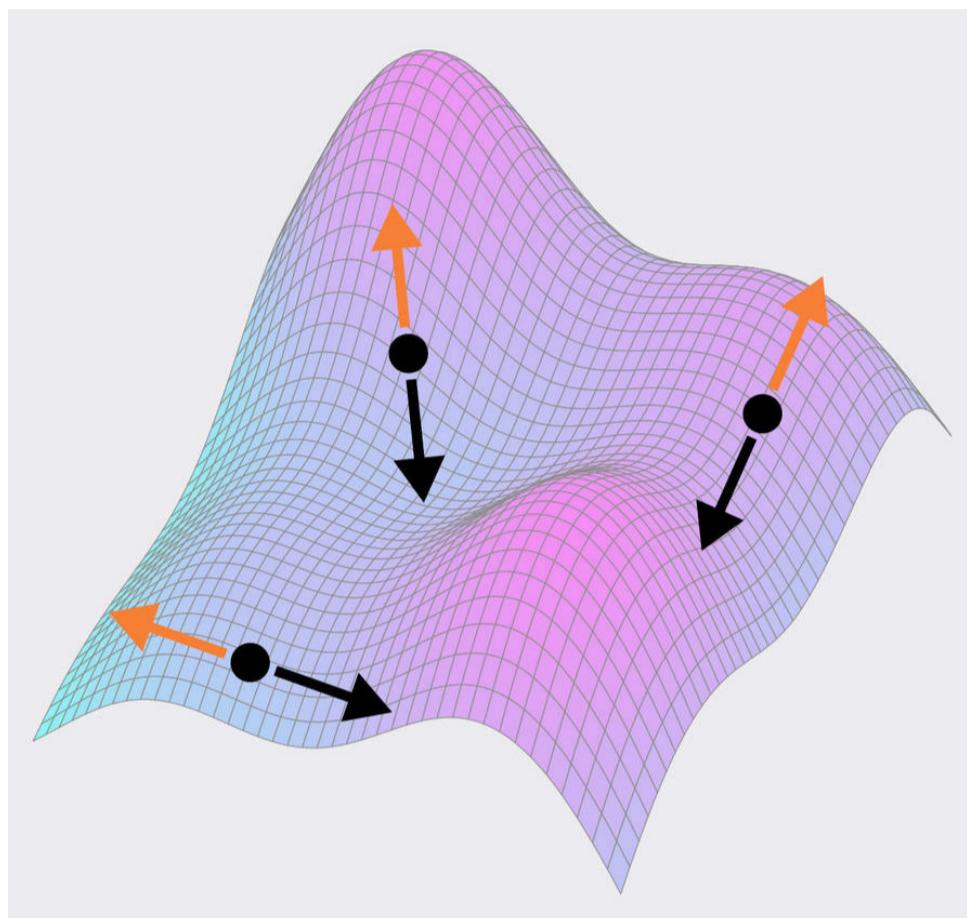


Figure 5.17: The direction of maximum ascent (in orange) always points in exactly the opposite direction as maximum descent (in black).

The direction of maximum ascent, shown in orange in Figure 5.17, is called the **gradient**. If we flip it around to find the direction of maximum descent, shown in black, we call that the **negative of the gradient**, or just the **negative gradient**. So the gradient points towards the steepest direction uphill, the negative gradient points in the steepest direction downhill. 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.

As Figure 5.17 shows, we can find both the gradient and the negative gradient at any point on a surface (well, not *any* point, as we'll see in a moment, but any point where there actually is an uphill or downhill to be found).

Now that we know the direction of maximum ascent, we can also find its **magnitude**, or strength, or size. That's simply how much we're going uphill. If we're going uphill slowly, the magnitude of our ascent is a small number. If we're going uphill by a lot, it's a bigger number.

We can use the gradient to find the local maximum, just as we used the derivative. 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 should *follow the negative gradient*, and always walk in the direction *exactly opposite* the gradient associated with each point under our feet as we descend. Figure 5.18 shows this step-by-step process in action.

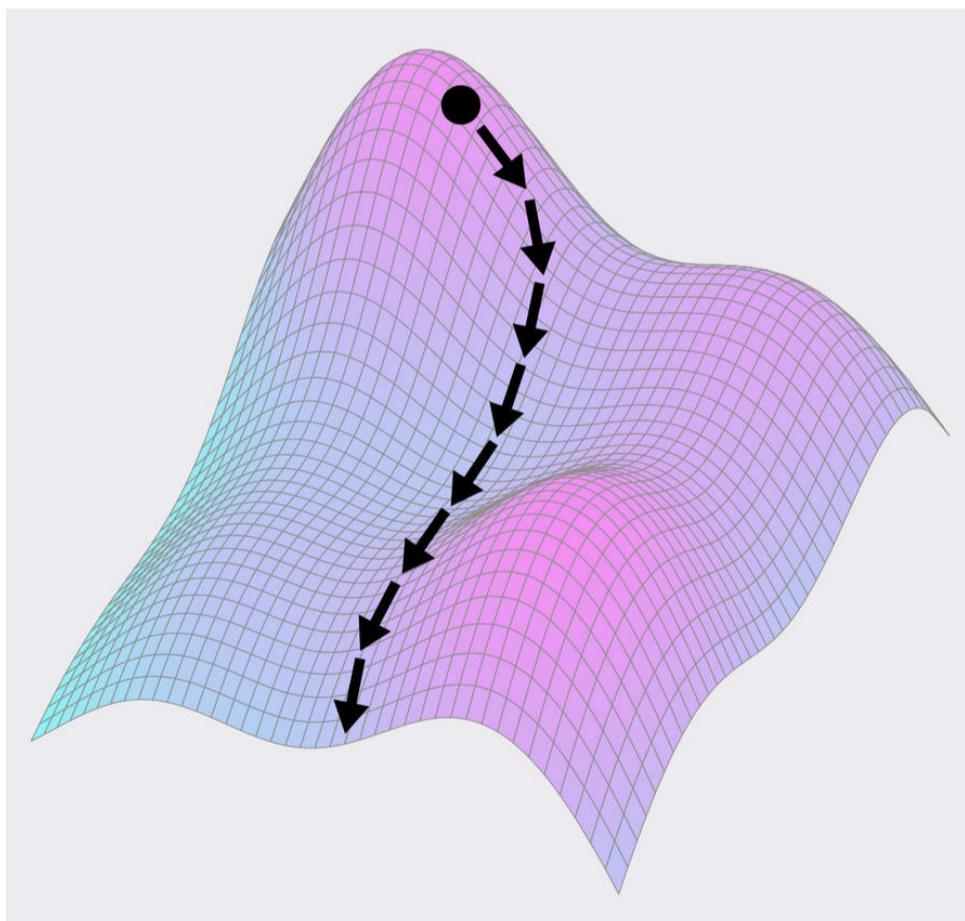


Figure 5.18: To get downhill, we can find the negative gradient (which points in the direction of maximum descent) and take a small step in that direction. Then we find the gradient at that new location, take another small step downhill, and so on.

Suppose that we're at the very top of a hill, as in Figure 5.19. This would be a local maximum (and maybe the global maximum). Here, there is no "uphill" direction to go in. Recall that our derivative had a value of 0 when we were at a locally flat section of a curve. Now we're at a locally flat section of the surface. Because there's no "uphill" to go to, our maximum rate of ascent is zero, and magnitude of the gradient is zero. There's no arrow at all! We sometimes say the gradient has **vanished**.

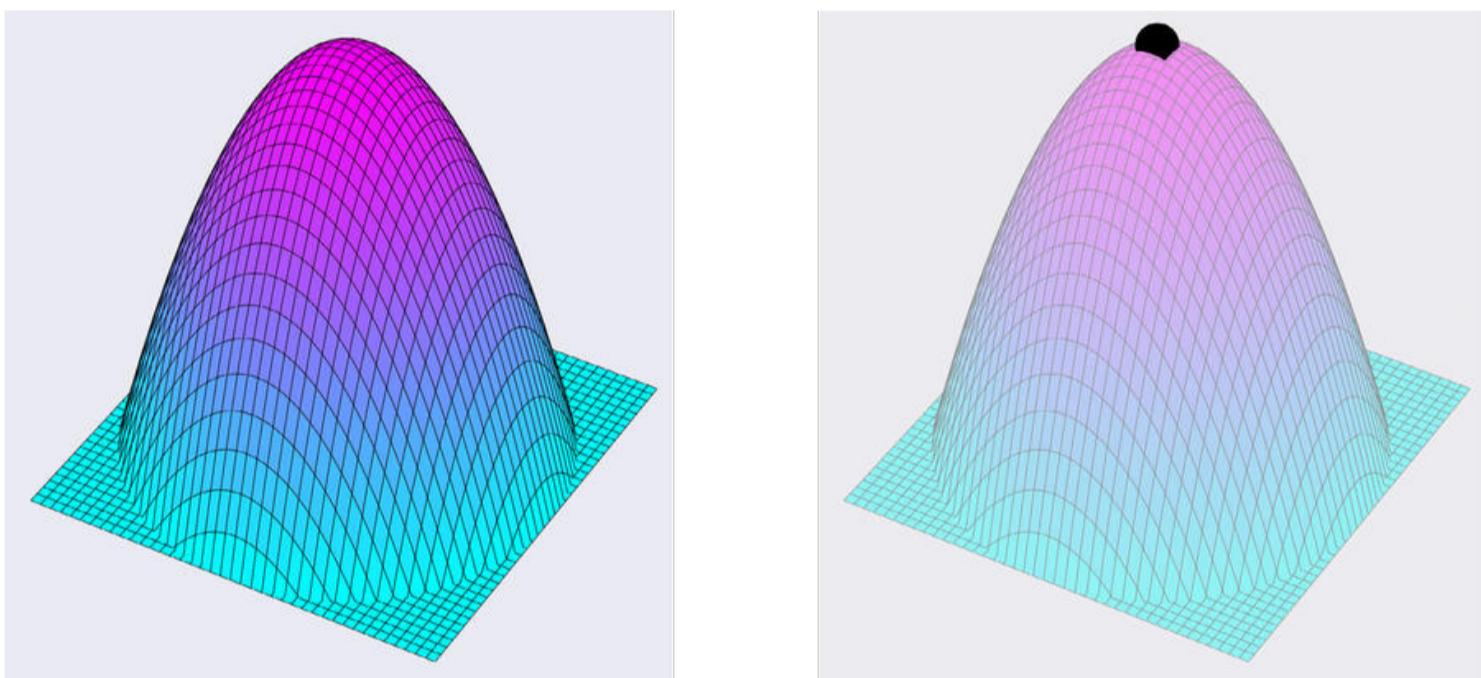


Figure 5.19: At the very top of a hill, there is no "uphill" left to go. Left: The hill. Right: Our location at the very top of the hill. At this point there is no gradient, because there's no way to go up. This is a local or global maximum.

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 5.20? This would be a local minimum (and maybe the global minimum).

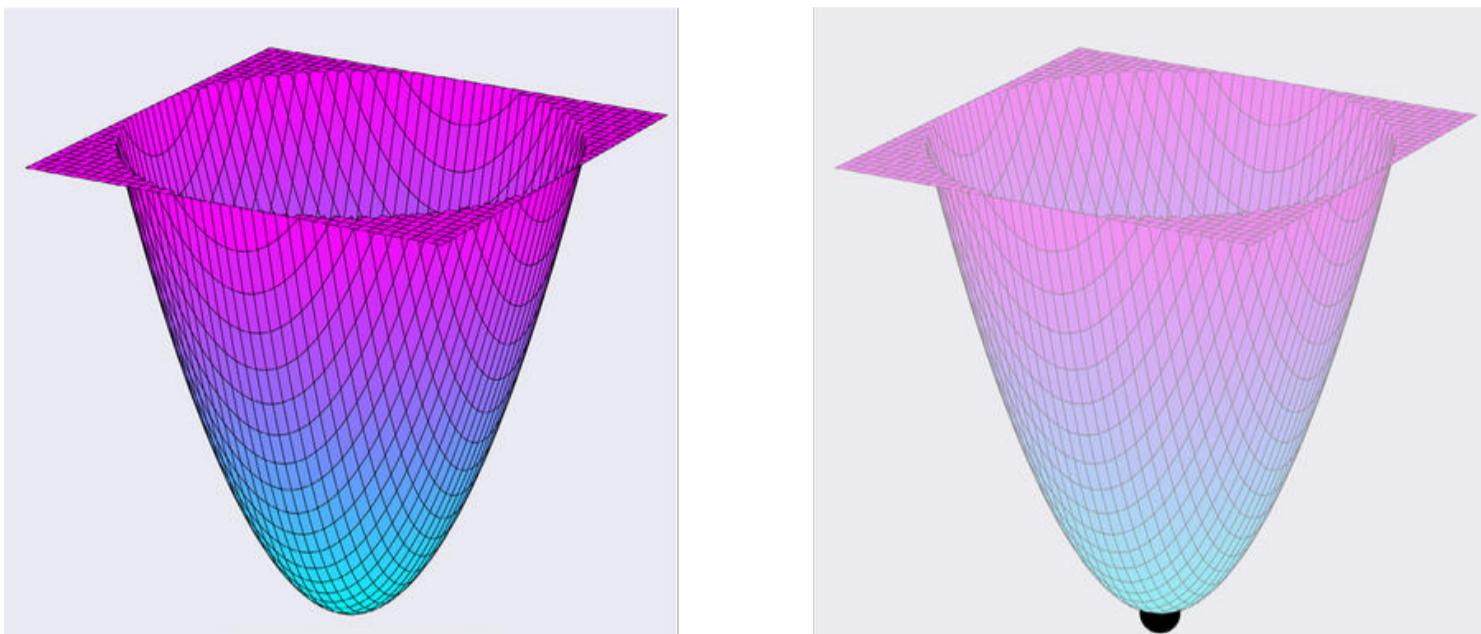


Figure 5.20: At the very bottom of a bowl, every move we make is uphill. Left: A bowl. Right: A point at the bottom of the bowl. Like the hilltop, at this point there is no gradient because right at the very bottom of the bowl the surface is locally flat. This point is either a local or global minimum.

At the very bottom of the bowl, like the top of a hill, the surface is flat. There's no gradient or negative gradient. In other words, there's either no direction that takes us either up or down, or just no “best” direction.

What if we're not on a hilltop or in a valley or on the side of slope, but we're just on a flat plain, or **plateau**, as in Figure 5.21?

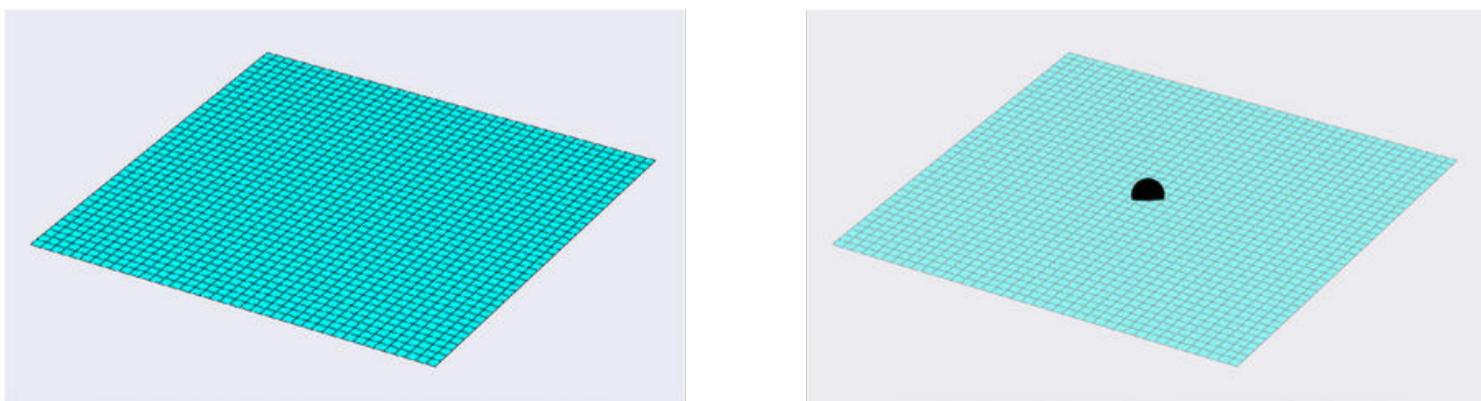


Figure 5.21: A flat surface, plain, or plateau. Left: The plateau. Right: The point on the plain is mainly on the plane. This point has no gradient.

Just like being on the hilltop, there's nowhere to go “up.” When we're on a plateau, we again have no gradient at all.

So far, we've seen local minima, maxima, and flat regions, just as we saw in 2D. But in 3D, there's a completely new type of feature. In its local neighborhood, it looks like the saddle that horse riders use.

In one direction, we're in the bottom of a valley, while in the other direction, we're at the top of a hill. Naturally enough, this kind of shape is called a **saddle**. An example saddle is shown in Figure 5.22.

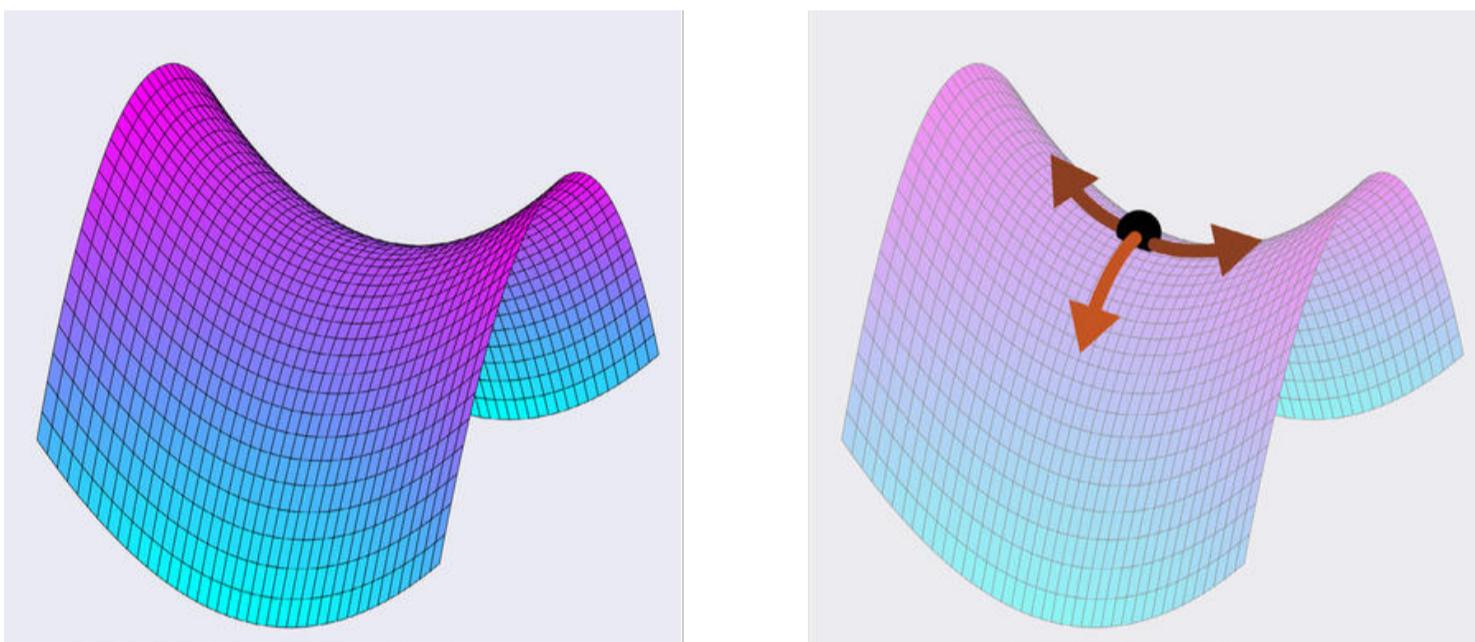


Figure 5.22: A saddle goes upwards in one direction and downwards in another. Left: A saddle. Right: A point on the saddle. The lighter arrow follows the downward direction, and the darker arrow points upwards. There's another light arrow heading downward on the far side of the saddle.

If we're in the middle of the saddle, as in Figure 5.22, 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, and there's no gradient.

But if we drift just a little bit in one direction or another, we'll find a little bit of curvature, and then the gradient re-emerges to show us the direction of maximum ascent from that spot.

Generally speaking, if we're on a part of the landscape where there's some up-and-down change near us, we'll have a gradient that points in the direction of maximum increase, and the length of the arrow tells

us how quickly that increase will happen. The arrow pointing in the opposite direction, with the same length, points in the direction of maximum decrease.

If we're on a flat local neighborhood (a plateau, a hilltop, a valley, or a saddle), then there is no gradient. We can imagine this as an arrow with no length and no direction, but since that's hard to draw, we usually just say that the gradient *vanishes*, or disappears, in those places. We also sometimes say that we have a *zero gradient*, which means the same thing. When we have no gradient, we don't know which way to go to climb uphill or downhill the fastest. Sometimes there's no uphill or downhill available.

We'll see that neural networks learn by following the gradient of their error. The networks adjust the numbers that control them in an effort to follow the negative gradient, so that their errors become smaller and smaller as they learn. This operation, naturally enough, is called **gradient descent**, and in Chapter 18 we'll see how to use it to train our deep networks.

References

[Apostol91] Tom M. Apostol, “Calculus, Vol. 1: One-Variable Calculus, with an Introduction to Linear Algebra, 2nd Edition”, Wiley, 1991.

[Berkey92] Dennis D. Berkey and Paul Blanchard, “Calculus”, Harcourt School, 1992.

Chapter 5: Curves and Surfaces

Chapter 6

Information Theory

How to think about and measure
the amount of information in our data,
calculations, and results, and how to compare
the efficiency of different representations of data.

Contents

6.1 Why This Chapter Is Here	233
6.1.1 Information: One Word, Two Meanings	233
6.2 Surprise and Context.....	234
6.2.1 Surprise.....	234
6.2.2 Context.....	236
6.3 The Bit as Unit	237
6.4 Measuring Information	238
6.5 The Size of an Event.....	240
6.6 Adaptive Codes.....	241
6.7 Entropy	250
6.8 Cross-Entropy.....	253
6.8.1 Two Adaptive Codes.....	253
6.8.2 Mixing Up the Codes	257
6.9 KL Divergence.....	260
References	262

6.1 Why This Chapter Is Here

In this chapter we'll look at the basics of **information theory**. This is a relatively new concept, introduced to the world in 1948 in a groundbreaking paper, but it's laid the groundwork for everything from modern computers and satellites to cell phones and the internet [Shannon48].

The goal of the original theory was to find the most efficient way to communicate a message electronically. But the ideas in that paper were deep, broad, and profound. They give us tools for measuring how much we know about something by converting it to a digital form that we can study and manipulate.

In this chapter we'll take a fast tour through some of the basics of information theory, as usual staying free of abstract mathematical notation. Knowing something about the field is important because its terms and ideas appear frequently in machine learning documentation and papers. In particular, the measurements provided by information theory are useful when we evaluate the performance of neural networks like deep networks.

6.1.1 Information: One Word, Two Meanings

This chapter is all about **information theory**, so before we get into the details, let's agree on what the word "information" will mean to us here.

Information is one of those words that has both an everyday meaning and a specialized, scientific meaning. In this case, they 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."

6.2 Surprise and Context

When we receive a communication of any kind, whether it's a word, a story, or a piece of pottery, it's a result of a change in the world. 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 presents some kind of communication to a **receiver**.

In this chapter we'll sometimes use the term "surprise" to represent how much new information this communication provides to the receiver. This is an informal term that refers to the gap between what we expect from some event, and what we actually learn from it. For example, suppose the doorbell rings. If we've recently ordered something online, and there's a delivery person from that company on the other side of the door, seeing them might be a little surprising, but it wouldn't be completely unexpected. But if the doorbell rings and there's a gorilla on the other side, or 25 rabbits stacked up on a turtle, that would be a lot more surprising! One of our goals in this chapter will be 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'd like to describe how surprised we are by the communication we receive. This is useful because we'll see that the greater the surprise, the greater the information that was delivered.

6.2.1 Surprise

Let's get specific. Suppose we get an unexpected text message from an unknown number. We open it up and the first word is "Thanks."

How surprised would we be? Surely we'd be 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.

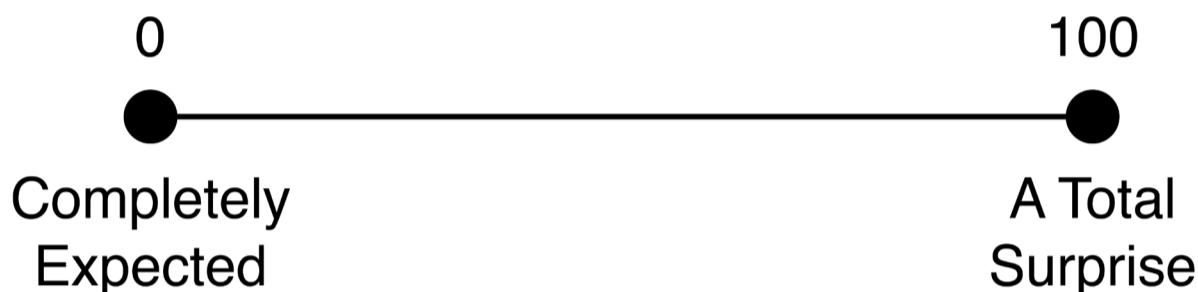


Figure 6.1: The surprise scale, expressed as a value from 0 to 100.

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 wasn't "Thanks," but instead was "Hippopotamus." Unless we're working with those animals or otherwise involved with them, that's likely to be a very surprising first word of a message. Let's say we'd rank that at an 80 on the surprise scale, as in Figure 6.2.

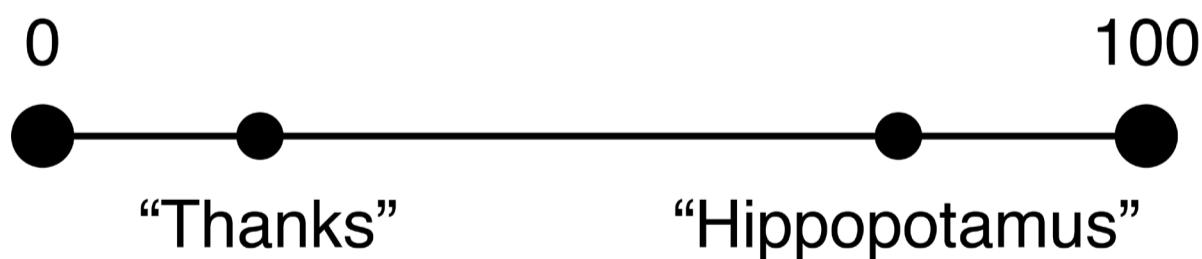


Figure 6.2: If the first word of an unexpected text is "Thanks," we might give that a surprise value of about 20. The word "Hippopotamus" would be much more unexpected, so we might give it a surprise value of 80.

Although "hippopotamus" might be a big surprise at the start of a message, it might not be surprising later on. The difference is **context**.

6.2.2 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 would represent 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 environment that we bring to any message, even before we're 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 capture our understanding of the environment and what we expect to learn from it.

At the other extreme is **local context**. In a text message, the local context for any given word would be 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 hippopotami on a regular basis, that word will likely be very surprising.

But if the message began 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" wouldn't be very surprising.

The difference between global and local context comes down to *dependency*. A word is more or less surprising in its local context because it depends on the other words around it.

When we interpret the words of our message in the global context, we don't have this dependency. Each word is taken by itself.

The global context still influences our surprise. If we work as river rafting tour guides and floating past hippos is a daily event, then the word "hippopotamus" could be a part of our everyday vocabulary, and thus not surprising to hear. But in that environment, the word "retro-rocket" might be completely unexpected and a big surprise. If we worked as spacecraft engineers, then in that context "retro-rocket" would be an everyday word, but "hippopotamus" would be a surprise.

We can describe the amount of surprise of 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 density function**, or **pdf (probability mass function or pmf)**, as we saw in Chapter 3.

That means we can draw a random variable from that pdf 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 would be 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.

6.3 The Bit as Unit

We'll use the probability of a word to help us get at the amount of information that's conveyed by sending that word, but first we need to talk about the units we'll use when discussing information.

Everyone knows that a “bit” is a single unit of information, either a 0 or 1. We also know that we can store bits using electronic circuits, so we might have a chip that has 1000 bits of memory.

Although the last paragraph uses everyday language and we all know what is meant, it’s a little imprecise. It will be helpful for us to tease apart a technical definition of a “bit” from its popular use, so we can use it accurately when discussing information.

A **bit** is a *unit*, like a kilogram, a second, or a gallon. We sometimes say something like “This jug of milk is a gallon,” but a more precise statement would be “This jug can hold one gallon of milk.” In the same way, rather than say “This memory is 1000 bits big,” we would more carefully say, “This memory is capable of holding 1000 bits of information.”

The difference is important because we want to remember that a bit is a unit, and not an electrical charge, quantum state, or any other specific mechanism. We can use a circuit or device to *represent* one bit’s worth of information, but those things are not bits themselves.

6.4 Measuring Information

We can measure the amount of formal information in a text message with a formula. 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 our probability mass function that describes the surprise inherent in each word it can contain. Taken together, this will produce a number that tells us how much surprise, or information, is in the message. This is usually expressed in bits.

The formula was designed so that the values it produced for each word (or, more generally, each **event**) have four key properties. We’ll illustrate each one using a context where we work in an office, not on a river.

First, likely events have low information. “Stapler” has low information.

Second, unlikely events have high information. “Crocodile” has high information.

Third, likely events have less information than unlikely events. “Stapler” conveys less information than “crocodile.”

Finally, the total information due to two *unrelated* events is the sum of their individual information values found separately. It’s rare in normal conversation for two consecutive words to be completely unrelated. But let’s suppose that we have an enormous number of office tools on our desk, in a huge array of colors. If someone asks to borrow the “green stapler,” we could consider the two words to be mostly unrelated, since even after hearing “green” there are a huge number of possible objects to ask for. If we suppose that the two words are completely unrelated, then we would find the information in the phrase “green stapler” by adding the information communicated by each word.

In normal conversation the words that lead up to any given word often narrow the possibilities of what it could be. So if someone says, “Today I ate a big...”, words like “sandwich” and “pizza” will carry less surprise than “bathtub” or “sailboat.” In this case, the sum of the individual surprises of each word taken alone will usually be greater than the surprise of the message as a whole. By contrast, suppose we’re sending a device’s serial number, which is essentially an arbitrary sequence of letters and perhaps numbers, like “K9GNNP4R.” If the characters really have no relation to each other, then adding the surprise due to each character will give us the overall surprise in the entire message representing the serial number.

The first three properties of our formula for information are satisfying to our common sense. Paraphrasing them, they say that common events are hardly surprising, uncommon events are quite surprising, and uncommon events are more surprising than common events.

The fourth property is what lets us move from single events to sets of them. The combined surprise of two unrelated words is the sum of their individual surprise values. So this is how we get from measuring the surprise, or information, in a single word to the surprise, or information, in an entire text message.

Though we haven't gone into the math, we have reached a formal definition of **information**: it's a little formula, or algorithm, that takes in one or more events and a probability distribution that tells it how surprising each event would be to us. From those two inputs it provides a number for each event, and guarantees that those numbers satisfy the properties above.

The output of this formula is an amount of information, and it's usually presented in bits, as we discussed in the last section.

6.5 The Size of an Event

The amount of information carried by each event is also influenced by the size of the probability function we hand to the 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 [Seuss60]. To represent a number between 0 and 49 we need 6 bits of information per word. Robert Louis Stevenson's book *Treasure Island* contains about 10,700 unique words [Stevenson83]. We'd have to use 14 bits per word to uniquely identify each word in that book.

So 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 the global context shared by sender and receiver.

Let's take that idea and run with it.

6.6 Adaptive Codes

In the last section we saw that it makes sense to adapt our transmission to the context shared by sender and receiver. For instance, if we're discussing orchestral music, then words like "tuba" and "violin" will be more common (that is, more probable) than words like "starfish" and "octopus". But if we're discussing underwater exploration, the opposite is true.

We saw that we can use this observation to improve the efficiency with which we can send messages.

A great example of this 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.

A	• —	J	• — — — —	S	• • •
B	— — • •	K	— • —	T	—
C	— — • — •	L	• — — • •	U	• • —
D	— — • •	M	— — —	V	• • • —
E	•	N	— •	W	• — —
F	• • — — •	O	— — — —	X	— — • • —
G	— — — •	P	• — — — •	Y	— — • — —
H	• • • •	Q	— — — • —	Z	— — — • •
I	• • •	R	• — — •		

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 single 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 [Longden87].

So 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).

This was a great way to send a message across a noisy telegraph wire, because the tone could be made out even in the presence of noisy static and other interference. It's still a good way to send a message over radio when there's a lot of interference.

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 its **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 takes a lot longer to send. The letter Q takes 13 dits to send (3 for

each of the 3 dashes, 1 for the dot, and 1 for each of the 3 spaces), but we'd need only 7 dits to send the letter H (4 dots, and 1 for each of the 3 spaces).

Morse code distinguishes its two symbols by their duration. Our discussion will be a little easier if we give both symbols the same duration. We can do this by making the “dot” one tone (say, a low frequency), and the “dash” another tone (say, a high frequency).

Then to send any character, we send a series of short bursts of one tone or the other, each burst being a single dit long. For example, to send an R, instead of “dot, dash, dot” we'd send “low, high, low.” We won't need silence between the symbols, but we'll include 1 dit of silence between characters.

Now every character that has 4 symbols will take 4 dits to send, regardless of the mix of dots and dashes.

Let's look at the patterns of the different characters. From Figure 6.3, it might not be clear that there's any principle behind how the various patterns were assigned. But there's a beautiful idea there waiting to be uncovered.

Figure 6.5 shows a list of the 26 Roman letters, sorted by their typical frequency in English [Wikipedia17].

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 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? Clearly Morse code is following our letter-frequency table, but it seems to be doing it a little inaccurately.

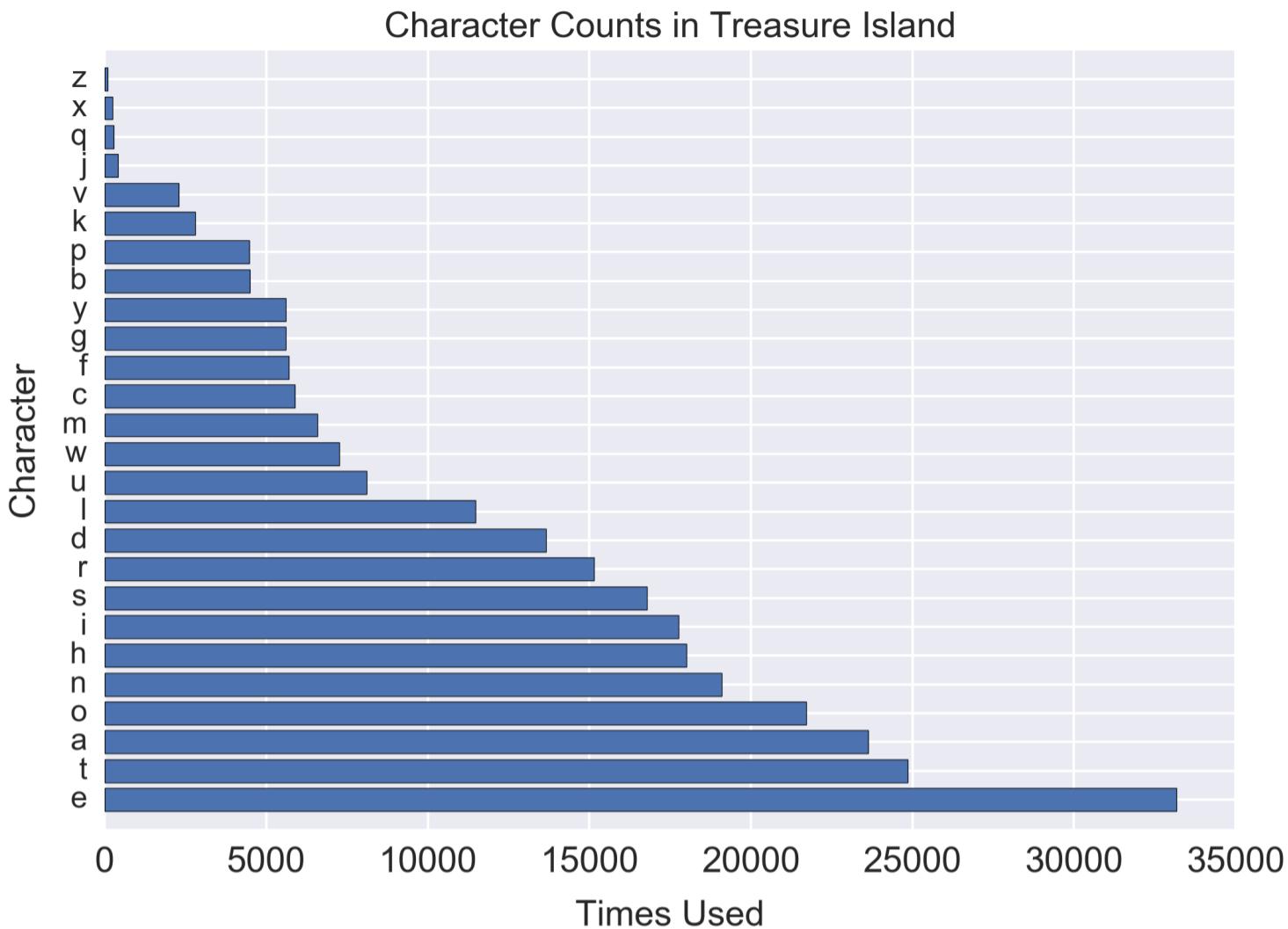
The explanation starts with Samuel Morse, who only defined patterns for the numbers 0 through 9 in his original code. Letters and punctuation were added to the code by Alfred Vail, who designed those patterns in about 1844 [Bellizzi11].

Vail didn't have an easy way to compute 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 (-.-)" [Pope88]

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, by placing a separate metal "slug" for every character into a giant tray. Vail reasoned that the most popular characters would have the most 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 [McEwen97].

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) line up with some actual text, Figure 6.6 shows the frequencies for the letters from “Treasure Island” by Robert Louis Stevenson [Stevenson83]. For this chart, we counted only the letters, which we turned into lower case 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. Upper and lower cases were combined.

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. All that requires is scaling the horizontal axis of the graph by just the right amount. The result is shown in Figure 6.7.

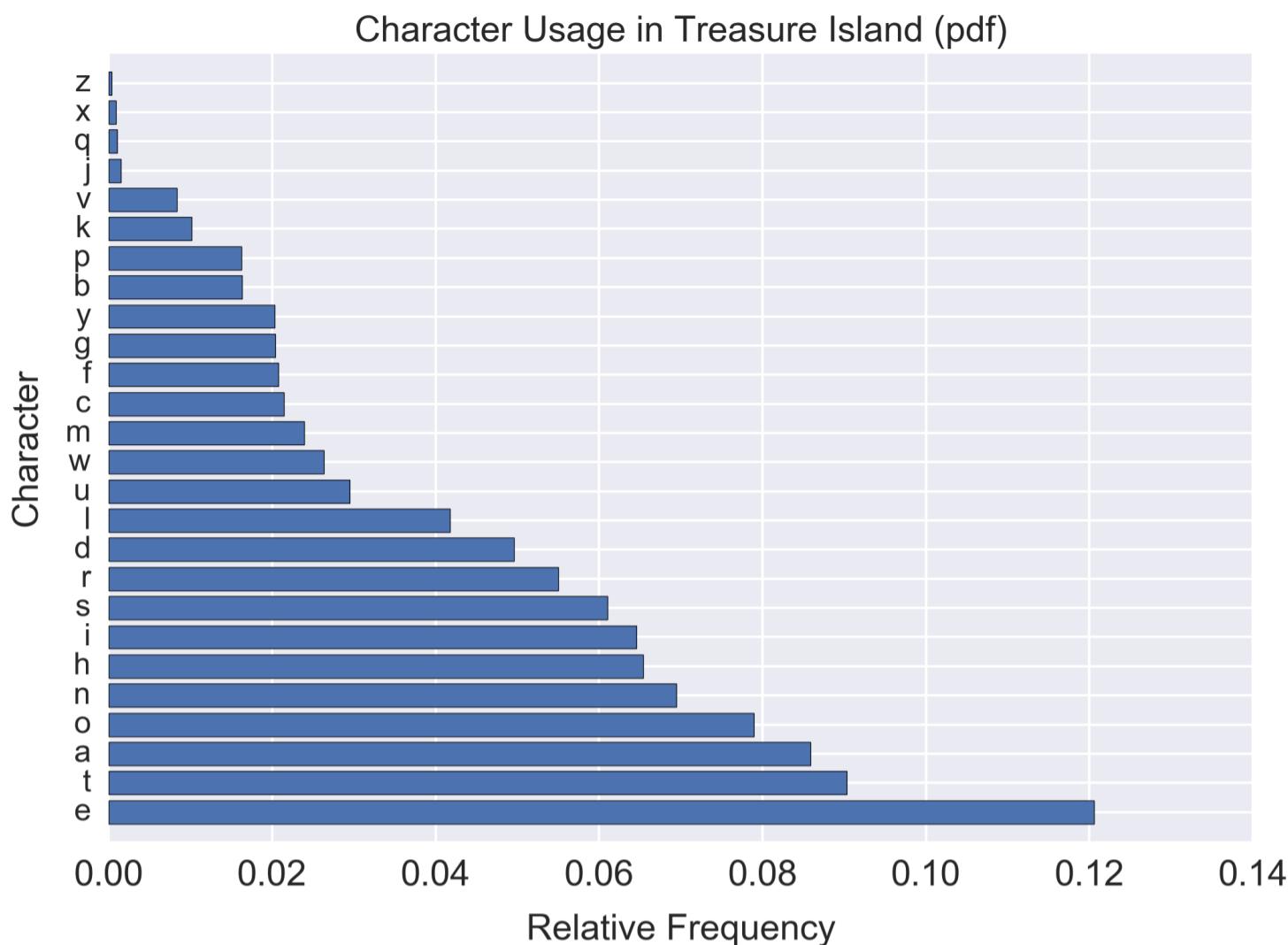


Figure 6.7: The probability distribution for characters in *Treasure Island*.

Now let's use our probability distribution of letters to improve the efficiency of sending *Treasure Island* via Morse code. We'll look at a couple of different ways to do this. We'll continue to use our two-toned version, where dots are a high tone and dashes are a low tone, and both last for a single dit.

Let's 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 4 symbols he could only label 16 characters, but with 5 symbols he could label 32 characters.

Figure 6.8 shows how we might arbitrarily assign such a 5-symbol pattern to each character. This is an example of a **constant-length code**, also called a **fixed-length code**.

A	• • • • •	J	• — • • —	S	— • • — •
B	• • • • —	K	• — • — •	T	— • • — —
C	• • • — •	L	• — • — —	U	— • — • •
D	• • • — —	M	• — — • •	V	— • — • —
E	• • — • •	N	• — — • —	W	— • — — •
F	• • — • —	O	• — — — •	X	— • — — —
G	• • — — •	P	• — — — —	Y	— — • • •
H	• • — — —	Q	— • • • •	Z	— — • • —
I	— • — • •	R	— • • • —		

Figure 6.8: Assigning 5 symbols to each character gives us a constant-length code.

Remember that the dot and dash are now sent with a low and high tone, respectively, each lasting for a single dit.

In Figure 6.8 we didn't create a character for the space, following in the footsteps of the original Morse code. Since spaces don't have an explicit character, counting them turns into a constant little detail, so we'll just ignore spaces between letters and words for the rest of this discussion.

The first two words in the text of *Treasure Island* are the name “Squire Trelawney.” Since every character takes five symbols, this phrase of 15 symbols (remember that we're ignoring the space) requires $5 \times 15 = 75$ symbols, as shown in Figure 6.9. That's 75 dots and dashes all told.

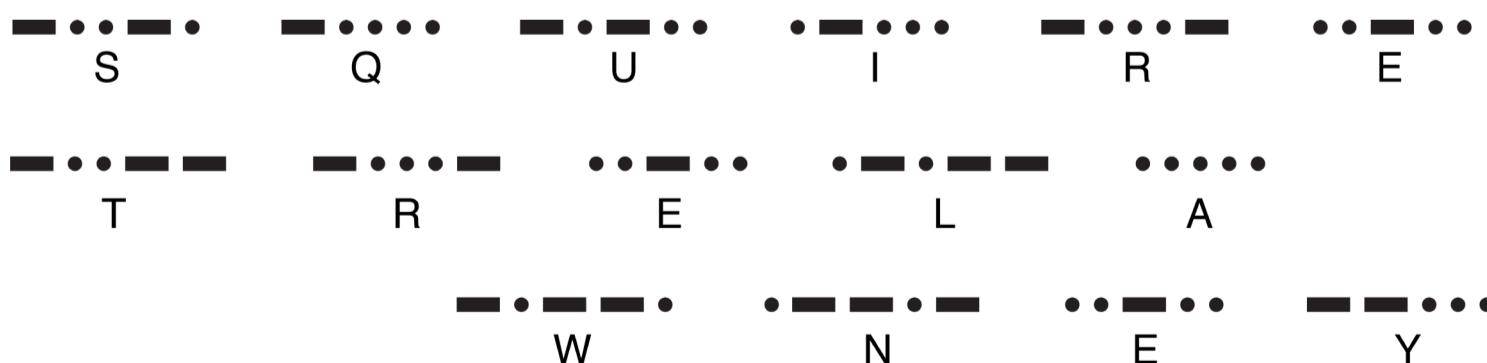


Figure 6.9: The first two words of *Treasure Island*, using our constant-length code.

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.

•••	— — • —	• • —	• •	• — — •	•			
S	Q	U	I	R	E			
—	• — •	•	• — • •	• —	• — —	— •	•	— • — —
T	R	E	L	A	W	N	E	Y

Figure 6.10: The first two words of *Treasure Island*, using Morse code.

If we count up the symbols, there are only 37 elements. Since each symbol (dot or dash) takes the same length of time in our modified two-tone code, the 37 elements of the Morse code version of the letters takes half the time of the 74 elements in our constant-length code ($37/74 = 0.5$). That savings came from adapting our code to the content we were 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 a the constant-length code, cutting our communication time almost in half.

Let's look at the whole text of *Treasure Island*, which contains about 338,000 characters (excluding spaces, punctuation, etc.). Using the 5 symbols per character of the fixed-length code, we'd need to send about 1.7 million symbols to transmit the whole book. Using the standard Morse code patterns, we'd need about 707,000 symbols. That's only about 42%! We could send the book in far less than half the time required by a non-adaptive code.

We could do even a better if, instead of using standard Morse code, we tuned the distribution of symbols to more closely match their actual percentages in the text of the book. 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 will be dwarfed by the message itself.

The idea of tuning a code to match the content of a message is central to our discussion, so let's state it plainly. By using Morse code, which is adapted to the average letter frequencies in English, we eliminated a lot of symbols from our message, speeding up its communication. But if we take another step, and instead of Morse code we build a custom *Treasure Island Code* that is perfectly adapted to the contents of *Treasure Island* specifically, we could expect even more savings. This idea led to the celebrated **Huffman code**, which greatly reduced the amount of time required to send a message [Huffman52].

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 receive, 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 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 will tell us just how much information we're about to communicate to our recipient.

6.7 Entropy

Recall that Figure 6.7 gives us a probability distribution for all the letters in *Treasure Island*.

Now suppose someone sends us a letter drawn at random from *Treasure Island*. That's saying exactly the same thing as supposing that someone draws a character from this probability distribution.

Not knowing which specific character they've chosen, how much information are we probably going to receive? In other words, given the probability distribution of Figure 6.7, if we receive a bunch of characters drawn from that distribution, what is the average information we receive per character?

There's a formula that gives us that value as a number in bits. That value is called the **entropy**, or sometimes **Shannon entropy**, in honor of the scientist who first presented the idea [Serrano17].

The entropy depends on both the message and the probability distribution. If we compute the entropy for a message using one probability distribution, and then compute entropy for the identical message with a different probability distribution, we will usually get two different results.

Let's change our focus now from letters to entire words. This gives us a chance to really dig in deeper into the differences between different manuscripts, because while all writers use the same 26 letters (plus capitals, punctuation, etc.), they definitely don't all use the same words.

To get started with thinking about entropy for collections of words, let's suppose that we've come up with a code that represents each word in a message with a unique number. For example, 7 might stand for "happy," 38,042 might stand for "pomegranate," and so on. If we're using an adaptive code, then shorter numbers would require fewer bits to send than longer numbers, so we could send the number 7 for

“happy” many times before we needed as many bits as we’d require to send the number 38,042 for “pomegranate” even once. This idea is like the surprise scale we discussed earlier in the chapter.

Let’s now suppose that we have two different codes, both of which have numbers for most of the words in the English language, but they assigned the numbers differently.

For example, suppose that one code was designed for text where the word “remember” comes up a lot, so that word has a small number assigned to it. But the other code was designed for text where “remember” is less frequent, and it gets a larger number. When we use these two codes to send a message, the number of bits required by each code will depend on how well the frequencies of words in that message match the frequencies assumed by that code. For example, if the word “remember” comes up frequently in a message, code 2 is going to require more bits to send that message than code 1.

Figure 6.11 shows the first few words from the second paragraph of *Treasure Island*, with the number of bits required to send their corresponding numbers using two imaginary codes.

I remember him as if it were yesterday									
Code 1	3	14	7	9	8	7	11	113	sum = 172 bits
Code 2	4	64	21	2	4	8	32	86	sum = 221 bits

Figure 6.11: The start of the second paragraph of *Treasure Island*. Each word has its own number, assigned to it by a specific code. Here we show how many bits are required to send the number for each word in two different codes. For example, using code 1, we’d require 14 bits to send the number for “remember,” but code 2 would require 64 bits to send its version of “remember.”

Code 1 in Figure 6.11 is better matched to this text than Code 2, even though for some words Code 2 assigned smaller numbers. Overall, we would require fewer bits to send this message using the numbers that came from Code 1 than the numbers from Code 2.

Because the entropy tells us how much information, on average, is transmitted by each element of a message, we can also work the arrangement backwards. Given a message, and a probability distribution, we can estimate the number of bits required to transmit that message.

Let's look at this way. Suppose we have a probability distribution, and we're going to pick some element from it randomly. Can we describe how certain, or uncertain, we are regarding which item was chosen? We can answer this using entropy [Hopper17].

Suppose our distribution is made up of words. In fact, it contains just one word, "rhubarb." Then we can be completely certain that a word chosen at random from this distribution will be "rhubarb."

Suppose the distribution has two words, where the probability of picking "rhubarb" is 0.25 and the probability of picking "sassafras" is 0.75. Then we can be 75% certain that a word chosen at random will be "sassafras."

We can extend this idea for larger distributions made up of many words (or any other kind of data). By combining the probabilities according to Shannon's formula [Shannon48], we get a value for the **entropy** of the distribution. If there's only one event in our distribution, the entropy is 0. If there are lots of events but they all have a probability of 0 except for a single event with a probability of 1, the entropy is again 0. When all of the entries have the same probabilities, the entropy is at a maximum.

Our events have been words, but they can be anything we want to communicate, such as notes of a melody, pictures of leopards, or dance steps. Whatever thing it is that we're communicating, each time we receive one instance of that thing we get some information. And the entropy is the average amount of information we'll get from things that come from that distribution.

So we can attach a value of entropy to an entire distribution, by combining the probabilities of its events in a specific way. That entropy tells us how much information we can expect when we receive an element from that distribution.

In other words, the entropy of a probability distribution is the expected value of the information in that distribution. So the entropy tells us, on average, how much information we expect to receive from an event drawn from that distribution.

The word “entropy” is used in similar but different ways in other fields, where it often stands for a measure of the amount of organization (or disorganization) in a system. We can build a bridge between these two common interpretations by connecting the organization of a system to the information the system contains. Generally speaking, we think of organizing something as requiring information, so the more structure there is in something (e.g., a company, a building, or a molecule), the more information it holds.

6.8 Cross-Entropy

We’re going to define a key idea called **cross-entropy**, which is related to the idea of entropy that we just discussed. This is going to play a role when we measure the error of some neural networks.

To get a feeling for the idea before we formalize it, let’s look at two novels, and build up a word-based adaptive code for each.

6.8.1 Two Adaptive Codes

Both *Treasure Island* and *The Adventures of Huckleberry Finn* were written in English [Stevenson83] [Twain85]. That is, they both draw on the same set of words. Though each book has some words,

particularly in the dialog, that are probably not in any dictionary. We'll ignore those for now, and assume both books are based on the same set of core words.

Both books have about the same vocabulary size: *Treasure Island* uses about 10,700 different words, and *Huckleberry Finn* uses about 7400 different words. 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 bottom-to-top in Figure 6.12. For the purposes of counting words, we converted all upper-case letters to lower case (so that "With" and "with" would be considered the same word). The single-letter pronoun "I" therefore appears in the charts as the lower-case "i."

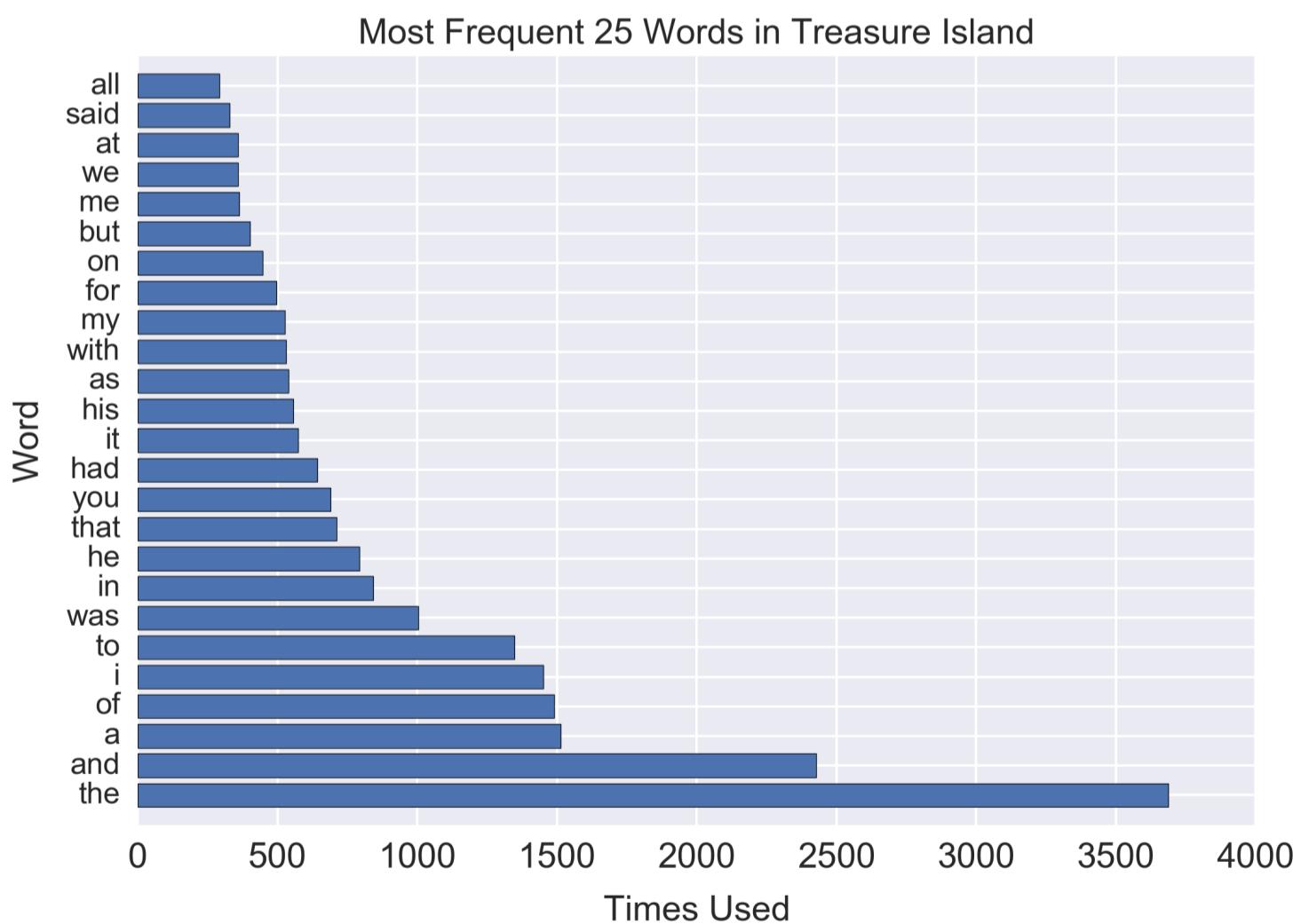


Figure 6.12: The 25 most popular words in *Treasure Island*, sorted by number of appearances.

Let's compare these to the 25 most popular words in *Huckleberry Finn*, shown in Figure 6.13.

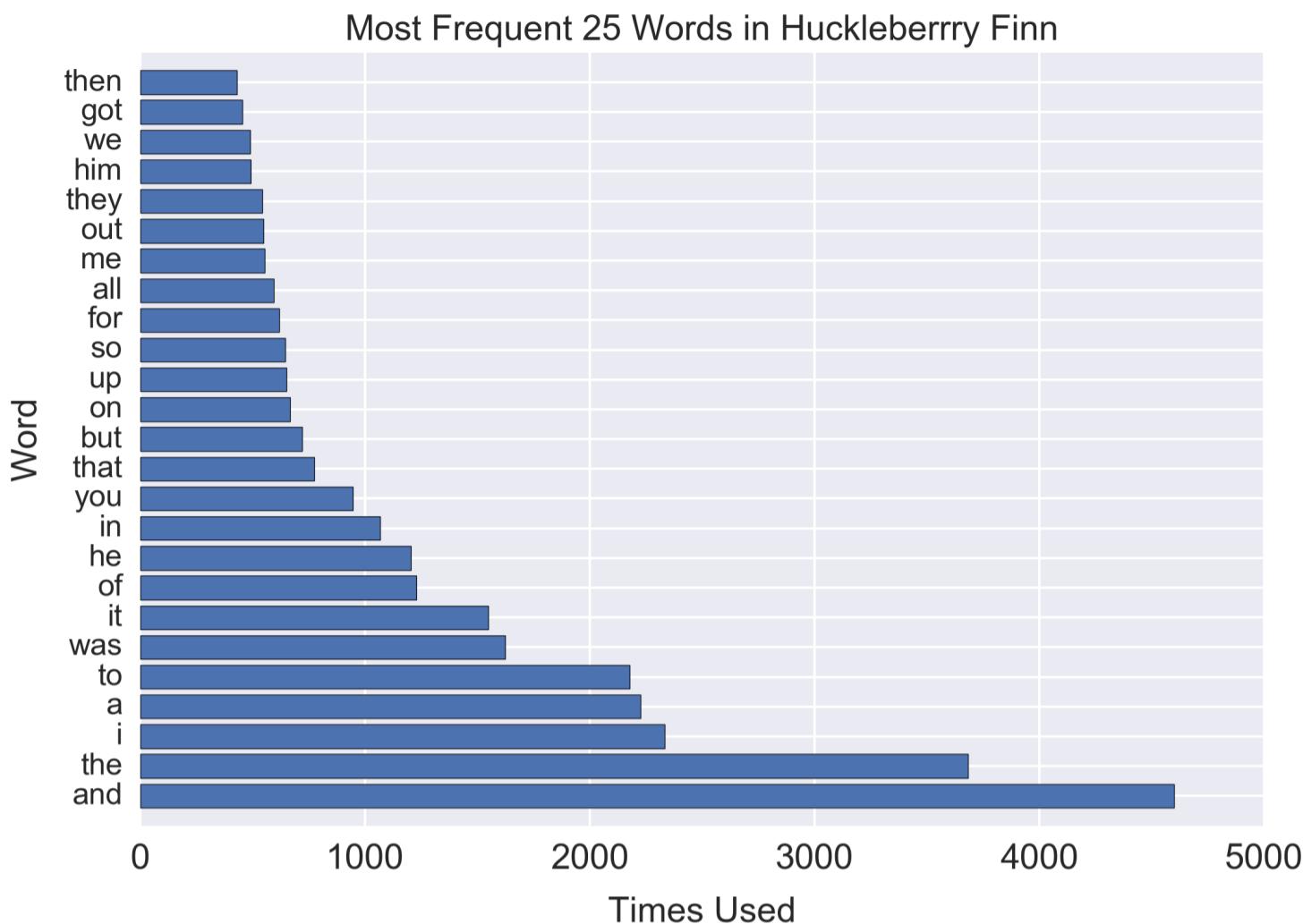


Figure 6.13: The 25 most popular words in *Huckleberry Finn*, sorted by number of appearances.

Perhaps unsurprisingly, the most popular dozen words in both books are the same (though in different orders), but then things begin to diverge.

Let's suppose we want to transmit the text of both books, word by word. We could take all the words in both books, alphabetize them, and then assign each word a number starting with 1, then 2, then 3, and so on.

But we know from our Morse code example above that we can send information much more efficiently by using a code that's adapted to the material being sent. So 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 and require us to send more bits (in *Treasure Island* there are about 2780 words that appear only once; in *Huckleberry Finn* about 2280 words appear only once).

Let's start with *Treasure Island*. We'll make an adaptive code for this text, starting with a tiny value for "the" and working our way up to huge values for one-time-only words like "wretchedness." Now we can send the whole book using that code and save time compared to using the code created by the alphabetic sorting, or a fixed code that uses the same number of digits for every word. To prepare for the discussions below, we'll also include one instance of every word that's in *Huckleberry Finn* but not in *Treasure Island*, so if we had to, we could send that book with this code.

Now we'll do the same thing for *Huckleberry Finn*. We'll make a code specifically for this text, giving the shortest code to "and" and leaving the big codes to one-time-only words like "dangerous" (shocking, but true: "dangerous" appears only once in *Huckleberry Finn*!). The *Huckleberry Finn* code will let us send the contents of this book more quickly than using the other codes we've discussed. As before, we'll also include one instance of every word that's in *Treasure Island* but not in *Huckleberry Finn*.

Note that both codes were built using the word frequencies of their respective books. There are two important observations to make about these codes.

First, our two codes are *different*. We can see that in the very first word, but as we get out of the top few words the word frequencies diverge. We'd expect that, because the two books are talking about very different subject matter, and so they use different words with different frequencies.

Second, both codes are drawn from the *same vocabulary*. That is, except for some oddball words in dialog, they both use English words you'd find in the dictionary. Remember that we assumed that all the words in *both* books are in both codes, so we can use either code to send either book.

Although the two books share the words of English, their words are in different orders. Most words they have in common appear with different frequencies, and some words in one book 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*.

6.8.2 Mixing Up the Codes

Now we have two codes, each of which can transmit every word in English. The *Treasure Island* code uses codes that are 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. If the ratio is exactly 1, then our adaptive code uses exactly as many bits as the non-adaptive code. If the ratio is 0.75, then the adaptive code sends only $3/4$ the number of bits needed by the non-adaptive 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 6.14 shows the compression ratio that we get from sending *Huckleberry Finn* with the code we built for it. We used a Huffman code, which we mentioned earlier, but the results would be similar for most adaptive codes [Wiseman17].

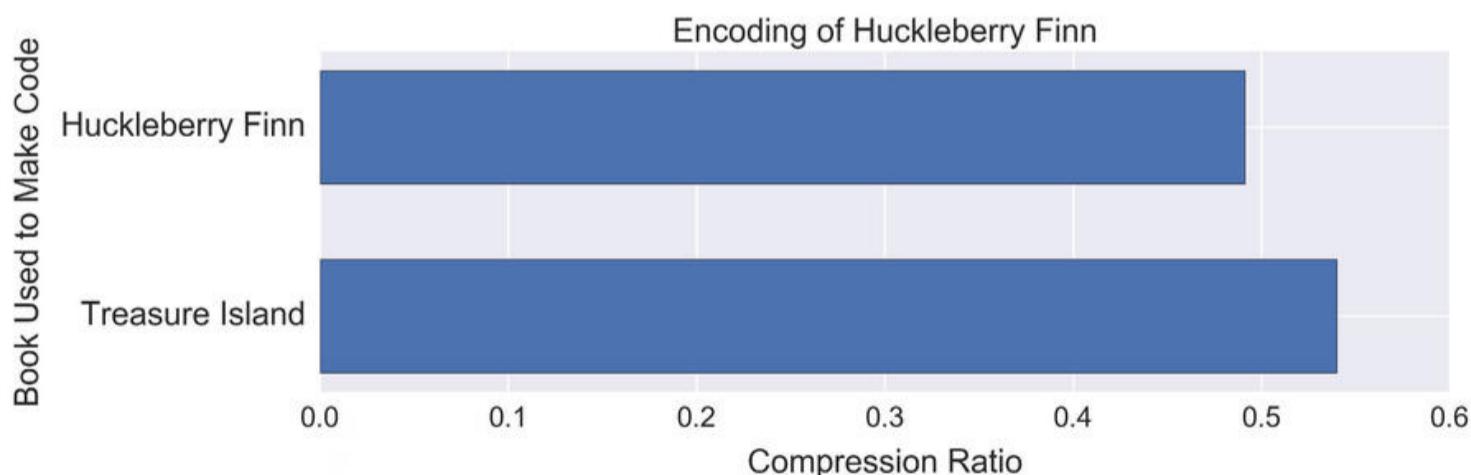


Figure 6.14: Sending *Huckleberry Finn* with adaptive codes. Top: The compression ratio from using the code built from *Huckleberry Finn* itself. Bottom: The compression ratio from using the code built from *Treasure Island*.

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 the fixed-length code.

Now let's try something new, and 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 6.14 shows our result, with a compression ratio of around 0.54. As expected, we didn't do quite as well.

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 6.15.

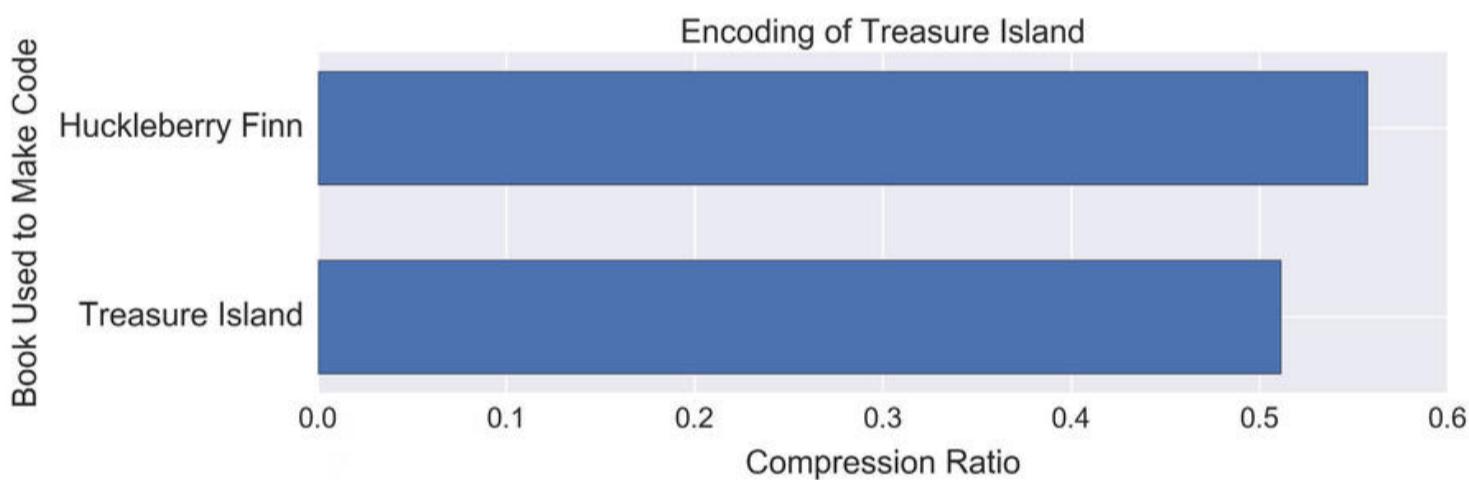


Figure 6.15: Comparing compression ratios for *Treasure Island*. Top: The compression ratio from using an adaptive code built from *Treasure Island* itself. Bottom: The compression ratio from compressing *Treasure Island* using a code built from the text of *Huckleberry Finn*.

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.

There is a formula that will tell us how the quality of the match between a code and a message, by telling us the average number of bits needed to send each word in the message with that code. The number produced by that formula is the **cross-entropy**.

The larger the cross-entropy, the more bits are required for each word. We can check how good a code would be for a given message by calculating its cross-entropy. If we compare two codes, the one that will send our message more efficiently is the one with the smaller cross-entropy.

We'll see in Chapter 20 that when we train neural networks, we often measure the error using a form of the cross-entropy.

Let's look at cross-entropy with a few more books. In Figure 6.16 we've compressed *Treasure Island* using adaptive codes built from that book, *Huckleberry Finn*, as well as *A Tale of Two Cities* [Dickens59] and an English translation of *Don Quixote* [Cervantes05], all as available on Project Gutenberg [ProjectGutenberg18].

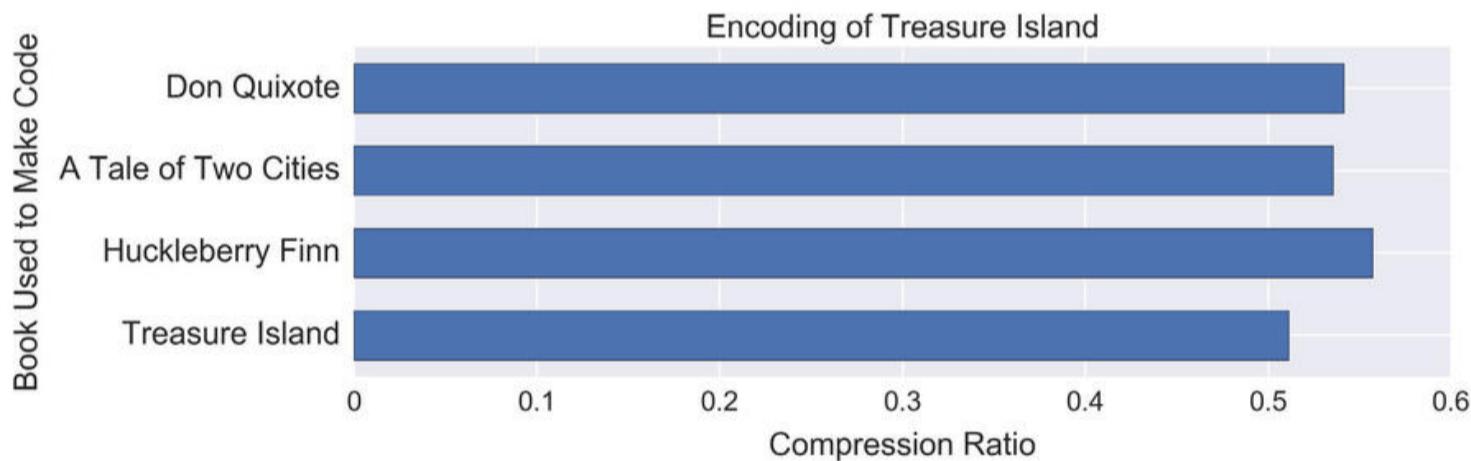


Figure 6.16: The efficiency of encoding *Treasure Island* with adaptive codes built from four different books. The code built for *Treasure Island* itself is the most efficient.

As we can see, the most efficient way to send *Treasure Island* is to use an adaptive code built from the words in that book, and not an adaptive code built from the words in any other book.

6.9 KL Divergence

The cross-entropy tells us how many bits we need, on average, to send a message with a particular code. This average will always be at its smallest when we're using the best possible code, which is a code that we make specifically for that message.

But creating a code takes time. Maybe we have a few codes already built and available. Might one of them be good enough? To answer that question, we'll look at the **expense** of a code. We can imagine that there's some cost associated with sending every bit, so the more bits we have to send, the more expensive the transmission. If one code has a lower expense for a given message than another code, it does better compression on that message, and thus requires fewer bits.

To determine the quality of a particular code for a particular message, we'll find the *added expense* we'd have to pay to send that message with that code, compared to sending that message with the best possible code. After all, if the added expense is 0, then we've already got a code as good as the best one. That means we have a perfect code.

The difference in the average number of bits between using any given code and the best one is the entropy (or cost) of using the wrong code, minus the entropy of using the right code.

That difference 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**, **information gain**, **relative entropy**, and **KLIC** (for Kullback-Leibler Information Criterion).

Another way to express this is that the KL divergence tells us what the penalty is for taking information generated by one probability distribution, and then representing it with a code made for a different probability distribution.

To compute the KL divergence for sending *Treasure Island* using the *Huckleberry Finn* code, we'll scale the word uses in *Treasure Island* to make them a probability mass distribution, shown in Figure 6.17.

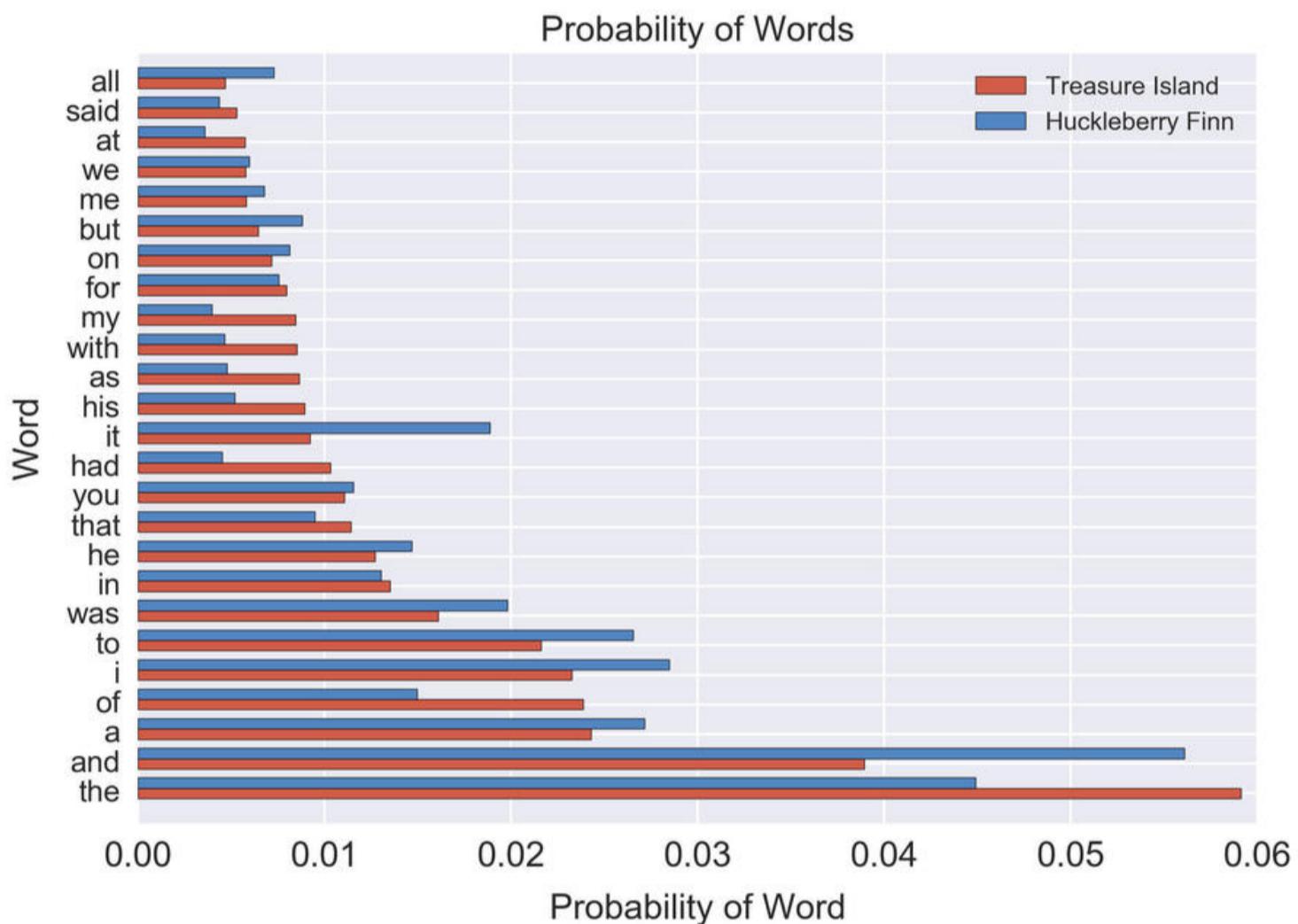


Figure 6.17: The probabilities of words in *Treasure Island* along with the probabilities of those words in *Huckleberry Finn*.

We also show the frequencies for the same words from *Huckleberry Finn*. We can see that the most popular word in *Treasure Island*, which is “the,” is used more frequently than it’s used in *Huckleberry Finn*. Therefore, the number of bits required to send “the” using the *Huckleberry Finn* code will generally be more than the number of bits we’d need using the *Treasure Island* code.

To find the KL divergence, we find the size of each of these discrepancies using a formula, and then add them all together, weighting the penalties for the more common words more than the rare ones.

If we run through the math, the KL divergence for sending *Treasure Island* with the *Huckleberry Finn* code, written “ $\text{KL}(\text{Treasure Island} || \text{Huckleberry Finn})$ ” is about 0.287. The two bars in the middle can be thought of as a single separator, like the more frequently-seen comma. 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 [Kurt17].

Note that the KL divergence isn’t symmetrical, because the word probabilities in the two books are different, and those probabilities are used to weight the mismatches. The KL divergence for sending *Treasure Island* with the *Huckleberry Finn* code, or “ $\text{KL}(\text{Huckleberry Finn} || \text{Treasure Island})$ ” is much higher, at about 0.5.

The KL divergence is great for telling us the cost of using a particular code, rather than a perfect code. Sometimes we may want to use an existing code that’s good enough, rather than build a new one.

References

[Bellizzi11] Courtney Bellizzi, “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>

[Cervantes05] Miguel de Cervantes, “The Ingenious Nobleman Mister Quixote of La Mancha”, published by Francisco de Robles, 1605. <https://www.gutenberg.org/ebooks/996>

[Dickens59] Charles Dickens, “A Tale of Two Cities,” Chapman & Hall, 1859. <https://www.gutenberg.org/ebooks/98>

- [Hopper17] Tim Hopper, “Entropy of a Discrete Probability Distribution”, [tdhopper.com, 2017.](https://tdhopper.com/blog/2015/Sep/04/entropy-of-a-discrete-probability-distribution/) <https://tdhopper.com/blog/2015/Sep/04/entropy-of-a-discrete-probability-distribution/>
- [Huffman52] David A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes,” Proceedings of the IRE, volume 40, number 9, 1952. http://www.cse.iitd.ernet.in/~pkalra/siv864/2017/assignment/huffman_1952.pdf
- [Kim16] Wonjoo Kim, Anupam Chattopadhyay, Anne Siemon, Eike Linn, Rainer Waser, and Vikas Rana, “Multistate Memristive Tantalum Oxide Devices for Ternary Arithmetic”, Nature Scientific Reports, Article 36652, 2016. <https://www.nature.com/articles/srep36652.pdf>
- [Kurt17] Will Kurt, “Kullback-Leibler Divergence Explained”, Count Bayesie blog, 2017. <https://www.countbayesie.com/blog/2017/5/9/kullback-leibler-divergence-explained>
- [Wikipedia17] Wikipedia authors, “Letter Frequency”, Wikipedia, 2017. https://en.wikipedia.org/wiki/Letter_frequency
- [McEwen97] Neal McEwen, “Morse Code or Vail Code?”, 1997, <http://www.telegraph-office.com/pages/vail.html> (quotes original article at <http://tinyurl.com/jobhn2b>)
- [Pope88] Alfred Pope, “The American Inventors of the Telegraph, which Special References to the Services of Alfred Vail”, The Century Illustrated Monthly Magazine, April, 1888, <http://tinyurl.com/jobhn2b> (Contains quotes attributed to William Baxter, without citation)
- [ProjectGutenberg18] Project Gutenberg creators, Project Gutenberg main site, 2018. https://www.gutenberg.org/wiki/Main_Page
- [Longden87] George Longden, “G3ZQS’ Explanation of how FISTS got its name”, FISTS CW Club, 1987. <https://fists.co.uk/g3zqsin-introduction.html>

[Serrano17] Luis Serrano, “Shannon Entropy, Information Gain, and Picking Balls from Buckets”, Medium, 2017. <https://medium.com/udacity/shannon-entropy-information-gain-and-picking-balls-from-buckets-5810d35d54b4>

[Seuss60] Dr. Seuss, “Green Eggs and Ham”, Beginner Books, 1960.
also see https://en.wikipedia.org/wiki/Green_Eggs_and_Ham

[Shannon48] Claude E. Shannon, “A Mathematical Theory of Communication,” Bell Labs Technical Journal, July, 1948 <http://worrydream.com/refs/Shannon%20-%20A%20Mathematical%20Theory%20of%20Communication.pdf>

[Stevenson83] Robert Louis Stevenson, “Treasure Island”, 1883.
<https://www.gutenberg.org/ebooks/120>

[Twain85] Mark Twain, “Adventures of Huckleberry Finn (Tom Sawyer’s Comrade)”, Charles L. Webster and Company, 1885.
<https://www.gutenberg.org/files/76>

[Wikipedia17a] Wikipedia, “shannon (unit)”, 2017. [https://en.wikipedia.org/wiki/Shannon_\(unit\)](https://en.wikipedia.org/wiki/Shannon_(unit))

[Wiseman17] John Wiseman, “A Quick Tutorial on Generating a Huffman Tree”, SIGGRAPH Education Materials, 2017.
https://www.siggraph.org/education/materials/HyperGraph/video/mpeg/mpegfaq/huffman_tutorial.html

Chapter 7

Classification

We often want to assign each piece of input data to one or more categories that best describe it, such as identifying the person whose face appears in a photo. We'll look at different approaches for meeting this challenge.

Contents

7.1 Why This Chapter Is Here	267
7.2 2D Classification.....	268
7.2.1 2D Binary Classification.....	269
7.3 2D Multi-class classification	275
7.4 Multiclass Binary Categorizing.....	277
7.4.1 One-Versus-Rest	278
7.4.2 One-Versus-One	280
7.5 Clustering.....	286
7.6 The Curse of Dimensionality	290
7.6.1 High Dimensional Weirdness.....	299
References	307

7.1 Why This Chapter Is Here

An important application of machine learning involves looking at a set of inputs, comparing each one to a list of possible categories, and picking the most probable category for that input. This process is called **classification**.

We could use the assigned category, or **class**, to identify the words someone has most probably spoken into their cell phone, what animals are visible in a photograph, or whether a piece of fruit is ripe or not.

Sometimes more than one class or category seems appropriate. For example, we might have a photo of a tiger next to a tree. It seems reasonable to give this photo two categories, “tiger” and “tree.” In this chapter, we’ll assume that there’s always one dominant aspect of each input, so we’ll assign the label corresponding to that aspect. The techniques we’ll see generalize to applying multiple classes to each piece of data, such as both “tiger” *and* “tree,” if we want that.

We’ll train our classifier by giving it data for which we’ve already determined the proper category. We call that category the **actual value**, the **ground truth**, the **expert’s label**, or simply the **label**. This is the class we have manually assigned to our sample, and the class we want the categorizer to assign our data to.

The category, or class, that the categorizer actually does assign to each input is called that input’s **predicted value**.

The goal of training is to get the predicted value to match the label as often as possible, in a way that will generalize to new inputs that the classifier hasn’t seen before.

In this chapter we’ll look at the basic ideas behind classification. We won’t consider specific classification algorithms, which we’ll get to in Chapter 13. Our goal here is just to become familiar with the principles, so later the algorithmic details will make more sense.

We'll also look at **clustering**, which is a way to automatically find useful ways to group together samples that don't have labels.

7.2 2D Classification

Classification is a big topic. Let's get started by simplifying the problem as much as possible.

For now, we'll assume that our input data belongs to only two different categories, or classes. So each input will come with an assigned label, which is one of these two classes. We'll strip off that label and ask the system to make a prediction, or assign its own class to the input. Then we'll compare the label that we started with and the label that the computer came up with, and see if they match.

This stripped-down version of classification lets us explore many of the most important ideas. 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 2D data. That is, every input sample is represented by exactly two numbers. This is just complicated enough to be interesting, but easy to draw, because we can draw each sample as a point on the plane. In practical terms, it means that we'll have a bunch of points, or dots, on the page.

We'll use color and shape coding to show which of the two labels each sample comes in with. Our goal will be to develop an algorithm that can predict those labels accurately. When it can do that, 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.

7.2.1 2D Binary Classification

We'll be working with samples that have two **features**, or two pieces of data, and which belong to two classes. We call this a **2D binary classification** system, where "2D," or "two-dimensional," refers to the two dimensions of the point data, and "binary" refers to the two classes.

The first set of techniques that we'll look at for working out which class should be assigned to each sample in a 2D binary classification 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 an important goal is to identify the simplest boundary shape we can find that still does a good job of splitting the groups.

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 could tell if it's fertilized or not. We'll bundle those values together to make a sample. Then we'll hand the sample to the classifier, which will then assign it either the class "fertilized" or "unfertilized."

Because each egg that we use for training needs a label, or a known correct answer, we'll use a technique called **candling** to decide if an egg is fertilized or not [Nebraska17]. Someone skilled at candling is called a **candler**. This involves holding up the egg in front of a bright light source. Originally that was a candle, but it can be 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 an egg is fertilized or not. Our goal is to get the classifier to give us the same results as the labels we determine by candling.

Before we carry on, let's note that there's no reason to think that measuring an egg's external qualities like weight and shape can accurately tell us if it's fertilized or not. We're just going to pretend that this data can do the job, and then look for ways to use that data to come to the right decisions.

Using our language from above, we want our *categorizer* (the computer) to consider each *sample* (an egg) and use its *features* (weight and length) to assign a *label* (fertilized or unfertilized).

Our problem is called a **binary classification** problem, because we have only two categories.

Let's start with a bunch of **training data** that gives the weight and length of each egg. We'll plot this data on a grid with weight on one axis and length on the other. Fertilized eggs will be shown with a red circle, and unfertilized eggs with a blue box. Figure 7.1 shows our starting data.

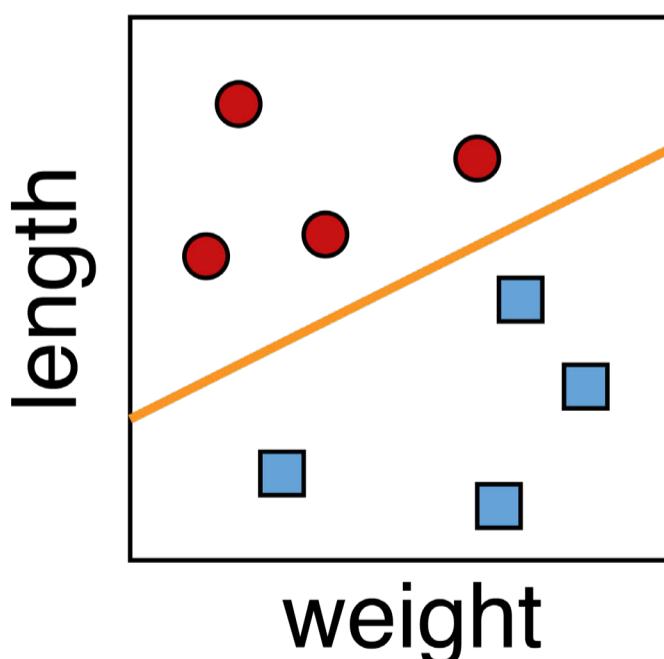


Figure 7.1: Categorizing eggs. The red circles are fertilized eggs. The blue squares are unfertilized eggs. Each egg is plotted as a point given by its two dimensions of weight and length. The orange line separates the two clusters.

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.

We're done with our classifier! When we get new eggs in the future (without manually-assigned labels), we can just look to see which side of the line they fall on. The eggs on the "fertilized" side get assigned the category fertilized, and the same for the unfertilized side. Figure 7.2 shows the idea.

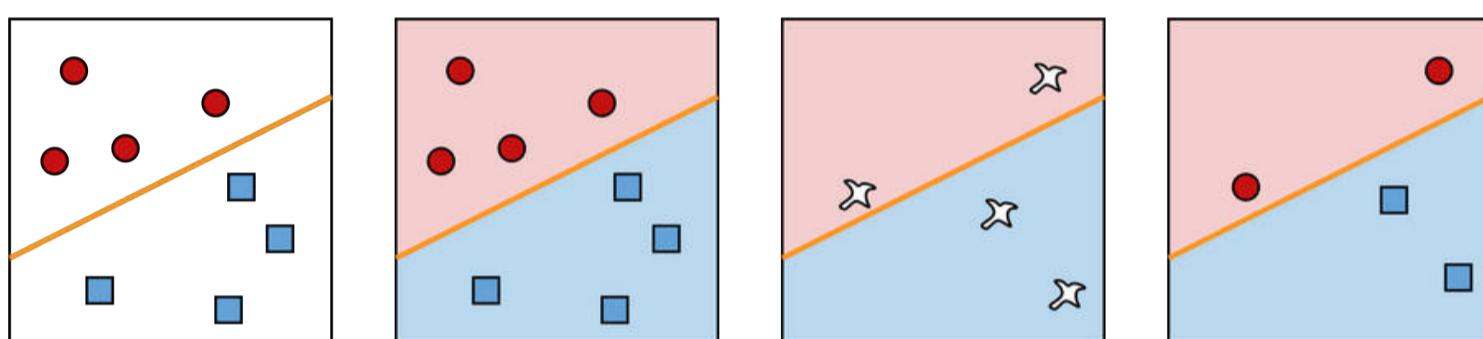


Figure 7.2: Categorizing new eggs. Far left: Our starting data. Second from left: The red and blue regions show how we've decided to split the fertilized and unfertilized eggs. Third from left: Four new eggs arrive to be categorized. Far right: We've assigned a category to each of the new eggs.

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 eggs manually to determine if they're fertilized or not, and then we plot the results as before. Figure 7.3 shows our new data.

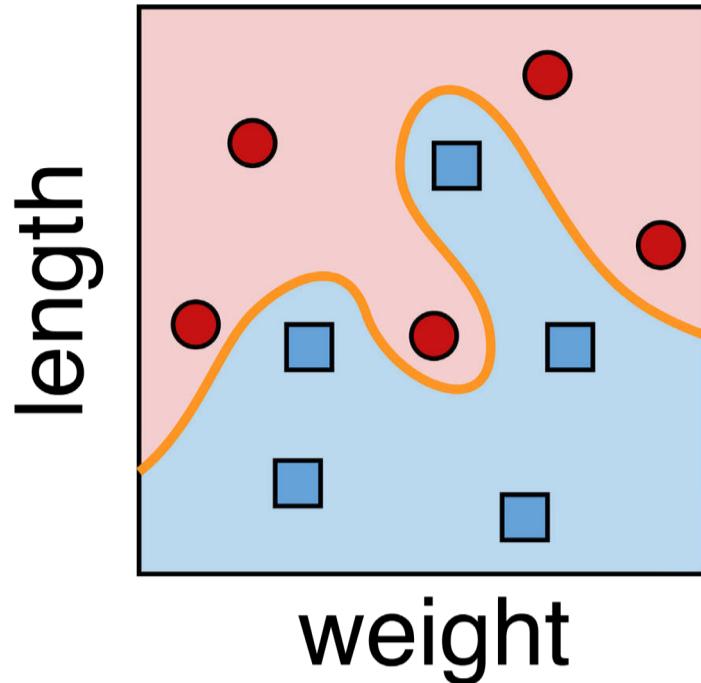


Figure 7.3: When we add some new types of chickens to our flock, it may get trickier to determine which eggs are fertilized based just on the weight and length.

The two groups are still distinct, which is great, but now they're separated by a squiggly curve, rather than a straight line.

But that's okay, because we can use the curve just like the straight line from before. Each new egg gets placed on this diagram, and if it's in the red zone it's fertilized, and if it's in the blue zone it's unfertilized.

When we can split things up this nicely, we call the sections into which we chop up the plane **decision regions**, 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 yet a different variety. As before, we manually candle a bunch of eggs and plot the data, this time getting the diagram of Figure 7.4.

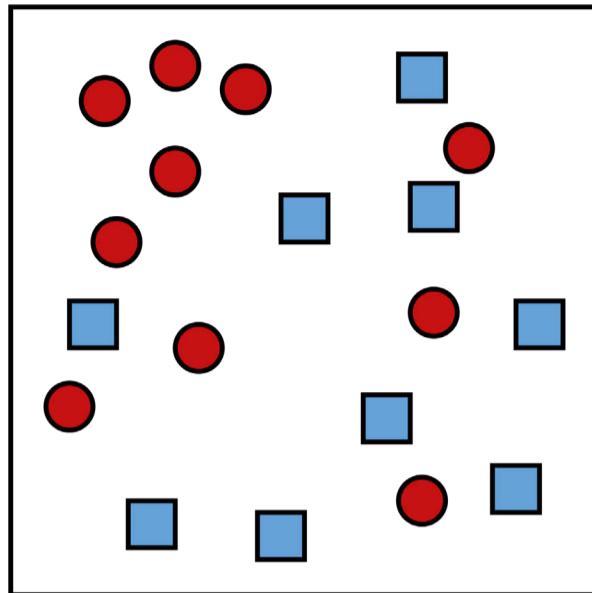


Figure 7.4: Our new purchase of chickens has made it much harder to distinguish the fertilized eggs from the unfertilized eggs.

There's still a mostly-red region and a mostly-blue region, but there's no clear way to draw a line or curve that separates them.

So a better way to talk about which sample is in which category is to consider **probabilities** rather than certainties.

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.

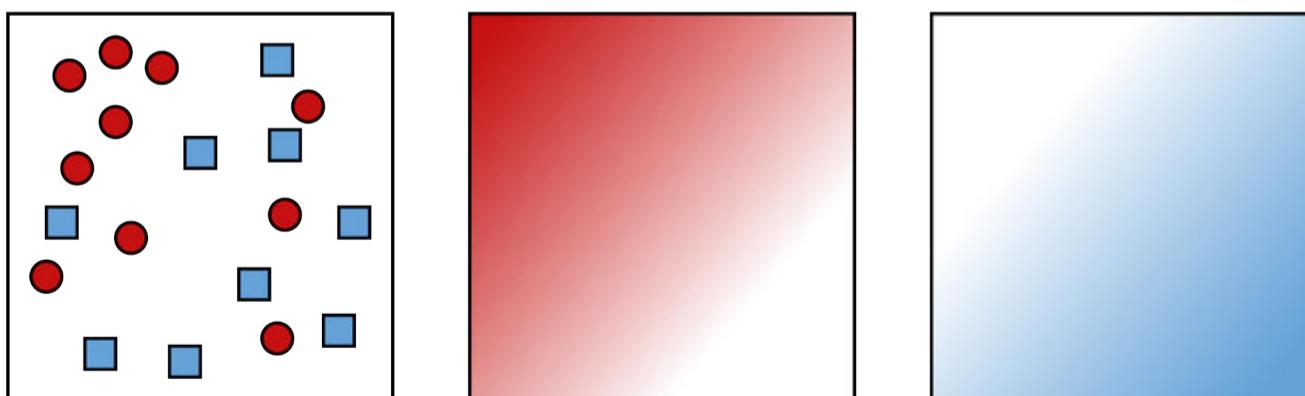


Figure 7.5: Given the overlapping results shown at the far left, we can give every point in our grid a probability of being fertilized, shown in the center where brighter red means the egg is more likely to be unfertilized. The image at the far right shows a probability for the egg being unfertilized.

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.

So there's going to be some ambiguity going forward. 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.

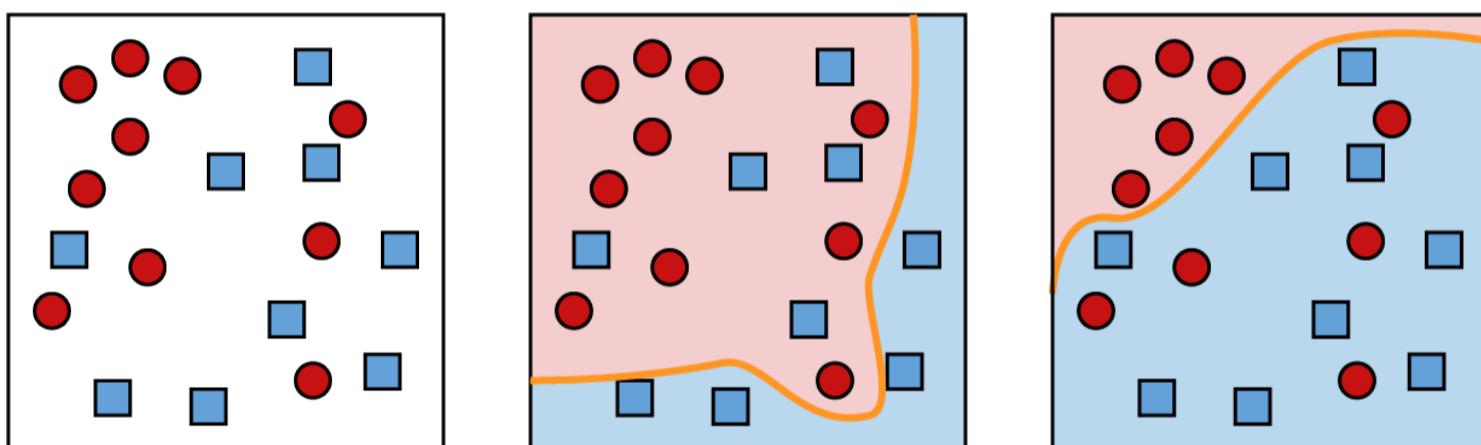


Figure 7.6: Given the results at the far left, we may choose a policy shown in the center, which accepts some false positives (unfertilized eggs classified as fertilized) to be sure we correctly classify all fertilized eggs. Or we may prefer to correctly classify all unfertilized eggs, with the policy shown at the right.

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.

In either case, the fuzziness of the overlap doesn't change our basic principle. Although we may use probabilities for each category during analysis, we ultimately have to declare each egg as either "fertilized" or "unfertilized" to decide how to handle it.

So we create a boundary that splits the data into two pieces, and each new egg gets classified based on which side of the boundary it lands in.

When the probabilities are fuzzy there's no right or wrong, since our decision on where to place the boundary is based not just on the data and an algorithm, but also on human and business factors.

7.3 2D Multi-class 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 can be unfertilized. An egg that is unfertilized because it was never fertilized is called a **yolker**. But in some fertilized eggs the developing embryo stopped growing for some reason and died. Such an egg is called a **quitter** [Arcuri16].

We can sell the yolkers, but the quitters can burst unexpectedly and spread harmful bacteria. So we want to identify the quitters and dispose of them.

Now we have three categories of egg: **winners** (viable, fertilized eggs), **yolkers** (unfertilized eggs), and **quitters** (dead fertilized eggs). As before, we'll hypothesize 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 categories we manually assigned to them by candling the eggs.

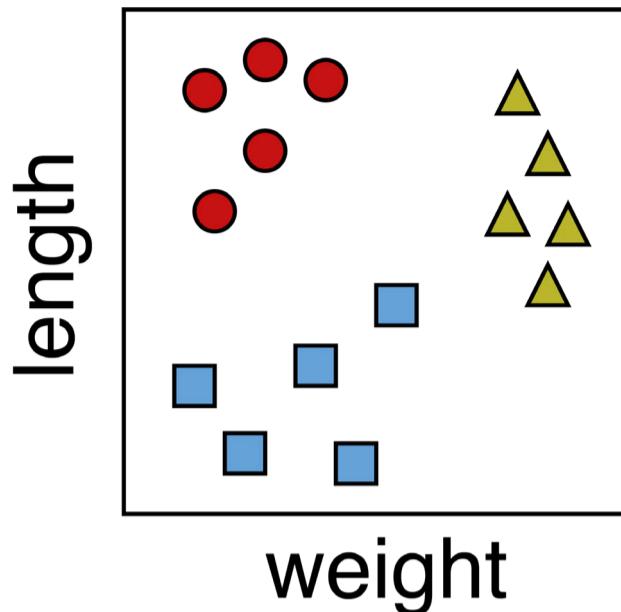


Figure 7.7: Three classes of eggs. The red circles are fertilized. The blue squares are unfertilized but edible yolkers. The yellow triangles are quitters that we want to remove from our incubators.

The job of assigning one of these three classes to new inputs is called **multi-class classification**. This name is usually applied to problems where we have three or more classes.

The ideas generalize from here to any number of categories or labels. Once we've found the boundaries between the regions associated with different classes, any new sample can be categorized just by seeing which boundary region it falls into.

We've been using 2D data (each egg is described by a weight and length), but we can also generalize our process to any number of features, or dimensions. So in addition to weight and volume, we might add other measurements like color and average circumference, and the time of day the egg was laid. That 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 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 will go up as well. Sometimes we'll want to modify our algorithms to work more efficiently in high dimensions.

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'll discuss this a little more near the end of the chapter). The problem with multiple dimensions is that we can't picture them well. But again, reasoning by analogy, we can pass some kind of a dividing shape between different groups of points. 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.

When we later receive the 5 measured values for a new egg, we can just determine which of these smaller 5D spaces it lands in, and that will tell us what class we'll predict for that point.

7.4 Multiclass Binary Categorizing

Perhaps surprisingly, we can do multiclass categorizing with nothing more than a bunch of binary classifiers. This is sometimes an efficient approach when we want to perform multiclass categorizing.

Here are two popular methods for using binary classifiers to do multiclass categorization.

7.4.1 One-Versus-Rest

Our first technique goes by several names: **one-versus-rest** (or **OvR**), **one-versus-all** (or **OvA**), **one-against-all** (or **OAA**), or the **binary relevance method**.

Let's suppose that we have 5 categories for our data. We'll name our categories, or classes, with the letters A through E.

Instead of building one classifier that will assign one of these 5 labels, we'll instead build 5 binary classifiers. Let's number these classifiers 1 through 5.

Categorizer 1 will tell us whether a given piece of data does or does not belong to class A. Because it's a binary classifier, it's concerned with only this one class, and it doesn't pay any attention to the other classes. In other words, this binary categorizer has one decision boundary that splits the space into two regions: class A and everything else. Every piece of data that comes into the classifier gets assigned one of the two regions, which correspond to "class A" or "not class A."

We can now see where the name "one-versus-rest" comes from. In this categorizer, class A is the "one," and classes B through E are the "rest."

Our second categorizer, named Categorizer 2, is another binary classifier. This time it tells us whether a sample is, or is not, in class B. In the same way, Categorizer 3 tells us whether a sample is or is not in class C, and Categorizers 4 and 5 do the same thing for classes D and E.

Figure 7.8 summarizes the idea. Here we used a library routine that takes into account all of the data when it builds the boundary for each classifier.

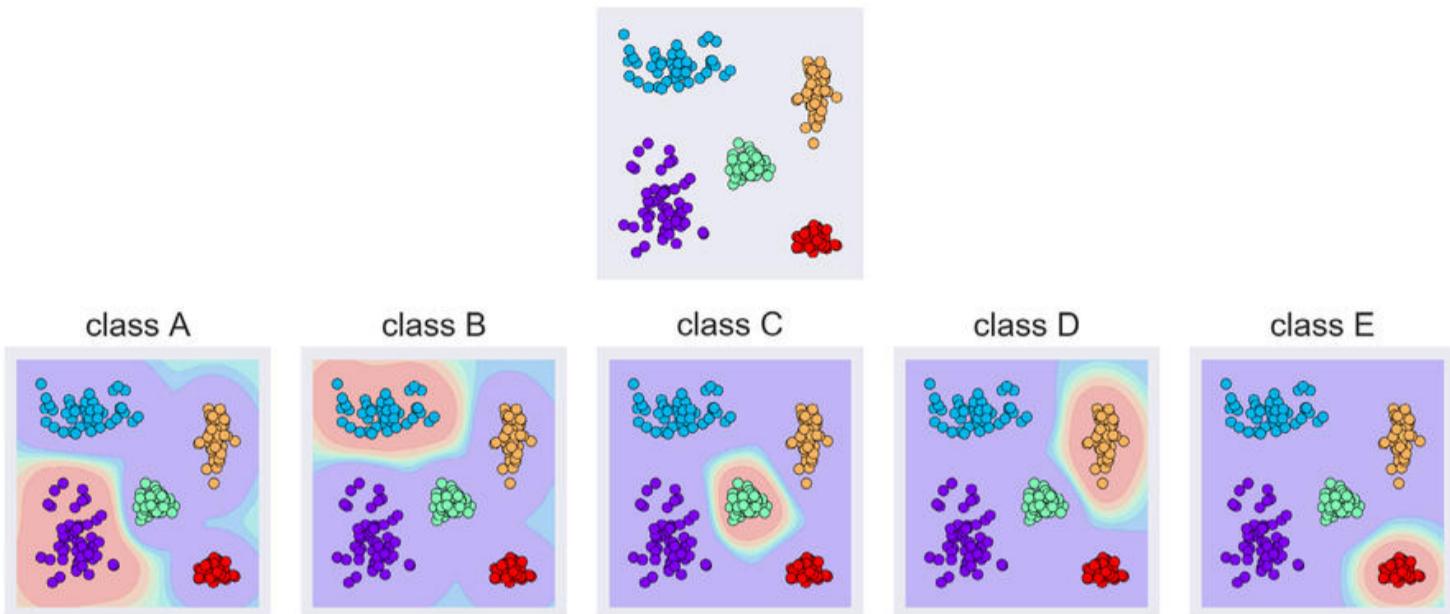


Figure 7.8: One-versus-rest classification. Top row: Samples from 5 different categories. Bottom row: The decision regions for 5 different binary classifiers. Each classifier is concerned only with finding a boundary between a single class and all other points. The colors from blue to red show increasing probability that a point belongs to that class.

Note that some locations can belong to more than one class. For example, points in the upper-right have non-zero probabilities from classes A, B, and D.

To categorize an example, we run it through each of our 5 binary categorizers 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.

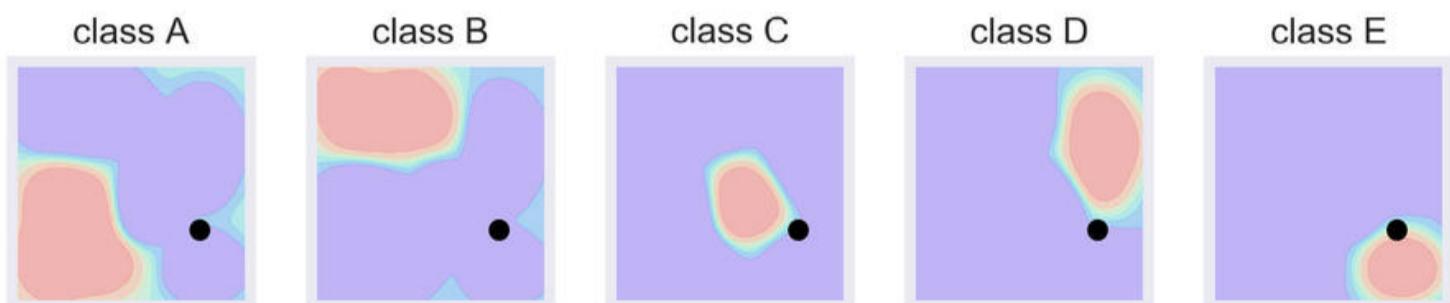


Figure 7.9: Categorizing a sample using one-versus-rest. The new sample is the black dot. The first four categorizers all report that the point is not in their class, so they return low probabilities. The categorizer for class E finds the point is in its class's region, and assigns it a higher probability than the others, so the point is predicted to be from class E.

The appeal of this approach is its conceptual simplicity, and its speed. Binary classifiers can be written to run very quickly. The downside is that we have to teach (that is, find boundaries for) 5 classifiers instead of just one, and we have to classify our example 5 times to find which category it belongs to.

When we have large numbers of categories with complex boundaries, the time required to run the sample through lots of binary classifiers can add up. As the collection of classifiers gets larger, and thus slower, it may be more efficient to switch to a single complex multi-class categorizer.

7.4.2 One-Versus-One

Our second approach for using binary classifiers to perform multi-class optimization is called **one-versus-one** (or **OvO**), and it uses even more binary classifiers than one-versus-rest.

The general idea is to look at each pair of classes in our data, and build a classifier for just those two classes.

Because the number of possible 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. So to keep things manageable, we'll work with only 4 classes this time, as shown in Figure 7.10.

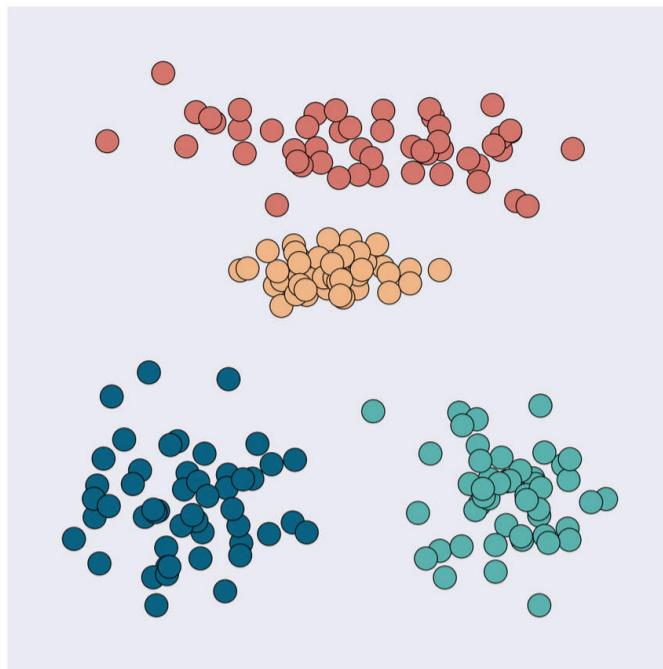


Figure 7.10: Four classes of points for demonstrating one-versus-one classification.

We'll 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 didn't even exist. This classifier finds a boundary curve that separates the classes A and B. Now every time we feed a new sample to this classifier, it will tell us whether it belongs to class A or B.

Since these are the only two options available to this classifier, it will classify 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.

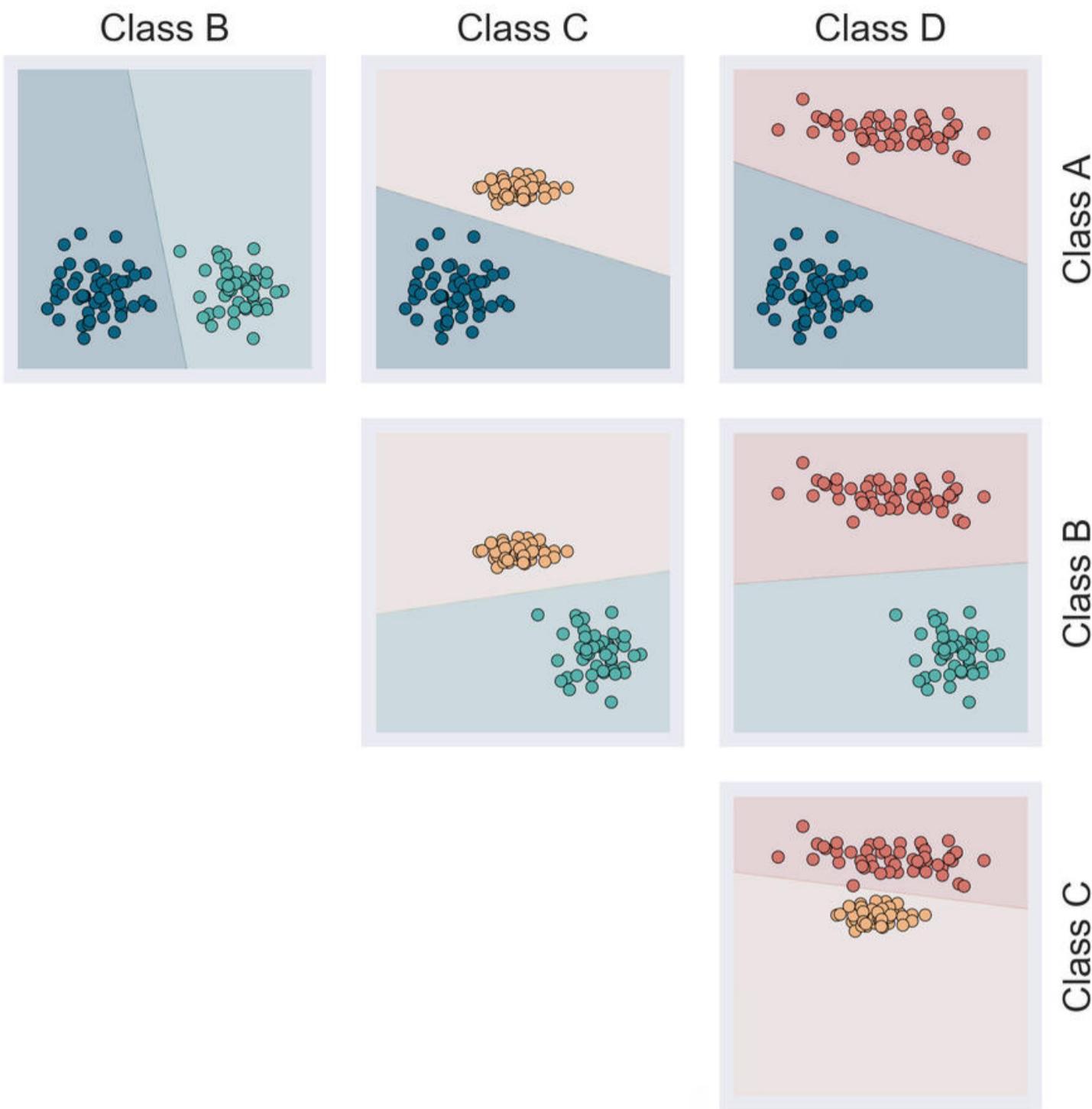


Figure 7.11: Building the 6 binary classifiers we use for performing one-versus-one classification on 4 classes. Top row: Left to right, binary classifiers for classes A and B, classes A and C, and classes A and D. Second row: Left to right, binary classifiers for classes B and C, and B and D. Bottom row: A binary classifier for classes C and D.

The top row of Figure 7.11 contains 3 classifiers, one each for classes A and B, classes A and C, and classes A and D.

Next, we build a binary classifier that is trained with data *only* from classes B and C, which will always return either B or C depending on which category the data falls into the “best.” We make another classifier for classes B and D.

The last remaining pair of classes are C and D.

The result is that we have 6 binary classifiers, each of which tells us which of two specific classes the data fits best.

To classify a new example, we always run it through *all* the 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.

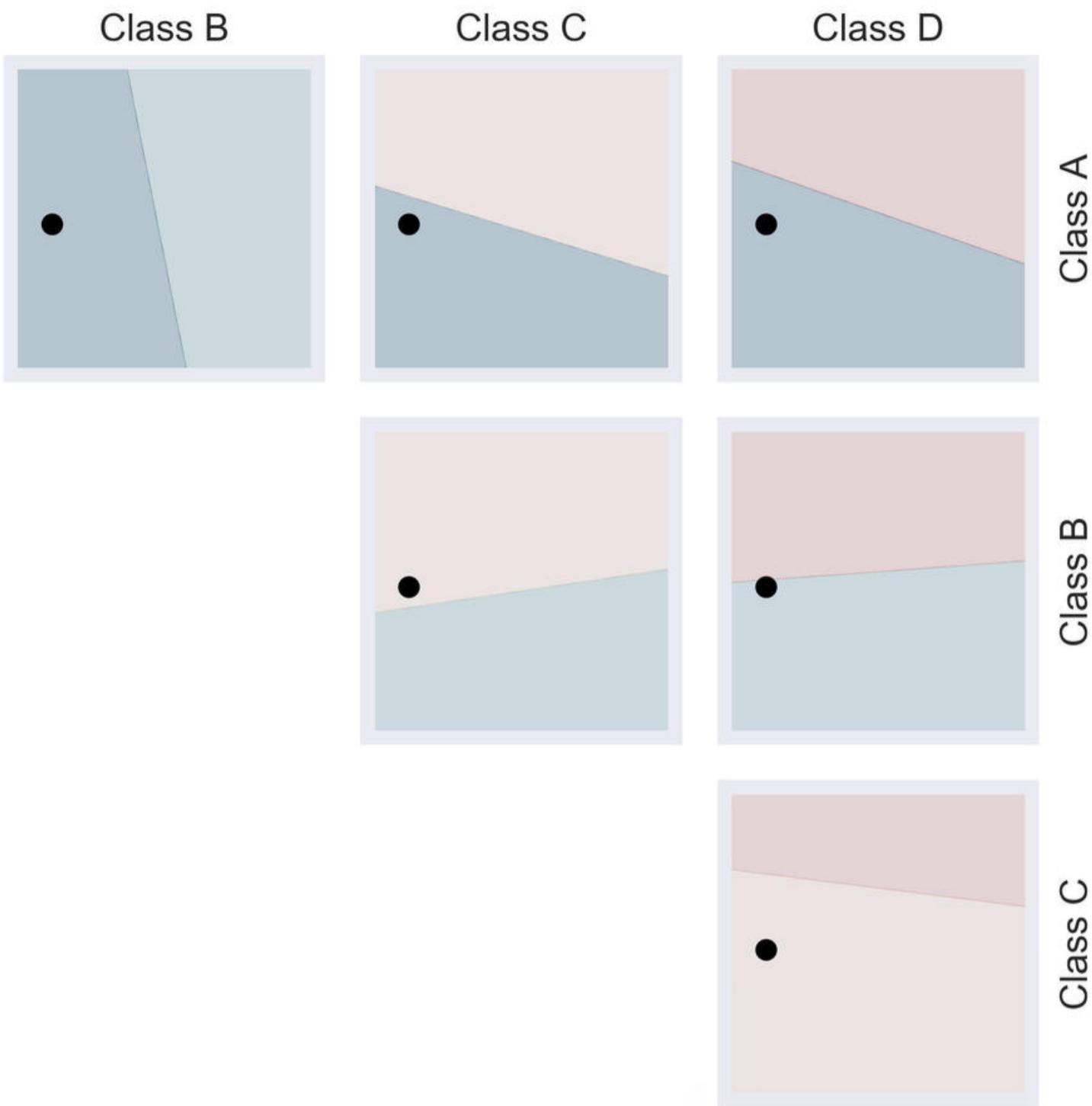


Figure 7.12: One-versus-one in action, classifying a new sample in black. Top row: These three binary classifiers each report that the sample is in class A, compared to B, C, and D. Middle row: These binary classifiers report the sample is in class C and class D, reading left to right, because the sample is not in class B. Bottom row: The sample is not in class C, so this binary classifier reports that it's in class D. Class A wins with the most votes.

One-versus-one requires many more classifiers than one-versus-rest, but it is sometimes appealing because it provides a deeper analysis of each sample with respect to all combinations of classes. 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 goes up [Wikipedia17]. Figure 7.12 gives us a clue as to how many there will be. For any number of classes, we can draw a square grid with that many classes horizontally and vertically, and then fill in the upper-right part of that grid with diagrams, one for each classifier. That means the number of classifiers we need is a little more than half of the number of classes times itself.

We've seen that with 4 classes we'd need 6 classifiers. 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 1000. Figure 7.13 shows just how dramatically the number of binary classifiers we need grows with the number of classes.

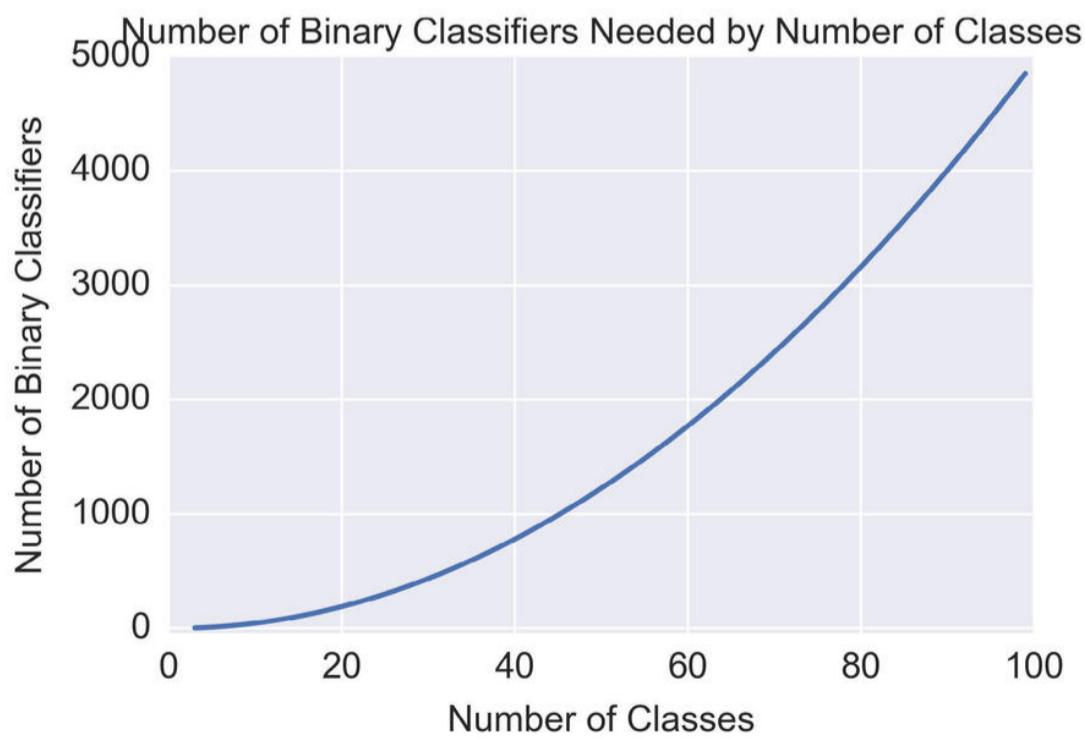


Figure 7.13: As we increase the number of classes, the number of binary classifiers we need for one-versus-one grows very quickly.

Each of these binary classifiers will have to be trained, and then we'll 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 will become more efficient to use a single complex multi-class categorizer.

7.5 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.

If the data has labels, as it does above, then we can use the labels to guide our creation of the clusters.

In the left image of Figure 7.14 we have data with 5 different labels. We can make clusters just by drawing a curve around each set of points, as in the middle image. If we extend those curves outwards 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.

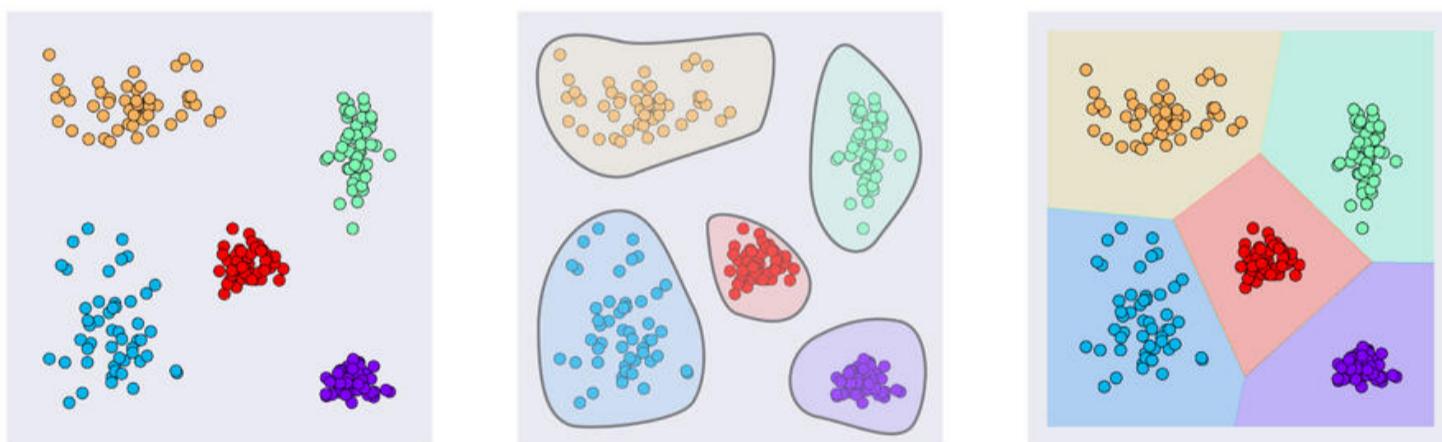


Figure 7.14: Growing clusters. Left: Starting data with 5 categories. Middle: Identifying the 5 groups. Right: Growing the groups outward so that every point has been assigned to one class.

This scheme required that our input data had labels. What if we don't have labels? We might have nothing but an unlabeled collection of points. If we could automatically break it up into clusters, we could apply the technique we just described. It's lacking in subtlety and options, but it's a place to start if we have nothing but un-ordered, unlabeled points.

Recall that problems that involve data without labels fall into the category 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 . 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 means , or averages, of groups of points to develop their clusters, it's 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 will produce what we want.

Keep in mind that the computer doesn't know where we think the cluster boundaries ought to go, so although it will chop things up into k pieces, they might not be the pieces we're expecting. But if our data is well separated, so there are lots of examples clumped together with big spaces between the clumps, we'll 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 up front is that we might not have any idea of how many clusters would best describe our data. If we pick too few clusters, then we won't separate our data into the most useful distinct classes. But if we pick too many clusters, then we'll end up with very 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 5 groups.

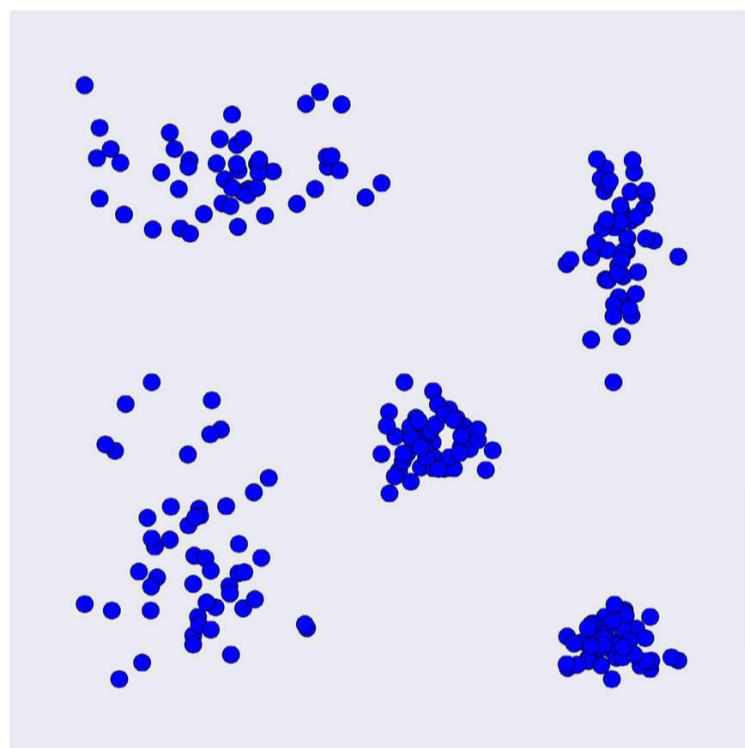


Figure 7.15: A set of 200 unlabeled points. They seem to visually fall into 5 groups.

Figure 7.16 shows how one clustering algorithm 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.

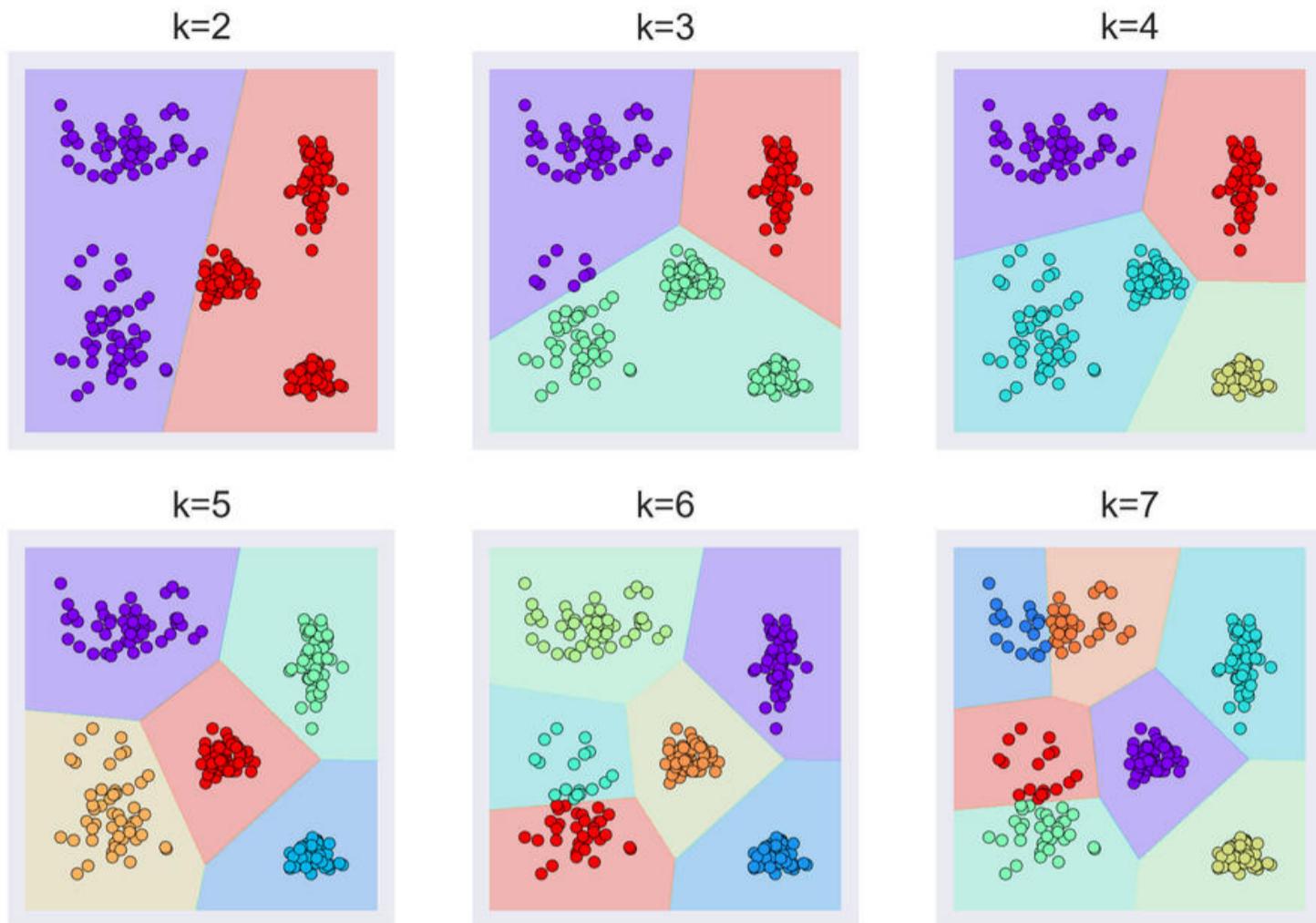


Figure 7.16: The result of automatically clustering our data in Figure 7.15 for values of k from 2 to 7. Unsurprisingly, $k=5$ produces the best results, but we deliberately made this data easy to separate. With more complex data, we might not know the best value of k to specify.

It's no surprise that $k=5$ is doing the best job on this data, but we're using a cooked example where the boundaries were easy to see. With more complicated data, particularly if it has more than 2 or 3 dimensions, it could be all but impossible for us to easily identify the right number of clusters just by looking at the raw data.

All is not lost, though. A popular option is to train our network several times, each time using a different value for k . 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 time. Perhaps a lot of time. Testing out 20 different values of k will take at least 20 times longer than if we knew the right value to start with.

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.

7.6 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 can 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. The first few features we add might give us better results, but after a certain point including more features will cause the system to degrade, and it will start to perform worse and worse.

So by describing our samples with more features, or dimensions, we caused the system's ability to assign the correct class to decline.

This counter-intuitive idea shows up so frequently that it has earned a special name: **the curse of dimensionality** [Bellman57]. This phrase has come to mean different things in different fields. We're using it in the sense that applies to machine learning [Hughes68].

One way to appreciate why this happens is to think about how the 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 divide them. In order to pick the one 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.

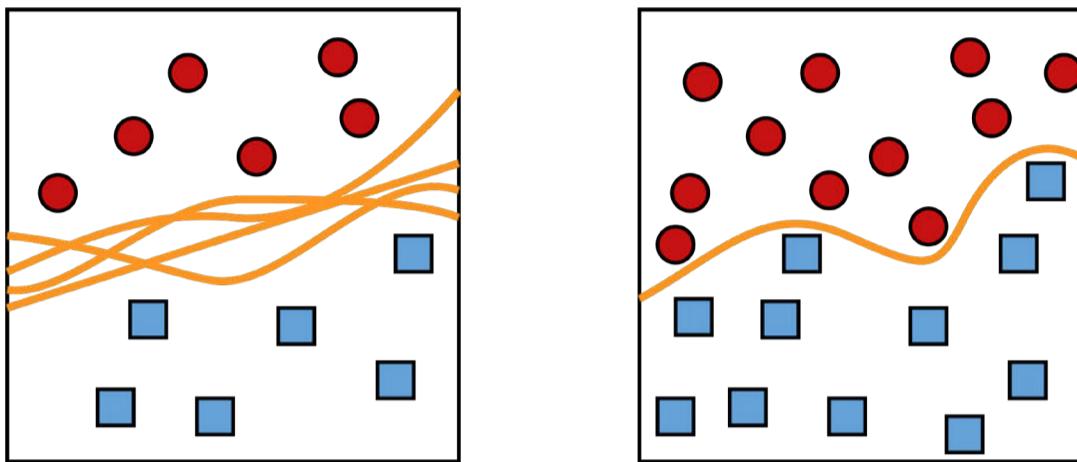


Figure 7.17: To find the best boundary curve we need a good density of samples. Left: We have very few samples, so we can construct lots of different boundary curves. We can't tell which of these (if any) would do the best job with future data. Right: With a higher density of samples, we can find a good boundary curve.

As Figure 7.17 shows, finding a good curve requires a dense collection of points. But as we add dimensions (or features) to our samples, the number of samples we need to maintain a reasonable density in the sample space explodes. If we can't keep up with the demand, the classifier will do its best, but it won't have enough information to draw a good boundary. It will be stuck in the situation of the left diagram in Figure 7.17, guessing at the best boundary.

Let's look at this dependence on the number of features, or dimensions, using our egg example. To keep things simple, we'll say that all the features we might measure for our eggs (their volume, length, etc.) will lie in the range 0-1.

We'll 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'll draw 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. Figure 7.18 shows how our data falls on an interval $[0,1]$ with 5 bins.



Figure 7.18: Our 10 pieces of data have one dimension each. We can draw them on a line. Here we've broken the line up into 5 equal pieces so we can get a feeling for how well the points cover the length of the line.

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 would be unchanged if we picked 300 eggs or 1700 bins.

The *density* of our space is the number of samples divided by the number of bins. In this case, it's $10/5$ or 2, telling us each bin will (on average) have 2 samples. Looking at Figure 7.18 we see that this is a pretty fair estimate for the average number of samples per bin.

Let's add length to our list of dimensions, or features, for each egg. Because we now have two dimensions, we'll pull our line segment of Figure 7.18 upwards to make a square, as in Figure 7.19.

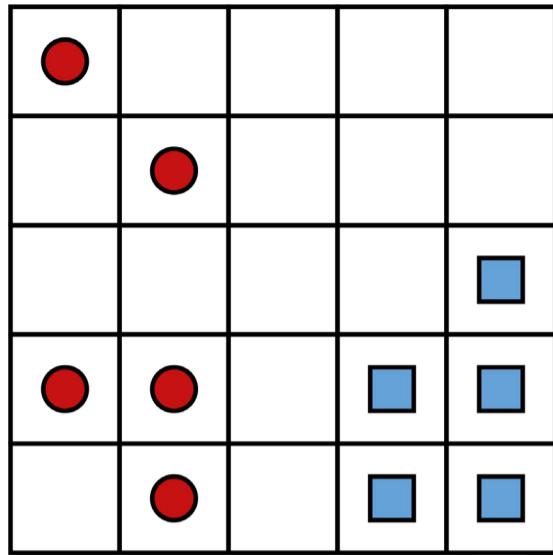


Figure 7.19: Our 10 samples are now each described by two measurements, or dimensions. We've drawn them on a 2D grid. As with Figure 7.18, we've split each side of the grid into 5 equal pieces, giving us 25 smaller squares, or bins. The average density is now 10 samples divided by 25 bins, or 0.4.

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 some of the regions won't have any data. The density now is $10/(5 \times 5) = 10/25 = 0.4$. We can draw lots of different boundary curves to split the data of Figure 7.19.

Now we'll 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'll push our square back into the page to form a cube, as in Figure 7.20.

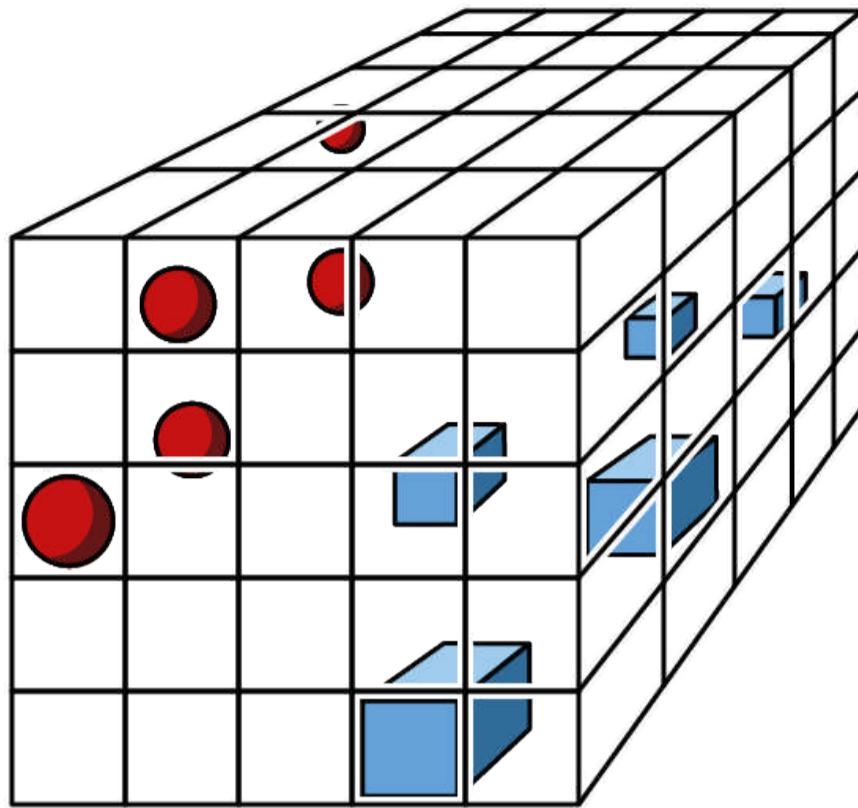


Figure 7.20: Now our 10 pieces of data are represented by 3 measurements, so we draw them in a 3D space. With 5 divisions on a side, we have $5 \times 5 \times 5 = 125$ little cubes. The density is now $10/125$ or 0.08 samples per cube.

Again splitting each axis into 5 pieces, we now have 125 little cubes, but we still have only 10 boxes. Our density has dropped to $10/(5 \times 5 \times 5) = 10/125 = 0.08$. In other words, there's only an 8% chance that any little cube has a sample in it.

Any classifier that splits this cube 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 not clear *how* to separate the data. 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 what samples are going to end up in those empty boxes 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.

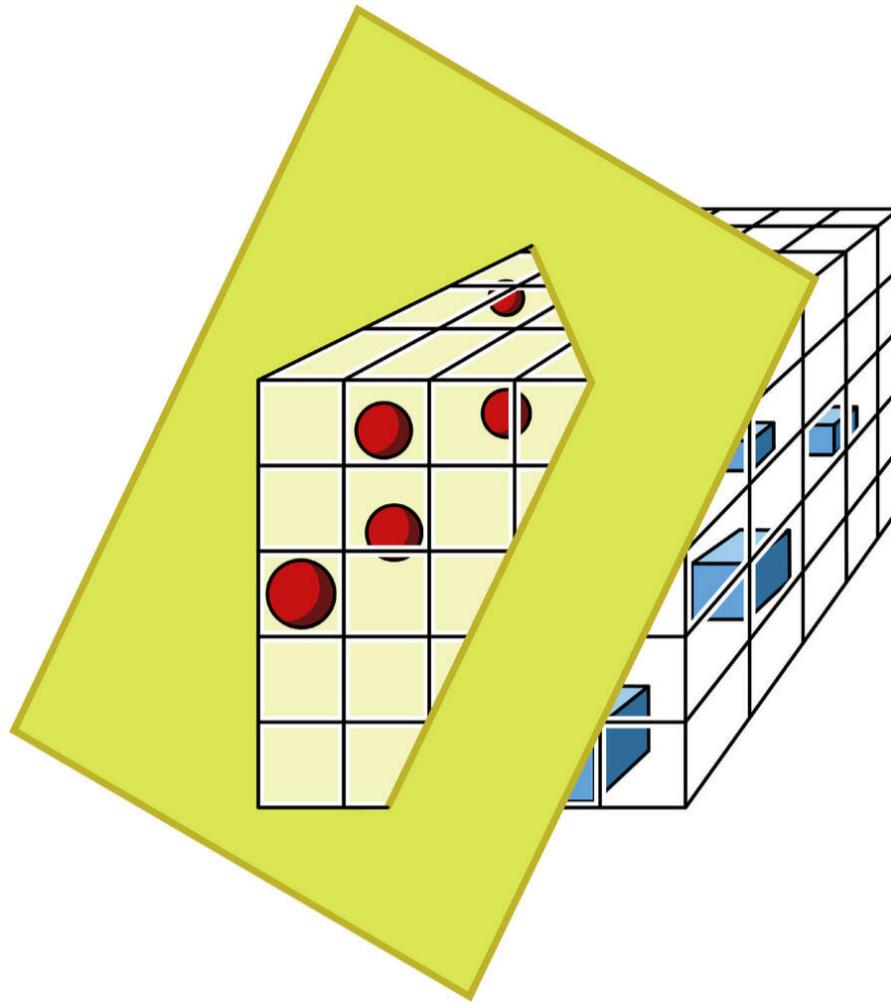


Figure 7.21: Passing a boundary surface through our cube, separating the red and blue samples. Whether or not this surface will do a good job on new samples is anyone's guess.

The expected low quality of this boundary surface is not the fault of the classifier, since it's doing the best it can with the data it has. The problem is that 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.

We can't draw pictures for spaces with more than 3 dimensions, but we can calculate their densities. Figure 7.22 shows a plot of the density of our space for our 10 samples as the number of dimensions goes up. Each curve corresponds to a different number of subdivisions for each axis. Notice that as the number of dimensions goes up, the density drops to 0 no matter how many bins we use.

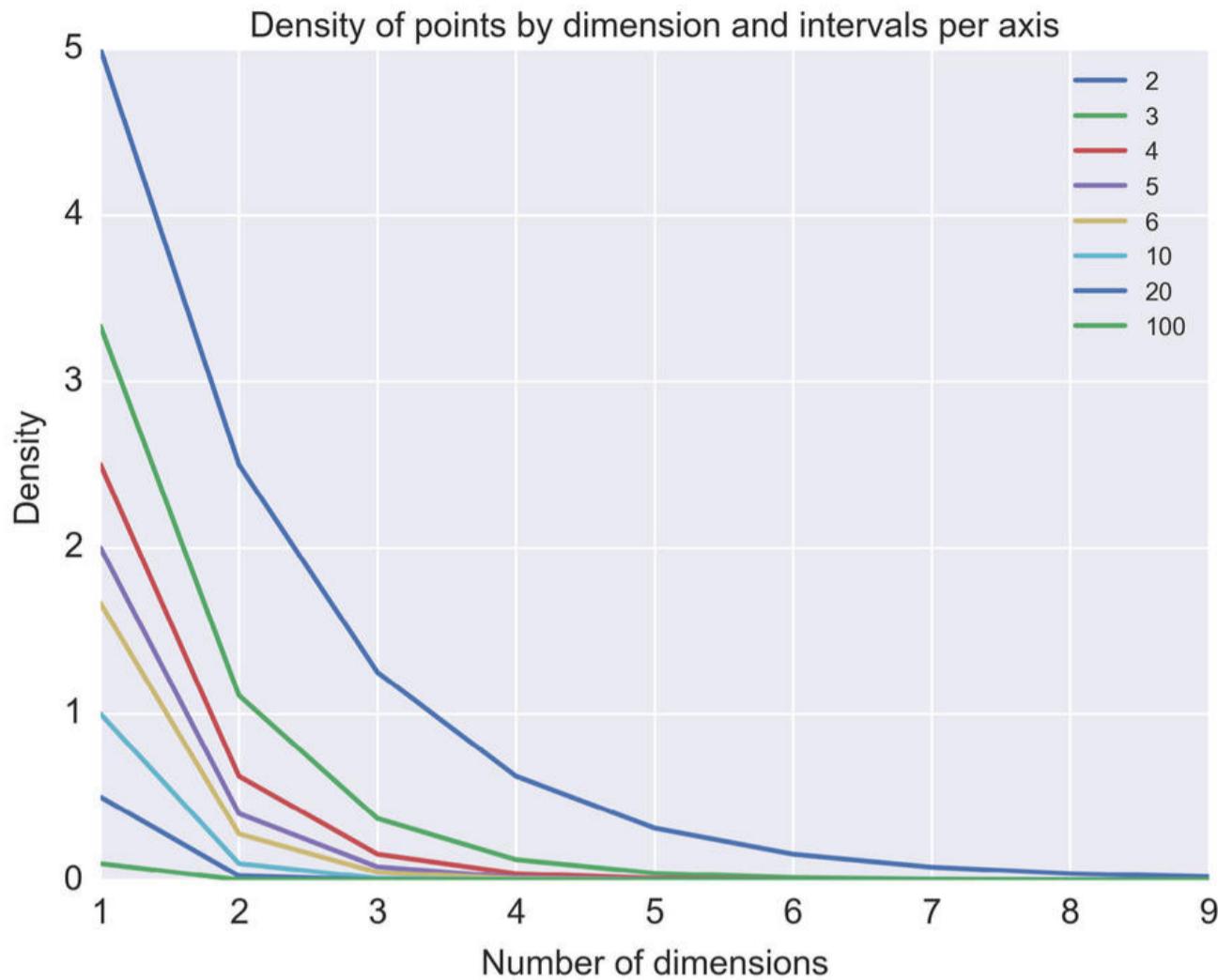


Figure 7.22: This is how the density of our points drops off as the number of dimensions increases, for a fixed number of samples. Each of the colored curves shows a different number of bins along each axis.

If we have fewer bins (that is, a smaller number for the curve in Figure 7.22), we have a better chance of filling them, but we can see that before long the number of bins becomes irrelevant. As the number of dimensions goes up, our density heads to 0.

This means our classifier will end up guessing wildly.

So adding more measurements to our eggs means adding more dimensions to our samples, and that's going to make it less and less likely that our predictor will find a boundary that's going to work well with new data.

The curse of dimensionality is a serious problem, and it could have made all attempts at machine learning fruitless. What saved the day is the **blessing of non-uniformity** [Domingos12], which we prefer to think of as the **blessing of structure**.

In practice, the features that we typically measure, even in very high-dimensional spaces, are not 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 very well.

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 other data will generally fall into the same regions. The upshot is that the density in those regions where the samples are mostly likely to fall will often be high enough that the system can either make a good guess at the boundary surface, or the exact location of that surface won't matter much and any boundary surface could be expected to work well.

For example, in our cube of Figure 7.20, instead of having the samples spread around more or less uniformly throughout the cube, the samples from each class might be located on the same horizontal plane. That means that any roughly horizontal boundary surface that splits the groups will probably do 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.

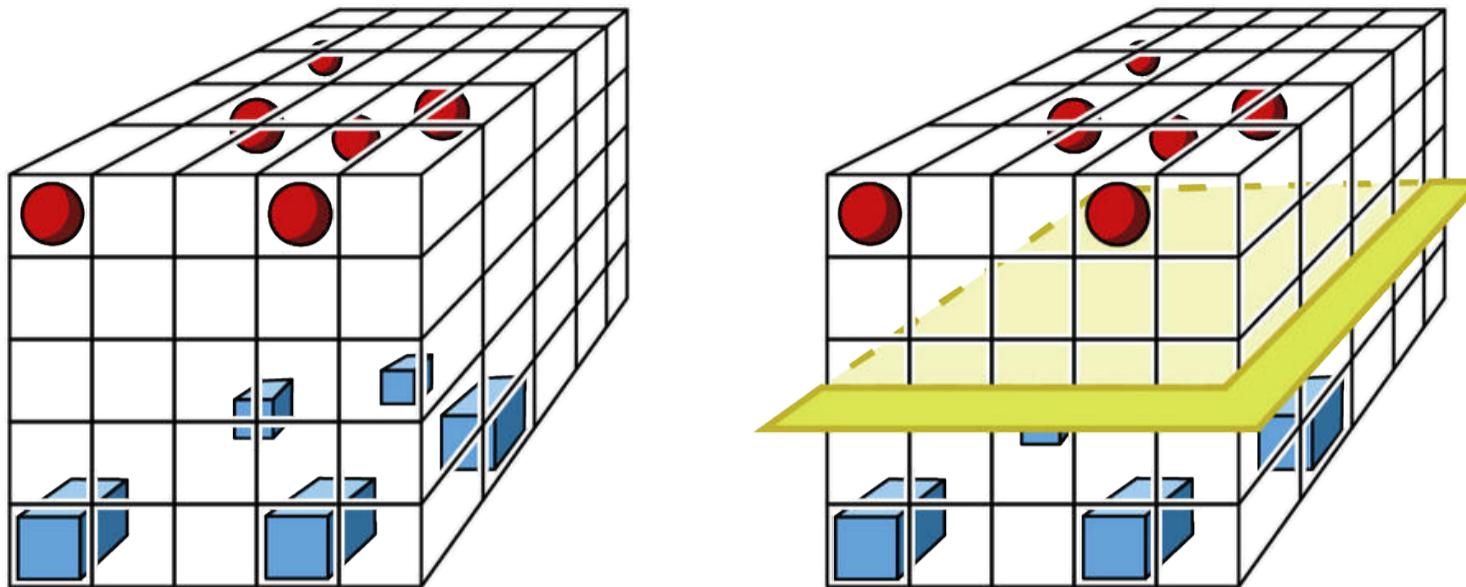


Figure 7.23: In practice, our data often has some structure in the sample space. Left: Each group of samples is mostly in the same horizontal slice of the cube. Right: A boundary plane passed between the two sets of points.

Although most of the cube in Figure 7.23 is empty, and thus has low density, the parts we’re interested in have a high density, and we can find a sensible boundary.

So although the curse of dimensionality dooms us to having a low density in our space, even with enormous amounts of data, the blessing of structure says that we will usually get reasonably high density where we need it. Note that this “curse” and “blessing” are both empirical observations, and not hard facts we can always rely on. Even 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.

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 measurements (or dimensions), we need lots and lots of samples to get enough density to make good boundary surfaces.

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 a hurry.

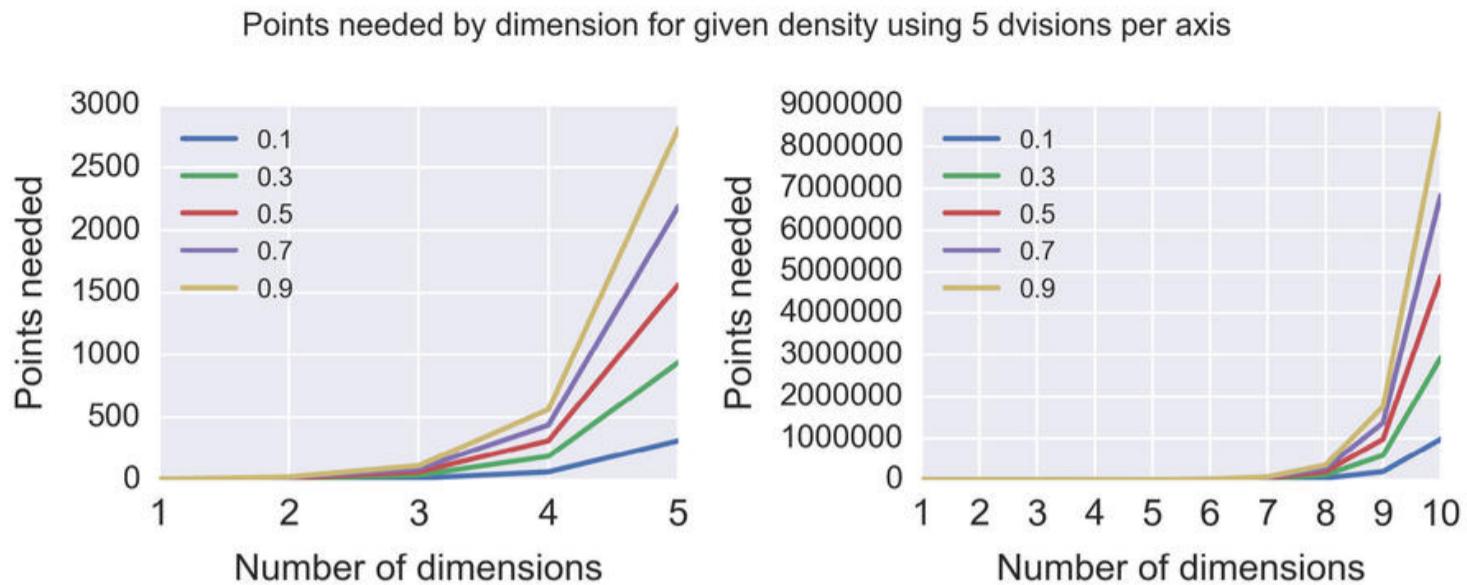


Figure 7.24: The number of points we need to achieve different densities, assuming we have 5 divisions on each axis. Notice how the values explode in size. Left: Results for dimensions 1 through 5. Right: Results for dimensions 1 through 10. To get a density of 0.5 in 5 dimensions requires about 1500 samples, but in 10 dimensions we need about 5 million samples.

Speaking generally, if the number of dimensions is low, and we have lots of points, then we'll 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.

7.6.1 High Dimensional Weirdness

Before we leave our discussion of high-dimensional spaces, it's interesting to note that these spaces often confound our intuition. For instance, suppose we have a collection of uniformly-distributed points in a high-dimensional space, and then we remove some of the points. That would cause the density to go down. We then let the points move around as necessary so that they're roughly uniformly distributed in the space again. We'd probably expect that as the density decreases, and the remaining points are farther away from each other, the distance between points would grow at a roughly equal rate. That turns out not to happen.

Let's pick some fixed number of points, and place them uniformly along a line. For every point, we'll find the distance from it to the nearest other point, and we'll find the average of all of those distances. Now place the same number of points on a 2D grid, and again find the average distance from any point to its nearest neighbor. Repeat this in a 3D space, then a 4D space, and so on. As we mentioned, we'd probably expect that as this fixed number of points is distributed in spaces of higher and higher dimensionality, the average distance between any point and its nearest neighbor would increase quickly.

Figure 7.25 shows the average distance between any two points, using the same number of points placed in spaces of different dimensions.

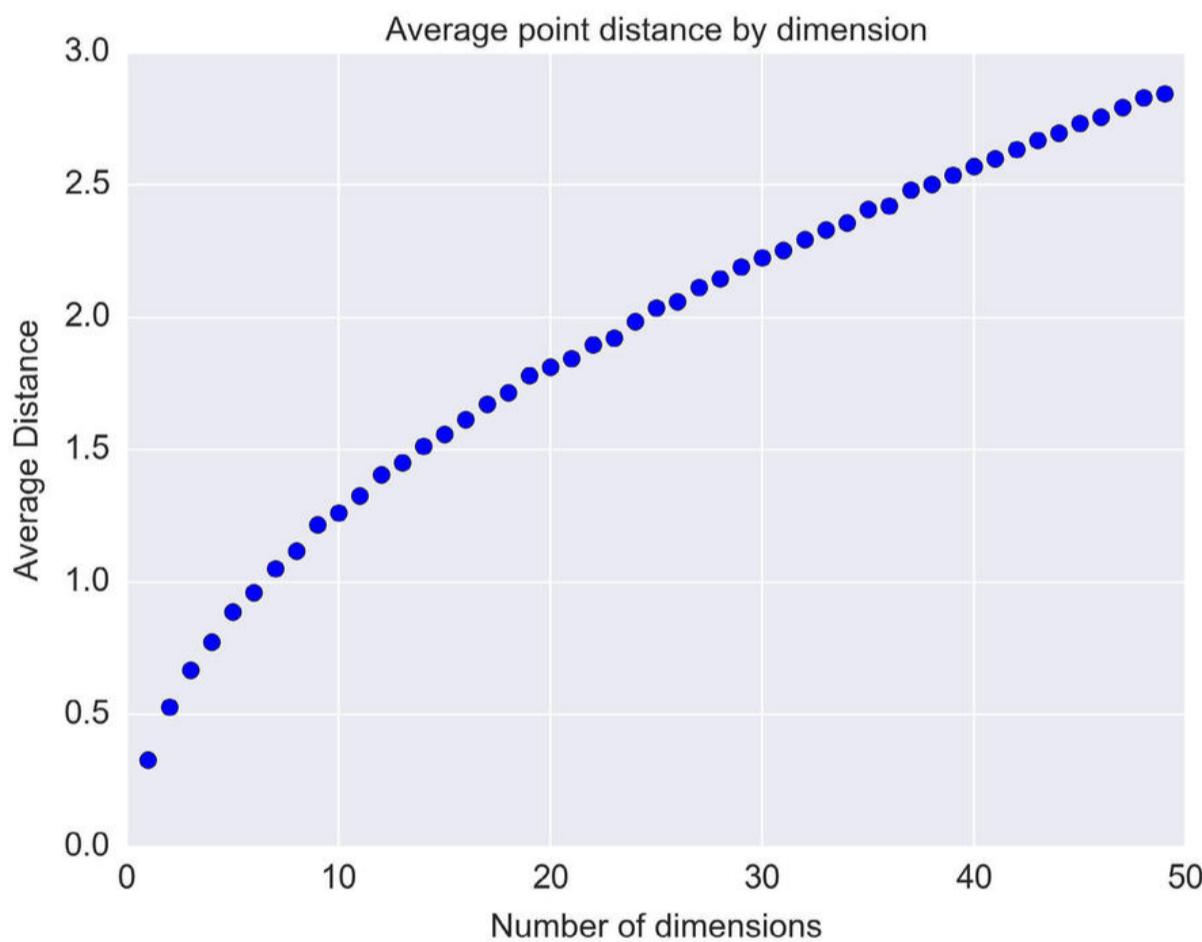


Figure 7.25: Our intuition can fool us when we get into higher dimensions. For each dimension, we uniformly scattered a fixed number of points through the space, and found the average distance between each point and its nearest neighbor.

The distance between points does grow, showing that the points are spreading apart, but the growth curve is very shallow.

Another famous example involves the volume of a sphere inside a box [Spruyt14]. Let's think about the volume of a sphere packed into a box in different dimensions, as shown in Figure 7.26, and consider how much of the box is occupied by the sphere.

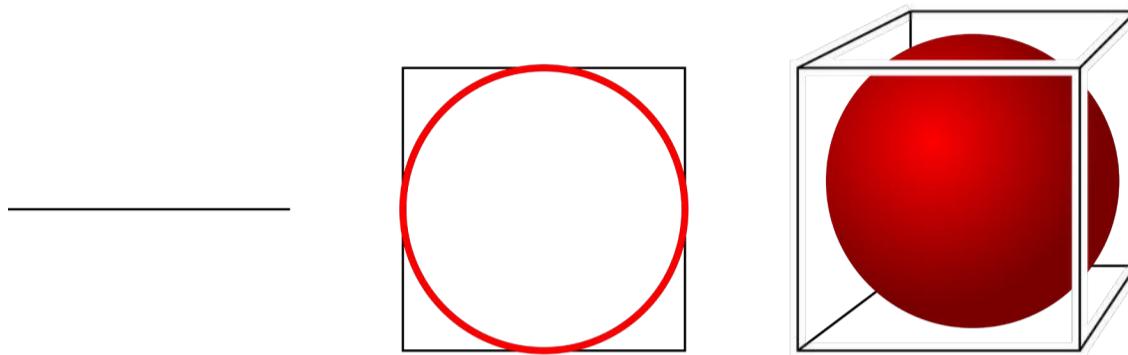


Figure 7.26: Spheres in boxes. (a) A 1D “box” is a line segment, and the “sphere” covers it completely. (b) A 2D box with a circle. (c) A 3D box with a sphere.

In one dimensions our “box” 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 “box” is 1.

In 2D, our box 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 box is a 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 box for higher dimensions, we get the curve in Figure 7.27.

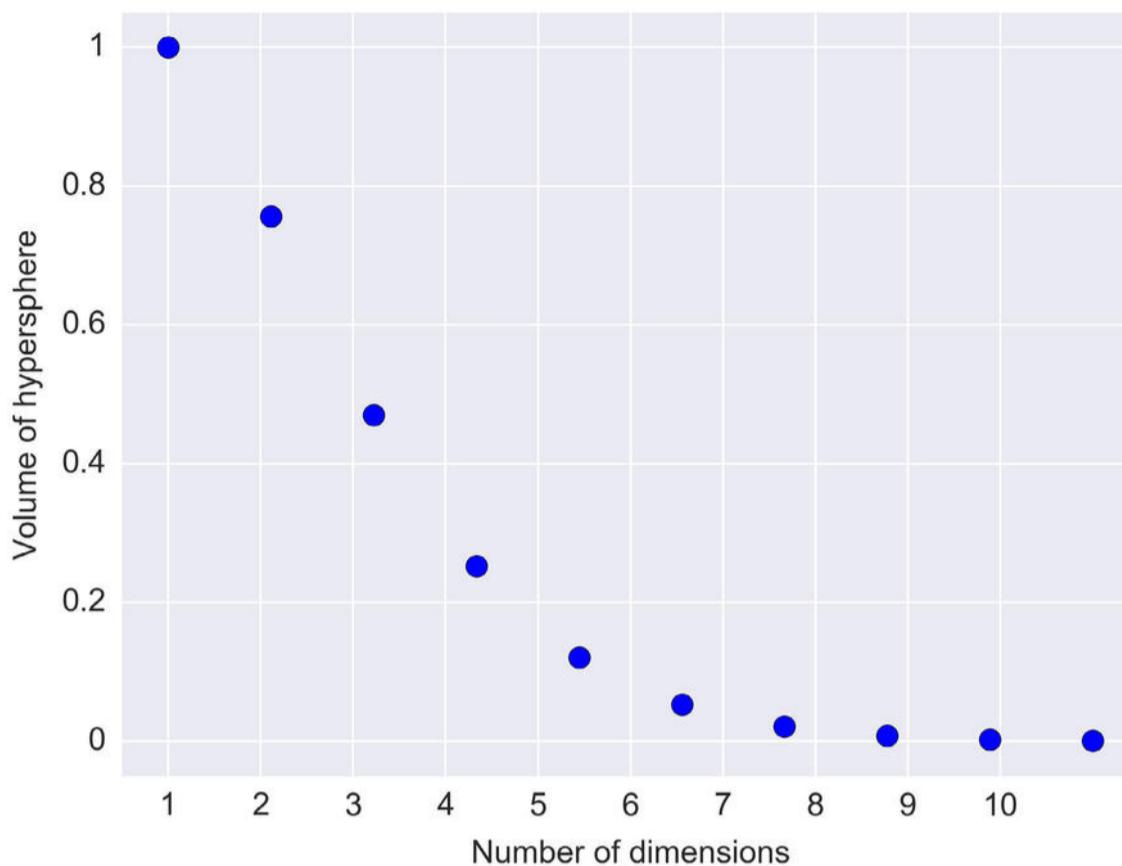


Figure 7.27: The amount of a box occupied by the largest sphere that fits into that box for different numbers of dimensions.

The amount of volume taken up by the sphere drops down towards 0. By the time we reach 10 dimensions, the largest sphere we can fit into enclosing box takes up almost none of the box's volume!

There's no trick to this, and nothing's going wrong. When we work out the math, this is what happens. Higher dimensions are weird. We need to keep this weirdness in mind and not rely on our intuition when we think about learning systems that work in high dimensions.

This result is important when we apply classification to data with lots of features, because every feature corresponds to one dimension. For example, if our samples have 7 features, then our classifier is implicitly operating in a 7-dimensional space. Our classifiers depend on having enough data with enough structure so that they can make good decisions about where the boundary surfaces should go. As the number of dimensions goes up, our classifiers can start to fail if those conditions

aren't met. Making sure that we have "enough" data that has "enough" density can be hard when we're in higher dimensions, because our intuition often fails us there.

There are more weirdnesses in the geometry of 4 or more dimensions. Let's look at a couple more of these surprises so we can get a good appreciation of how risky it is to generalize from 2D and 3D into higher dimensions.

As the number of dimensions climbs, we find that more and more of the points belonging to the sphere are located near its surface, and not inside [Carpenter17]. If we think of the sphere as an orange, it becomes all skin and no pulp.

Here's another weirdness. Let's suppose we'd like 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'll put one balloon in each corner of the box, so that they all just touch each other. What's the biggest orange we can put into a box of a given size?

We want to answer this question for boxes (and balloons and oranges) with any number of dimensions, so let's start with just two dimensions. We'll also assume that our balloons and oranges are all perfectly round.

To make the 2D version of our orange shipment, we'll draw a 2D square of size 4 by 4, and we'll put a circle of radius 1 in each corner, as in Figure 7.28(a). These circles will be our 2D balloons.

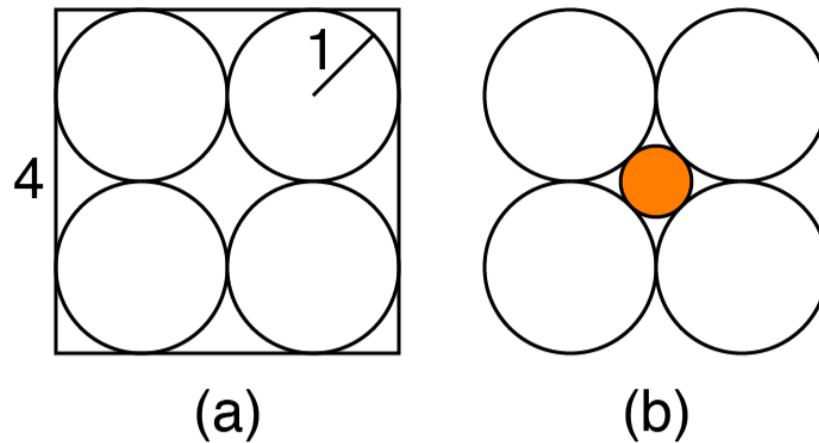


Figure 7.28: Shipping a circular 2D orange in a square box, surrounded by circular balloons in each corner. (a) The four balloons each have a radius of 1, so they fit perfectly into a square box of side 4. (b) Our orange fits in the middle of the box, surrounded by the balloons. The radius of the orange is about 0.4.

Our orange in this 2D diagram is also a circle. In Figure 7.28(b) we show the biggest orange we can fit. A little geometry tells us that the radius of this circle is about 0.4.

Now let's get back to our original 3D problem. We'll add depth to our 4 by 4 square by lifting it up out of the plane, creating a cube that's 4 by 4 by 4. We can now fit 8 spheres, each of radius 1, into the corners, as in Figure 7.29(a).

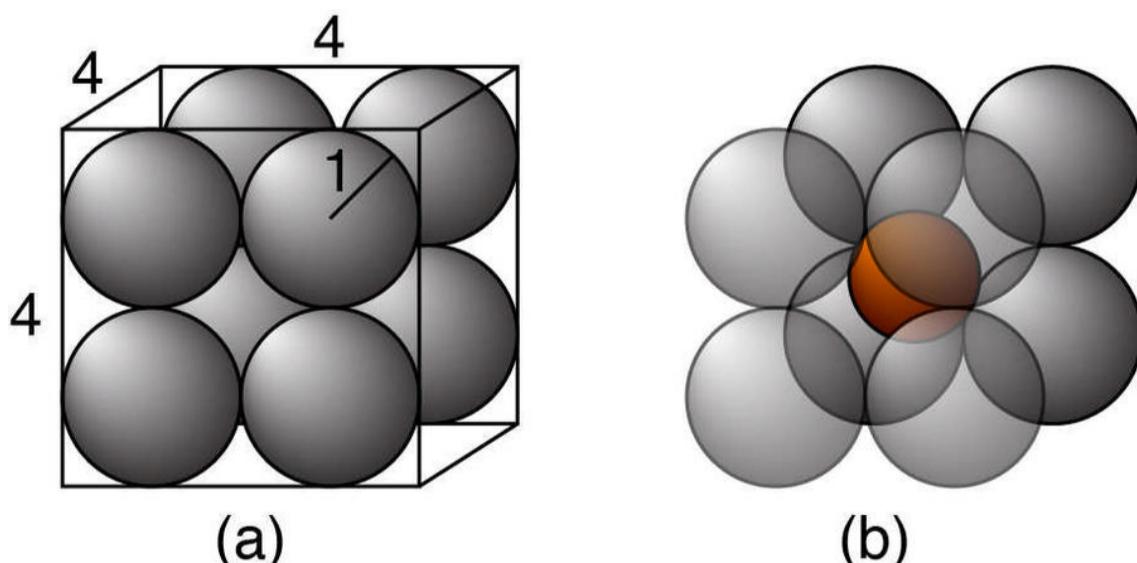


Figure 7.29: Shipping a spherical orange in a cubical box surrounded by spherical balloons in each corner. (a) The box is 4 by 4 by 4, and the 8 balloons each have a radius of 1. (b) Our orange fits in the middle of the box, surrounded by the balloons. The radius of the orange is about 0.7.

Figure 7.29(b) shows the orange we can fit into the gap in the middle. Another bit of geometry tells us that this sphere has a radius of about 0.7. This is a little 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.

When we talk of cubes and spheres of dimensions greater than 3, we call them **hypercubes** and **hyperspheres**. By analogy, we can call our higher-dimensional orange a **hyperorange**. For any number of dimensions, our hypercubes will always be 4 units on every side, there will always be a hypersphere balloon in every corner of the hypercube, and these balloons will 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 [Numberphile17]. Figure 7.30 plots this radius for different numbers of dimensions.

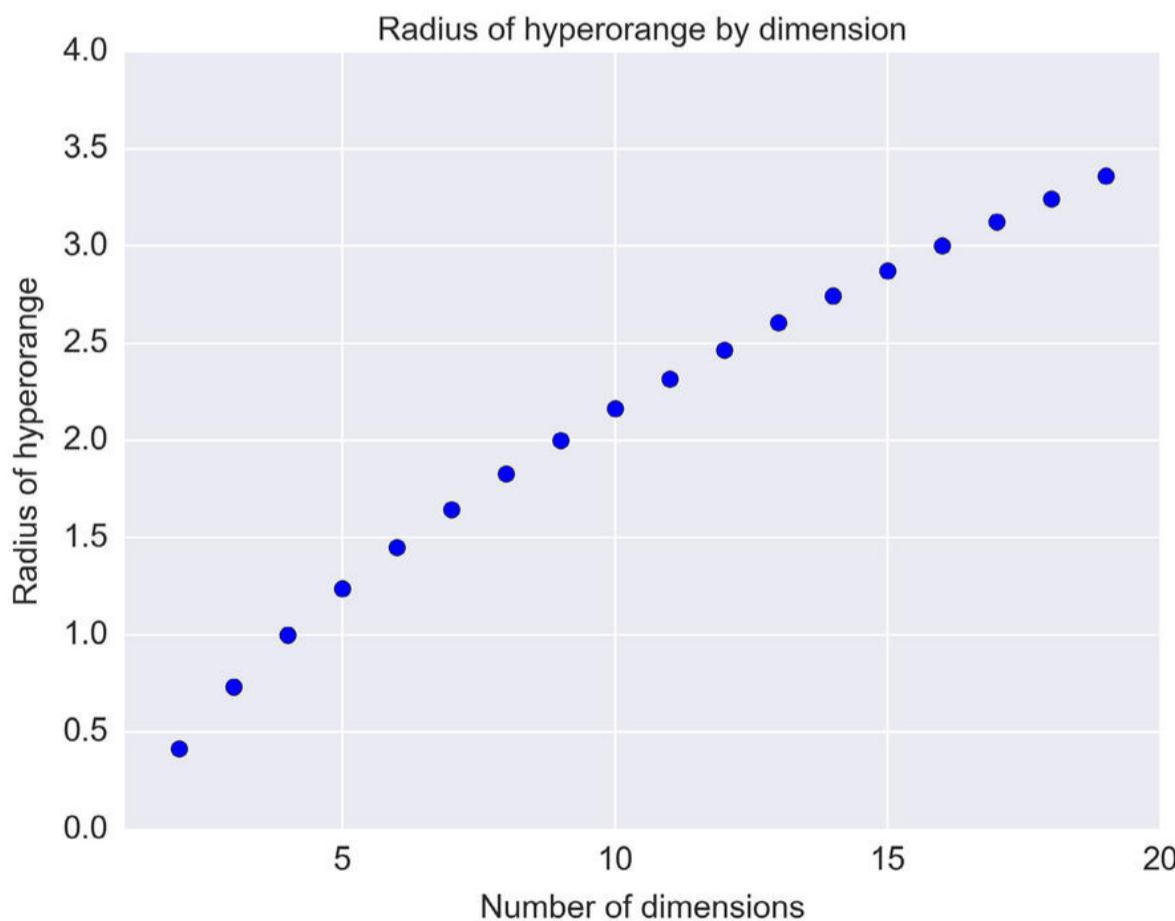


Figure 7.30: The radius of a hyperorange in a hypercube with sides 4, surrounded by hyperspheres of radius 1 in each corner of the cube.

We can see from Figure 7.30 that in 4 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 stranger.

Figure 7.30 tells us that in 9 directions, the hyperorange has a radius of 2, so its diameter is 4. That means the hyperorange is as large as the box itself. That's despite being surrounded by 512 hyperspheres of radius 1, each in one of the 512 corners of this 9-dimensional hypercube.

But things get even crazier. At 10 dimensions and higher, the hyperorange has a radius that's *more* than 2. The hyperorange is now *bigger* than the hyperbox that was meant to protect it, even though there's still a protective balloon in every corner.

How can this be? It's hard to get a good intuitive picture for what's going on. A popular explanation is that the hyperorange is a "spiky sphere," somehow (though nobody's quite sure just how) managing to send out spikes that fit between the hyperspheres around it [Strohmer17]. Figure 7.31 shows the idea conceptually.

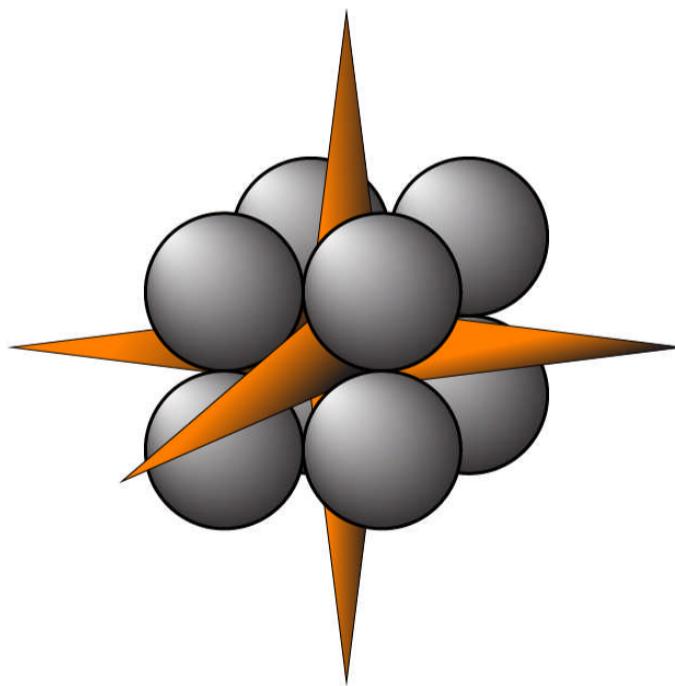


Figure 7.31: A hyperorange of more than 10 dimensions has a radius that's greater than 2. One way to visualize this is to imagine a "spiky sphere," though it's not very much like the spheres we're used to.

Though this is a common way to visualize how the hyperorange might be so big, these spikes seem difficult to reconcile with the idea that the shape is any kind of “sphere” as we usually understand it. It might be easier to think of the spiky sphere as something like a water balloon that’s being squeezed in a fist, with pieces of the balloon oozing out between each pair of fingers. This still doesn’t feel terribly spherical, but at least there aren’t any spikes.

The moral here is that our intuition can fail us when we get into spaces of many dimensions [Aggarwal01]. 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.

References

[Aggarwal01] Charu C. Aggarwal, Alexander Hinneburg, and Daniel A. Keim, “On the Surprising Behavior of Distance Metrics in High Dimensional Space”, ICDT 2001. <https://bib.dbvis.de/uploadedFiles/155.pdf>

[Arcuri16] Lauren Arcuri, “Definition of Candling - How to Candle an Egg,” The Spruce Blog, 2016. <https://www.thespruce.com/definition-of-candling-3016955>

[Bellman57] Richard Ernest Bellman, “Dynamic Programming”, Princeton University Press, 1957 (republished 2003 by Courier Dover Publications)

[Carpenter17] Bob Carpenter, “Typical Sets and the Curse of Dimensionality,” Stan blog, 2017. <http://mc-stan.org/users/documentation/case-studies/curse-dims.html>

[Domingos12] Pedro Domingos, “A Few Useful Things to Know About Machine Learning”, Communications of the ACM, Volume 55 Issue 10, October 2012. <https://homes.cs.washington.edu/~pedrod/papers/cacm12.pdf>

[Hughes68] G.F. Hughes, “On the mean accuracy of statistical pattern recognizers”, IEEE Transactions on Information Theory. 14 (1): 55–63, 1968

[Nebraska17] Nebraska Extension, “Candling Eggs”, Nebraska Extension in Lancaster County, University of Nebraska-Lincoln, 2017. <http://lancaster.unl.edu/4h/embryology/candling.shtml>

[Numberphile17] Numberphile, “Strange Spheres in Higher Dimensions,” YouTube, 2017. https://www.youtube.com/watch?v=mceaM2_zQd8

[Spruyt14] Vincent Spruyt, “The Curse of Dimensionality”, “Computer Vision for Dummies” blog post, 2014. <http://www.visiondummy.com/2014/04/curse-dimensionality-affect-classification/>

[Strohmer17] Thomas Strohmer, “Surprises in High Dimensions”, Mathematical Algorithms for Artificial Intelligence and Big Data Analysis, UC Davis 180B Lecture Notes, 2017. <https://www.math.ucdavis.edu/~strohmer/courses/180BigData/180lecture1.pdf>

[Wikipedia17] Wikipedia authors, “Combination”, 2017. <https://en.wikipedia.org/wiki/Combination>

Chapter 8

Training and Testing

How to train our system to learn from data, then measure how well it's learned, and how well we expect it to perform on new data that it's never seen before.

Contents

8.1 Why This Chapter Is Here	311
8.2 Training	312
8.2.1 Testing the Performance	314
8.3 Test Data.....	318
8.4 Validation Data	323
8.5 Cross-Validation	328
8.5.1 k-Fold Cross-Validation.....	331
8.6 Using the Results of Testing	334
References	335
Image Credits.....	336

8.1 Why This Chapter Is Here

The point of building a learning system is to extract meaning from data. Sometimes we want to understand a dataset that we already have, and other times we want to understand data that is yet to come.

In this chapter we'll look at **training**. This is the process of taking a system that's been initialized with default or random values, and reconfiguring it so that it's tuned to the data we want to understand.

Efficient training requires a way to evaluate how well the system is learning. We call that **validating** the system. We validate the system by showing it new data it hasn't seen before, and measure how accurately it is able to extract meaning from it.

We'll look at a validation tool called **cross-validation**, which guides our training to help us squeeze every bit of learning we can get from our training data. This is particularly useful when our dataset is small, and we can't easily collect more samples.

If the predictions on the validating data are good enough for our application, we can start using the system for real. If they're not good enough, then we'll go back and train it some more.

We'll illustrate the ideas in this chapter using a supervised categorizer, which we'll teach with labeled data. In particular, we'll use a neural network type of classifier, which learns by repeated exposure to the training data. Most of the techniques we'll discuss are general, and can be applied to almost all types of learners.

8.2 Training

When we do supervised learning with a categorizer, every sample has an associated **label** describing the category we've manually determined to be the correct one. 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 category.

If the prediction is correct (that is, it matches the label we already know to be accurate), 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, an *updater* uses the correct label, the predicted label, and the current state of the classifier to 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.

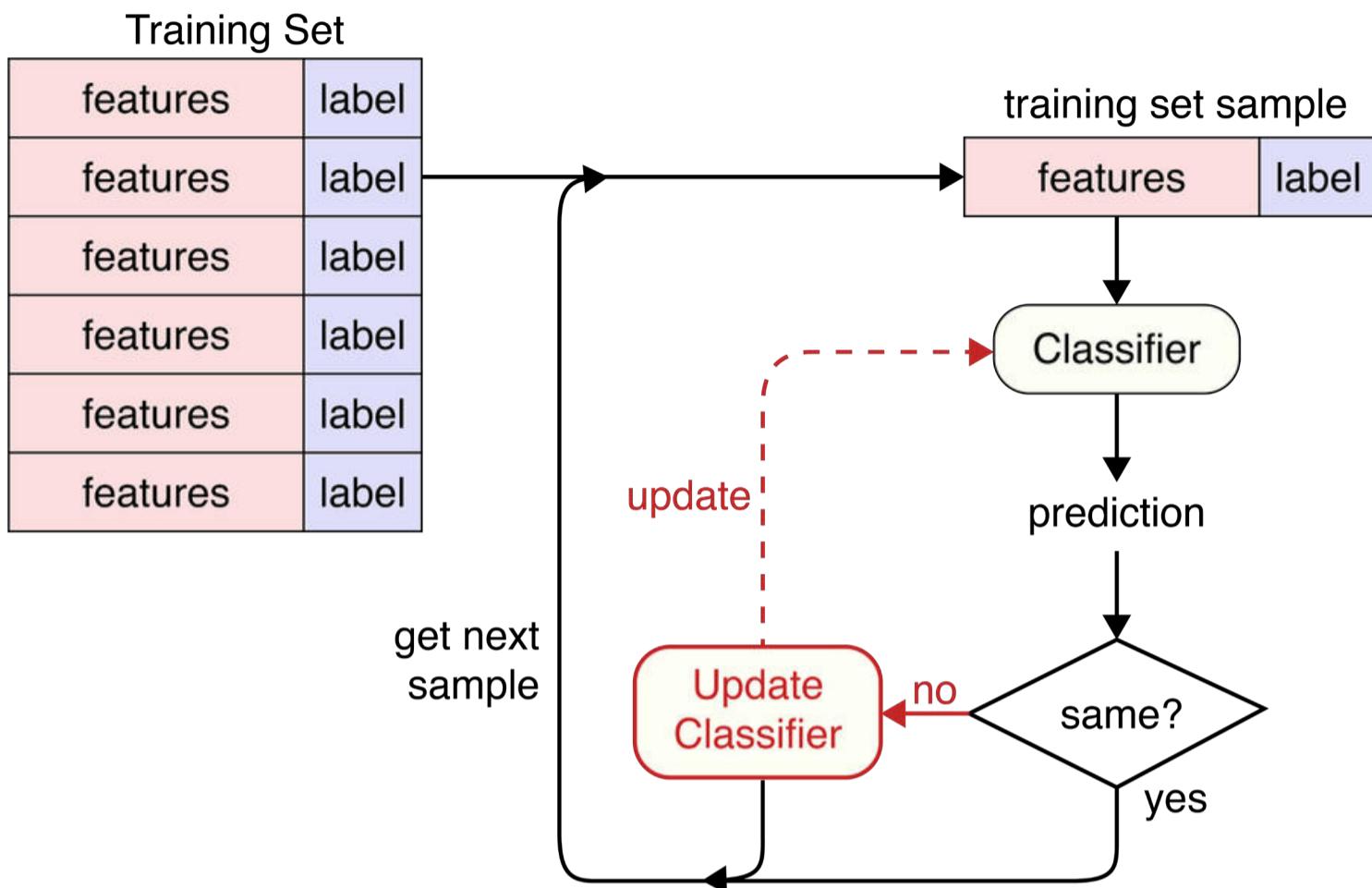


Figure 8.1: The essence of training our classifier.

This explanation and diagram are a simplified version of the full process. As we'll see in later chapters, our test may also include other factors that could lead us to update the categorizer's variables. We'll also find that it's more efficient to update less frequently than after every sample.

By running through this loop, one sample at a time, the classifier's internal variables are nudged into 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. Typically, we'll keep training as long as the system is still learning and improving its performance on the test data. But we might stop if we run out of time, or if the system's performance is good enough for the task we want to apply it to.

Let's now look at how we might measure our classifier's skill at predicting correct labels.

8.2.1 Testing the Performance

The data we use to train the system is called the **training data**, or **training set**. The training data is what we're starting with, so it's not the data the classifier will see in the real world, once it's been **released**, or **deployed**. That's the **real-world data** (or **deployment data**, **release data**, or **user data**) that will only arrive after our running system has been released.

We'd like to know how well our categorizer will perform on real-world data before we deploy it. We may not need perfect accuracy, but we probably have in mind a quality threshold we want the system to meet or exceed. 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 that's not enough. If we just use the quality of the classifier's results from this data, we'll be misled.

To see why, let's suppose we're going to use our supervised classifier to look at pictures of dogs and assign a label to each picture identifying the dog's breed. Our goal is to put the system online, so people can drag a picture of their dog onto their browser, and we'll come back either with the dog's breed, or the catch-all "mixed breed."

To train our system, we'll collect 1000 photos of different purebred dogs, each labeled by an expert. Now we show the classifier each picture and ask it to predict that dog's breed. If the answer is wrong, we tell it the correct answer, and the system will adjust its internal parameters to make it more likely that it will get the correct answer the next time it sees that picture (again, we'll be getting into the details of that operation later).

We'll show the system all 1000 pictures, and then we'll show it all of them again, and again, over and over, though we'll usually scramble the order so they don't always arrive in the identical sequence. If our

system is designed well, it will gradually start producing more and more accurate results, until it's perhaps correctly identifying the breed of the dog in 99% 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 we don't really know what the system learned from the photos. For example, suppose our images of Poodles look like Figure 8.2.

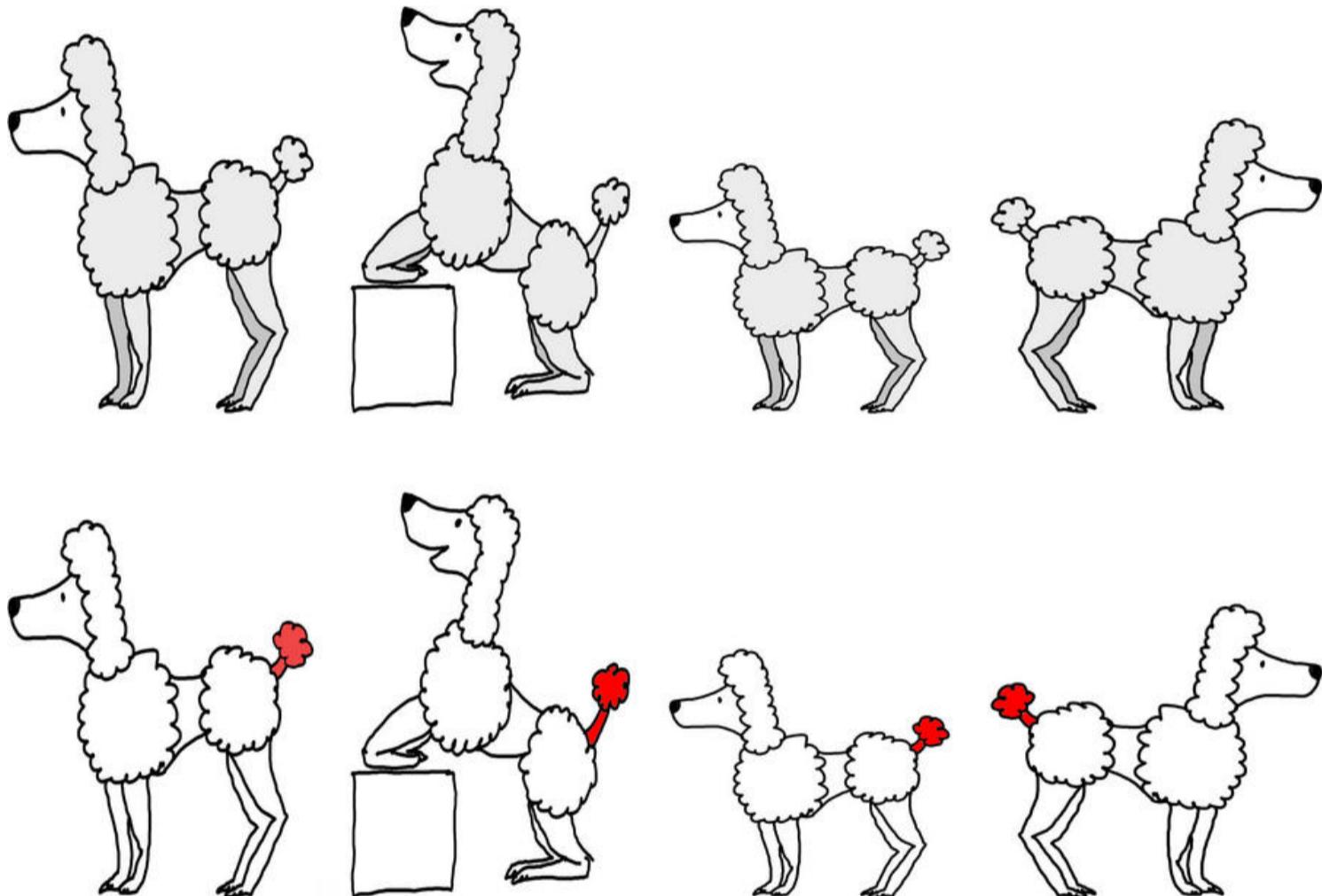


Figure 8.2: Our system might learn to recognize poodles just from the bob on their tail. Top row: Our input poodle images. Bottom row: The feature that our system learned to identify a photo as a Poodle is highlighted in red.

When we prepared our training set, we didn't notice that all of the Poodles had a little bob at the end of their tails, and none of the other dogs did. The system could discover the rule that a white bob on the end of a tail means it's a Poodle. Using that rule it would categorize all of our images correctly.

Perhaps all the pictures of Yorkshire Terriers were taken when the dogs were sitting on a couch, as in Figure 8.3. We hadn't noticed this, nor that none of the other pictures had couches in them. The system might discover that if there's a couch in the image, then it should label it a Yorkshire Terrier. This rule, too, would work perfectly for our training data.

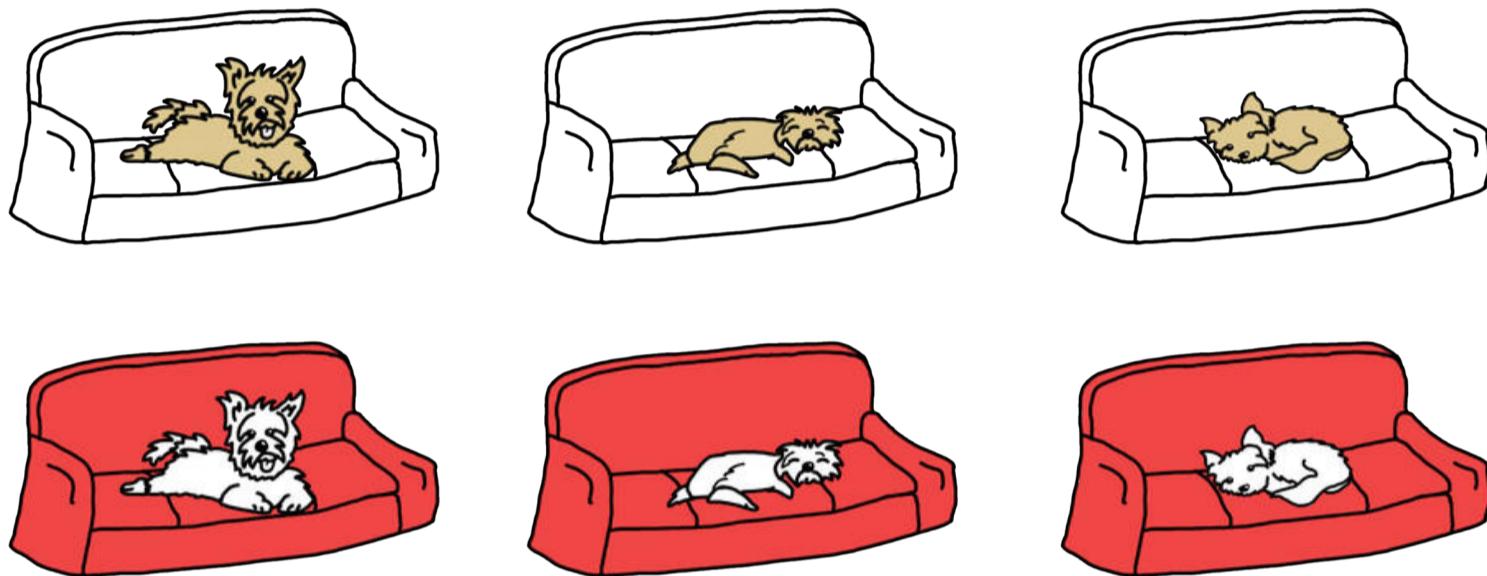


Figure 8.3: Top row: three Yorkshire Terriers (Yorkies) on couches. Bottom row: The feature our system has learned, shown in red: the couch.

Suppose we 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 Huskie on a couch, as in Figure 8.4. Our system will notice the white bob at the end of the Great Dane's tail and say that it's a Poodle, and it will see the couch and ignore the dog and report that it's a Yorkie.

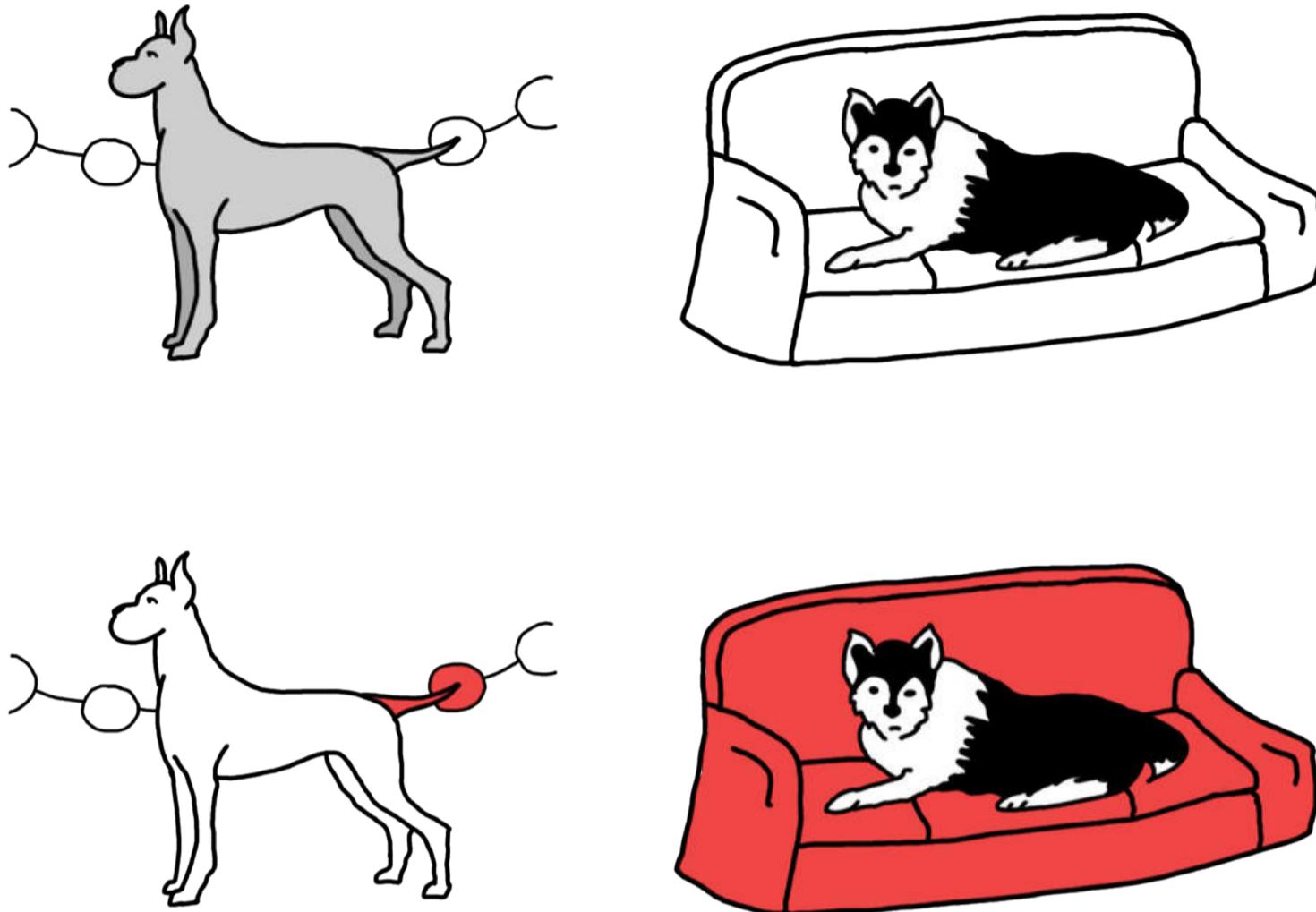


Figure 8.4: Top: A Great Dane is standing in front of a holiday display of white balls on a string, and a Huskie is lying on a couch. Bottom: The system sees the white ball on the end of the Great Dane's tail and tells us that the dog is a Poodle, and notices the couch and so categorizes the dog as a Yorkie.

This isn't just a theoretical concern. A famous example of this phenomenon describes a meeting in the 1960's where a researcher was demonstrating an early machine-learning system [Muehlhauser11]. The details of their training data are murky, but it seems that they had a collection of photos containing multiple matched pairs of images: a stand of trees, and the same trees with a camouflaged army tank in their midst. The researcher presented results that showed the system could pick out the image with the tank every time. The claim was that the system knew how to recognize the camouflaged tanks, which would have been a shocking level of ability.

At the end of the talk, someone in the audience stood up and observed that the photos with the tanks in them were all taken on sunny days, while the photos without the tanks were all taken on cloudy days. It seemed likely that the system had merely to distinguish a bright picture from a dim picture, and the results had nothing to do with trees or tanks at all.

So just 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.

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 good our system is 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.

8.3 Test Data

The best way anyone has found to work out 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 new collection of data points, or samples, the **test data** or **test set**.

Like the training data, we hope that the test data will be representative of the data that we're going to see once our system is in actual use.

The typical process is to train the system using the training data until it's doing as well as we think we 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 performance on the test data isn't good enough, then no matter how well it's done on the training data, we need to train it some more. Since training on more data is almost always a good way to improve performance, we'd probably go gather more pictures of dogs, each with its breed, to add to our training set. We'd either re-train from scratch or just train more with the new data, and then try the test data again to evaluate our classifier.

Another benefit of getting more pictures is that we will probably get examples that avoid the idiosyncrasies we found during training. For example, we might find dogs other than Poodles with a bob on their tail, or dogs other than Yorkies on a couch. Then our system would have to find other ways to classify those dogs, and hopefully those idiosyncrasies in the original training set would cease to matter.

An *essential* rule of the training and testing process is that *we never learn from the test data*. As tempting as it might be to put the test data into the training set, because then we'll have even more examples to learn from, this would ruin the reliability of our measure of the general ability 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 could all too easily key in on idiosyncrasies in the training data (now with the test data included). If we then use the test data to see how well the classifier works, it applies the correct label to each image, but it's "cheating," because it's already learned from our test data.

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.

The problem of learning from the test data is important enough to have its own name: **data leakage**, also called **data contamination**, or **working with 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.

If we're not meticulous about keeping these data sets separate, our system might learn something subtle, like the statistics of the test data. These can include the range of values, or their mean, or their standard deviation. Sophisticated algorithms can learn from more subtle statistics that we might never even think to look for, but can skew the program's performance to appear better than it really is.

In many fields there are some ideas that come up over and over again in different and subtle ways, and data contamination is one of those ideas in machine learning. We'll revisit this idea in Chapter 12 where we'll see that we have to take explicit steps during pre-processing to keep our training and test data from influencing one another.

Figure 8.5 shows one flow of information when training, where results are used to help the classifier learn from mistakes, and another flow for testing, where the results are used to measure the classifier's performance, and are *not* fed back to help the classifier learn.

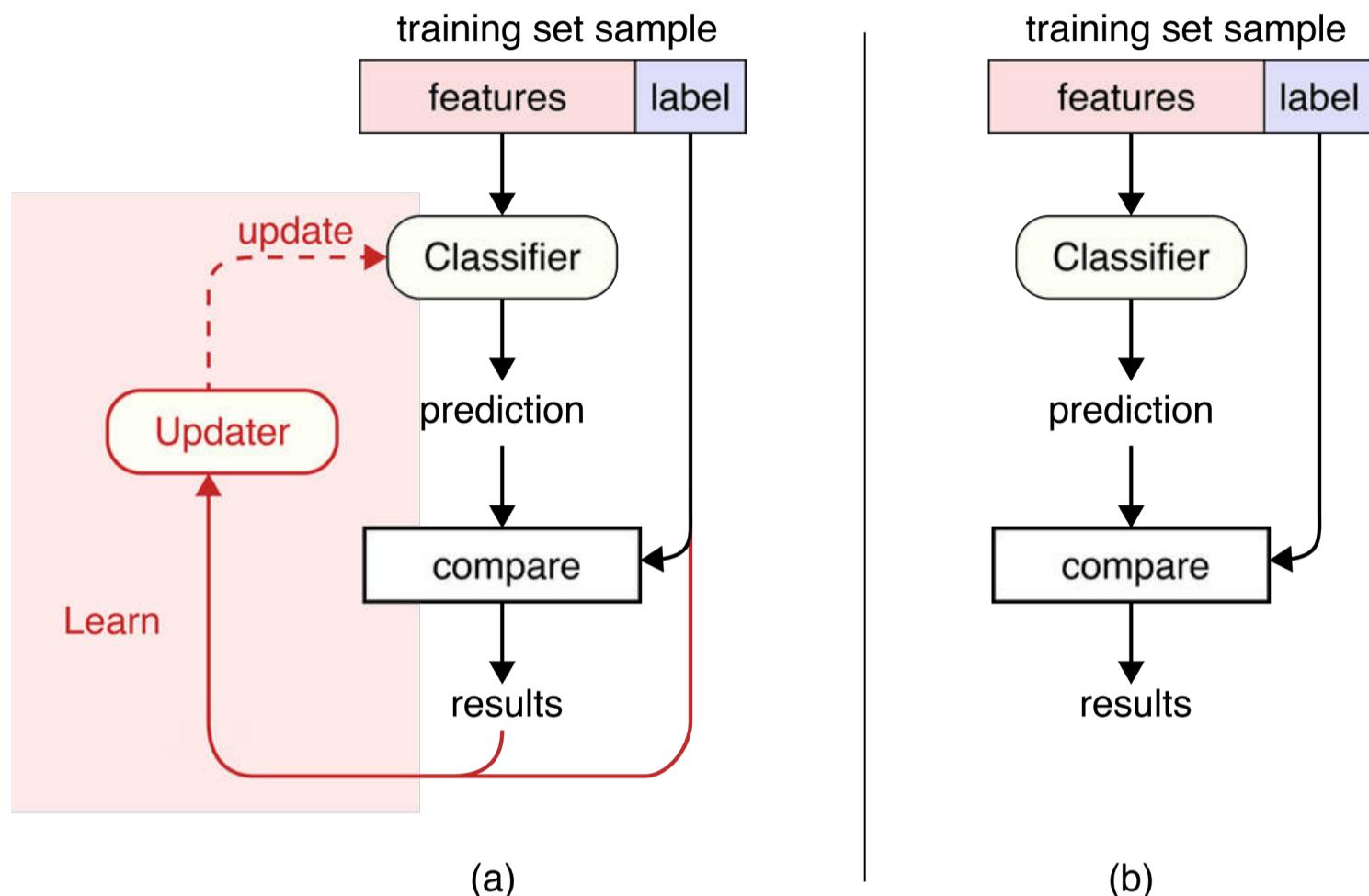


Figure 8.5: The difference between training and testing. (a) The training flow takes the results of comparing the prediction and the real label, and uses any error between them to help the classifier learn to do better. (b) When we run the test data through the system, we omit the learning step.

As Figure 8.5 shows, when we're evaluating the test data the learning mechanism is disconnected, and the classifier is not changed even when it predicts the wrong label. The learning stage is over, and now we're only evaluating the system we created.

Another way to look at data leakage is to think about our training data as a bunch of multiple-choice exams, where each time we get a question wrong, we're told what the correct answer should have been.

Let's suppose that over time we get very good at answering a set of these exams. We don't know if it's because we've learned the subject matter, or because we've memorized all the answers. To find out, we take a new test with new questions. It's essential that all the questions have to be completely new to us, because if we've seen them before, we could just answer them with our memorized correct answer. That wouldn't be a good test of our knowledge.

So we must keep our test set hidden from the system until training is over, and then use it just once to estimate its performance.

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% 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 set has a broad enough collection of samples that it's a good approximation of the full input data. Most machine-learning libraries offer routines to perform this splitting process for us.

Figure 8.6 shows the idea.

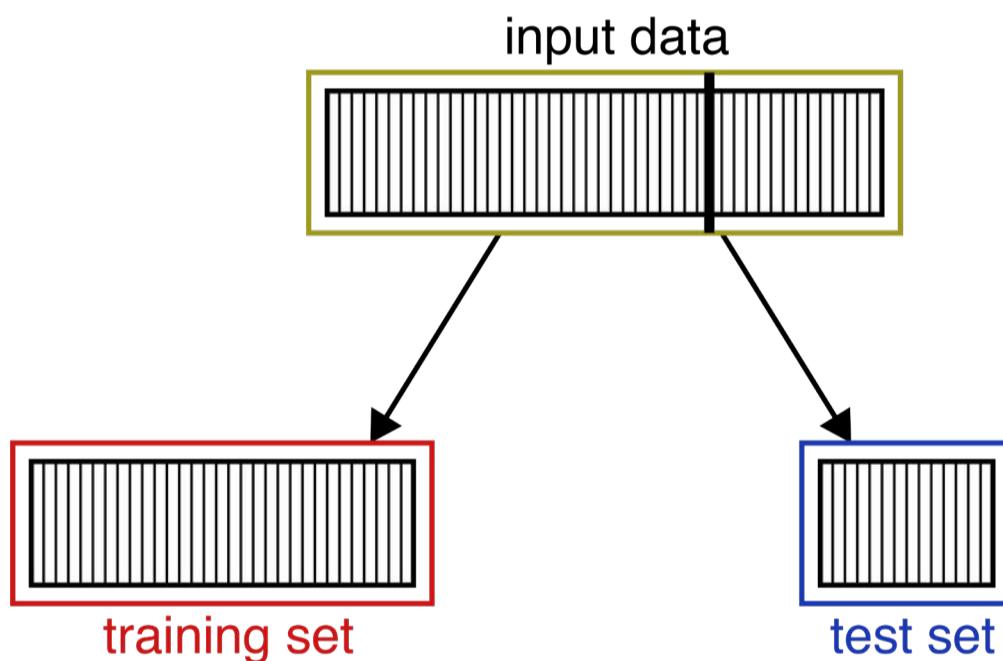


Figure 8.6: Splitting our input examples into a training set and a test set. The split is often about 75-25 or 70-30.

Evaluating the quality of a learner with the test set is usually the last thing we do before deployment. If the system does well enough, we're good to go. If its performance isn't good enough, then we typically go back to training.

Even if our test data is representative of the overall input data, we might still wish we had more cases to work from. Returning to our example of classifying dogs, our input data might contain images only of Standard Poodles. But today there are lots of dogs that are partly

poodle and partly another breed, such as Cockapoos (Cocker Spaniel and Poodle) [VetStreet17a], Labradoodles (Labrador Retriever and Poodle) [VetStreet17b], and Schnoodles (Miniature Schnauzer and Poodle) [VetStreet17c]. Dogs of these breeds look something like a standard poodle, but there are differences, as we can see in Figure 8.7. If we want our system to recognize these different types of poodle-like dogs, we'll need to train it with labeled images of each breed.



Figure 8.7: Different dogs that have some Poodle in them. From left to right, a Standard Poodle, a Cockapoo, a Labradoodle, and a Schnoodle.

8.4 Validation Data

We often derive another set of data from the original input, in addition to the training set and the test set. We call this the **validation data**, or **validation set**.

This is yet another chunk of data that is meant to be a good proxy of the real-world data we'll see when we deploy the system. We typically make this set by splitting the original data into three pieces, often assigning about 60% of the original data to the training set, 20% to the test set, and the remaining 20% to the validation set. Figure 8.8 shows the idea.

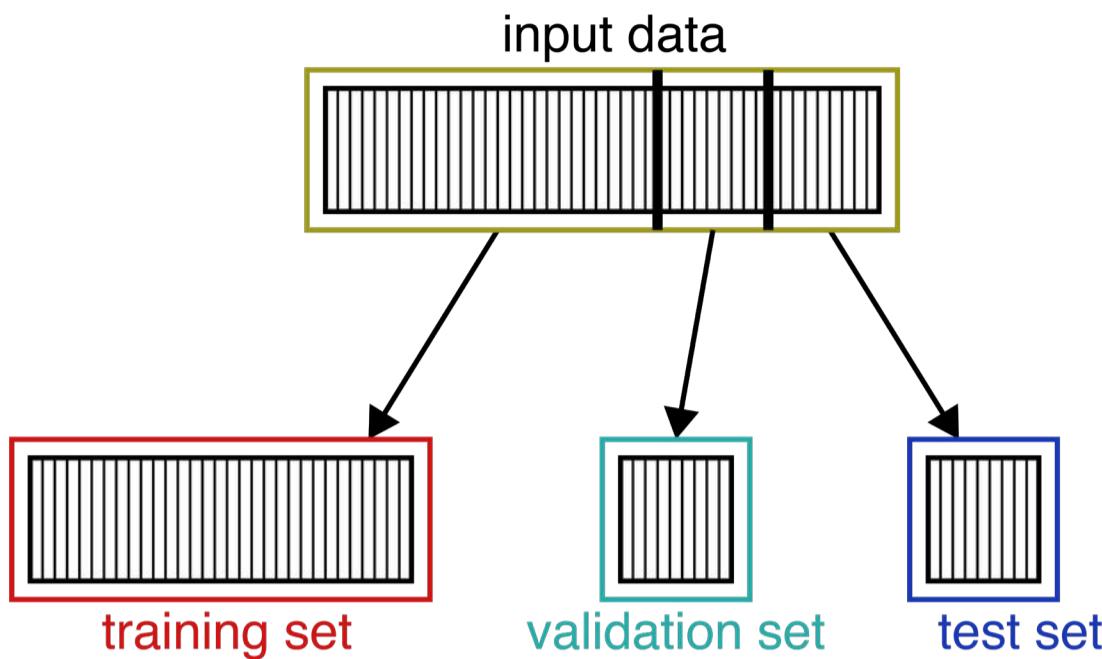


Figure 8.8: Splitting our input data into a training set, validation set, and test set.

This way to split up the data is useful when we use automated search techniques to try out many values of hyperparameters. Recall from Chapter 1 that hyperparameters are variables that we set before we run our system to control how it operates. For each variation, we train on the training set, and evaluate the system's performance on the validation set. We then usually select the hyperparameters that gave us the best results, and then we run that through the test set to evaluate the system's performance on data that it's never seen before.

In other words, we use the validation set as part of the learning process. The test set is still held aside for that one-time final evaluation.

Our approach to trying out different sets of hyperparameters is based around running a loop. Let's look at a simplified version of that loop now, where we'll use just one split of the training set into a smaller training set, a validation set, and a test set. We'll soon see a more sophisticated method called **cross-validation** that doesn't re-use the same validation set over and over in this way.

To run our loop, we'll select a set of hyperparameters, train our system, and then test it with the validation set. This tells us how well the system that was trained with those hyperparameters will do at predicting new

data. Now we'll set that system aside, pull out the next set of hyperparameters, train a new system, and use the validation set to evaluate its performance. We'll repeat this process over and over, once for each set of hyperparameters.

When we've run through all sets of hyperparameters, we just pluck out the set of values that resulted in the system that gave us the best performance. We then take that system, run the test set through it, and check how good its predictions are.

This tells us how good it will be on new data. Figure 8.9 shows this graphically.

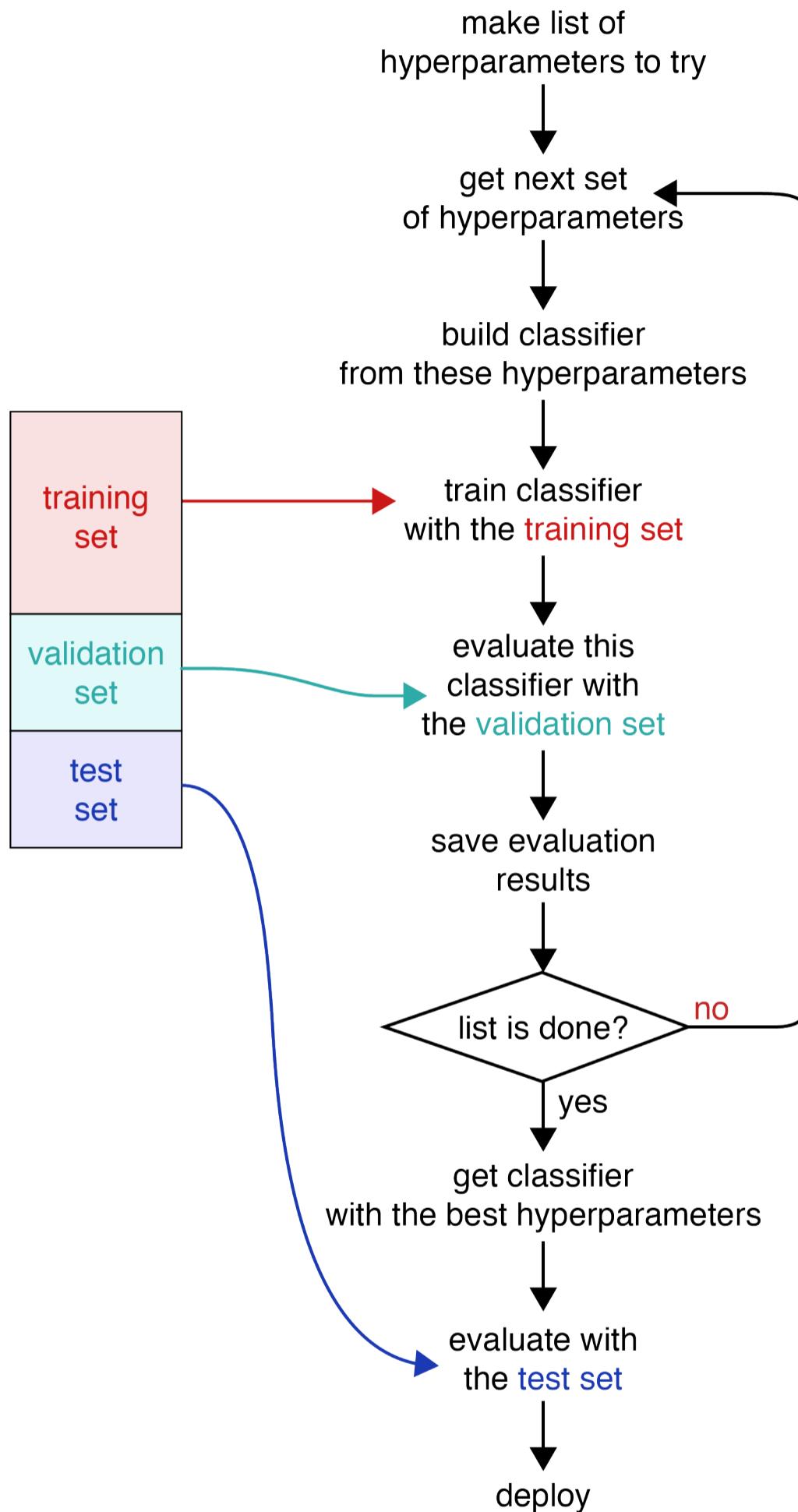


Figure 8.9: We use the validation set when we try out lots of different hyperparameter sets. Note that we still keep a separate test set, which we use just before deployment.

When the loop is done, we might 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 might seem that we can save ourselves the trouble of making a separate test set, and then running 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, something in that data has contributed to our final classifier. So our knowledge of the classifier's performance on the validation data would "leak" into our final evaluation before deployment.

If this seems subtle, 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 deploying a system that isn't good enough for our intended use.

To get a good estimate for how our system will perform on brand-new data that it's never seen before, we need to test it on brand-new data that it's never seen before.

The validation set is necessary in a loop like Figure 8.9, but it takes a second big chunk out of our precious training data, so clever techniques have been developed to get around the need to set aside a separate validation set. Let's look at them now.

8.5 Cross-Validation

The discussion above assumed that when we split up our original data into pieces, the training set was still large enough to do a good job of training the system.

But what if it's not? Sometimes we only have a small amount of labeled data, and getting more is impractical or impossible. Perhaps we're working with images of a comet taken by a spacecraft during a short-lived flyby. Or maybe we're working with data measured from a hurricane that has dissipated. In these situations, and many others, it's either inconvenient, expensive, or just impossible to get more data.

Because we have only a limited number of samples, we want to make use of every one. In the last section we saw that if we want to try out multiple learners, we need to split up our original training data into three pieces, leaving perhaps only 60% for actual training.

We can do better. The trick is that while we still need to remove some samples from our input data in order to make a test set, we can cleverly avoid the need to carve out a separate, permanent validation set as well.

The technique for pulling this off is called **cross-validation** or **rotation validation**. There are different types of cross-validation algorithms, but they all share the same basic structure [Schneider97].

The core idea is that we'll run a loop that trains and tests our system multiple times, as we saw in Figure 8.9.

But we'll change our focus from looking at trying out different sets of parameters to instead evaluating just one system. Our goal will be to use the basic loop of Figure 8.9 to evaluate how well our system will respond to new data, but without making an explicit validation set. This will let us use all of our data for training (though as we'll see, not all at the same time).

Before we start, we'll pull out the test data and set it aside. We'll use that as before, just before deployment.

Now we're ready for the trick. Now each time through the loop, we'll split all of our remaining training data into a training set and validation set. This split is temporary, and only for that time through the loop.

So our goal now is how to determine the performance of our system given a set of hyperparameters, but without sacrificing 20% of our precious training data into a dedicated validation set that we can't use for training. We'll do this by creating a new, smaller loop inside of the one in Figure 8.9. This loop will let us evaluate the performance of just the current classifier.

We begin by building a fresh instance of our classifier. We train our system on the temporary training set, and evaluate it with the temporary validation set. This gives us a **score** for the classifier's performance.

Now we go through the loop again, but this time when we split the remaining training data into temporary training and validation sets we split that data into *new sets*, different from any we've tried before.

In this way we go through the loop over and over, splitting the data into fresh sets, training and validating, and getting a score.

When we're done, the average of all the scores is our estimate for the overall score of our classifier.

A visual summary of cross-validation is shown in Figure 8.10.

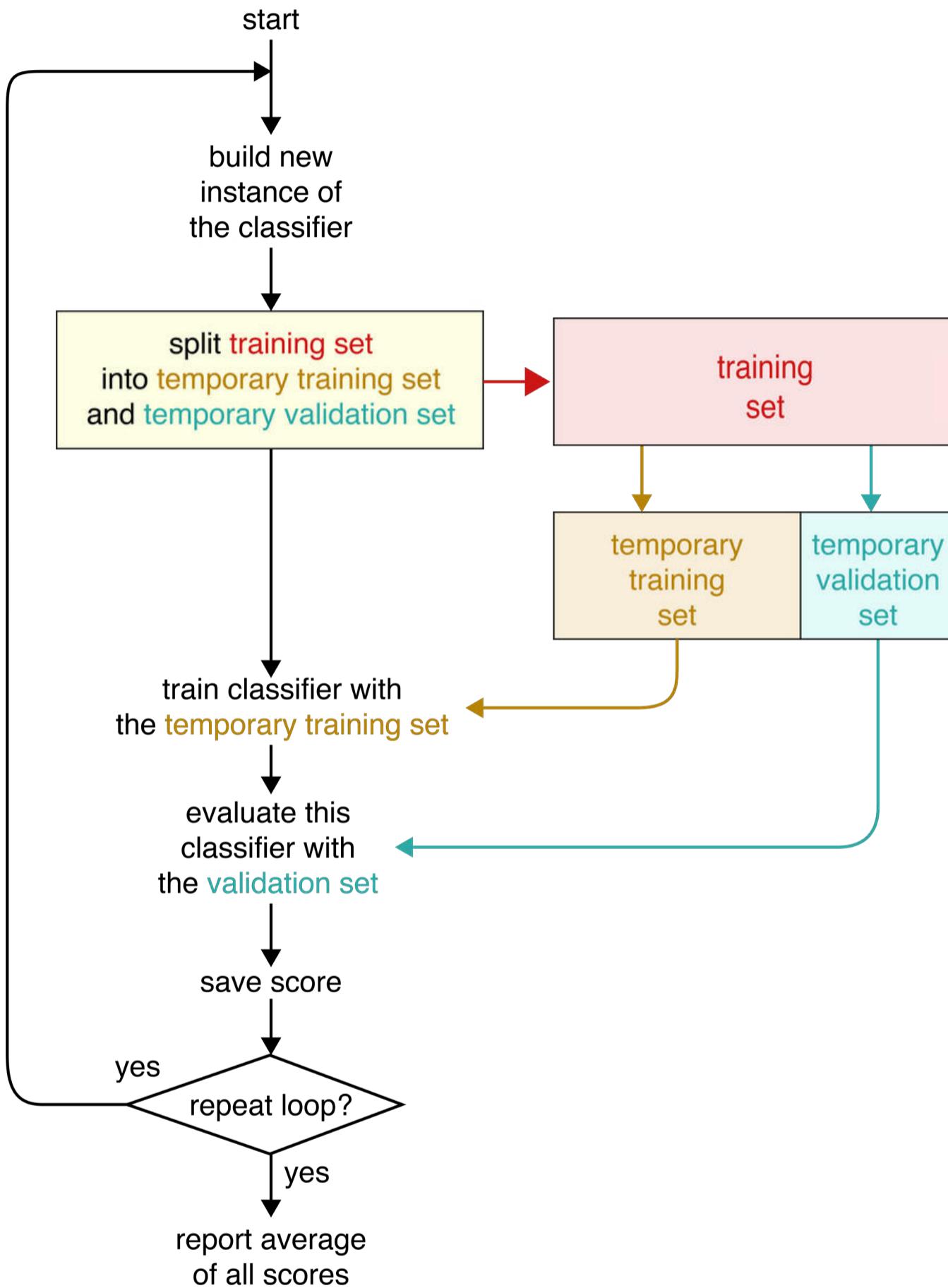


Figure 8.10: Using cross-validation to evaluate our system's performance.

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 validation set.

This algorithm doesn’t have data leakage issues because each time through the loop, we create a new classifier, and the temporary validation 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.

There are a variety of different algorithms available for constructing the temporary training and validation sets. Let’s look at a popular approach.

8.5.1 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, but 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.

The algorithm starts before the cross-validation loop of Figure 8.10 begins. We take our training data and split it up into a series of equal-sized chunks. We call each chunk a **fold**. The word “fold” 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 8.11 shows the idea.

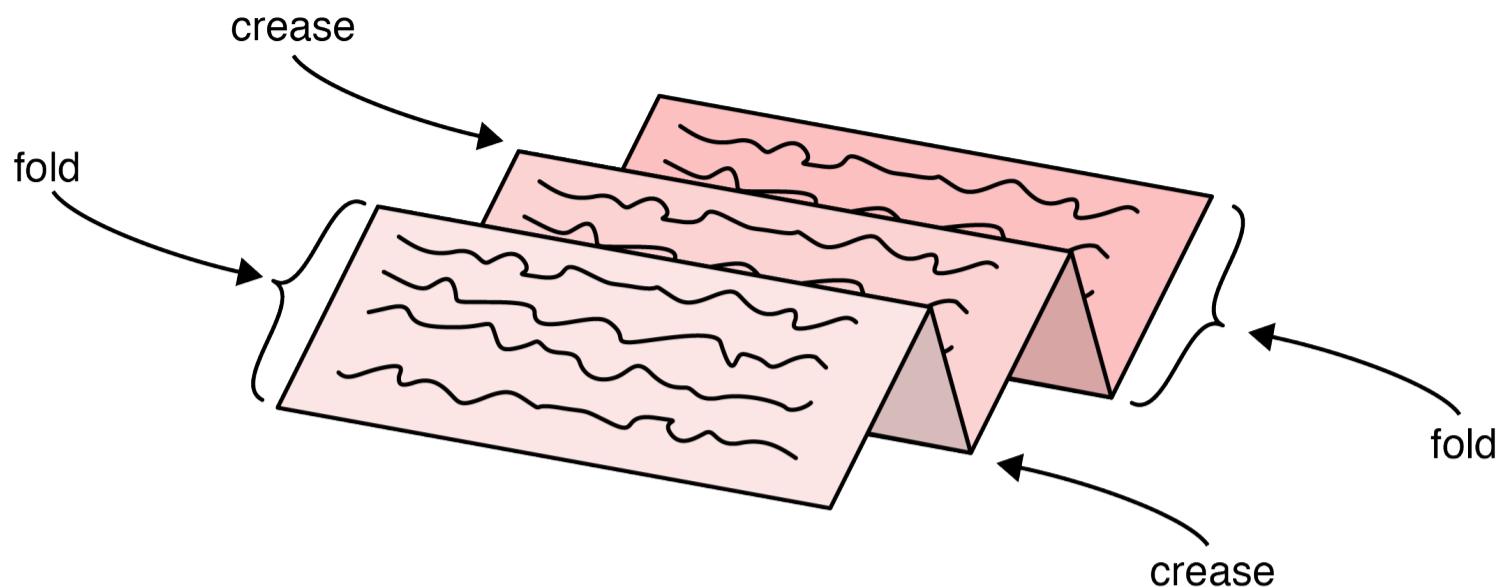


Figure 8.11: Creating the folds for k-fold cross-validation. Here we have 4 creases and 5 folds.

We'll always build equal-sized folds from our training data, with each fold holding the information between two creases (or one crease and the top or bottom of the list). We can flatten out Figure 8.11 to create the more typical picture of 5 folds, shown in Figure 8.12.

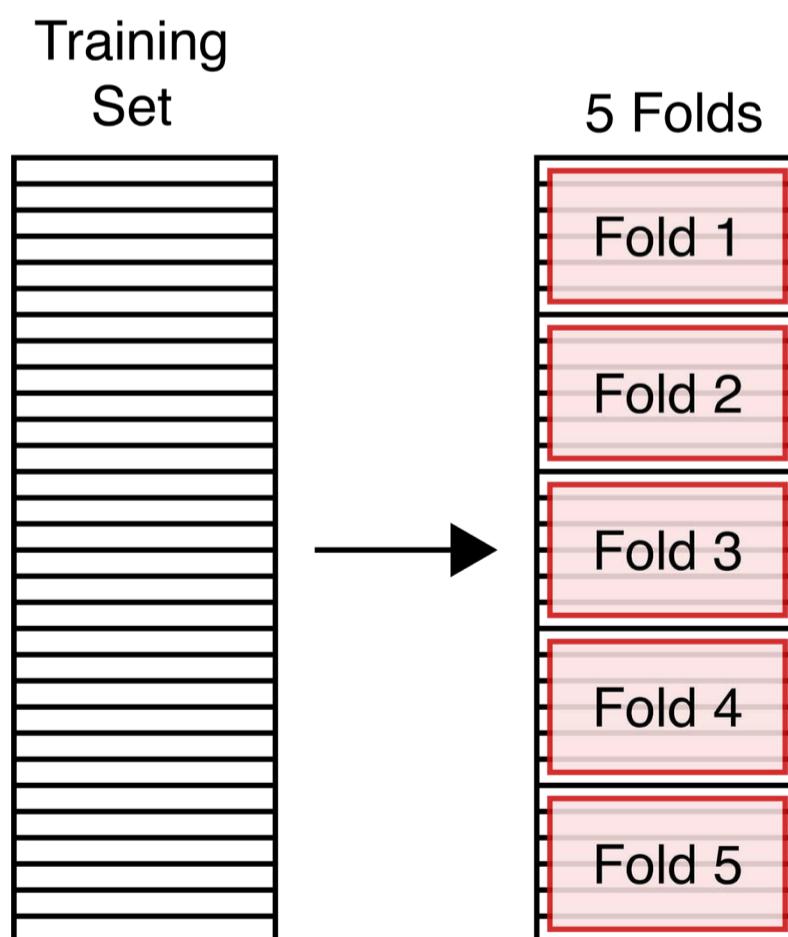


Figure 8.12: Splitting our training set into 5 equally-sized folds, named Fold 1 through Fold 5.

Let's use these 5 folds to see how the loop proceeds. The first time through the loop, we treat the samples in folds 2-5 as our temporary training set, and the samples in fold 1 as our temporary validation set. That is, we'll train the classifier with the samples in folds 2 through 4, and then evaluate it with the samples in fold 1.

The next time through the loop, we'll use the samples in fold 2 for our temporary validation set, and the samples in folds 1, 3, 4, and 5 for the training set. We train and validate as usual with these two sets, and continue with the remaining folds. Figure 8.13 shows the idea visually.

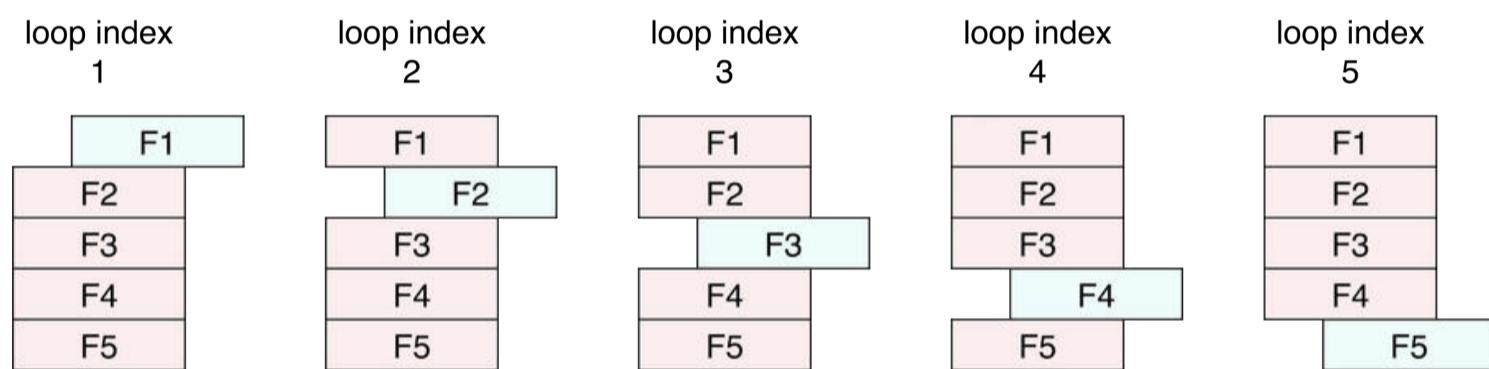


Figure 8.13: In each pass through the loop, we choose one fold for validation (in blue), and train with the others (in red). If we go through the loop more than 5 times, we just repeat the pattern.

We usually set the number of folds to be the same as the number of times through the loop. But because training often involves some random numbers, we can choose to repeat the loop as many more times as we like, just repeating the cycle of fold selections.

The beauty of this scheme is that we get to use all of our training data, without splitting it into permanent test and validation sets. Instead, we make a new training/validation split each time through the loop. The average testing score for all those different splits is our final score for the system.

This makes cross-validation a great option when we don't have much data, but we really want to get a good estimate for the quality of our system. We do have to repeat the train-validation cycle many times,

which is a downside, but we trade that running time for 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.

As we noted above, we've demonstrated k -fold cross-validation with a classifier, but the algorithm is broadly applicable to almost any kind of learner.

8.6 Using the Results of Testing

There are two principal applications of the types of testing we've been looking at.

The first is to predict how well our system will do before we release it into the wild. We typically use the test data to determine this. If the performance is good enough for our application, then we can deploy it and let people use it. If it's not good enough, we have to think about our many options (such as changing the algorithms, gathering more data, or picking new hyperparameters) and take another shot at building our learner. We can also use the test results to quantify the performance of our system, which can be useful if we're publishing them in a paper, or evaluating different algorithms in some kind of contest or competition.

The second application is during training time, when we want to pick the best hyperparameters for a particular system. While we're doing this, we're committed (at least for the moment) to a specific learner and training regimen, including the training data, and we're just trying to find the best settings of the hyperparameters that control the system.

When that's done, we can evaluate the results with the test data. As before, if those results aren't good enough, we can re-think the whole system, or we might decide that it's worth our time to go looking for better hyperparameters. In that case, we'd stick with the algorithm and go back to doing more cross-validation.

There are no hard rules for this process. We proceed based on experience and intuition, both of which generally improve the more we work with a specific system and data set.

References

- [Muehlhauser11] Luke Muehlhauser, “Machine learning and unintended consequences”, LessWrong blog, 2011. Note the comment from 2016 in which Ed Fredkin reports that he was the person in the audience who spotted the sunny or cloudy “tell” in the pictures. http://lesswrong.com/lw/7qz/machine_learning_and_unintended_consequences/
- [Schneider97] Jeff Schneider and Andrew W. Moore, “Cross Validation”, in “A Locally Weighted Learning Tutorial using Vizier 1.0”, Department of Computer Science, Carnegie Mellon University, 1997. <https://www.cs.cmu.edu/~schneide/tut5/node42.html>
- [VetStreet17a] VetStreet, “Cockapoo”, 2017. <http://www.vetstreet.com/dogs/cockapoo>
- [VetStreet17b] VetStreet, “Labradoodle”, 2017. <http://www.vetstreet.com/dogs/labradoodle>
- [VetStreet17c] VetStreet, “Schnoodle”, 2017. <http://www.vetstreet.com/dogs/schnoodle>

Image Credits

Figure 8.7, Poodle

<https://pixabay.com/en/poodle-the-poodle-dog-the-dog-breed-1561405>

Figure 8.7, Cockapoo

<https://pixabay.com/en/beach-dog-water-sea-cockapoo-2239440>

Figure 8.7, Labradoodle

<https://pixabay.com/en/labradoodle-lab-dog-puppy-2441859>

Figure 8.7, Schnoodle

<https://pixabay.com/en/dog-schnoodle-pet-animal-canine-2689514>

Chapter 9

Overfitting and Underfitting

Determining when our systems have not yet learned enough from the training data, or have learned too much, and how to keep them balanced between the extremes.

Contents

9.1 Why This Chapter Is Here.....	339
9.2 Overfitting and Underfitting	340
9.2.1 Overfitting	340
9.2.2 Underfitting	342
9.3 Overfitting Data	342
9.4 Early Stopping.....	348
9.5 Regularization	350
9.6 Bias and Variance	352
9.6.1 Matching the Underlying Data	353
9.6.2 High Bias, Low Variance	357
9.6.3 Low Bias, High Variance	359
9.6.4 Comparing Curves	360
9.7 Fitting a Line with Bayes' Rule	363
References	372

9.1 Why This Chapter Is Here

We build machine-learning systems to help us extract meaning from data. To help them learn to do this, we usually start with a finite set of examples and try to learn general rules from them. When our system has learned those rules, we can release it to the world, so we (or other people) can give it new data. By applying its learned rules to the new data, the system can give us back useful information about that data.

But 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, and when we get to working on new data we're likely to come to wrong conclusions. 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 be likely to come to wrong conclusions.

These phenomena are respectively called **underfitting** and **overfitting**. The more common and troublesome problem of the two is overfitting, and if unchecked, overfitting can leave us with a system that is all but useless. We control overfitting and rein it in with techniques known collectively as **regularization**.

These ideas are often discussed using the associated concepts of **bias** and **variance**, which we saw in Chapter 2. These ideas give us another way to think about the same phenomena.

In this chapter we'll look at the causes of overfitting and underfitting, and how to address them.

9.2 Overfitting and Underfitting

When our system learns from the training data too well, and does poorly when presented with new data, we say that it is **overfitting**. When it doesn't learn well enough from the training data, and does poorly when presented with new data, we say that it is **underfitting**.

We'll look at these phenomena in turn.

9.2.1 Overfitting

We'll approach our discussion of overfitting with a metaphor.

Let's 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 at the park, 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 [Foer12] [Proctor78].

Suppose we meet 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.

The day goes on, the wedding goes great, and at the reception afterwards 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 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 someone 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 party, 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, we had no way to tell us that this was someone new.

We saw the same problem in Chapter 8, where we associated a white bob on the end of a dog's tail with that dog being a Poodle, and said that any dog on a couch was a Yorkshire Terrier. We made those connections because those characteristics were unique to those dogs in our training data.

We thought we were doing well in both cases, because when evaluating the **training data**, we got most of the results right. Focusing on the positive, we say that we got up to a high **training accuracy**. If we focus instead on the errors, we'd say we had a low **training error** (or **training loss**).

But when we then went out into the world and got new, more **general** data to evaluate, our **generalization accuracy** was low, or equivalently, or **generalization error** (or **generalization loss**) was high.

Our errors with both people and dogs were due to **overfitting**. We "over-learned" or "over-fit" the input data. In other words, we learned too much about it. We paid too much attention to the individual details of each person we met at the party, and used those instead of general principles.

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 make the right prediction, it will find and

exploit that quirk, like our story in Chapter 8 of how a system was supposed to be solving the hard problem of finding tanks in photos of trees, but really was taking the easy way out and simply noted whether the sky was sunny or cloudy.

We can take two actions to control overfitting. First, using **regularization** methods, we can encourage the system to keep learning general rules as long as possible, preventing it from slipping into the memorizing of details.

Second, we can catch when this descent into memorization starts, and stop the learning process at that moment.

9.2.2 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. This is usually much less of a problem 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.

9.3 Overfitting Data

Let's look more closely at overfitting.

Recall from Chapter 1 that we learn from **samples** in the **training set**, and we periodically check our performance using samples in the **validation set**. The score we get from the validation set is our **generalization error**. When that 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.

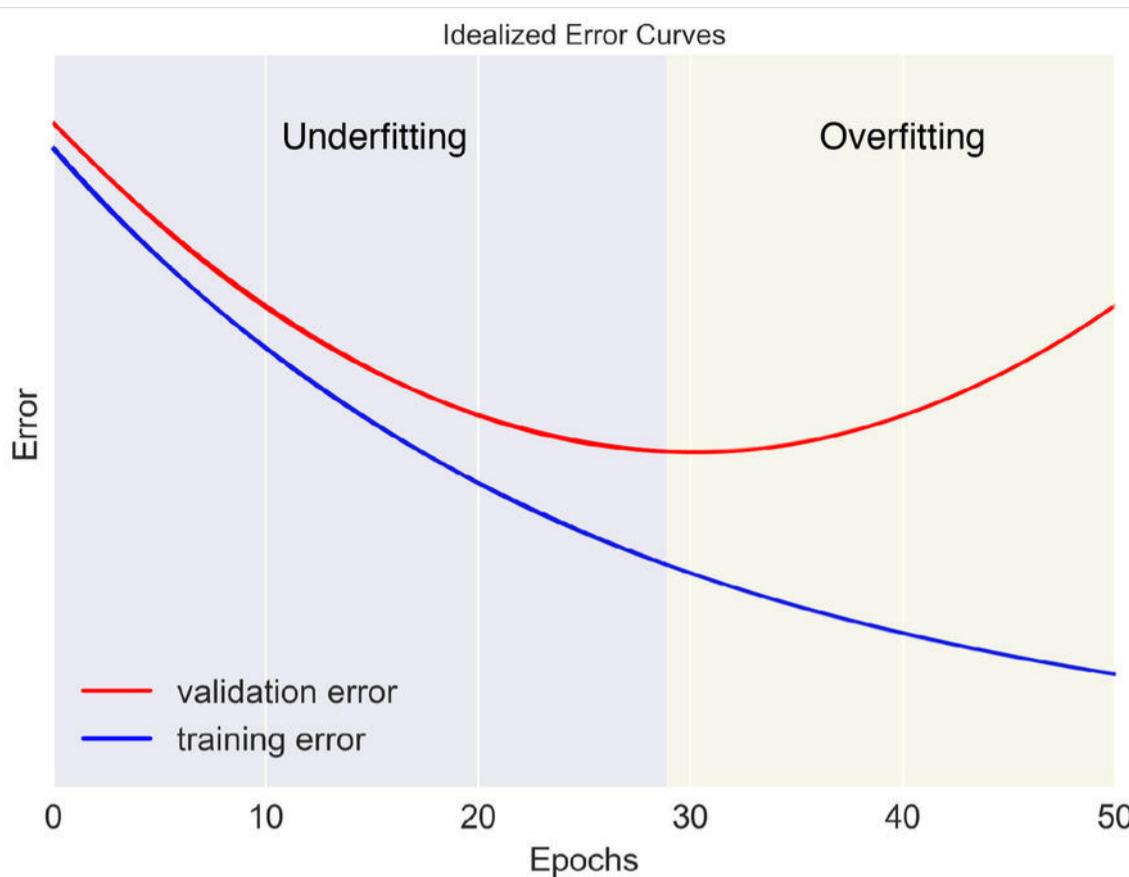


Figure 9.1: An idealized set of error curves. Recall that an epoch means a single pass through all of the training data. The training and validation errors both go down steadily near the start of training, but after a certain point the validation error starts to increase. That's where we enter the realm of overfitting. We want to stop just before that happens (figure after [Duncan15]).

Note that in Figure 9.1 we show the **validation error** rather than the generalization error. Recall from Chapter 1 that during training we usually estimate the generalization error with a **validation set** that we keep separate from the training data. When the validation set is a good proxy for new and unseen data, the two types of error should be similar.

Note something important in this process: *the error on the training set 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 digging too deeply into the specific details (such as that all people with walrus mustaches are named Walter). It's the performance on the validation set that lets us see that

we've crossed the line, and although we're getting better at the training data, we're no longer improving on new data. In fact, we could be getting worse.

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 will automatically adjust the music the way she wants it.

We start by gathering data. 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.

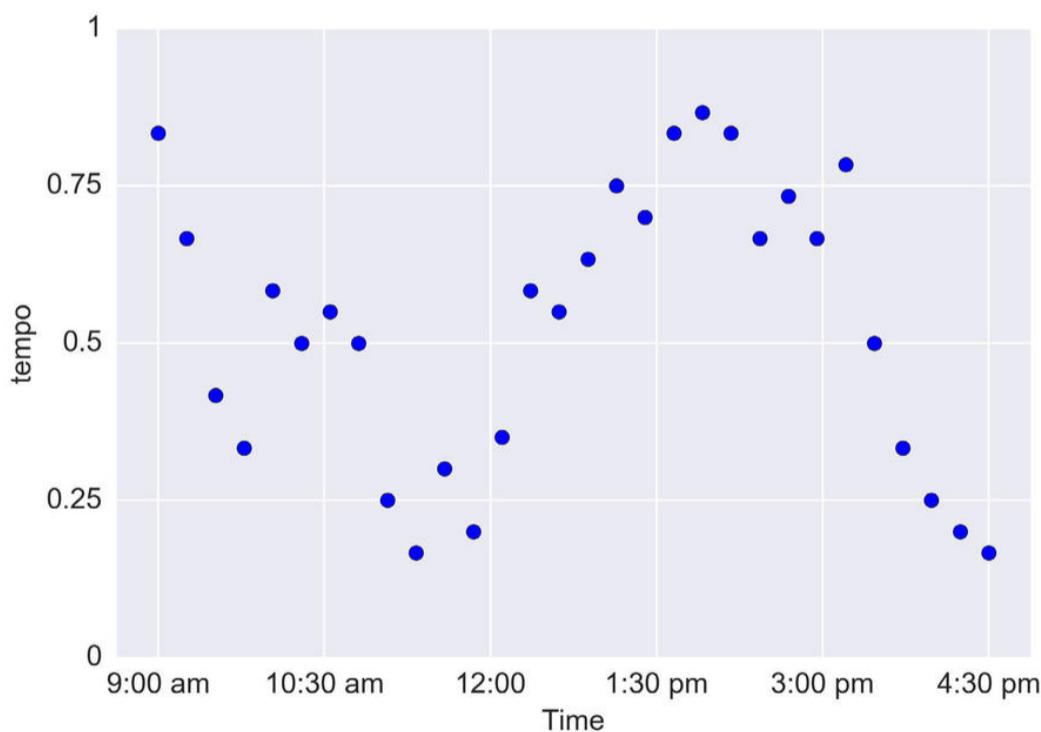


Figure 9.2: Our recorded data showing the tempo chosen by our store owner each time she adjusted it during the day.

Back in our labs that evening, we fit a curve to the data, as in Figure 9.3.

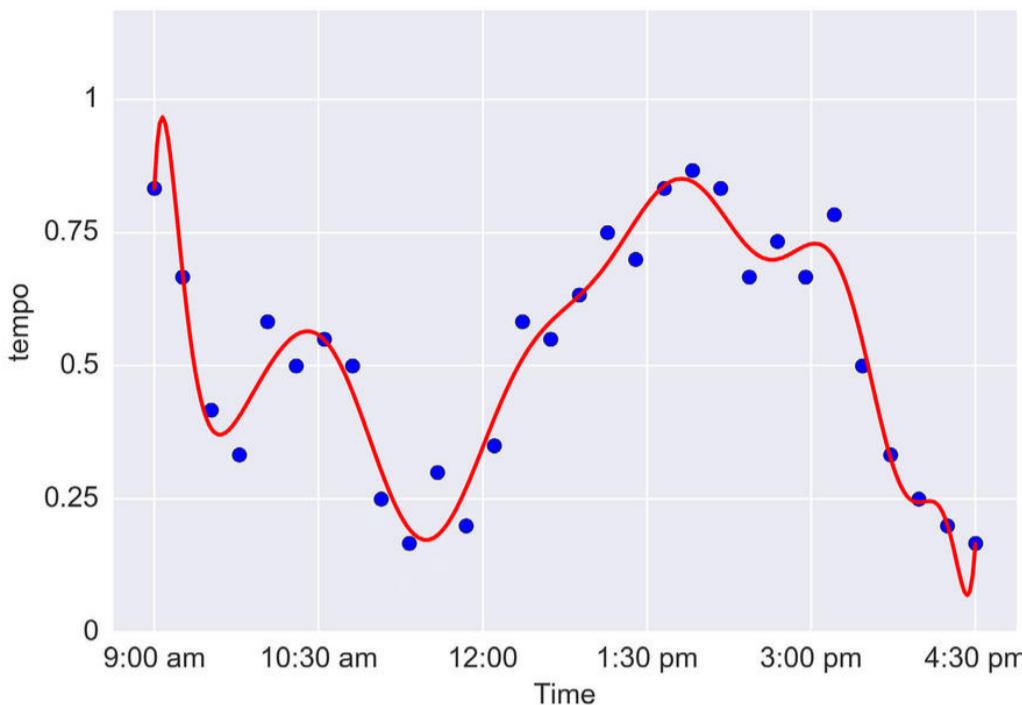


Figure 9.3: A curve fit to the data of Figure 9.2.

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 day the owner is complaining because the tempo of the music is changing far too often, and too dramatically, and 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 to watch her choices for several more days and use all of that data to come up with a more general plan, but she's insistent on having this done as soon as possible. The data we have is all we're going to get.

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 gentler curve of Figure 9.4.

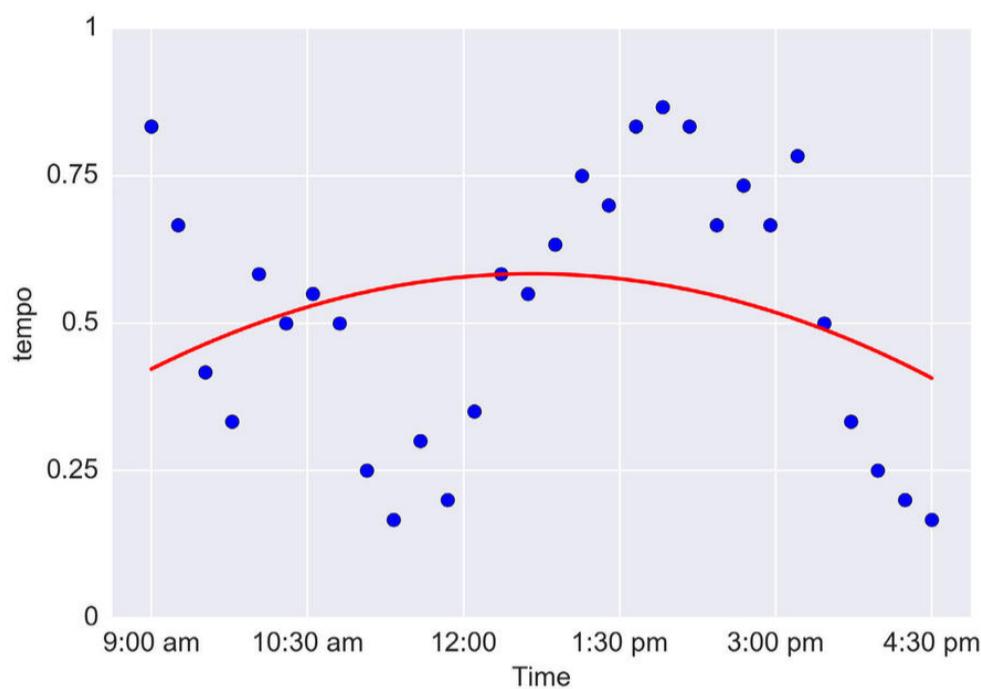


Figure 9.4: A gentler curve for matching our tempo data.

We find the next day that our client isn't satisfied by this choice of tempo selection either, because it's 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." So the next day we set up the system according to Figure 9.5.

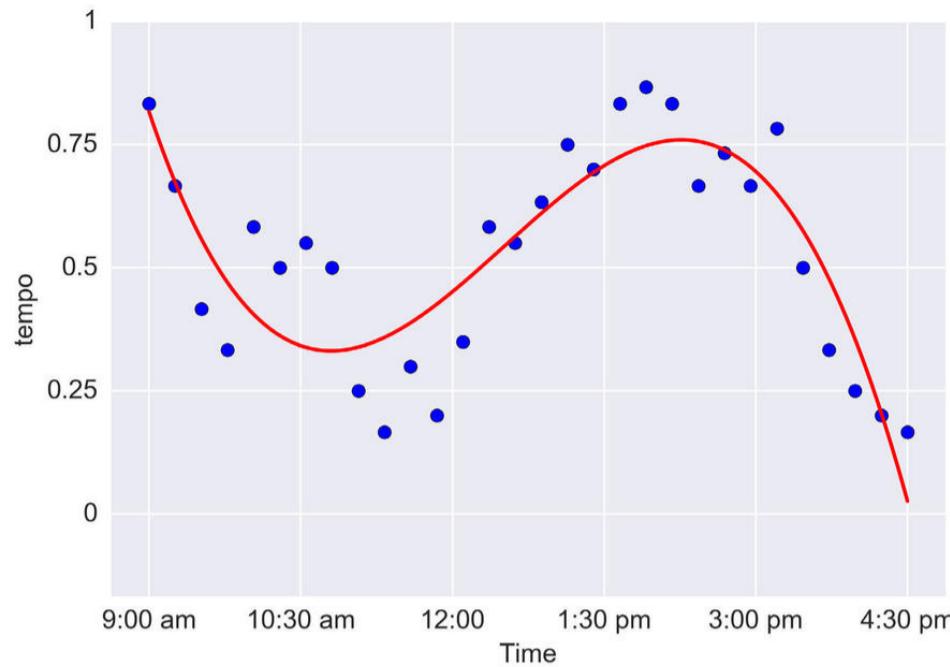


Figure 9.5: A curve that matches our tempo data well enough, but not too well.

Our client is happy with this curve and the tempos of the songs it chooses over the day. We've found a good place between under- and over-fitting. In this example, finding the best curve was a matter of personal taste. We'll see later that when we use these ideas during training, we can come up with specific rules that will let us know when we pass over the boundary from underfitting to overfitting.

Figure 9.6 shows another example of overfitting, this time in categorizing two categories of 2D points.

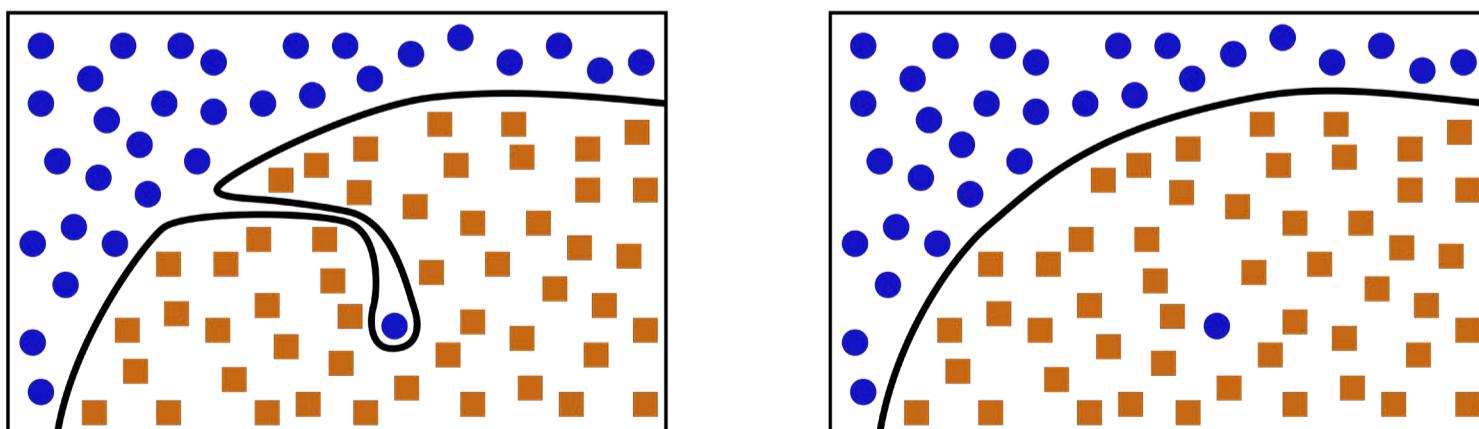


Figure 9.6: A likely case of overfitting. Left: The input data with a boundary curve that separates the two different sets of data points. The curve makes a weird plunge to include a single oddball data point. Right: Probably a better curve. (after [Bullinaria15]).

Here we have one circular point deep in square territory. 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 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, it seems reasonable to infer that this one isolated circular point is probably unusual, and unlikely to recur in future batches of measurements. In other words, we expect that future data that lands near that one isolated circular point will belong to the square category.

This leads us to say that the curve on the left of Figure 9.6 is probably overfitting. In that case, it's probably better to use the simpler boundary curve in the right diagram of Figure 9.6.

9.4 Early Stopping

Generally speaking, when we start training our model we'll be underfitting. The model won't have seen enough examples to figure out how to handle them properly, so it's using first-guess rules that are general and vague.

As we train more and the model refines its boundary curve, the training and validation errors will both typically drop. To discuss this, we'll repeat Figure 9.1 for convenience here as Figure 9.7.

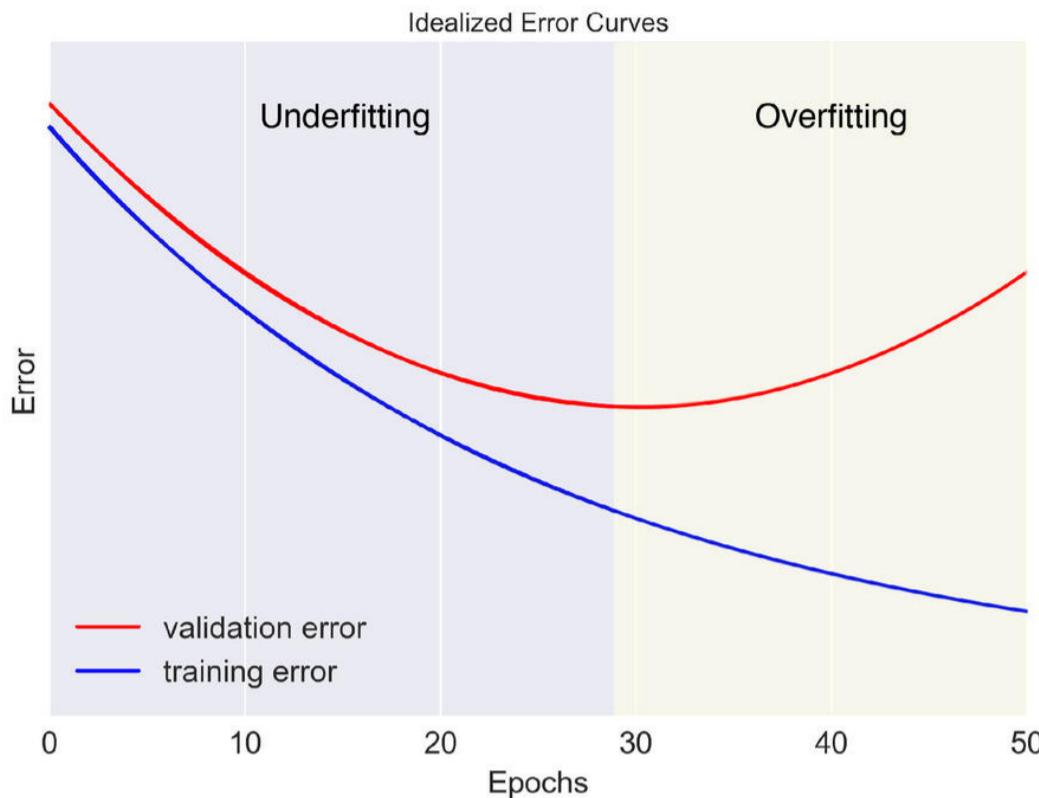


Figure 9.7: A repeat of Figure 9.1 for convenience. In these idealized curves, while validation error is dropping we're underfitting. When it starts to rise, we're overfitting.

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 stand in, 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 pass the boundary at around 28 epochs in Figure 9.7 and we find that the validation error is going up, even as training error is dropping, we 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 easier to think of this idea as **sensible stopping**, since we're stopping our training neither too early nor too late in a broader sense, but at just the right moment when we've stopped improving the model's validation error.

In practice, our error measurements are unlikely to be as smooth as the idealized curves in Figure 9.7. They'll 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. Usually we use a variety of tools to smooth out the errors so we can detect when the validation error really is rising, and not just experiencing a momentary increase. We'll see in Chapters 23 and 24 how easily we can include early stopping in our training process.

9.5 Regularization

We always want to squeeze as much information as we can out of our training data, stopping just short of overfitting. Early stopping is one way to manage that. But there are other techniques that let us reduce the tendency to overfit, allowing us to train longer and continue to push down both training and validation errors.

These techniques for controlling 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 passes the line from “good knowledge of the input data” to “overly-specific knowledge this particular data.” So anything that controls overfitting requires us to add in some new information beyond the basic learning algorithm.

A popular type of information to include is a step that limits the size of the parameters used by the classifier. Conceptually, the core argument for why this staves off overfitting is that by keeping all of these values to small numbers, we prevent any one of them from dominating [Domke08]. 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 by 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 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 could force all of the features we notice to have roughly equivalent importance, then “has a walrus mustache” won’t get the chance to dominate, and the other features will continue to matter when we remember the name of a new person.

We can say that each of our parameters has an associated **weight**, or strength. This is just a number that we use to scale the importance of that parameter. The basic idea is similar to how we might assign weights to different criteria when we draw up a list to help make a decision. For instance, to decide whether or not to move to a new apartment, we might assign a large weight to the walking score of the new neighborhood, and assign a smaller weight to the size of the hallway closet. **Regularization** is a method to make sure that no one weight, or no small set of weights, dominates all the others.

Note that we’re not trying to set all the weights to the *same* value, which would make them useless. We’re just trying to make sure they’re all the same ballpark, so no one weight swamps the others.

Pushing the weights down to small values goes a long way to preventing overfitting, and lets us continue to learn more information from our training data.

The amount of regularization we want 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 parameter that’s traditionally written as a lower-case Greek λ (lambda), though sometimes other letters are used. This is another

example of a **hyperparameter**, or a parameter that we set externally to our learning algorithm. Most commonly, larger values of λ mean more regularization.

By keeping the weights low, this means that the classifier's boundary curves don't get as complex and wiggly as they otherwise could, which is the whole point. So we can use the regularization parameter λ to choose how complex we want our boundary to be. High values will give us smooth boundaries, while low values will let the boundary adapt more precisely to the data it's looking at. In other words, low values of regularization lead to the wiggly overfitting curves we saw earlier, and larger values of regularization smooth those curves out. We'll see this below when we try out different values of λ on a noisy set of data.

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** and **batchnorm** that can help control overfitting on those types of architectures. Both are designed to prevent any elements of the network from dominating the results. Dropout lets us temporarily disconnect some of the elements of the network, forcing other elements to take up the slack [Srivastava14]. Batchnorm adjusts the outputs of each layer so that none of the elements computing those values can dominate the others [Ioffe15].

9.6 Bias and Variance

Recall our discussion of the statistical terms **bias** and **variance** from Chapter 2. These ideas are intimately related to underfitting and overfitting, and often come up when those topics are discussed.

There are many ways to think about bias and variance. Looking at them from the perspective we'll be using in this chapter, 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 [Domingos15].

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, in a classification problem. The ideas of bias and variance are definitely not limited to any one type of algorithm, or to 2D data. But we'll stick to 2D curves because they make the discussion concrete, and we can draw and interpret them.

We'll focus on *finding a good fit to an underlying noisy curve*, and we'll see how the ideas of bias and variance let us describe how our algorithms behave.

9.6.1 Matching the Underlying Data

Let's suppose that an atmospheric researcher friend of ours has come to us for some help. She's measured some data, say the wind speed at a certain spot at the top of a mountain, at the same time every day for a year. Her measured data is in Figure 9.8.

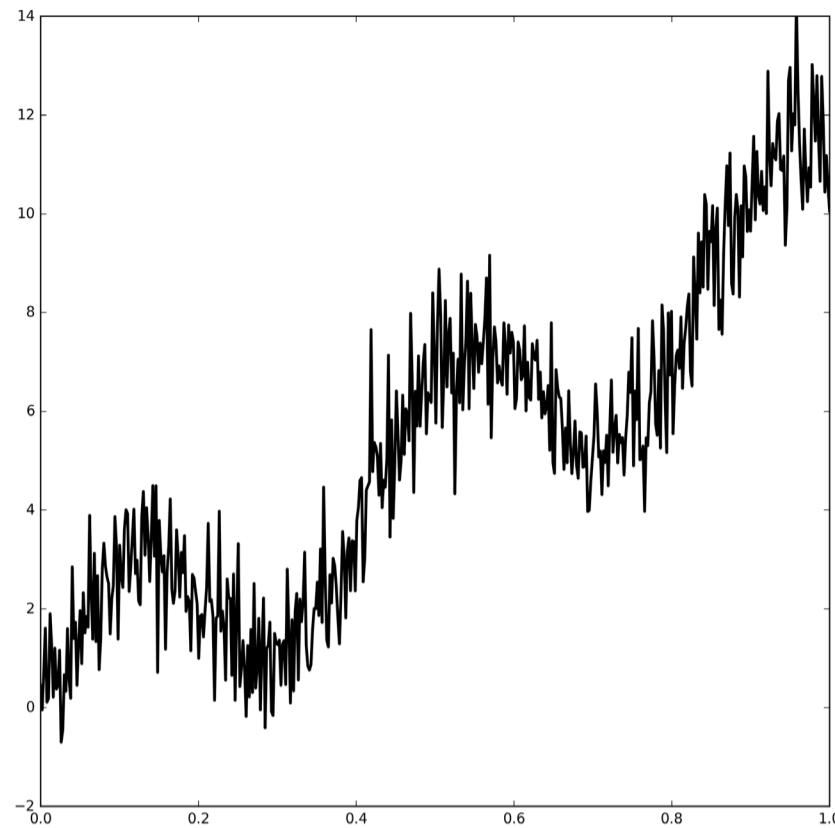


Figure 9.8: Wind speed over time as measured by an atmospheric scientist. In this data there's a clear underlying curve but there's also plenty of noise (base curve after [Macskassy08]).

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 the **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 added together to make Figure 9.8.

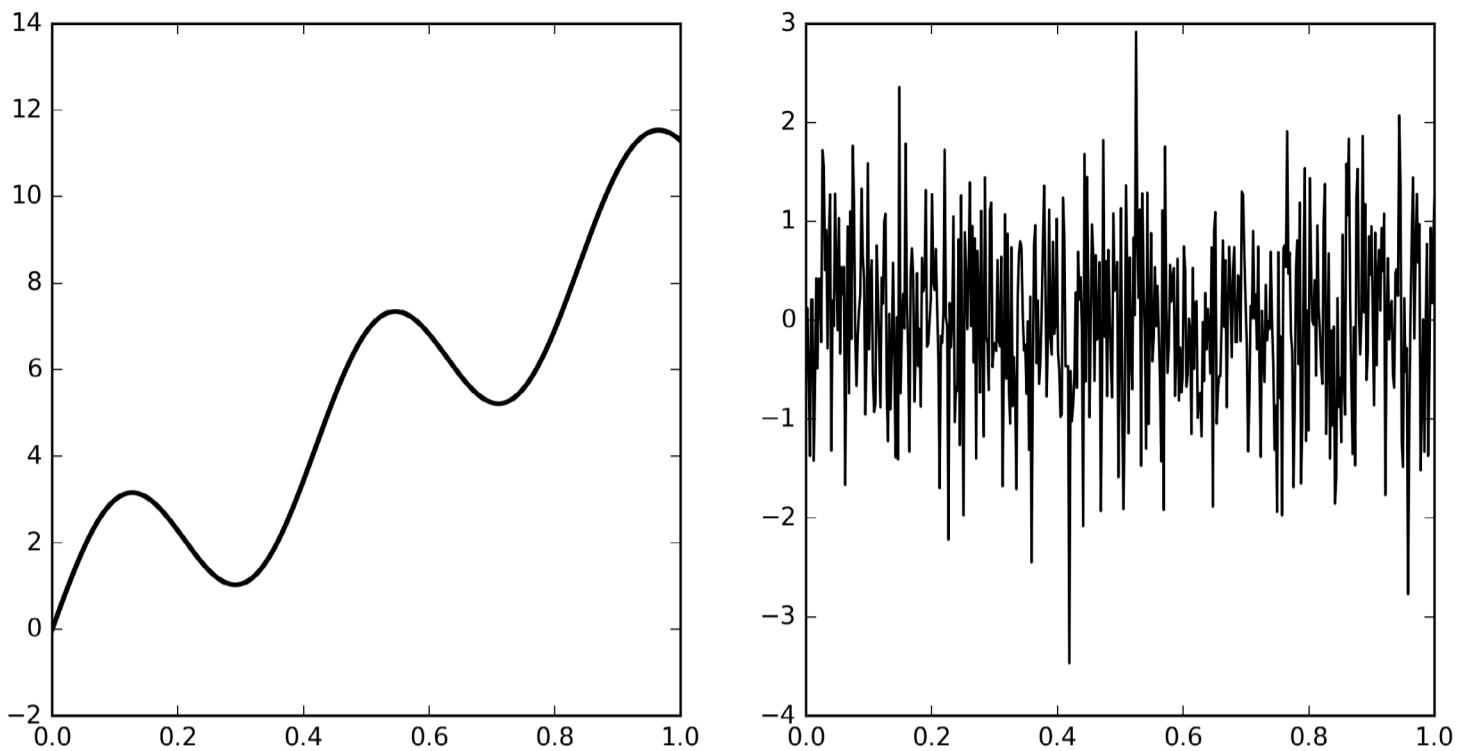


Figure 9.9: The data from Figure 9.8 split into two pieces. Left: the underlying “idealized” curve we seek. Right: The noise which nature added to the idealized curve to give us the noisy, measured data.

She 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, she believes that a split like that in Figure 9.9 describes her data, but she doesn't know the exact values of the noise that's in her data, so she can't subtract them to get the idealized curve.

We could try to fit a curve to the noisy data of Figure 9.8 [Bishop06]. 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 could hope to get a pretty fair match to the general shape of the curve, which would be a good starting point for finding the underlying smooth curve.

Figure 9.10 shows a typical 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 when it runs out of data.

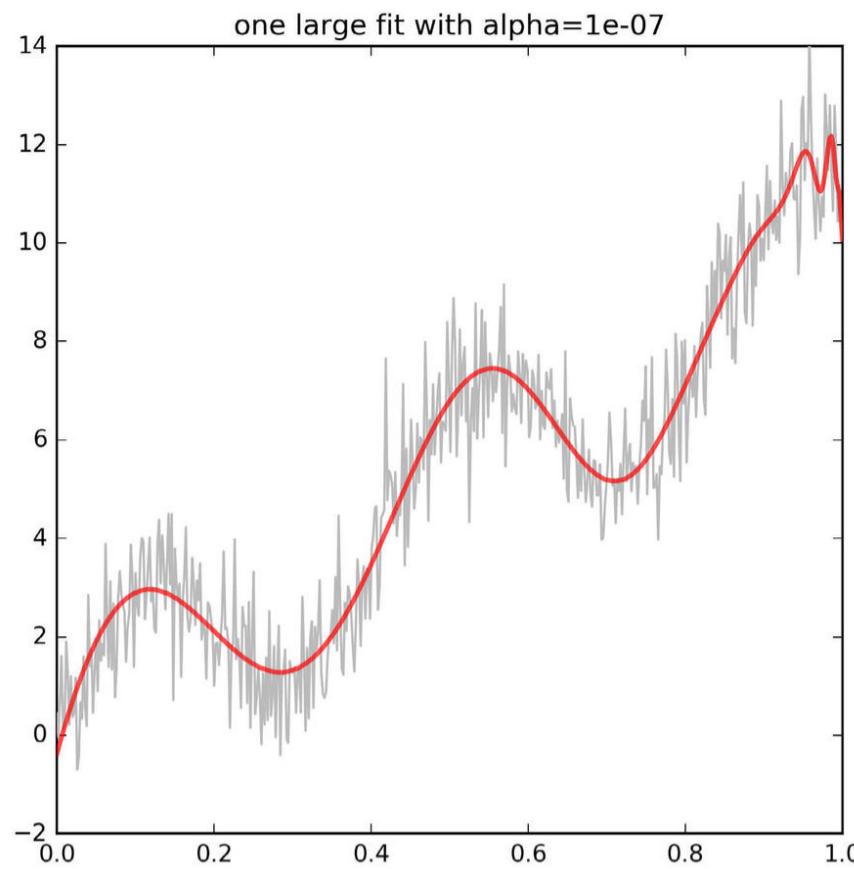


Figure 9.10: Fitting our noisy data with a curve using a curve-fitting algorithm. We used Ridge regression with a regularization parameter value of 0.0000001 (written in scientific notation as 1e-7).

To illustrate the ideas of bias and variance, let's take a different approach. The idea is inspired by the method of bootstrapping that we discussed in Chapter 2, but we won't actually use the bootstrapping technique.

We'll make 50 versions of the original noisy data, but each version will use just 30 points selected randomly, without replacement. So to generate this data, we could just work our way along the noisy data left to right, plucking out 30 points at random. The first 5 of these **subsampled** curves are shown in Figure 9.11.

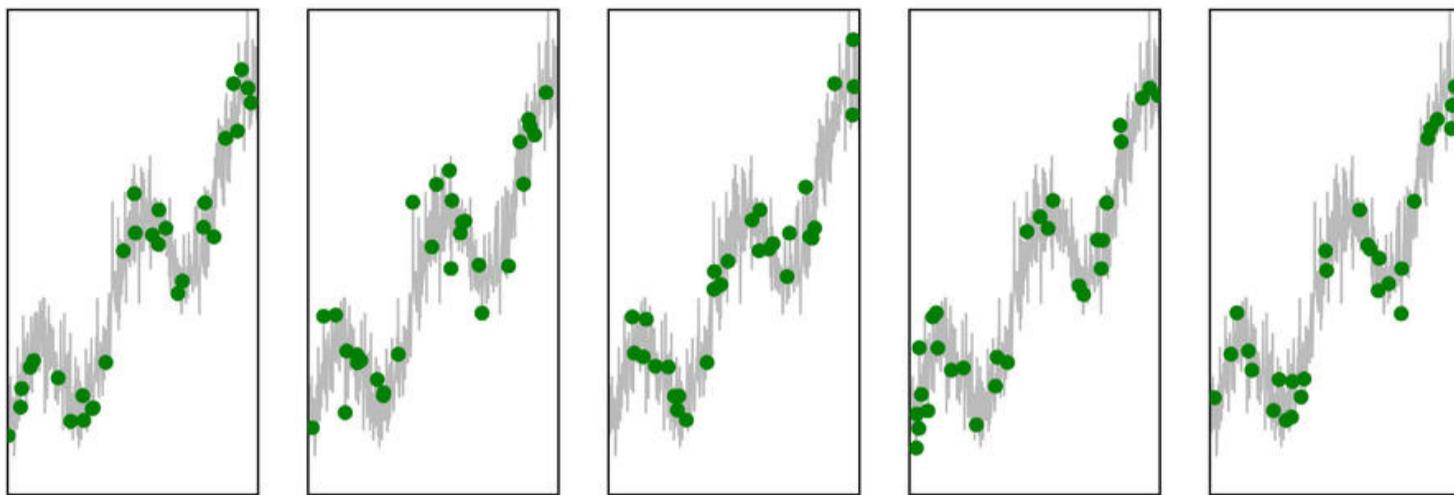


Figure 9.11: We'll make 50 versions of our noisy starting data. Each version consists of 30 samples chosen from the original data. These are the first 5 sets, with the chosen points shown as green dots.

Let's try matching these sets of points with simple curves and with complex curves, and compare the results in terms of bias and variance.

9.6.2 High Bias, Low Variance

We'll first fit our data with simple, smooth curves. We'll pick curves that have a gentle upward bend. Because we're picking out this shape 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.

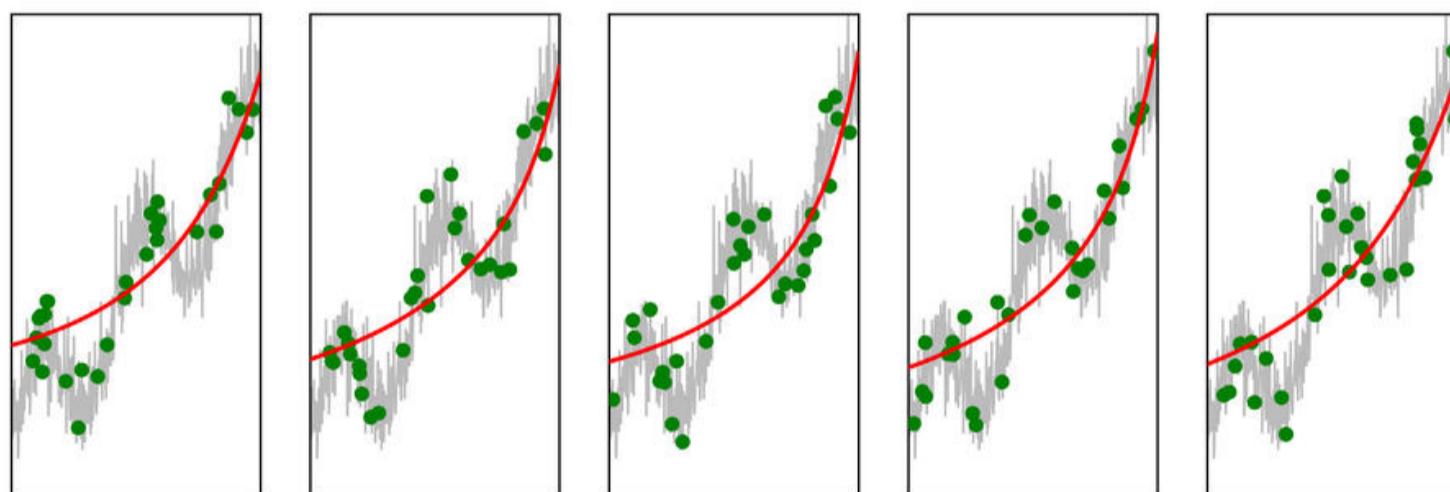


Figure 9.12: Fitting simple curves to our first 5 sets of points in Figure 9.11

As expected, the curves are all simple and similar. Because these curves are so simple, this collection of curves are showing a **high bias**. In other words, they're passing through very few of the green data points. Our chosen shape doesn't give them the flexibility to pass through more than a few points at the most.

To see the variance of the curves, or how much they vary from one to the next, we can draw all 50 curves on top of one another, as in Figure 9.13.

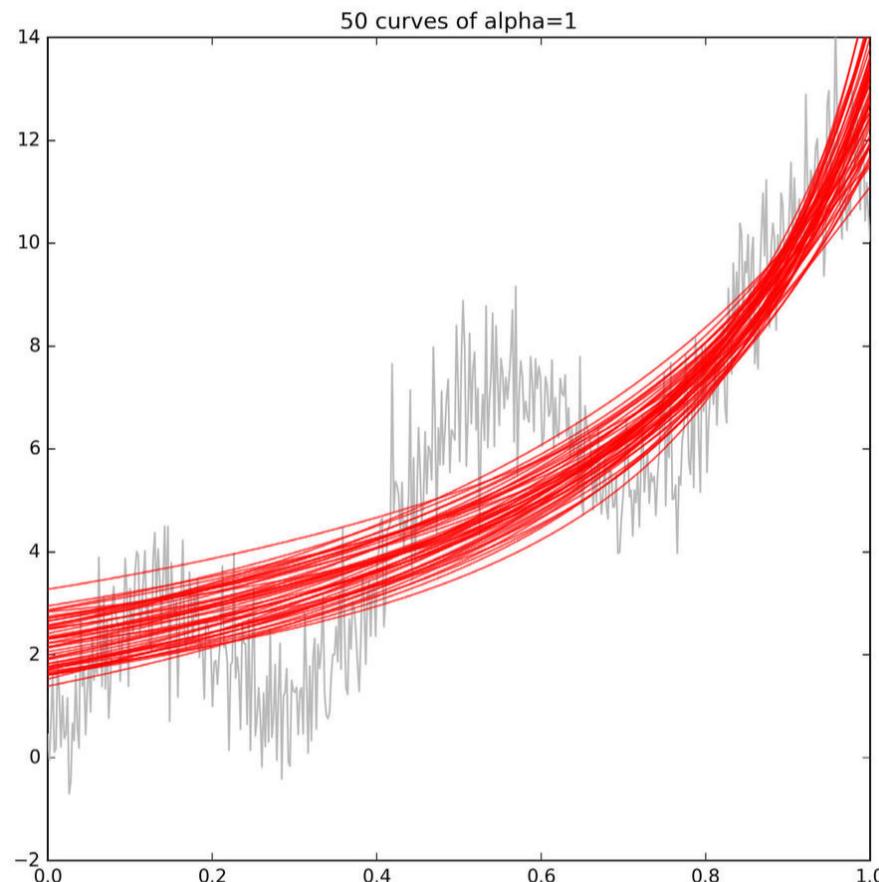


Figure 9.13: The curves for all 50 of our 30-point samples from the original data, overlaid on one another.

As expected, the curves are pretty much all the same. That is, this collection of curves has **low variance**.

9.6.3 Low Bias, High Variance

Now let's try fitting complex curves to our data, so that they come closer to fitting the data.

Figure 9.14 shows more complex curves applied to our first 5 sets of data. Compared to Figure 9.12, the 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.

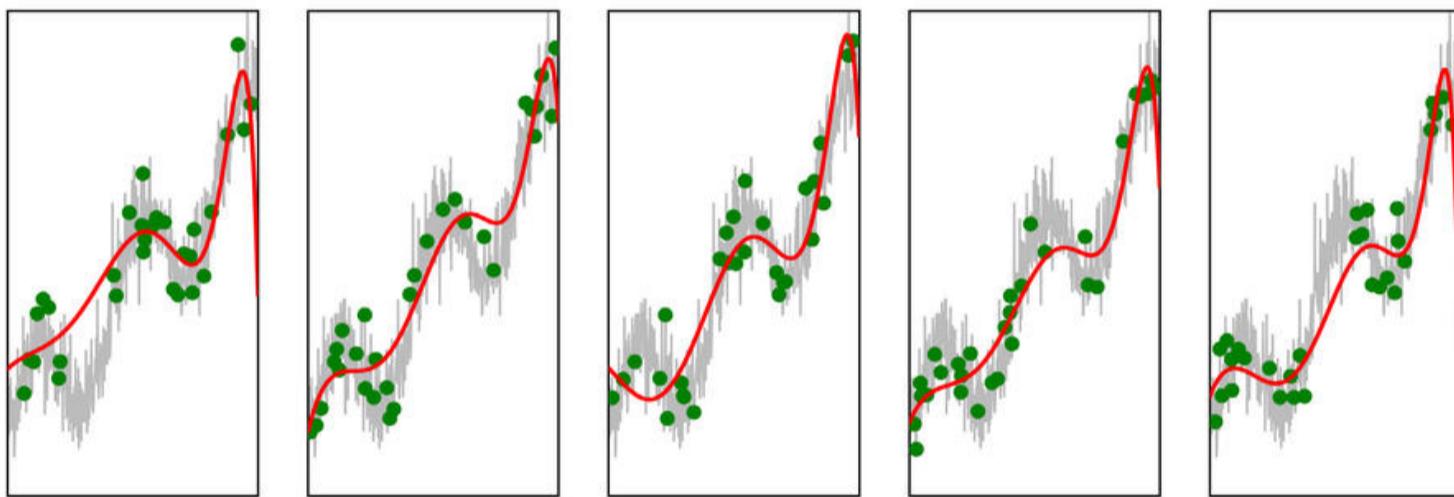


Figure 9.14: The complex curves created for the first 5 sets of points from our noisy data.

Since the shapes of these curves are more complex and flexible, they're more influenced by the data than by any starting assumptions. That is, they have **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 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.

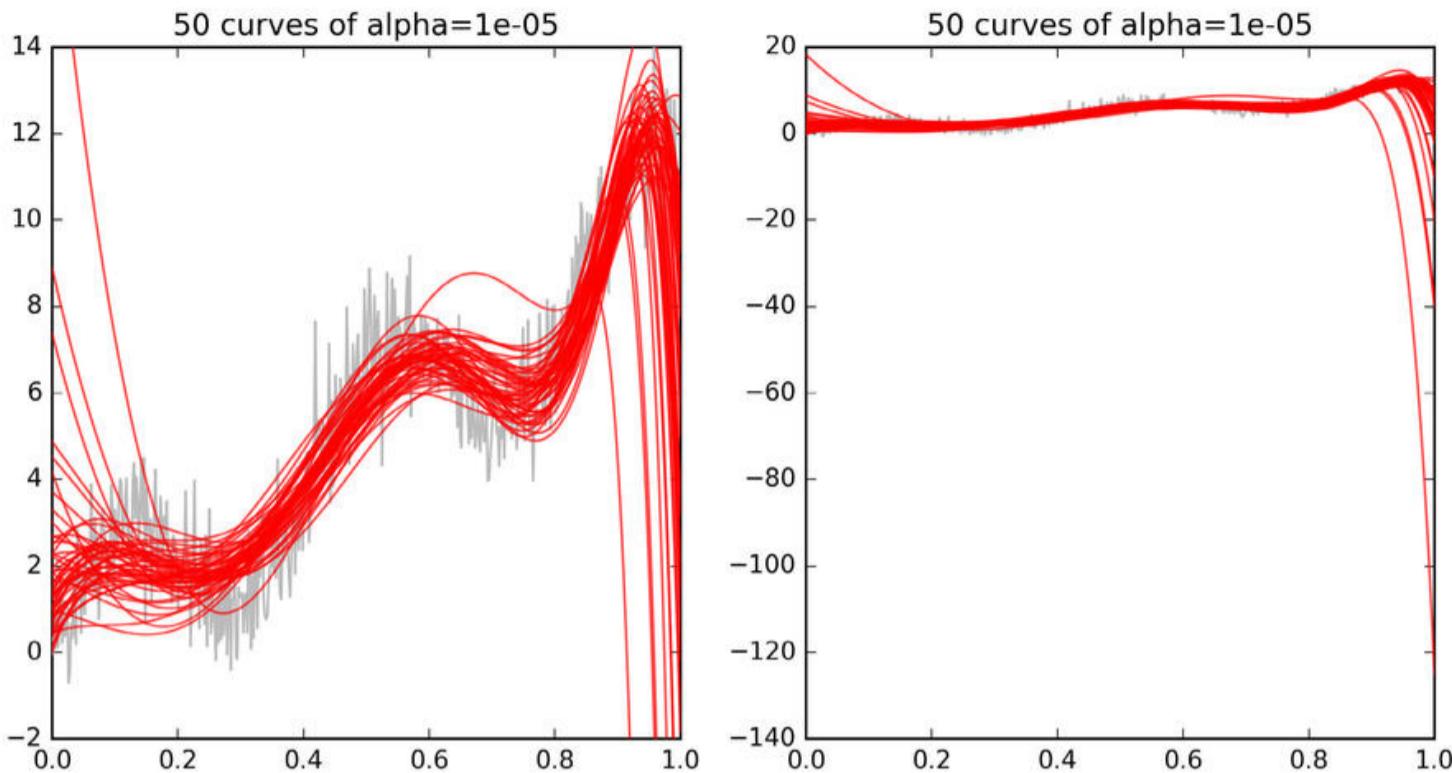


Figure 9.15: The complex curves fit to our 50 sets of data. The right plot shows the entire vertical scale of the curves. To make these curves, we used a regularization value of 0.00001 (written 1e-05 in scientific notation).

These curves are quite different from one another. In other words, they have **high variance**.

9.6.4 Comparing Curves

Our atmospheric scientist asked us for a curve that would match the underlying idealized curve in her data. From what we've seen, the curve in Figure 9.10 would be a good bet (though some more data would help us get rid of that bump at the end).

But to explore bias and variance, we instead created 50 small sets of points, randomly extracted from the 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), regardless of the data they were intended to match. Because they were so similar, 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 adapt 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, we say that set of the 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**.

Notice that these are properties of *families*, or collections, of curves. We can't say much about the bias and variance of a single curve. We have to try fitting a lot of curves to a lot of data to see the effects of bias and variance.

Bias and variance are real phenomena, and references to them show up often in machine learning discussions as a conceptual device for understanding the **complexity** or **power** of a model or algorithm.

For example, suppose we were trying to match our noisy data above using a simple model based on just a few parameters. These parameters are used to describe the curve that the model is creating in hopes of matching the noisy data's smooth, underlying curve. Because the model is simple, it can only produce simple curves, because it just doesn't have enough parameters to represent something more complex.

If we trained the system several times with multiple similar datasets, the system would consistently produce about the same simple curves, so the set of curves would show low variance. Those curves would miss a lot of the data points, so the curves would show high bias. Since the curves made by the system would be high bias and low variance, we sometimes take a little shortcut in language and say that the system itself has high bias and low variance.

We could also get these results from a complex model in the early stages of training, when the curves it's discovering are just starting to match the data. We can say that this model is **underfitting** the training data with curves that are too simple.

Suppose instead that our model is complicated, and capable of making complicated curves.

The curves produced by this model, when trained on multiple datasets, would be quite different as they adapt to the particular data (that is, they'd have high variance), but they'd come close to or even match many of the data points (that is, they'd have low bias). As before, we often speak of these properties of the curves produced by the system as belonging to the system itself, so we'd say this model has low bias and high variance.

We can get these results when a complicated model is allowed to train for too long. These could **overfit** the input, or match the training data too well.

We'd love to describe our data with curves that simultaneously have low bias (so all the points in the dataset are correctly matched) and low variance (so that the curve is not overly-trained on one dataset and thus will generalize nicely).

Unfortunately, in most real-world cases we can't have both. As we saw in Chapter 2, bias and variance are inversely related: when either one goes up, the other goes down, and vice-versa.

That's because when simple curves get wigglier and thus match the data better, and their bias decreases, their variance increases, because curves that match the data well will naturally be different from one another. If we make those complex curves less wiggly, the variance will come down, because they'll be more like each other, but the bias will go up, because they'll be less able to match the data. We can't have low bias and low variance at the same time.

Neither bias or variance is inherently better or worse than the other in general, 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 might be acceptable.

For instance, if we know for sure that our training set is absolutely representative of all future data, then we wouldn't care about the variance and instead would 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 a terrible representative of future data (but it's the best we have at the moment), we might not care about bias, since matching this awful dataset isn't important, and we'd want the lowest variance we could 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 specific goals of any particular project, given the specific learner we've selected and the data we've got.

9.7 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 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 value, but instead we use averaging of all the values to find the "true" answer that each value approximates. Those ideas fit the frequentist approach very well.

By contrast, the Bayesian approach to fitting data considers 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 an absolute answer.

Let's see this in practice. Our discussion is based on a lovely visualization due to Bishop [Bishop06]. We'll use Bayes' Rule to find a nice approximation to the noisy data atmospheric data we used above.

Rather than fit a complicated curve to our data, we'll restrict ourselves to straight lines. That's only because that lets us show everything with 2D plots and diagrams. To show pictures of fitting data with curves, we'd need to draw diagrams in 3, 4, 5, or more dimensions. So for clarity, we'll stick to lines.

In fact, we'll stick to lines that are *mostly horizontal*. Again, this is just so we can draw nice diagrams. Handling any lines at all would require 3D diagrams that would be much harder to interpret.

We'll describe our lines using two numbers: a **slope** and an **intercept**. These are often represented by the letters m and b respectively, but we'll call them by name [MathForum03] [MathForum94].

A line that's perfectly horizontal has a slope of 0. As the line rotates counter-clockwise, the slope increases until it has a value of 1 when the line runs northeast-southwest. As the line rotates clockwise from horizontal, the slope decreases until it has a value of -1 when the line runs northwest-southeast.

As a line becomes more vertical the slope becomes larger and larger, but we'll stick to slopes in the range $[-2, 2]$ here.

The intercept tells us where the line crosses the Y axis. We'll restrict our attention to intercept values that also in the range of $[-2, 2]$.

These ranges give us the ability to represent any line with a slope from -2 to 2, and an intercept from -2 to 2. Figure 9.16 shows some of these lines.

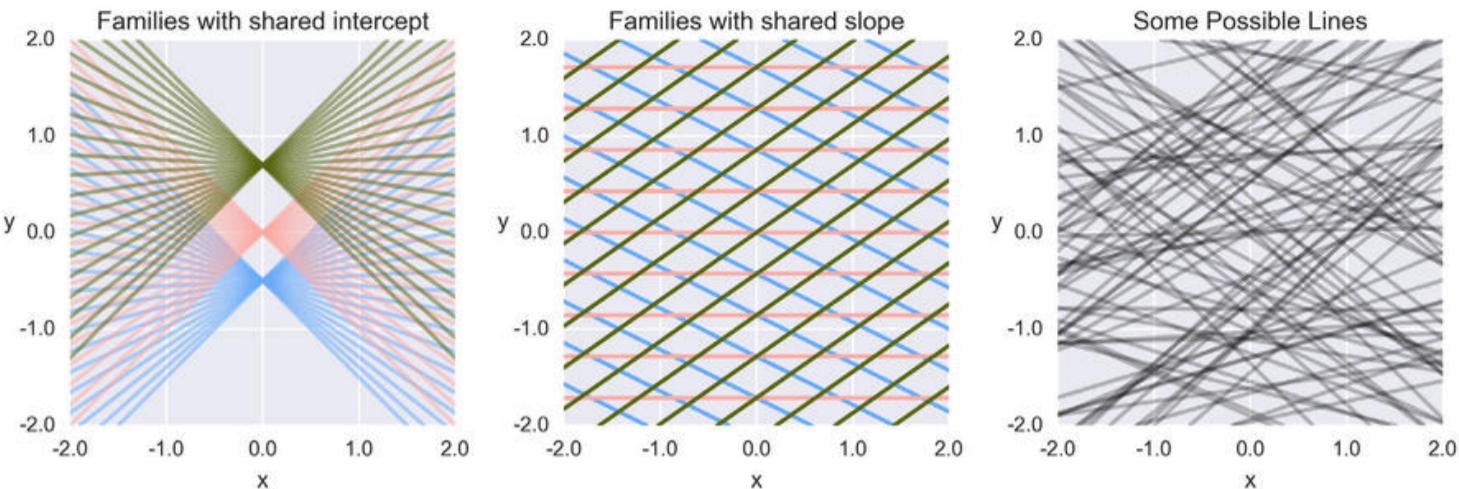


Figure 9.16: The lines we can work with in this discussion. Left: Each family of lines with the same color has the same intercept (the point where the line crosses the Y axis), but different slopes. Middle: Each family of lines with the same color has the different Y intercepts, but the same slope. Right: Some random lines with slopes from -2 to 2 , and intercepts from -2 to 2 .

Because of our limited range of available lines, we'll vertically compress our original noisy data by a little bit so it doesn't rise at such a steep angle. Again, this is just to make it easier to draw and interpret the figures.

We'll be working with multiple lines at once, but drawing lots of lines on top of one another can get confusing. So let's pick another way to represent all of our possible lines. We'll make a grid that has our range of slopes from -2 to 2 running left to right, and our range of Y intercepts from -2 to 2 running bottom to top. Then any point in this 2D box corresponds to a particular line in the ranges we're thinking about. Figure 9.17 shows this idea.

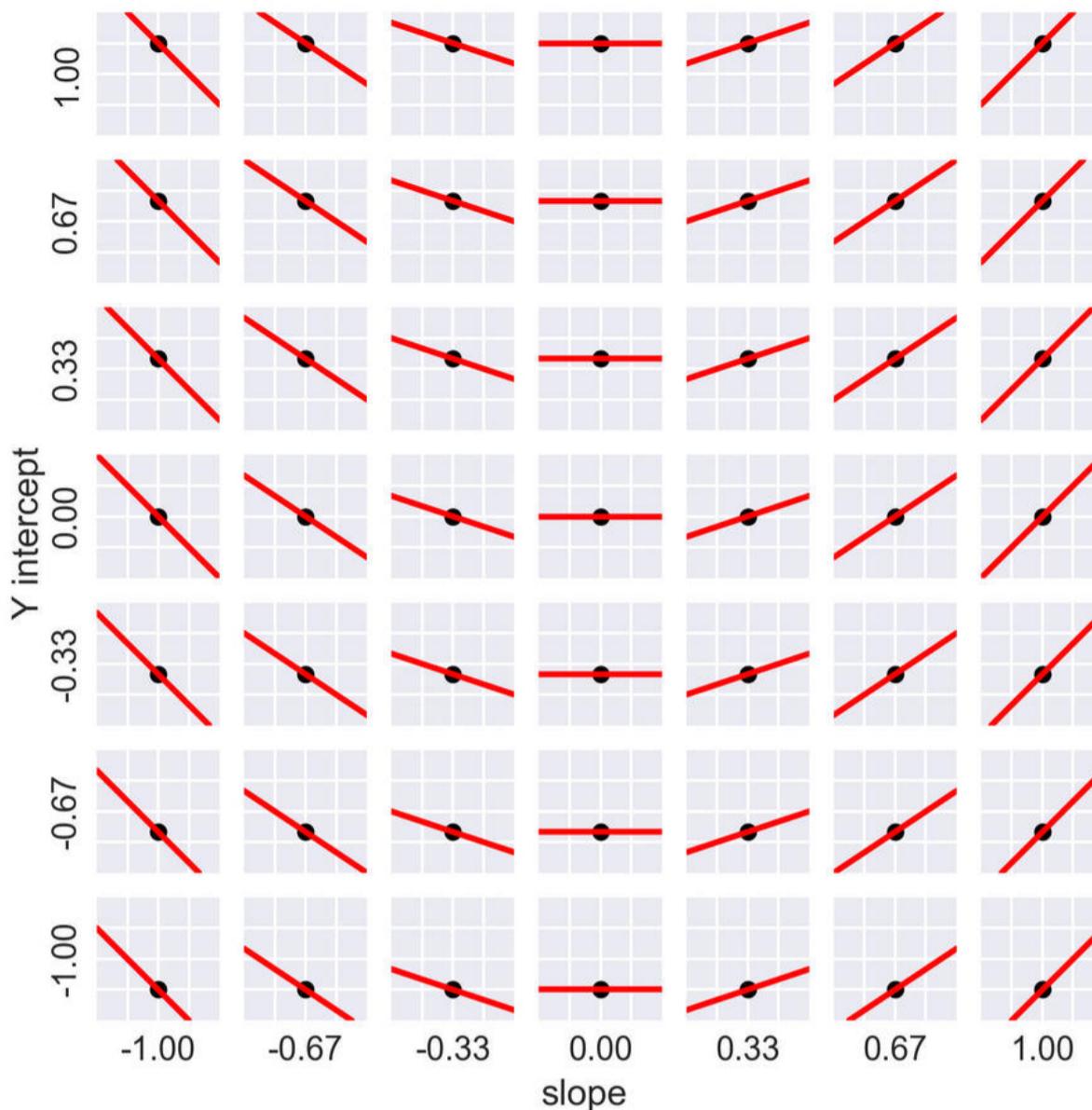


Figure 9.17: We can describe the lines we’re thinking about in terms of their position on a grid with slopes from -2 to 2 running horizontally left to right, and intercepts from -2 to 2 running vertically bottom to top. In this arrangement, each row is like the row beneath it with all lines shifted upwards, and each column is like the column to its left with all lines rotated counter-clockwise around the point where they meet the Y axis.

We have all of our tools set up now. Let’s consider what we want to do with them.

We want to find a straight line that will go through a bunch of noisy data. That means we know for sure that it won’t go through all the data points. But we’d like it to pass through as many points as it can, and come as *near* as possible to as many other points.

Let's consider just a single point from our noisy data. What lines should we consider as good fits for this point? Certainly all the lines that pass through the point are good candidates. Because we know the final line is going to only come near many (if not most) of the points, we should also include lines that only come near this point.

Let's assign a probability to each of these lines. The lines that go directly through our point will be the most probable, but we'll include plenty of nearby lines with lower probabilities too. Then if we later find that we only want to come near this first point, those options are available.

Now comes the trick that lets us use Bayes' Rule to find the best line through the data.

We can show the probabilities for all possible lines we can draw in our system at once using a grayscale version of Figure 9.17, which we'll call a *slope-intercept diagram*. White will mean that line has a very high probability, and darker shades of gray will represent lower probabilities.

Figure 9.18 shows this in action.

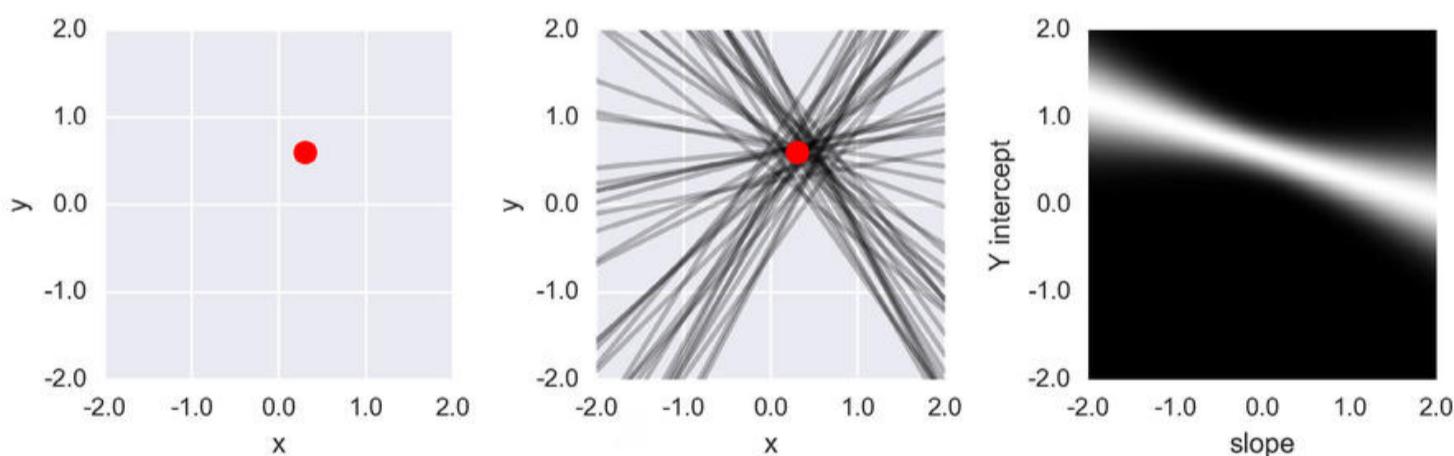


Figure 9.18: Finding the Bayesian likelihood of lines that go through a point. Left: A point at $(0.3, 0.6)$ in a typical (x,y) plane. Middle: Some randomly-chosen lines that go through, or come close to, the red dot. Right: A slope-intercept diagram, where each location (or pixel) represents a line, as in Figure 9.17. Here we've marked the probability of every line on the diagram from 0 (black) to 1 (white). The closer each line comes to passing through the point, the higher its probability.

The rightmost image of Figure 9.18 is the Bayesian likelihood of all lines that we might use to match the red point in the leftmost image. The closer each line comes to passing through the point, the more likely it is, but lines that just come nearby have some possibility, too.

What lines are represented by the rightmost diagram in Figure 9.18? Some of them are shown in Figure 9.19, where we've picked five points in the non-black region. We've plotted each line corresponding to those points along with the original red point from Figure 9.18. Note that the higher the probability of a line, the closer it comes to the red dot.

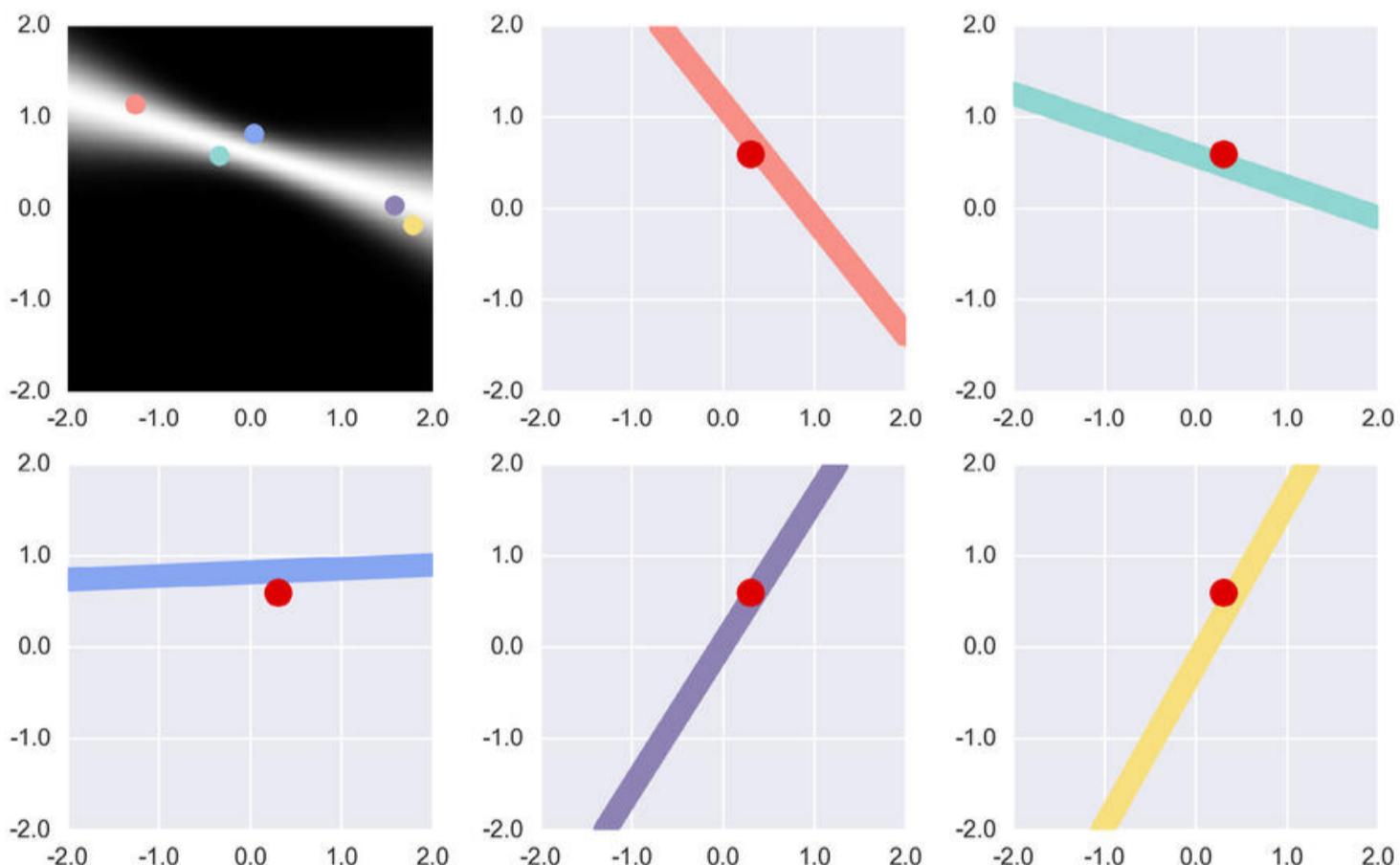


Figure 9.19: The rightmost diagram of Figure 9.18 tells us the probability of each line with slope -2 to 2 , and intercept -2 to 2 . We've repeated that figure in the upper left, along with five dots from the non-black region. The line corresponding to each of these dots is drawn in the rest of the figure, where the line's color shows which dot it corresponds to. Notice that lines with higher probability pass nearer to the original red point at $(0.3, 0.6)$, also shown.

Now we're ready to use Bayes' Rule to match our data! Figure 9.20 shows the process.

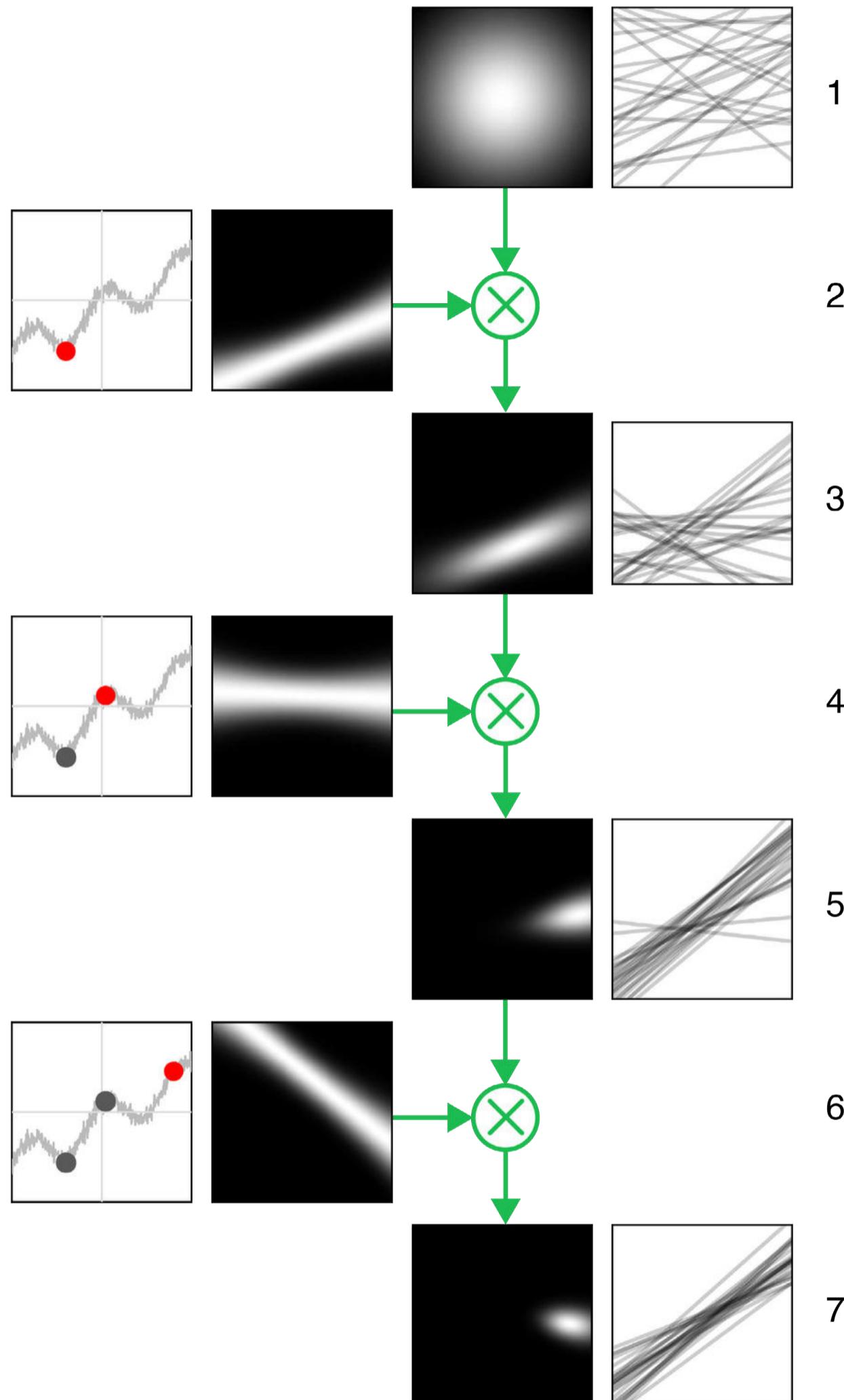


Figure 9.20: Fitting a straight line through our data with Bayes' Rule (figure inspired by [Bishop06]).

In Figure 9.20, the leftmost column shows our noisy dataset, with each new point shown in red. The column to the left of center shows the likelihood for that red point. The column to the right of center shows our prior at the top of the column, and then each figure below it shows the evolving posterior (or prior). Each posterior is used as the prior for the next step. The rightmost column shows some lines pulled at random from the distribution to their left.

In row 1 we show our prior, or our starting guess of the distribution of straight lines that will fit our data. We're using a Gaussian with its center in the middle. This prior means that we're guessing that our data will mostly likely be 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 tried several other priors for this figure, and they all ended up with about the same final results. We stuck with the Gaussian prior here because it produced the most revealing illustrations.

The image on the right of row 1 shows 25 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. We also need to divide by the evidence, but we're leaving that step out of the diagram because in this example it's just book-keeping.

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 each line will fit this piece of data). Notice that the posterior is a new blob. To its right we see another 25 lines drawn at random from

that distribution. We can see a huge empty space near the top of the figure that wasn't there in row 1. The system has learned from this data point that none of those lines near the top of the region are likely.

The posterior from line 3 becomes our new prior for the next data point that comes along.

In row 4 we introduce a new random point, 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 range of lines that will fit both points is smaller than the range that would fit 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, though there are also a few low-probability lines present.

We add another new data point in line 6, with the resulting posterior 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'll get an ever-smaller posterior (or prior), and thus a more tightly limited range of probable lines through all the data.

It might be tempting to look at the lines in the bottom right of Figure 9.20 and apply our ideas of bias and variance to them, but that's not thinking like a Bayesian. In the Bayes framework, these aren't a family of lines that approximate some "true" answer, which we can discover by using various forms of averaging. Instead, a Bayesian sees all of these lines as potentially accurate and correct, but with different probabilities. Using averages and distances to compute the bias and variance of lines drawn from this collection is possible, but not meaningful.

Both the frequentist and Bayesian approaches let us fit lines (or curves) to data. They just take very different attitudes and use different mechanisms, so we can't always use the same tools to evaluate the results of both approaches.

References

- [Bishop06] Christopher M. Bishop, “Pattern Recognition and Machine Learning”, Springer-Verlag, 2006, pp. 149-152
- [Bullinaria15] John A. Bullinaria, “Bias and Variance, Under-Fitting and Over-Fitting”, Neural Computation lecture notes, University of Birmingham, 2015. <http://www.cs.bham.ac.uk/~jxb/INC/l9.pdf>
- [Domke08] Justin Domke, “Why does regularization work?”, Justin Domke’s Weblog, 2008. <https://justindomke.wordpress.com/2008/12/12/why-does-regularization-work/>
- [Domingos15] Pedro Domingos, “The Master Algorithm”, Basic Books, 2015
- [Duncan15] Brendan Duncan, “Bias, Variance, and Overfitting”, Machine Learning Overview part 4 of 4, 2015 <http://blog.fliptop.com/blog/2015/03/02/bias-variance-and-overfitting-machine-learning-overview/>
- [Foer12] Joshua Foer, “Moonwalking with Einstein: The Art and Science of Remembering Everything”, Penguin Books, 2012.
- [Ioffe15] Sergey Ioffe, Christian Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, arXiv 1502.03167, 2015. <https://arxiv.org/abs/1502.03167>

- [Macskassy08] Sofus A. Macskassy, “Machine Learning (CS 567) Notes”, 2008. <http://www-scf.usc.edu/~csci567/17-18-bias-variance.pdf>
- [MathForum03] Math Forum, “Why b for Intercept?”, 2003. <http://mathforum.org/library/drmath/view/65307.html>
- [MathForum94] Math Forum, “Why m for Slope?”, 1994. <http://mathforum.org/library/drmath/view/52477.html>
- [Proctor78] Philip Proctor and Peter Bergman, “Brainduster Memory School,” from “Give Us A Break,” Mercury Records, 1978. [https://en.wikipedia.org/wiki/Give_Us_a_Break_\(Proctor_and_Bergman_album\)](https://en.wikipedia.org/wiki/Give_Us_a_Break_(Proctor_and_Bergman_album)) Sketch audio at <https://www.youtube.com/watch?v=4o43EFd5fho>
- [Srivastava14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, JMLR, 15(Jun), pgs. 1929–1958, 2014. <http://jmlr.org/papers/v15/srivastava14a.html>



Chapter 10

Neurons

How real biological neurons inspired the artificial neurons we use in machine learning, and how those little bits of processing work alone and in groups.

Contents

10.1 Why This Chapter Is Here	376
10.2 Real Neurons	376
10.3 Artificial Neurons.....	379
10.3.1 The Perceptron	379
10.3.2 Perceptron History.....	381
10.3.3 Modern Artificial Neurons	382
10.4 Summing Up.....	390
References	390

10.1 Why This Chapter Is Here

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 the brain, and are largely responsible for our cognitive abilities.

In this chapter we'll look very briefly at real neurons, and then look at the structure of the highly simplified artificial neurons that are based on them.

It's important to note that not all machine learning algorithms depend on these artificial neurons. Many important and useful techniques are based on traditional forms of analysis, and use direct mechanisms, such as explicit equations, to find a solution for us. No neurons are needed or used.

Artificial neurons come into their own when we look at more general kinds of learning systems, such as the **neural networks** that make up **deep learning** systems we'll see starting in Chapter 20. In those systems, artificial neurons are indispensable.

Because of their critical role in those algorithms, knowing about these artificial neurons is important to understanding how to design, build, teach, and use modern deep learning systems.

10.2 Real Neurons

A key element of our human thinking apparatus is a particular kind of nerve cell called a **neuron**.

The term “neuron” is applied in biology to a wide variety of rich and complex cells distributed throughout every human body. These cells all have similar structure and behavior, but they are specialized for different tasks. They are complex and sophisticated pieces of biology, using chemistry, electricity, timing, proximity, and other means for controlling their behaviors and communication [Julien11] [Lodishoo] [Mangels03] [Purves01]. A structural sketch of a neuron is shown in Figure 10.1.

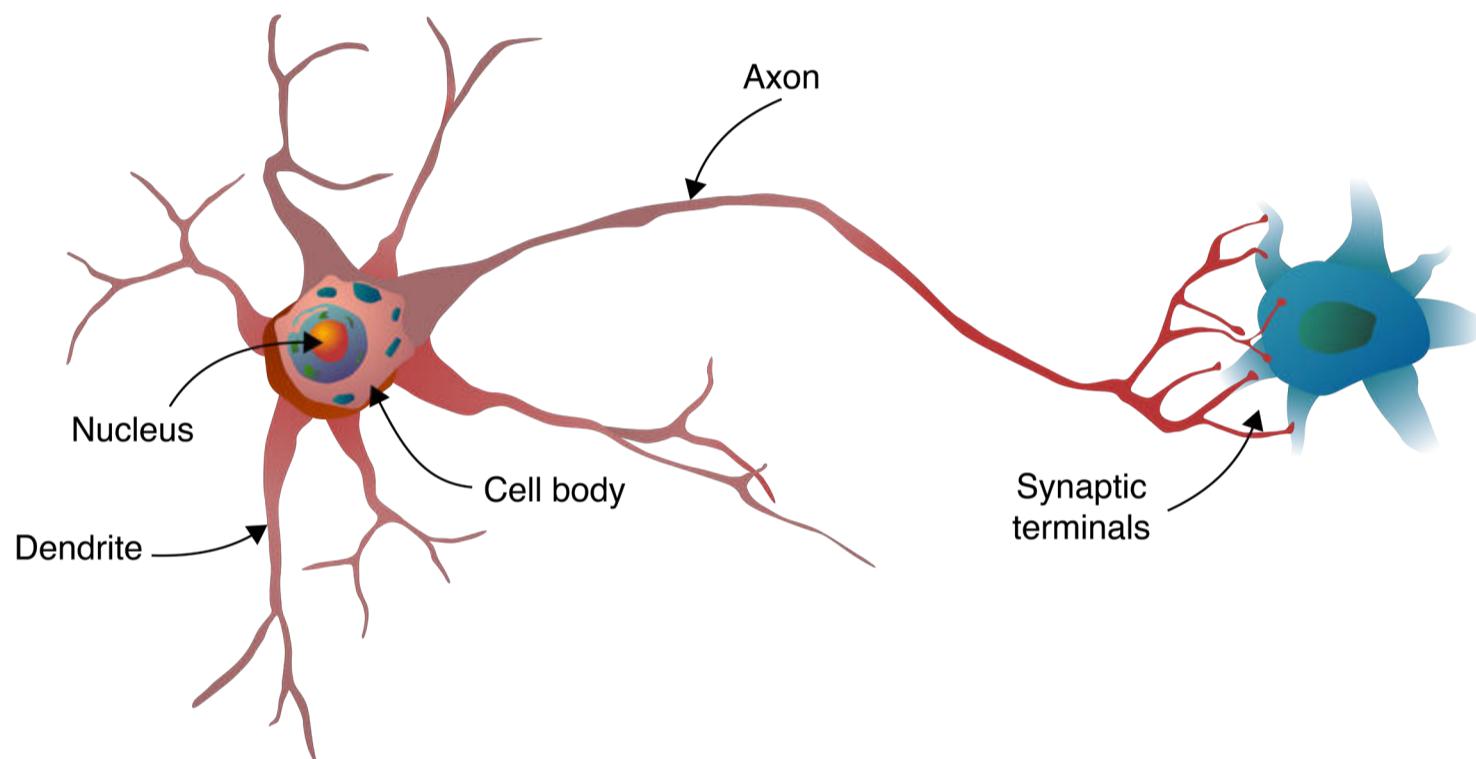


Figure 10.1: A sketch of a biological neuron (in red) with a few major structures identified. This neuron’s outputs are communicated to another neuron (in blue), only partially shown (Image after [Wikipedia17b]).

Neurons are information processing machines. Information arrives in the form of chemicals called **neurotransmitters** that temporarily **bind**, or attach, onto **receptor sites** on the neuron itself [Goldberg15]. These attachments cause electrical signals to travel into the body of the neuron. Each of these electrical signals can be either positive or negative, depending on a wide range of factors. 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 electrical signal is sent to another part of the neuron,

where specific new neurotransmitters are 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 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 there is usually no physical overlap. There is some evidence that each individual's particular pattern of connections between neurons is as essential to cognition and identity as the neurons themselves [Sporns05] [Seung13]. A map of an individual's neuronal connections is called their **connectome**. Connectomes are as unique as fingerprints or iris patterns.

It is fascinating to realize that the constant flow of electrical and chemical signals among billions of neurons is an essential part of our entire mental world, enabling phenomena such as consciousness, intelligence, emotions, thoughts, a sense of self, a sense of others, and language. Many people believe that some or all of these qualities also require a physical body with senses in order to emerge [Wilson16].

Although neurons and their surrounding environment are electro-chemically complex and subtle, the basic mechanisms described above have 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, and hoping that interesting behavior will emerge [Furber12] [Timmer14]. So far this has not delivered results that most people would call intelligence.

But there are specific ways that we can connect up neurons that have proven to produce great results on a wide range of problems. Those are the types of structures we'll discuss in this book.

10.3 Artificial Neurons

The “neurons” we use in machine learning are inspired by real neurons in the same way that or 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.

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 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 to instead 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’ll use them in this book as well.

10.3.1 The Perceptron

The history of artificial neurons may be said to begin in 1943, with a seminal paper by a pair of psychiatrists with a mathematical bent. They presented a simple mathematical abstraction of a neuron’s basic functions, and described how these models could be connected into a **network**, or **net**, inspired by real neurons. The big contribution of this work was that they proved mathematically that “for any logical expression satisfying certain conditions, one can find a net behaving in the fashion it describes” [McCulloch43].

In other words, neurons could perform mathematical logic. And mathematical logic is the basis of machine calculation, which means that neurons could perform mathematics.

This was a big deal, because it provided a bridge between the fields of math, logic, the biology of neurons, and brains.

Building on that insight, in 1957 the **perceptron** was proposed as a simplified mathematical model of a neuron [Rosenblatt62]. Figure 10.2 gives a block diagram of a single perceptron with 4 inputs.

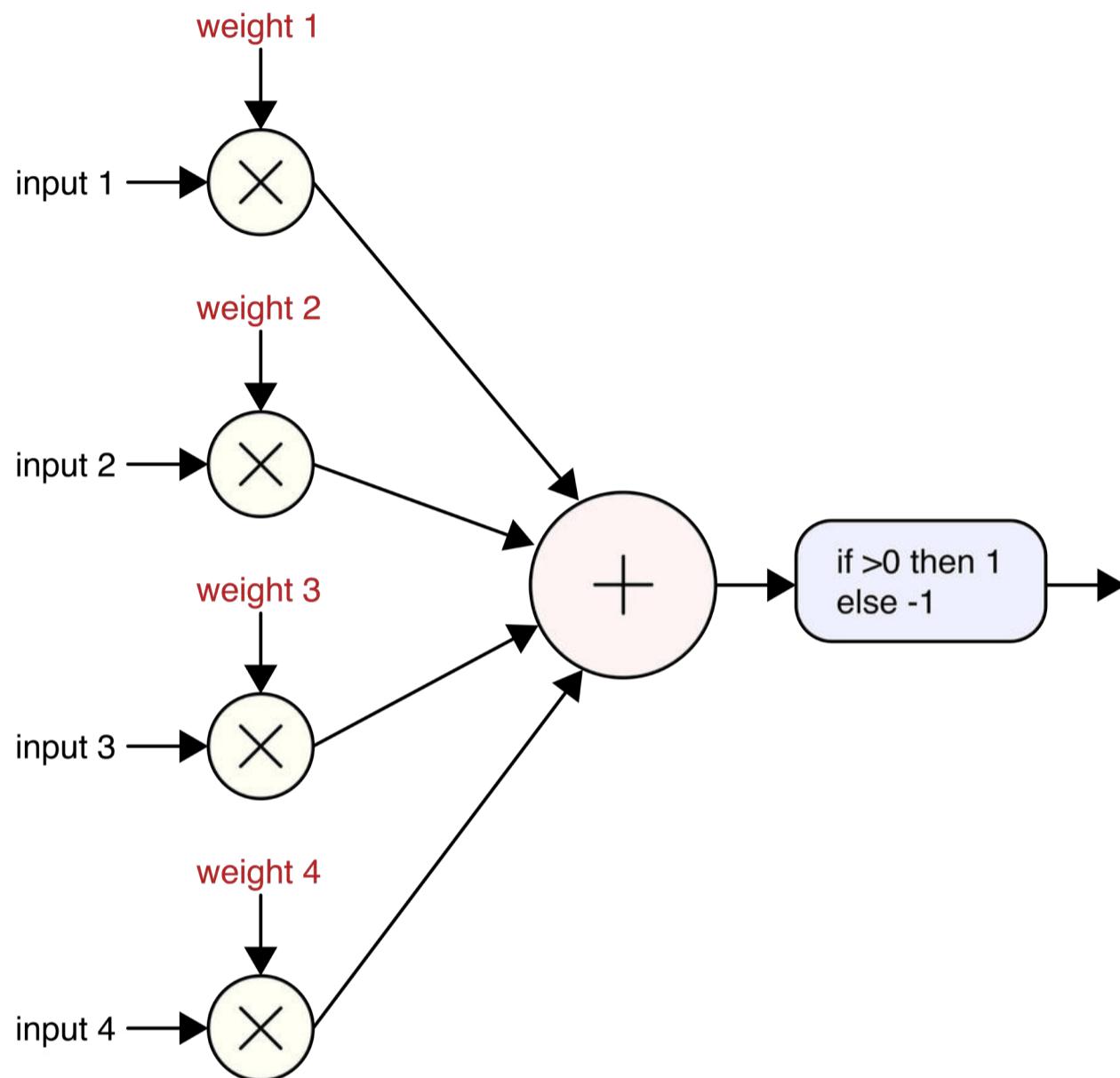


Figure 10.2: A schematic view of a perceptron. Each input is a single number, and it's multiplied by a corresponding real number called its weight. The results are all added together, and then tested. If the results are positive, the perceptron outputs +1, otherwise -1.

The workings of a single perceptron mimic the functional behavior of a neuron. A perceptron has a number of **inputs**, each a single floating-point number. Each of these inputs is multiplied by a corresponding floating-point number called a **weight**. The results of these multiplications are all **summed**, or 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 has proven to be a terrific building block for machine learning systems. The basic mechanism has persisted, though we now usually replace the final threshold with something a little more complicated. We'll review a number of those replacements in Chapter 17.

10.3.2 Perceptron History

The history of the perceptron is 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 [Estebon97] [Wikipedia17a]).

After the principles of the perceptron had been verified in software, a perceptron-based computer was built at Cornell University. It was a rack of wire-wrapped boards the size of a refrigerator, and called the Mark I Perceptron [Wikipedia17c].

The device was built to process images. It had 400 photocells which could digitize an image at a resolution of 20 by 20. The weight on each input of each 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 images 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 was published that proved that the original perceptron technique was fundamentally limited. It showed that the lack of progress wasn't due to a lack of imagination, but because

of built-in theoretical limits in the structure of the perceptron. Most interesting problems, and even some very simple ones, were provably beyond the ability of a perceptron to solve [Minsky69].

This result seemed to many people to close the door on perceptrons, 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.

But the book had only shown that perceptrons were so severely limited when they were used the way they'd always been used. Some people thought that writing off the whole idea was an over-reaction, 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 [Rumelhart86]. These combinations easily surpassed the limitations of any single unit. A series of papers were published that 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.

10.3.3 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 often called “perceptrons,” but there’s rarely any confusion because the new versions are used almost exclusively. More commonly, they’re just called **neurons**.

The first change to the original perceptron is that we provide each neuron with one more input, which we call the **bias**. This is a number that does not come from the output of a previous neuron. Instead, it’s a

number that is directly added into the sum of all the weighted inputs. Each neuron has its own bias. Figure 10.3 shows our original perceptron, but with the bias term.

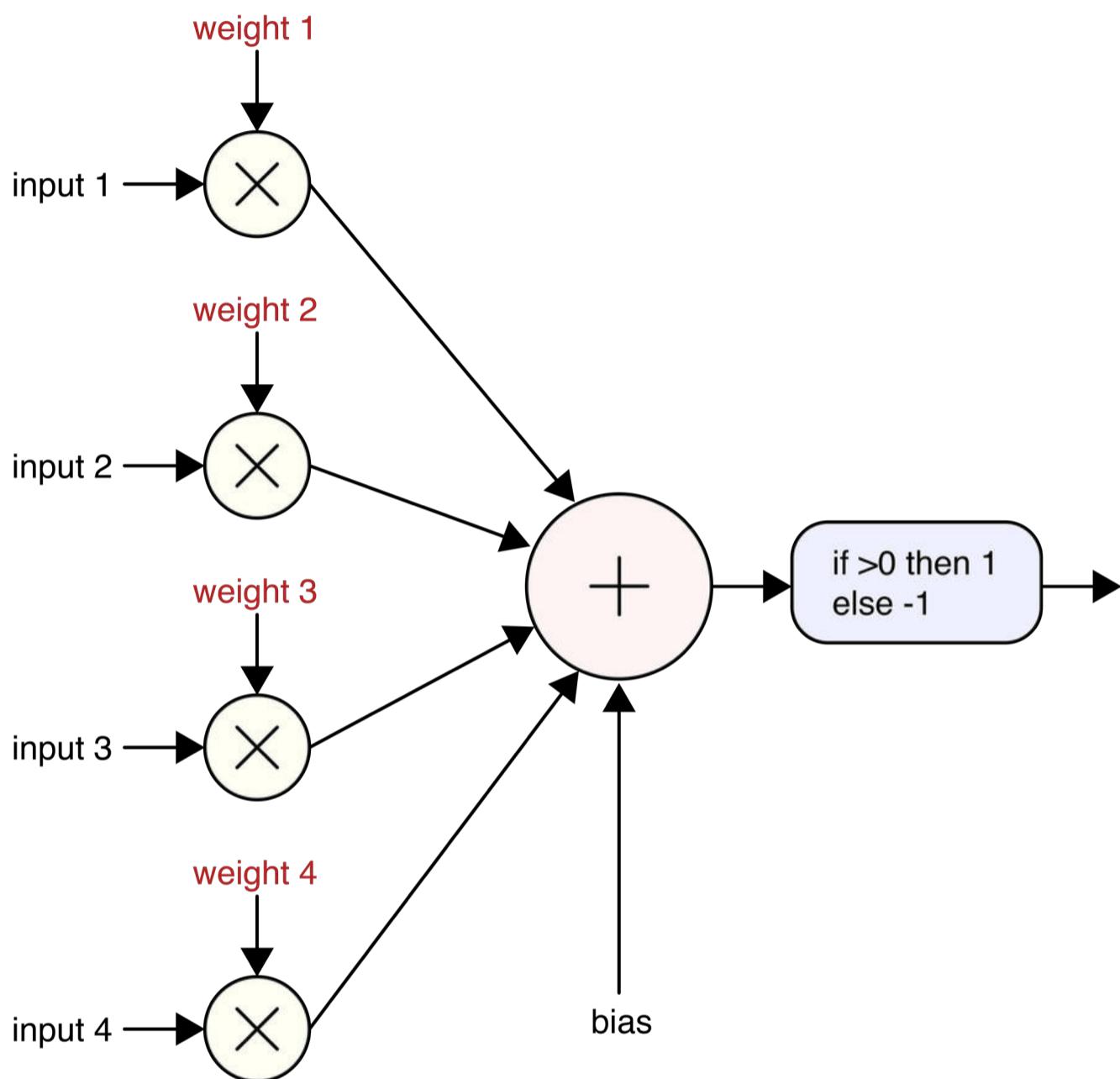


Figure 10.3: The perceptron of Figure 10.2, but now with a bias term. This is a single number that is added to the summed results of the weighted inputs.

Our other change to the perceptron is at the output. Recall that the original perceptron tests the sum against a threshold, and then produces either a -1 or 1 (or 0 or 1). To generalize this, we replace that entire step of test-and-output with a mathematical **function** that takes the sum (including the bias) as input and returns a new floating-point value as output. We call this the **activation function**. The

output of the activation function might take on any value. There are a variety of functions that are popular, each with its own pros and cons. We'll run through them in Chapter 17.

Let's look for a moment at how we draw our neurons, and identify a convention that is used by most of these drawings. In Figure 10.3 we showed the weights explicitly, and also included little multiplication nodes 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 are implied**.

This is important, and bears repeating.

In neural network diagrams, the weights, and the nodes where they multiply the inputs, are not drawn. Instead, we're supposed to know that they are there, and mentally add them to the diagram. **The weights are always there, and they always modify the inputs.** They're just not drawn.

Instead, if we show the weights at all, we typically label the lines from the inputs to the node that sums them up with the name of the weight. Figure 10.4 shows Figure 10.3 drawn in this style.

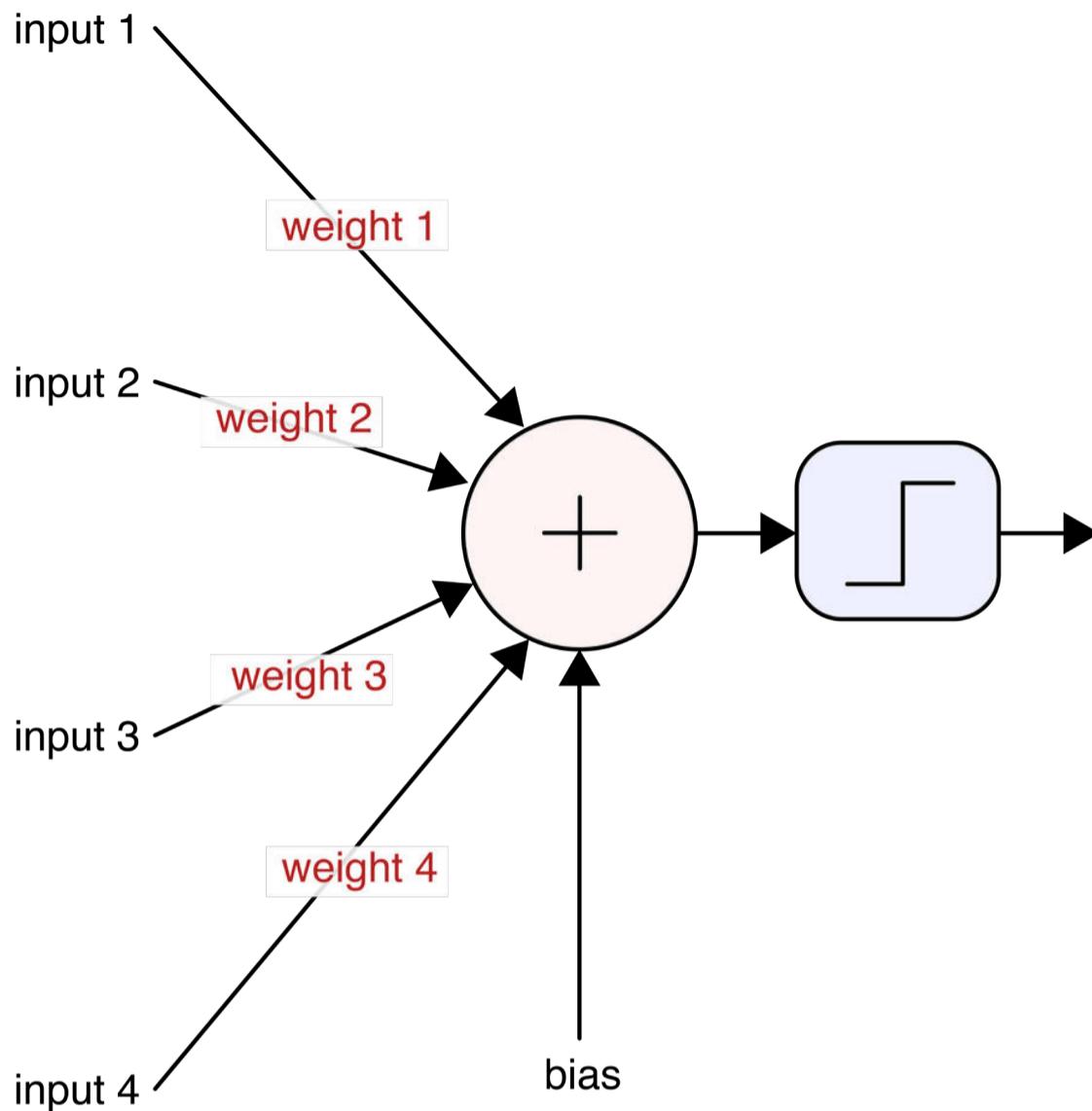


Figure 10.4: A neuron is often drawn with the weights on the arrows. This “implicit multiplication” is common in machine learning figures. We’ve also replaced the threshold function with a step, to remind us that any activation function can follow the summation step.

In this figure, we’ve changed the threshold step 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 box, and a new number comes out, determined by whichever function we select for the job.

We can simplify things again. This time we’ll 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. Because relabeling the bias as an input seems a little sneaky (although it’s perfectly fine to do), this step 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). So the bias is considered to be just another input, like all the others, and this it is also multiplied by a weight. Figure 10.5 shows this change in labeling.

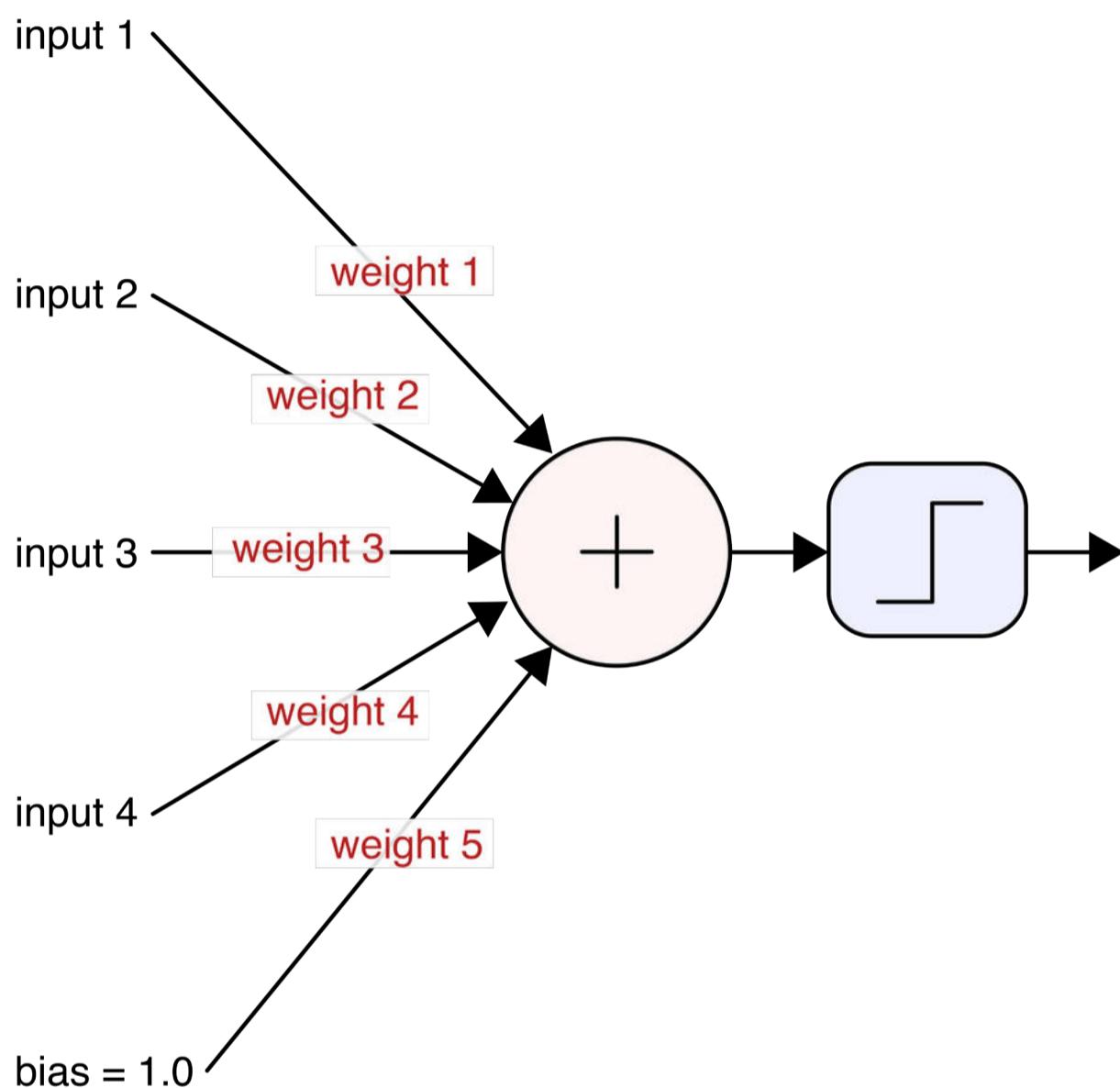


Figure 10.5: The “bias trick” in action. Rather than show the bias term explicitly, as in Figure 10.4, we pretend it’s another input with its own weight.

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 explicitly. We’re supposed to remember that the bias is included (along with its weight), but it’s not shown.

Second, as we mentioned earlier, the weights are often omitted as well. Remember that even though we don't show their names, **the weights are there**. We are supposed to keep in mind that every input to an artificial neuron gets multiplied by its corresponding weight, even when we don't show those weights, as in Figure 10.6.

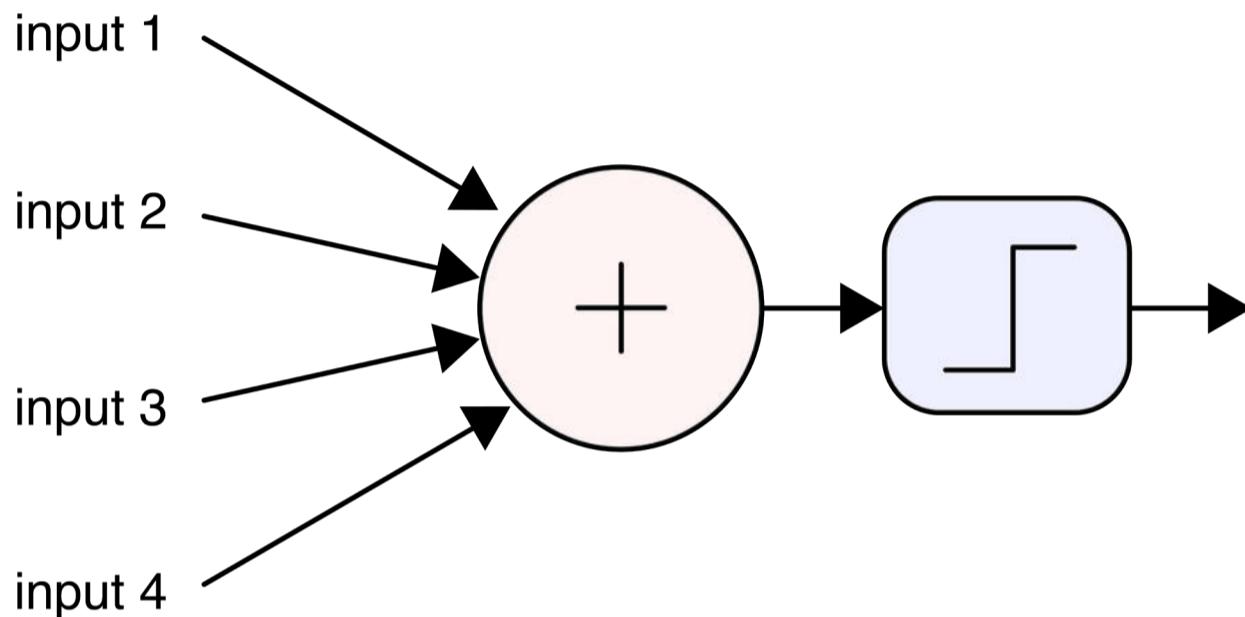


Figure 10.6: A typical drawing of an artificial neuron. The bias term and the weights are not shown, but the bias and weights are in there. We are supposed to add them back in mentally.

When we connect neurons together into networks, we draw “wires” to connect one neuron’s output to one or more other neuron’s inputs.

Like real neurons, artificial neurons can be wired up in dense networks, where each input comes from the output of another neuron. Figure 10.7 shows this idea visually.

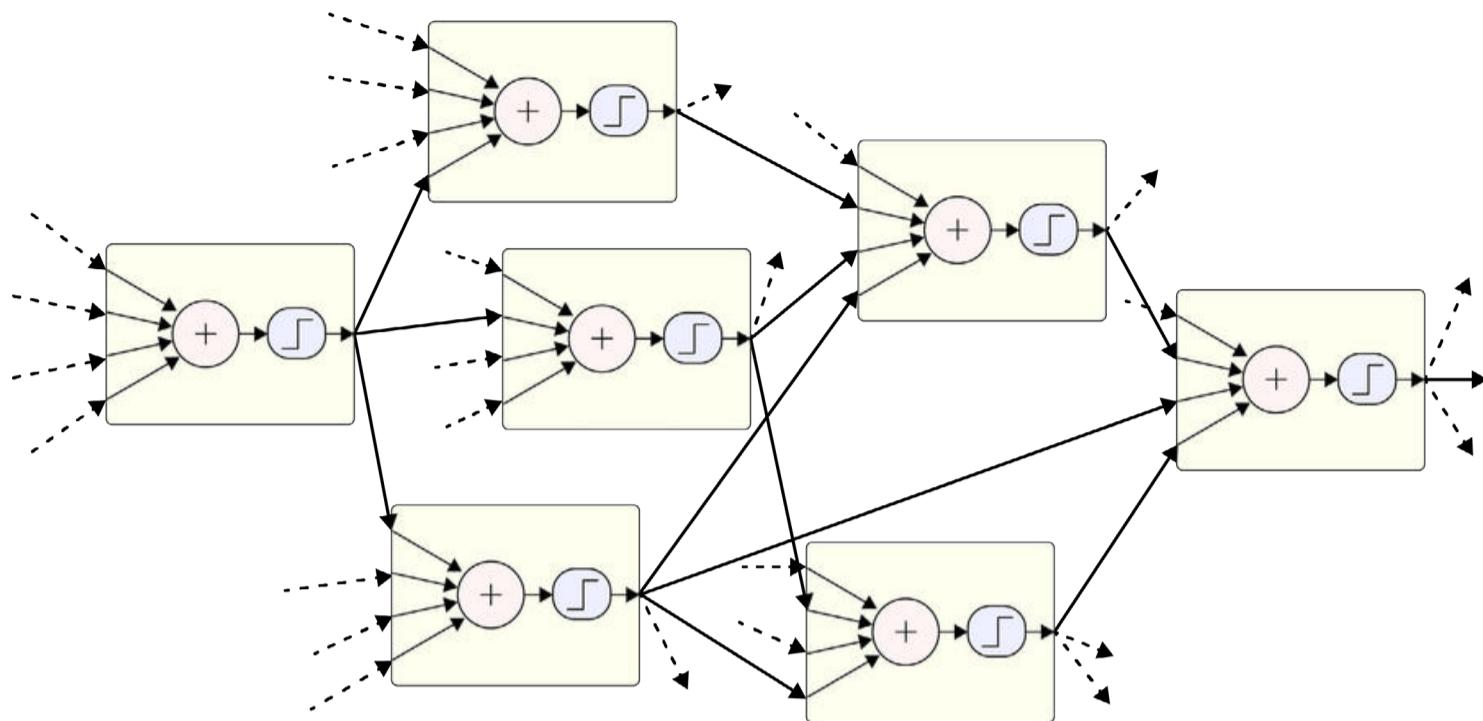


Figure 10.7: A piece of a larger network of artificial neurons. Each neuron receives its inputs from other neurons. The dashed lines show connections coming from outside this little cluster.

Usually the goal of a network of neurons like Figure 10.7 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 usually don't draw the weights in a diagram like Figure 10.7, sometimes it's useful in discussions to refer to individual weights. To make this easy, a convention has developed where each weight is given a two-character name.

For example, in Figure 10.8 we show six neurons. For convenience, we've labeled each neuron with a letter. To name a weight we just combine the name of the output neuron and the name of the input neuron for the wire associated with that weight. For example, the weight that multiplies the output of A before it's used by D is called AD.

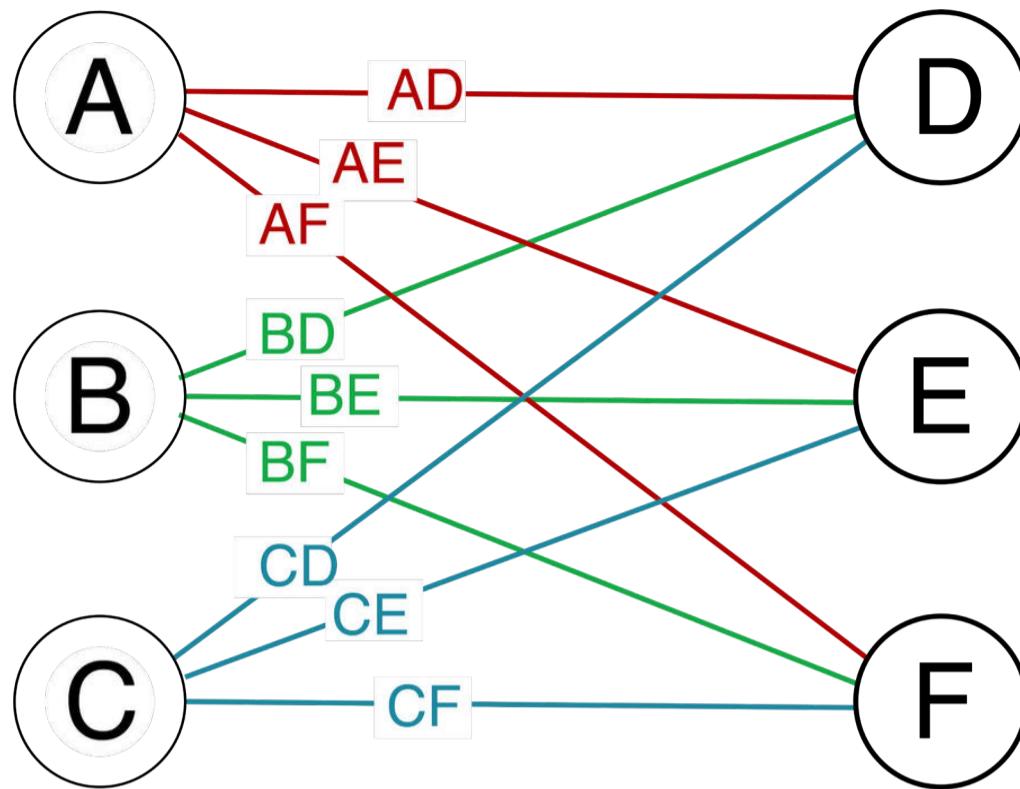


Figure 10.8: The weights are shown explicitly in this diagram. Following convention, each weight is given a two-letter name formed by combining the names of the neuron that produced the output on that weight's wire, with the name of the neuron that receives the value as input. For example, BF is the weight that multiplies the output of B for use by F.

From a structural point of view, it makes no difference whether we place the weights inside the neuron, or on the wires that carry values to it. Various authors assume one case or the other if it makes their discussion easier to follow, but we can always take the other viewpoint if we like.

In Figure 10.8 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 can write the equations. It's always worth a moment to check which order is being used in diagrams like this.

10.4 Summing Up

Real neurons are complicated electrochemical nerve cells. They accept chemical inputs, turns those into electrical charges, and then add the charges. If the charge is above a threshold, the neuron releases new chemicals that travel to other neurons. In this way neurons receive messages from many other neurons, and transmit their messages to many others.

Researchers found that the basic functions of neurons can be used to represent logic statements. From there, it's a short step to using neurons to perform mathematics. This inspired research in artificial neurons.

Today's artificial neurons accept as input a bunch of numbers from other neurons, as well as a number called the bias, scale each of these by a corresponding weight, and add them together. That result is run through a little mathematical formula called an activation function which computes a new number that serves as the neuron's output. That output will usually then serve as an input to other neurons. Taken together, these interconnected neurons form a neural network.

References

[Estebon97] Michele D. Estebon, “Perceptrons: An Associative Learning Network,” Virginia Institute of Technology, 1997.
<http://ei.cs.vt.edu/~history/Perceptrons.Estebon.html>

[Furber12] Steve Furber, “Low-Power Chips to Model a Billion Neurons”, IEEE Spectrum, July 2012. <http://spectrum.ieee.org/computing/hardware/lowpower-chips-to-model-a-billion-neurons>

[Goldberg15] Joseph Goldberg, “How Different Antidepressants Work”, August 2015. <http://www.webmd.com/depression/how-different-antidepressants-work>

[Julien11] Robert M. Julien, “A Primer of Drug Action”, 12th Edition, Worth Publishers, 2011.

[Lodishoo] Harvey Lodish, Arnold Berk, S Lawrence Zipursky, Paul Matsudaira, David Baltimore, and James Darnell, “Molecular Cell Biology, 4th edition”, Section 21.1: Overview of Neuron Structure and Function, New York: W. H. Freeman; 2000.
<http://www.ncbi.nlm.nih.gov/books/NBK21535/>

[Mangels03] Jennifer Mangels, “Cells of the Nervous System”, Columbia University, Department of Psychology. <http://www.columbia.edu/cu/psychology/courses/1010/mangels/neuro/neurotutorial.html>

[McCulloch43] Warren S. McCulloch and Walter Pitts, “A Logical Calculus of the Ideas Immanent in Nervous Activity,” Bulletin of Mathematical Biophysics, Vol. 5, pp. 115-133, 1943. <http://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf>

[Minsky69] Martin Minsky and Seymour Papert, “Perceptrons: an introduction to computational geometry”, The MIT Press, Cambridge, MA. 1969.

[Purves01] D. Purves, G.J. Augustine, D. Fitzpatrick, et al., “Neuroscience, 2nd edition”, Sinauer Associates; 2001. <http://www.ncbi.nlm.nih.gov/books/NBK11117/>

[Rosenblatt62] F. Rosenblatt, “Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms”, Spartan, 1962.

[Rumelhart86] David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams, “Learning Representations by Back-propagating Errors,” Letters to Nature, Nature, vol. 223, no. 9, 1986. http://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf

[Seung13] Sebastian Seung, “Connectome: How the Brain’s Wiring Makes Us Who We Are”, Mariner Books, 2013

[Sporns05] Olaf Sporns, Giulio Tononi, Rolf Kötter, “The Human Connectome: A Structural Description of the Human Brain”, PLoS Computational Biology, Volume 1 Issue 4, 2005. <http://journals.plos.org/ploscompbiol/article/file?id=10.1371/journal.pcbi.0010042&type=printable>

[Timmer14] John Timmer, “IBM Researchers Make A Chip Full of Artificial Neurons”, Ars Technica, August 2014. <https://arstechnica.com/science/2014/08/ibm-researchers-make-a-chip-full-of-artificial-neurons/>

[Wikipedia17a] Wikipedia, “History of Artificial Intelligence”, 2017. https://en.wikipedia.org/wiki/History_of_artificial_intelligence

[Wikipedia17b] Wikipedia, “Neuron”, 2017. <https://en.wikipedia.org/wiki/Neuron>

[Wikipedia17c] Wikipedia, “Perceptron”, 2017. <https://en.wikipedia.org/wiki/Perceptron>

[Wilson16] Robert A. Wilson and Lucia Foglia, “Embodied Cognition”, The Stanford Encyclopedia of Philosophy, Edward N. Zalta, editor 2016. <http://plato.stanford.edu/entries/embodied-cognition/>

Chapter 11

Learning and Reasoning

How to represent statements that we believe to be true, how to use them to discover new statements, and how to decide if those new statements are true or not.

Contents

11.1 Why This Chapter Is Here	395
11.2 The Steps of Learning.....	396
11.2.1 Representation	396
11.2.2 Evaluation	400
11.2.3 Optimization	400
11.3 Deduction and Induction	402
11.4 Deduction	403
11.4.1 Categorical Syllogistic Fallacies	410
11.5 Induction	415
11.5.1 Inductive Terms in Machine Learning	419
11.5.2 Inductive Fallacies	420
11.6 Combined Reasoning.....	422
11.6.1 Sherlock Holmes, “Master of Deduction”..	424
11.7 Operant Conditioning.....	425
References	428

11.1 Why This Chapter Is Here

When people are offered a new piece of information, we make a judgment call about how much to trust it. Based on a wide variety of criteria and processes, we decide how reliable the new information is likely to be, and to what degree we will choose to believe it. If this sounds familiar, it's because updating our current beliefs in something based on new information is just what Bayesian reasoning is all about, as we discussed in Chapter 4.

It's rarely an easy decision. We know that we're surrounded all the time by truths, partial truths, honest misunderstandings, deliberate misrepresentations, and even outright lies. How can we make good decisions about what to believe?

Starting with the ancient Greeks, philosophers and logicians have been developing a body of knowledge that attempts to provide formal structures for deciding how to judge the reliability of new information [Graham16].

Computer programs that learn are deeply influenced by these ideas. The tools of logical thinking form the conceptual foundation for how machines can learn from the information we provide to them, and integrate that new knowledge with what they already know. Since a computer can't use common sense, or a gut feeling, or even life experience, it has to use explicit, formal rules to process information and draw conclusions from it.

In this chapter we'll quickly survey some of the highlights of this kind of logical thinking.

11.2 The Steps of Learning

Learning is a big and complicated subject, even when limited just to algorithms.

To help us get a grasp on things, we can break down the learning process for any machine learning system into three conceptual pieces: **representation**, **evaluation**, and **optimization** [Domingos12].

Let's look at these in turn.

11.2.1 Representation

A system's **representation** describes what sorts of things it is capable of "knowing." It describes the structure for how information is saved and interpreted. The representation we use places limits on what can be saved, learned, and remembered.

The representation is both the abstract structure that's inside our learner, and the data that goes into that structure. In machine learning, the representation is an architecture for organizing numerical parameters, and algorithms that modify and interpret those parameters.

For example, consider the perceptron from Chapter 10. This little algorithm adjusts its weights to find a straight line between two sets of data. The weights in the perceptron, and how it uses those weights, are the perceptron's representation of the line that divides the two groups.

Because of this simple structure, a single perceptron can't represent a curved dividing line. We say it doesn't have enough **representational power**, or simply **power**, to describe anything more complex than a line. Even if we somehow "told" the perceptron about a curve that divided our two groups of data perfectly, the perceptron's limited representational power means that it couldn't learn or remember that curve.

What a system can't represent, it can't "know."

If we want to find and remember a curve that splits two groups of data, we need an algorithm that has **more representational power**, or is simply **more powerful**.

We sometimes say that an algorithm, its way of representing information, and a particular set of values, together make up a **model**.

More power doesn't always work to our benefit. As we saw in Chapter 9, the ability to represent a very curvy boundary can lead to that curve following the data so closely that it overfits. The trick to teaching a powerful representation is to start slowly and let the model it's building evolve from simple to complex.

This is why selecting the right algorithm for a task often requires human judgment. We need to use what we know of our data and our application to pick an algorithm whose representation is powerful enough to learn what we want to teach, but not so much more powerful that it over-learns (and thus over-fits). We also want the representation to be well-tuned to what's being learned, so that training is efficient.

For instance, if we're trying to draw a curve between two clusters of 2D data, then we want a representation that is good at describing the shape of curves. In this case, the representation might be a list of numbers that give us the parameters of that curve, sufficient to allow us to draw it.

But if we're trying to recognize words spoken into a cell phone, we'd like a representation that's good at describing words and what they sound like. This might be an audio waveform, and a list of words that correspond to what was said.

For any given choice of representation, we can think about what it can learn in terms of a nested structure from what is theoretically possible down to what is efficient given a set of resources. This hierarchy is shown in Figure 11.1.

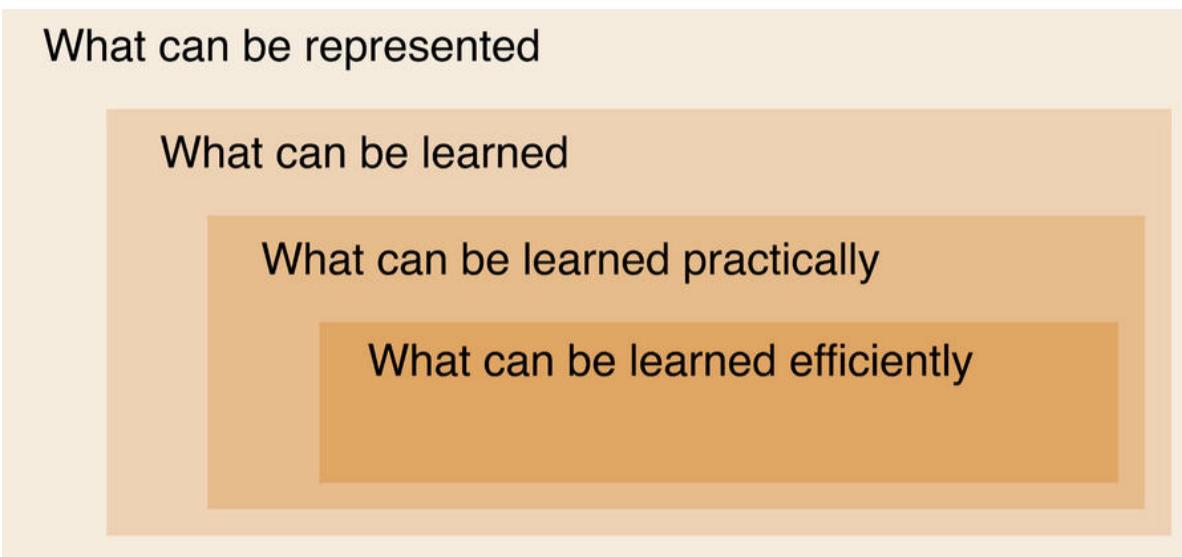


Figure 11.1: Organizing what information an algorithm can represent and learn as a hierarchy.

At the outermost level, a system's representation tells us what it can describe and remember, so this describes everything that it can learn in theory. But we can imagine some things that can be represented but not learned.

Recall from our discussion of information theory in Chapter 6 that each piece of data we want to describe will require a certain amount of information, usually measured in bits. In practice, we usually have only a fixed, limited number of bits for each piece of data. For example, if we store integers using 8 bits, we can represent any number from 0 to 255, but we cannot represent negative numbers, or anything larger than 255. But we can be sneaky. Suppose we want to use the value of pi, which has digits that go on forever. We can't represent the whole thing. But even in a finite amount of memory we can store a program that can be used on demand to calculate as many digits of pi as we like.

As another example, we can represent every hurricane that will form in the upcoming year. We only need to allocate memory to hold the date it will form, the date it will dissipate, its maximum wind speed, and so on. But we can't learn those things right now. Right, now all we can do is assign labels to those hurricane's qualities that we know about, while leaving their values blank.

A more abstract example comes from computer science. The **halting problem** takes a specific computer program, and a specific input, and asks whether the program will eventually stop with an output, or if it will run forever [Kaplan99]. This result is easy to represent with just a “yes” or “no,” but it turns out that it is theoretically impossible for us to predict the answer. The answer is not just difficult or time consuming to find, but rather it is literally impossible. Even if we’re allowed to look at the program itself, and use every tool that exists or could exist, we can prove that the question simply cannot be answered.

We can take a practical approach and run the program. If it stops, then we know the answer to “Does this program stop?” for that program was “Yes, eventually it stops.” But we could watch the program run for decades and never be able to say “No, it will never stop,” because it might stop a fraction of a second after we decided to give up. There’s no way to predict the result. So we can represent whether a program working on a specific input stops or not, but we can’t learn the answer.

So we want to narrow our focus on those things that can theoretically be learned. Beyond that, some things might require an impractical amount of resources (like infinite time or compute power), so now we want to further limit our attention to those things that can be learned in practice. Finally, we want to limit our attention yet again to just those things that we can learn efficiently, given the time and resources available to us.

We often use representations that can be tuned using parameters that we can set, so we can shape our representation to match our problem and data. This can limit our representations to those that will work for us (e.g., a curvy 2D boundary), saving us the time of learning other information that we don’t care about.

We often make these choices, or the settings of these parameters, based on the architecture we select for our system. But it’s important to keep in mind that our algorithms can only discover and learn the kinds of information that they are able to describe, or represent, in some way. What we cannot represent, we cannot know.

11.2.2 Evaluation

Learning is a continuum. To determine how well a system has learned something, we perform an **evaluation**.

Evaluation is where we compare the system's response to some kind of stimulus to the response we were hoping for.

In machine learning we usually quantify this process with a numerical score, called the **error score** or the **loss score** (or just the **error** or **loss**), that in some way captures how close the algorithm has come to what we wanted. The smaller the loss, the better the system's performance. We can measure the error in just about any way we want, if we think it will help the system learn what we're trying to teach it.

For example, we can use the concepts of accuracy, precision, recall, and so on, which we saw in Chapter 3. If we're trying to teach our system to recognize cats, we can aim for a high accuracy if we want to make sure that we only label as cats those things that we are sure really are cats. If we want to make sure that most cat pictures are correctly labeled, we might instead assign a high error when the precision is low.

Most libraries offer a variety of error functions, plus the ability to write our own.

11.2.3 Optimization

If a system hasn't learned something perfectly, we often want it to improve. This is the process of **optimization**. We also say that we're **optimizing** our system.

Note that this isn't the same as saying that the system is **optimal**, or at an **optimum**. These terms refer to a system that has reached an ideal state in some context. In other words, anything that is at optimum cannot be improved. The process of optimization can be seen as moving a system closer to that optimum, but perhaps never quite reaching it.

When we think about an algorithm and its optimal performance, things can become subtle.

As an analogy, suppose we want to dry some washing by hanging our clothes out on a clothesline. We could just throw them over the line, but we'd prefer to hang them. We might say that the optimal solution is to use one or more clothespins, since they are simple, cheap, and cause no harm to the clothes. But let's suppose that we instead start with a lousy solution, and we use duct tape to attach our clothes to the line. This has lots of problems, so we start to optimize. Perhaps we find a brand of duct tape that is slightly less sticky, so it doesn't shred our clothes when we pull it off of them. Maybe we then cut the strips of tape lengthwise, so they don't weigh so much and drag down the line. In this way, we're optimizing our poor solution by making it better, but we're not moving towards the overall optimum because our modified duct tape is not turning into a set of clothespins.

This is conceptually similar to the problem of finding the global minimum of a curve or surface, which we discussed in Chapter 5. Optimizing corresponds to improving by getting closer to a local minimum, while reaching optimal usually means finding the global minimum.

The process of optimizing a learner is carried out by a specialized algorithm called an **optimizer**. There are many different optimizers available in machine learning, with more showing up all the time. We'll meet some of the most popular optimizers in Chapter 19. But why do we have so many? After all, we always want to learn in the most efficient way. Isn't there a single best optimizer that we should always use?

The answer is definitely “No.” Not only is there no known optimizer that is better than all the others for all the problems we might ever want to solve, but we can *prove* that no optimizer could *ever* be better than all the others in all situations. This famous result is colorfully named the “No Free Lunch Theorem” [Wolpert97].

So just as we pick the right representation for each learner, so too do we pick an optimizer that we decide, by benefit of experience, hunch, or investigation, will best help the learning process for our particular representation, algorithm, and data.

11.3 Deduction and Induction

At the heart of learning is the ability to turn data into knowledge. That is, we want to convert raw data, in the form of numbers, labels, strings, and so on, into patterns and descriptions that make sense to us and help us make sense of the data, and sometimes even the broader world both we and the data share.

People have spent millennia studying how we learn. Philosophers, educators, neuropsychologists, and many others have looked into learning in enormous depth, and the ideas that have been put forth and debated are vast. We won't even begin to survey the subject.

Luckily, most machine learning algorithms have so far been based on just two general approaches to learning.

These two ideas are **deduction** and **induction**. These often work hand-in-hand when we do science.

Generally speaking, deduction is a process where we start with a theory, or **hypothesis**, which makes predictions. We then gather evidence to see if those predictions are accurate. If a piece of evidence supports our theory, we may be able to learn from it to refine our hypothesis. If a piece of evidence refutes our theory, we typically narrow the scope of our predictions to exclude the circumstances where they're incorrect. In this way the theory is made stronger and more precise.

If in the end there's enough supporting data, we announce our theory as a potential explanation of the subject matter under discussion. The goal of a theory is to say with some confidence that if certain conditions are met, certain results will follow.

If there isn't enough supporting data, we abandon or refine the hypothesis and start a new search for data to confirm or refute it.

By contrast, induction starts by making observations, which we then examine in a quest to find patterns. The more that new observations support those patterns, the more likely we consider those patterns to be. As time goes on we can start to draw tentative ideas about what is possible, what is likely, and what is neither. When one of those ideas becomes specific and seem to be supported by a lot of data, we can express it as hypothesis and use it as the starting point for deduction.

While reasoning inductively, we never assert definitely that some set of observations mean that some conclusion is true, because we always hold out the possibility that a new observation could come along and completely ruin our pattern. Instead, we assert that something is usually, or probably, true, with the degree of certainty coming from the nature and quantity of the examples that we've seen.

We sometimes say that deduction is **top-down**: we start with a general hypothesis, and gather specific data to support and refine it. On the other hand, induction is **bottom-up**: we start with observations, and gradually grow a hypothesis from them based on the patterns we find in the data.

Let's look more closely at both of these ideas in turn. We'll see that they both contribute to how machine learning algorithms work.

11.4 Deduction

Deduction, also called **deductive reasoning**, **deductive logic**, and **logical deduction**, is a process of moving from the *general* to the *specific*.

Before we get into abstractions, let's look at an example of deduction: solving a murder in the style of classic mystery books.

The scene is Rawlinson Manor, a mansion set atop the tallest hill on Forlorn Island. One night when there are no guests in the house and no boats at the dock, Lady Vivian retires to bed as usual, surrounded by her late husband's hunting trophies, guns, and swords hanging on the wall. In the morning, as the maid enters her bedroom with breakfast, she screams and drops the tray. Lady Vivian is dead, her husband's scimitar deep in her chest.

Mere hours later, Inspector Stanshall is on the scene. He assembles the entire staff of the house in the parlor, and proceeds to use deduction to solve the crime.

"Someone killed Lady Vivian last night, and everyone in this room is a suspect," he says, stating his theory. "There is no way she could have stabbed herself with such a long sword. Someone else did it, and everyone who possibly could have done so is in this room. Any one of you could be the murderer."

He now starts to test his theory. Inspector Stanshall observes that the maid is a frail woman of 80 years. He hypothesizes that she would be physically unable to lift the heavy sword.

He tests his hypothesis by pulling the helmet off of a suit of armor in the corner. He casually asks the maid to come over and hands her the helmet, which weighs less than the scimitar that killed Lady Vivian. As soon as the maid takes the helmet, she gasps, staggers, and drops it, nearly on her own feet. Having now confirmed to his satisfaction that the maid does not have the physical strength to lift the murder weapon, Inspector Stanshall **deduces** that the maid cannot be the murderer.

Thus he refines his theory. "Someone killed Lady Vivian last night," he says, "and everyone who possibly could have done so is in this room. Any one of you could be the murderer, except for the maid."

Figure 11.2 shows this process of elimination graphically.

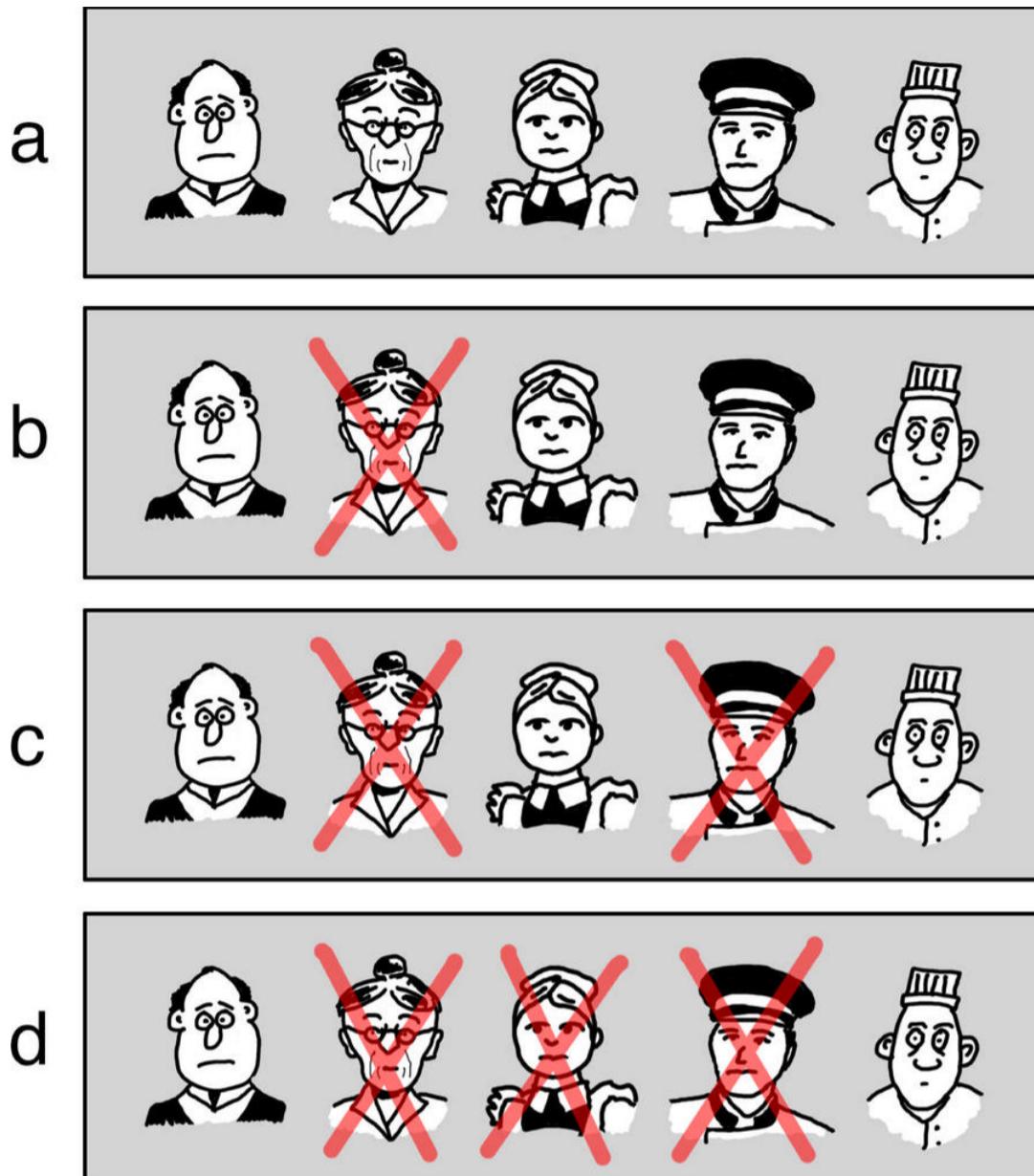


Figure 11.2: Figuring out the murder using deduction. (a) The original 5 suspects. (b) Elimination of the elderly maid. (c) Elimination of the valet. (d) Elimination of the young maid. Only the butler and the cook can't be excluded.

By making repeated hypothesis and observations, the Inspector continues to refine his theory. Some of his hypotheses are designed to rule out a suspect, others are to confirm that a suspect could have done the deed. One observation at a time, Inspector Stanshall makes his theory stronger and narrower, until only the cook and butler remain. He concludes that the murder was done by these two men, working together.

We say that the process of deduction starts with a **domain of discourse**, or the subject matter that the hypothesis addresses. In our example the domain of discourse started out containing all the servants

in the room. By using hypotheses, prediction, and observations, the inspector **reduced the possibilities** in the domain until his theory was correct for all the remaining servants: the cook and the butler.

The most pared-down form of deductive reasoning is the **syllogism**. This is a list of three sequential statements that reason from a general idea to a specific one [Kemerling97].

There are several common forms of syllogisms. We'll begin with the **categorical syllogism**, which may be the best known. Here's an example of a categorical syllogism in **standard form**:

1. No fish live on the moon.
2. Frank is a fish.
3. Therefore, Frank does not live on the moon.

The final statement is called the **conclusion**. In terms of sentence structure, it expresses a relationship between the **subject** of the sentence (in this case, "Frank") and its **predicate** (in this case, "lives on the moon").

Figure 11.3 illustrates the terms in this syllogism.

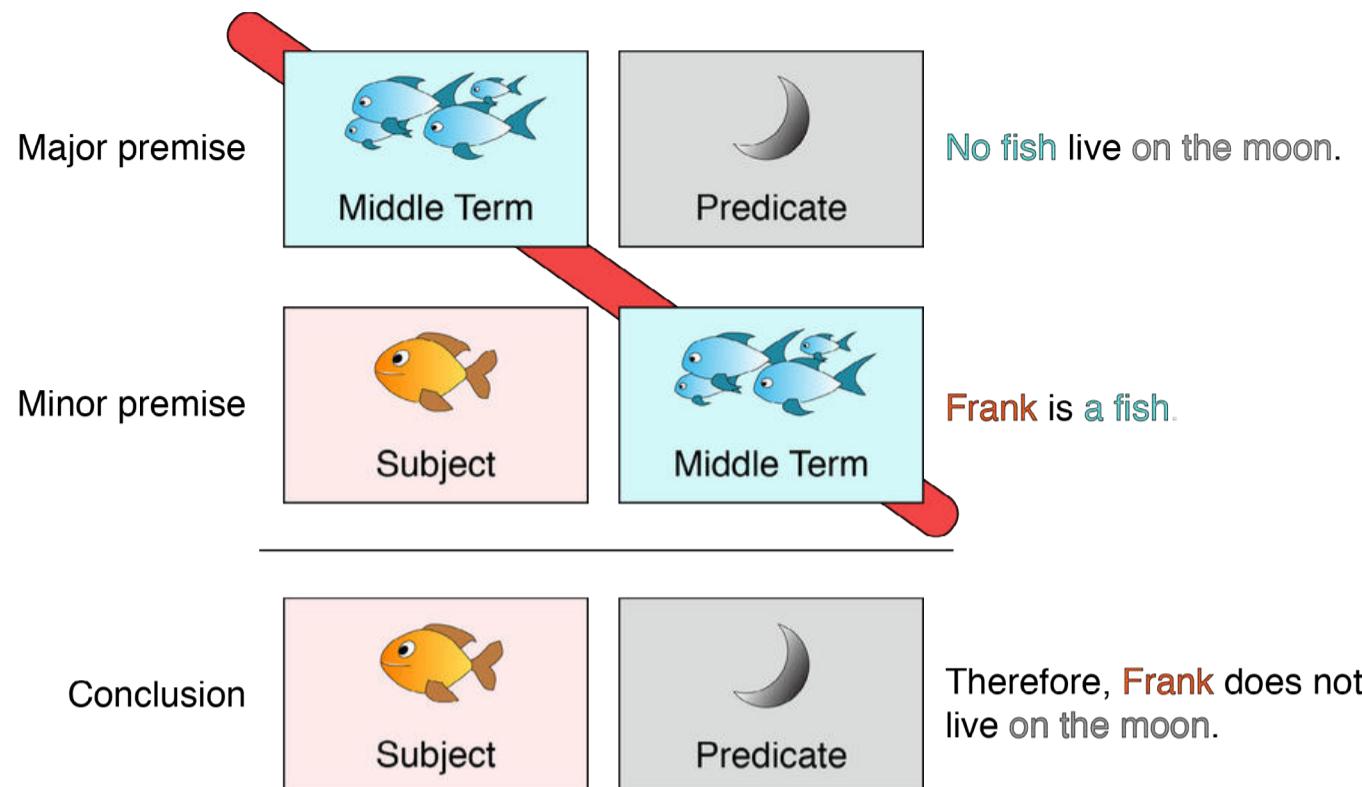


Figure 11.3: The names of the pieces in our categorical syllogism. The conclusion brings together the subject and predicate. As shown by the red bar, the middle terms drop out.

The first statement is called the **major premise**, and it tells us something about the predicate of the conclusion (“living on the moon”). The second statement is called the **minor premise**, and it tells us something about the subject of the conclusion (“Frank”). The major and minor premises share a **middle term** that is used to join them, but doesn’t appear in the conclusion (“things that are fish”).

If the conclusion flows correctly from the premises, as in this example, we say that the syllogism is **valid**, otherwise it’s **invalid**.

When a syllogism is invalid, it’s because something went wrong in the logic. There are several common ways to mess up the logic. Each kind of error commits a different form of **sylogistic fallacy**, or simply a **fallacy**, many of which have their own names. We’ll see some of these below.

Note that validity has nothing to do with the **accuracy** of the premises, or whether they make any sense in the real world or not. As long as the logic holds, the syllogism is valid.

It's precisely this mechanical nature of the syllogism that makes it so appropriate for computers. We can encode the steps in software, and as long as our program doesn't have bugs, we are guaranteed that the process will produce a conclusion that follows from the premises. We don't have to interpret or think about what the statements mean or what they refer to.

When accuracy, or relevance to the real world, is an issue, it's covered by the concept of being **sound**. If a syllogism is valid, *and* all of its premises are in fact true, then it's not just valid but **sound**. If the syllogism is valid but the premises are not true, the syllogism is **unsound**.

Take this example:

1. All giraffes like to fly kites.
2. All bowling balls are giraffes.
3. Therefore, all bowling balls like to fly kites.

This syllogism is valid, because the conclusion flows correctly from the premises. But it's unsound, because the premises themselves are not true statements about the world.

The most famous categorical syllogism is probably one which has appeared in one form or another in works of logic since at least the 1200's [Allegranza16]. It goes like this:

1. All men are mortal.
2. Socrates is a man.
3. Therefore, Socrates is mortal.

Let's illustrate the steps in this syllogism by drawing boxes to represent categories of objects, as in Figure 11.4.

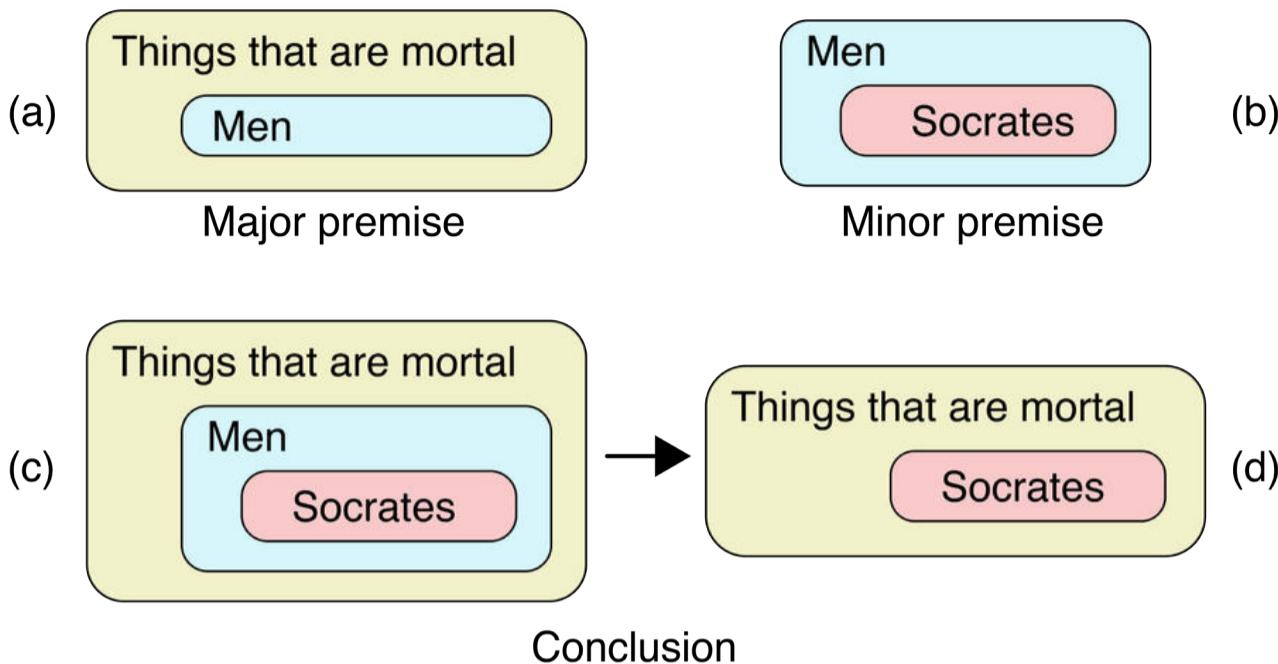


Figure 11.4: The most famous categorical syllogism in diagram form. (a) The major premise, “All men are mortal.” (b) The minor premise, “Socrates is a man.” (c) The result of combining the premises. (d) The logically valid conclusion, found by eliminating the middle term in blue.

There are multiple types of syllogisms, each with their own structure. Here are some of those other forms [Fiboni17].

A **conditional syllogism** states on the first line that a general relationship, or condition, is true. The second line provides some information about the elements in a specific instance of the relationship. The conclusion appears on the third line.

For example,

1. If it is raining, then John is wearing a raincoat.
2. It is raining.
3. Therefore, John is wearing a raincoat.

This is valid (whether or not it's sound depends on today's weather and John's habits). Alternatively, consider this syllogism:

1. If it is raining, then John is wearing a raincoat.
2. John is wearing a raincoat
3. Therefore, it is raining.

This syllogism is invalid, because the conclusion does not follow. John might be wearing his raincoat even when it's not raining because it's comfortable, or it keeps him warm.

A **disjunctive syllogism** asserts on the first line that one situation *or* the other is true, but not both at the same time. The second line then gives us the information needed to decide.

For example,

1. Either the room is painted red or blue.
2. The room is not painted red.
3. Therefore, the room is painted blue.

This is a valid syllogism, because the added information in the second line forces the conclusion. For contrast, consider this one:

1. The sky tonight is either clear or cloudy.
2. Bob likes to look at the stars.
3. Therefore, the sky is clear.

This syllogism is invalid, because there's no logical reason why Bob's desire to view the stars should mean the sky is clear.

11.4.1 Categorical Syllogistic Fallacies

There are lots of ways that we can make logical errors in a syllogism, particularly when we use complicated language for the various pieces.

We'll look at some typical errors, called **fallacies**, in this section.

Logicians often simplify the language by using letters rather than verbal descriptions. For example, the syllogism about Socrates is often presented this way [Hurley15].

1. All A are B. (All men are mortal.)
2. C is A. (Socrates is a man.)
3. Therefore, C is B. (Socrates is mortal.)

This can be helpful but it takes some getting used to, so we'll stick with language.

All of the fallacies we'll consider here have names that have been around for a long time, and tend to be somewhat wordy.

We'll assume that our syllogisms refer to the fruit in a particular store one day, so our domain of discourse is "fruit in the store." We'll usually be specifically interested in the categories of "fruits that are apples," "fruits that are bananas," and the more general "fruits that are ripe." It just so happens that today we just got in a new shipment, so all the apples, bananas, and apricots are ripe. Unfortunately, there was a delivery mix-up, and all the other fruit in our store is still green and not yet ripe.

For our first logical error, let's start with **affirming the consequent**:

1. All apples are ripe.
2. This fruit is ripe.
3. Therefore, this fruit is an apple.

The problem is that there are plenty of other ripe fruits that are not apples. Figure 11.5(a) diagrams this visually.

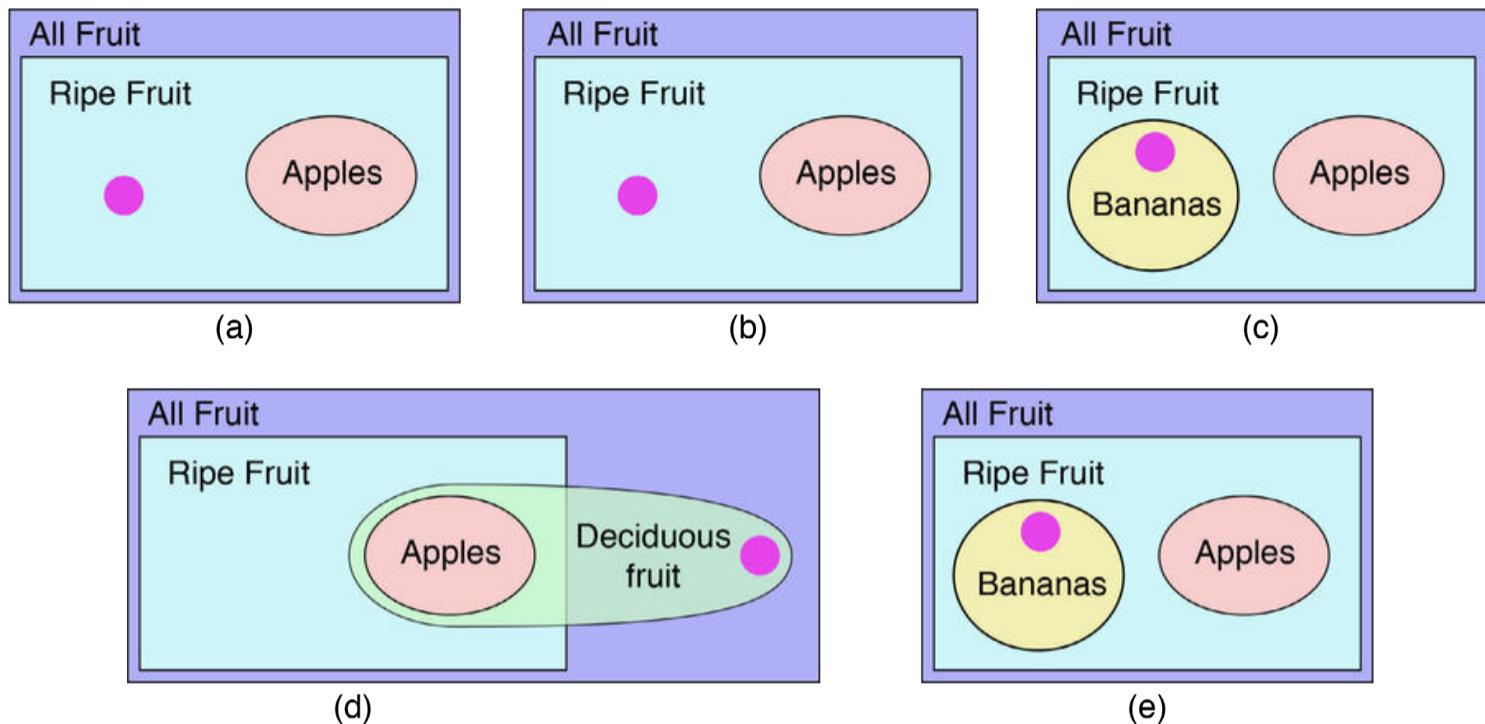


Figure 11.5: Some common syllogistic fallacies, as described in the text.
 (a) Affirming the consequent. (b) Denying the antecedent. (c) The illicit major.
 (d) The illicit minor. (e) The undistributed middle.

In Figure 11.5, the condition that “All apples are ripe” is shown by the red ellipse (“Apples”) that is entirely within the blue box (“Ripe fruit”). So any point in the red ellipse is an apple, and since the whole ellipse is inside the blue box, that apple is also ripe. All other points that are inside the blue box are also ripe fruit, just not apples. Any point outside the blue box is either not ripe, not a fruit, or both. An example of a fruit that contradicts the conclusion of each fallacy is shown as a purple dot.

The fallacy of affirming the consequent says that because all apples are ripe, and this fruit is ripe, it’s an apple. But the purple dot is a ripe fruit that is not an apple.

The error of **denying the antecedent** is very much like the error of affirming the consequent, only it comes at it from the opposite direction:

1. All apples are ripe.
2. This fruit is not an apple.
3. Therefore, this fruit is not ripe.

The problem is that there are plenty of fruits other than apples that are ripe, as shown in Figure 11.5(b). The purple dot here is not an apple, and yet it's ripe.

The error of **the illicit major**:

1. All apples are ripe.
2. No banana is an apple.
3. Therefore, no bananas are ripe.

The conclusion doesn't follow because apples aren't the only things that are ripe, as shown in Figure 11.5(c). In this example, the purple dot is a ripe banana.

The error of **the illicit minor**:

1. All apples are ripe.
2. All apples are deciduous fruit.
3. Therefore, all deciduous fruits are ripe.

The problem is that there are plenty of other deciduous fruits, like apricots and peaches, and perhaps some of them aren't ripe, as in Figure 11.5(d). The purple dot shows another deciduous fruit (perhaps a pear or plum) that isn't ripe.

Finally, the error of **the undistributed middle**:

1. All apples are ripe.
2. All bananas are ripe.
3. Therefore, all bananas are apples.

The error comes from the fact that bananas and apples can be different but ripe at the same time, as shown in Figure 11.5(e). The purple dot shows a banana that is not an apple.

These are just a few of the common fallacies that we see with syllogisms.

Listening critically to popular arguments in contemporary media reveals that all of these fallacies are alive and well and are being committed regularly. When people speak quickly and passionately, it's easy to miss fallacies, particularly if the premises are themselves made up of compound arguments. Whether the fallacy is being committed with deliberate intent to deceive, or as an innocent error of reasoning, the result is still that we can be convinced of something that is not accurate. Generally speaking, it takes attention and practice to spot these errors of reasoning as they happen and thereby know that the conclusion doesn't follow [Garvey16].

It's useful to keep in mind that just because an argument reaches a conclusion via a fallacy, that doesn't mean the conclusion is wrong. It could be correct for other reasons. All we know is that it doesn't correctly follow as a result of that argument.

The clear logic of the deductive approach explains why it is popular in science. There's no room for interpretation or guesswork. By the rules of logic, either the conclusion follows from the premises or it does not.

The result is that we are able to state with total confidence that if the premises are correct, the conclusion is either true or not. There's no gray zone.

11.5 Induction

Inductive reasoning, also called **hypothesis construction**, is a process of moving from the *specific* to the *general*.

Induction begins with observations. As the observations accumulate, we look for patterns. When we find a pattern we put it forward as a hypothesis, but always with a cautionary disclaimer that the hypothesis could be wrong. The more observations that we find that agree with the hypothesis, the more confident we become, but we never become certain. There's always a chance that we're wrong.

For example, we might observe that all the crows we've ever seen are black. We can then say, "It's extremely likely that all crows are black." We hedge our bets a little bit because maybe tomorrow we'll step outside and see an orange crow. We can be very sure that won't happen, but we can't be absolutely certain. The key issue is that we're generalizing from observations, and there's always a chance that a new observation will come along that contradicts the pattern we've seen. The process can be formalized using Bayes' Rule, as we saw in Chapter 4.

In inductive reasoning, it's possible to start with true premises, and make no errors in logic, but still come to a false conclusion.

For instance, we might look at 1000 apples and note that every single one grew on a tree. From this, we might say, "It's extremely likely that all apples grow on trees." Despite our 1000 correct observations, tomorrow we might find an apple that had been grown hydroponically, without a tree in sight. This one counter-example would destroy our conclusion, and any other conclusions that used this as part of their evidence.

In this case, we could recover some of our conclusion by re-wording it, perhaps as “Most apples grow on trees.” As we see more examples of hydroponically grown apples, we might weaken that conclusion further, perhaps to “Some apples grow on trees,” or even “A few apples grow on trees.”

Let’s look at some of the principles of induction [Fieser11]. We’ll see that some of the terms we’ve been using to describe machine learning (like *generalization* and *prediction*) are taken directly from these principles.

We’ll talk about a **population**, which is some group or category of things that we can observe. A **sample set** is some number of things taken randomly from the population (or perhaps just observations of those things). An **individual** is a single object from the population (or an observation of that object). Some percentage of the population has a **property** (e.g., it’s over a certain weight, has a certain color, or sheds its leaves in the fall). Figure 11.6 illustrates these ideas.

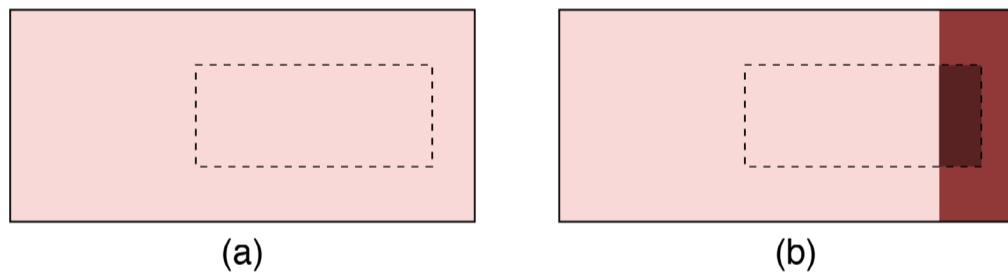


Figure 11.6: How we’ll look at induction principles. (a) Our population is the larger rectangle. The sample set is the smaller rectangle drawn with dashed lines. (b) Suppose that 15% of the members of the population are dark red. Because our sample set is a good representative of the population, 15% of the members of the sample set are also dark red.

Returning to our fruit store example, our population might be the store’s stock of apples. The sample set could be someone’s shopping cart with a few apples chosen at random. An individual would be one apple, chosen either from the sample set or the population. The property could be the apple’s weight.

The principle of **generalization** says that if we learn something about our sample set, it will also be true of the population. For example,

1. 15% of the apples in our shopping cart weigh more than 3 ounces.
2. Therefore, 15% of the apples in the store weigh more than 3 ounces.

We could qualify the second statement in the form, “Therefore, *it’s likely that* 15% of the apples...” but in practice we usually treat these extra qualifying words as implicit, so we don’t have to repeat them over and over. Figure 11.7 shows the idea.

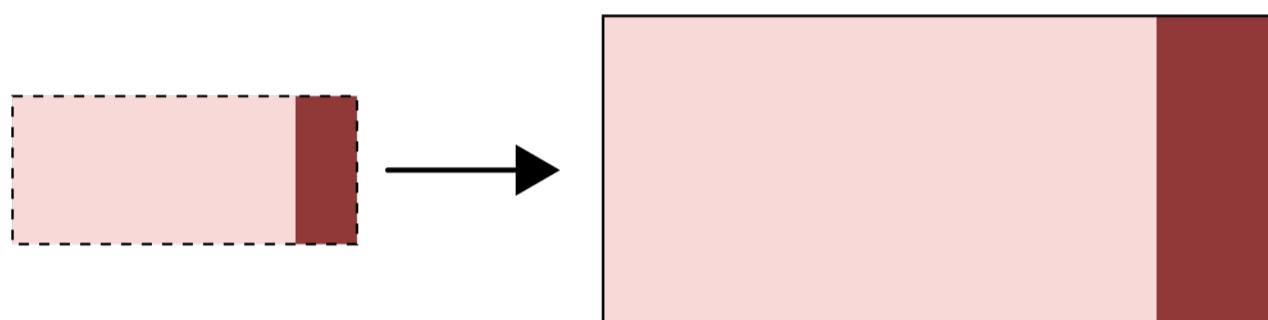


Figure 11.7: The principle of generalization, which says that properties of the sample set also apply to the population. Here, 15% of the sample set is dark red, which tells us that 15% of the population is also dark red.

This principle is exactly what we rely on when we train a learner with our training set. Because we believe that the training set is representative of the more general data the system will be exposed to, we trust that the properties the system learns about the training data will also apply to the data it receives after it’s been deployed.

The principle of the **statistical syllogism** lets us reason from knowledge about the population to knowledge about an individual drawn from the population.

For example,

1. 15% of the apples in the store are ripe.
2. This apple was picked randomly from the store.
3. Therefore, there is a 15% chance that this apple is ripe.

Figure 11.8 shows the idea.

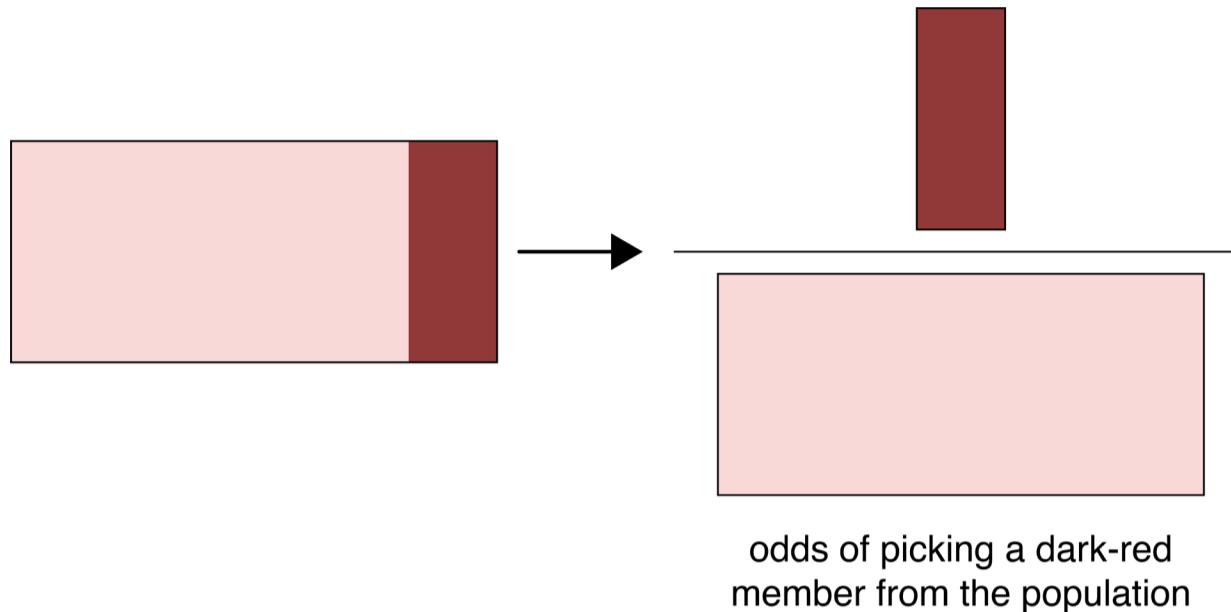


Figure 11.8: A statistical syllogism, using the dart-throwing style of figures from Chapter 3. 15% of this population is dark red, so if we choose a random member of the population, there is a 15% chance it will be dark red.

Finally, **prediction** lets us say that a property of our group will be shared by an individual chosen from the population.

1. 15% of the apples in our shopping cart are ripe.
2. Therefore, there is a 15% chance that the next apple we select from the store will also be ripe.

Figure 11.9 shows the idea visually.

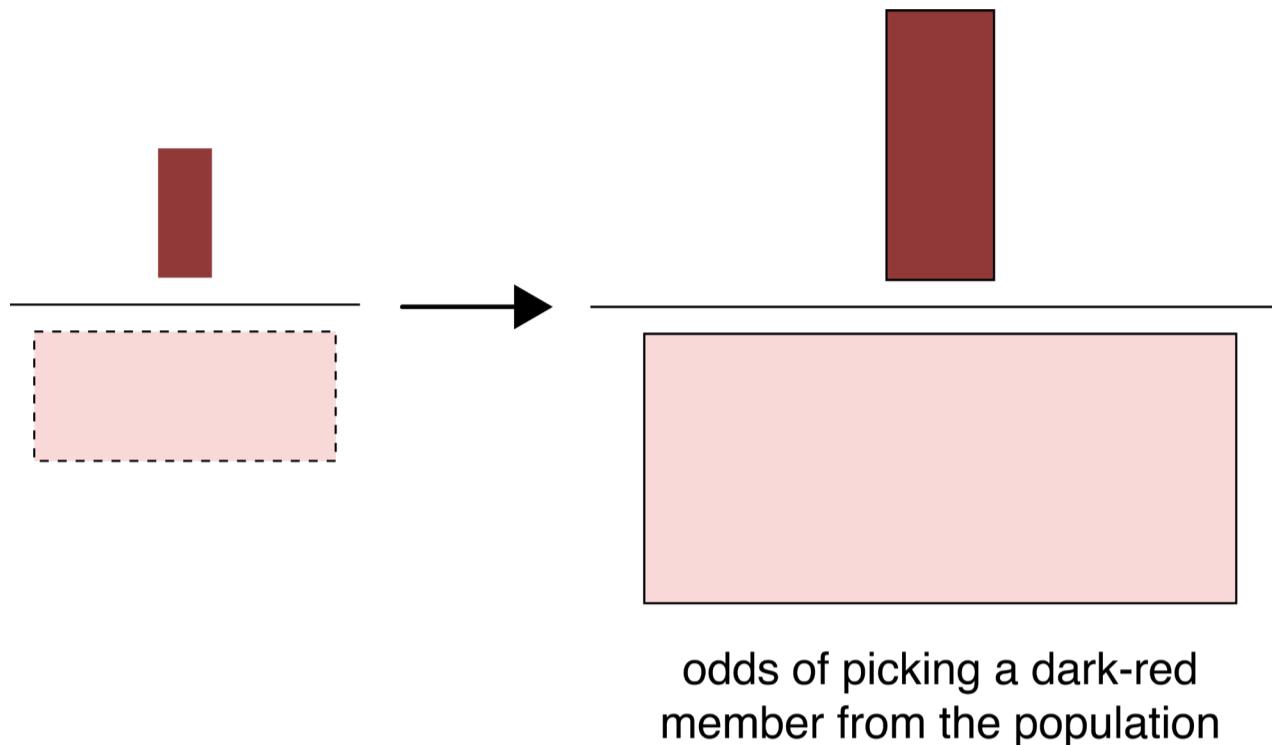


Figure 11.9: The principle of prediction. 15% of our sample set is dark red. Because the sample set matches the population, if we select a random member of the population, we have a 15% chance of getting a dark-red element.

11.5.1 Inductive Terms in Machine Learning

Now we can see that the terms *generalization* and *prediction* that we use frequently in machine learning weren't just plucked out of the air. They come from the theory of inductive reasoning.

As we've seen, *generalization* in induction tells us that if we've learned something about the sample set, then we should expect to see that the population at large is similar. In terms of machine learning, we say that an algorithm's ability to *generalize* describes how well it can translate knowledge about the training set to knowledge about real-world data.

In induction, *prediction* tells us that if we learn something about the sample set, then an individual chosen from the population will share that quality. In terms of machine learning, we say that once we've learned from the training set, we can apply that to each sample by *predicting* a value (like a label or number) for it.

11.5.2 Inductive Fallacies

Because induction is so flexible, there are lots of ways we can make mistakes. We'll look at some of those here.

Figure 11.10 shows our first set of 4 induction fallacies. We'll suppose that our population is made up of points on a circle, so our sample set will always be a collection of those points. We'd hope that we would always conclude that our data forms a circular shape. Instead, we show in red the non-circle shape that we infer as the result of a fallacy.

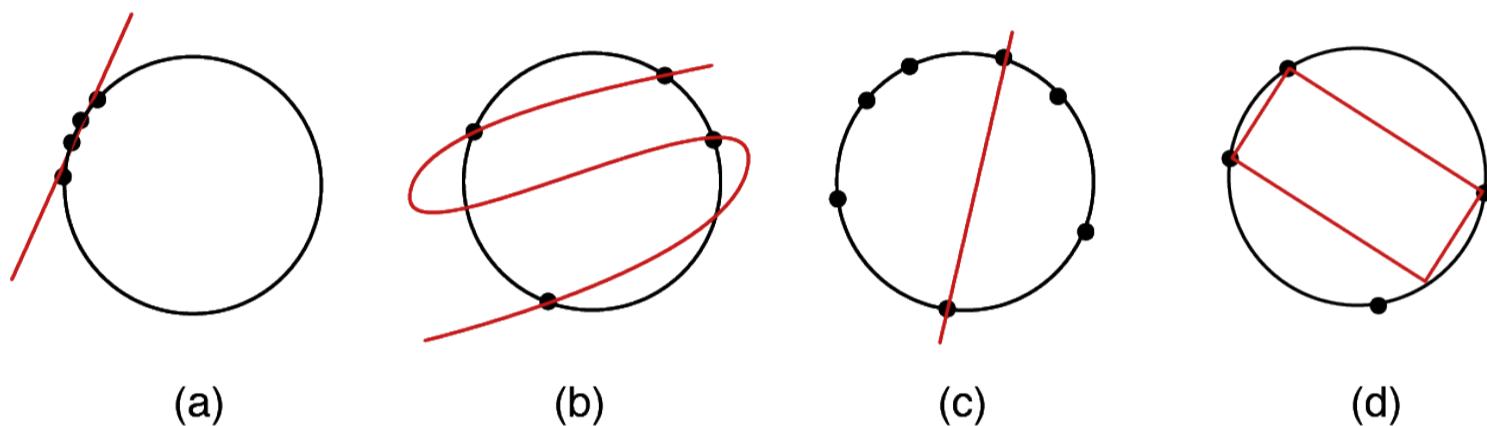


Figure 11.10: Some induction fallacies. (a) Hasty generalization. (b) Faulty generalization. (c) Slothful induction. (d) Biased sampling.

Hasty generalization in Figure 11.10(a) comes from working with a small, non-representative set of samples. Here, our data points seem to form a line. More data would help us avoid this mistake. **Faulty generalization** in Figure 11.10(b) comes from working with too few samples. The S-shape here is plausible given these four points, but more points would rule it out. **Slothful induction** in Figure 11.10(c) is when we ignore the most simple, reasonable, or obvious conclusion and choose something else. The most obvious shape for this set of points is a circle, not a line. **Biased sampling** in Figure 11.10(d) comes from seeing what we want to see, despite the evidence. In this case, we're expecting a box, so that's what we see.

Figure 11.11 illustrates another set of induction fallacies.

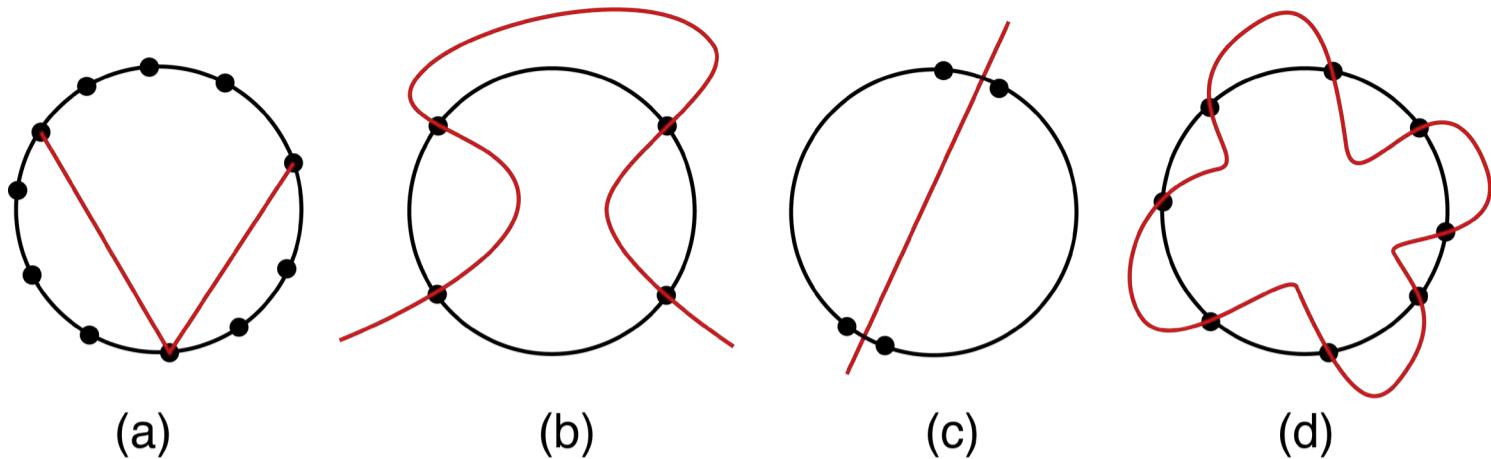


Figure 11.11: Some more induction fallacies. (a) Overwhelming exception.
(b) Appeal to coincidence. (c) Misleading vividness. (d) Special pleading.

Overwhelming exception in Figure 11.11(a) is like hasty or faulty generalization, but we explicitly argue away the data that doesn't fit our conclusions. This is also called the **fallacy of exclusion**. Here, we're making the rather unlikely argument that the 8 points we're not using are experimental errors and so should be ignored. **Appeal to coincidence** in Figure 11.11(b) means that we ignore the most obvious explanation and pick something else. These four points are more likely to be a rectangle or circle or other simple shape, and not this complex curve. **Misleading vividness** in Figure 11.11(c) refers to times when our senses are swamped with a perception that we just can't get past. Here, we might just "see" a straight line and be unable to imagine an alternative. **Special pleading** in Figure 11.11(d) involves re-interpreting the results selectively, usually with an appeal to authority. We might have someone in our group with a lot of experience who insists that these 8 points form a flower, and so we defer to their judgment, justifying this decision not with the data but by citing their expertise.

11.6 Combined Reasoning

As we mentioned earlier, science is frequently referred to as a deductive discipline. This is an appealing characterization, since deduction is firmly bound by logic, while induction is flexible and never completely sure of its conclusions.

But deduction rarely stands on its own. Consider the classical example we gave of a categorical syllogism:

1. All men are mortal.
2. Socrates is a man.
3. Therefore, Socrates is mortal.

The minor premise, “Socrates is a man,” is something we can observe directly (well, not anymore, but let’s pretend that he’s still around for this discussion).

But the major premise “All men are mortal” (that is, all men eventually die), is an idea we can’t confirm on the spot. That comes from making observations, and then developing a general principle, or hypothesis, from those observations.

Let’s return to the major premise of our syllogism, “All men are mortal,” and describe it inductively:

1. All living things that we’ve seen are mortal.
2. All men are living things.
3. Therefore, all men are very likely to be mortal.

The first statement is still taking a leap, because there might be living things out there that are not mortal. They might already have been discovered, depending on just how we define terms like mortality [Wikipedia17].

When we remove the qualifiers from our conclusion (since the evidence for this conclusion appears to be extremely strong), we can say “All men are mortal.” That now becomes our major premise in the syllogism about Socrates.

So although the internal logic of our syllogism about Socrates involved pure deduction, the process of establishing the major premise, which was necessary to get the ball rolling, required induction.

This leads us to ask what kinds of premises can appear in a deductive syllogism. We can break down premises into two categories: **rational** and **empirical**.

A **rational** premise emphasizes the products of pure intellect, such as mathematics and logic. Rational premises include “ $2+3=5$,” and “If $A=B$ and $B=C$, then $A=C$.”

On the other hand, an **empirical** premise emphasizes the things in the world, experienced and understood through our senses. Empirical premises include “Mature Redwood trees are taller than most birds,” and “All men are mortal.”

The philosophical tool that captures this distinction is known as **Hume’s fork** [DeMichele16]. Speaking very broadly, Hume asserted that our rationalist ideas were preferable to our shaky empirical observations [Hume77]. In one of the great philosophical counter-arguments, Kant “crossed Hume’s fork” by saying that empirical perceptions of the real world inform our rationalist ideas, and vice-versa [Kant81].

Any premise that involves accumulated observations by our senses (e.g., “All men are mortal,” “Fish live in water,” or “The market is open on Tuesdays and Thursdays”) is the result of an inductive process. When such a premise is used as part of a deductive syllogism, the deductive conclusion is only as sound as the inductive conclusions that form its premises.

11.6.1 Sherlock Holmes, “Master of Deduction”

Before we leave the ideas of induction and deduction, let’s consider perhaps the most famous fictional logician of all time, Sherlock Holmes.

Sherlock Holmes was often said to be the “master of deduction” [Doyle01]. Indeed, Holmes frequently lectured his colleague Watson on the value of deductive reasoning.

Here he’s explaining the need for reducing the domain of discourse, as we discussed above in our murder-mystery example:

“...when you have eliminated the impossible, whatever remains, however improbable, must be the truth” [Doyle90].

But many of Holmes’ flashier pronouncements were entirely inductive (and because they were based on so little evidence, also extraordinarily lucky). Consider his explanation for concluding that Watson’s brother was careless. In this passage, Holmes is pointing out features on a hand-wound pocket watch carried by Watson’s brother.

“...your brother was careless. When you observe the lower part of that watch-case you notice that it is not only dented in two places, but it is cut and marked all over from the habit of keeping other hard objects, such as coins or keys, in the same pocket. Surely it is no great feat to assume that a man who treats a fifty-guinea watch so cavalierly must be a careless man” [Doyle90].

Because Holmes draws his conclusion from a set of observations, this is a purely inductive explanation. At least Holmes isn’t claiming it’s anything else.

But he does just that in another story. In this tale, Watson was quietly contemplating some paintings at home. He was thinking about the folly of war when Holmes said that he agreed that war was “preposterous.” How had Holmes known what he was thinking?

Holmes explained that he merely watched Watson:

“...you looked hard across as if you were studying the character in his features. Then your eyes ceased to pucker, but you continued to look across, and your face was thoughtful [...] a moment later I saw your eyes wander away from the picture, I suspected that your mind had now turned to the Civil War, and when I observed that your lips set, your eyes sparkled, and your hands clenched I was positive [...] At this point I agreed with you that it was preposterous and was glad to find that all my deductions had been correct” [Doyle92].

Although Holmes refers to “all my deductions,” the only thing even close to deduction was his guess (we could call it a hypothesis) that Watson was thinking of the Civil War. All of his other conclusions were drawn from a series of observations, so they were the result of applied inductive reasoning.

Any detective, whether real or fictional, who uses observed clues to come to a conclusion is using at least some inductive logic.

11.7 Operant Conditioning

When we learn, it’s usually imperfect and not immediate. In the early stages, we can find ourselves guessing at answers. The feedback we receive tells us if we’re on the right track or not.

In machine learning we need to help our algorithms through this same process: we train our algorithms, they make guesses, and we give them feedback. What’s the best form of feedback to provide?

Like learning itself, the issue of educational feedback has been explored for millennia. Let’s look at the approach developed by people working in the field of **behaviorism**, also called **behavioral psychology**. This discipline studies how creatures learn and respond to stimuli based solely on their actions, or behaviors (as opposed to theories about their mental or emotional states) [Skinner51].

We say that someone's actions **operate** on the environment. The environment responds with feedback that **reinforces** or **punishes** each action. This way of studying actions and feedback is called **operant conditioning**, or **instrumental conditioning**. We'll see in Chapter 26 that this approach can be applied directly to machine learning, where we call it **reinforcement learning**.

A common way to look at this approach is that we're trying to **shape** the behavior of a learner (e.g., a person, dog, or computer algorithm). We distinguish the learner's actions into those that we consider **desirable** and those that are **undesirable**. Our goal is to increase the **frequency** of the desired behaviors, and decrease the frequency of the undesired behaviors. In machine learning, the desired behavior is doing a good job on extracting correct meaning from our data, and undesired behavior is doing a lousy job at that task.

In all but the most dispassionate situations (like when training a computer), learning goes in both directions. Though we usually think of the “trainer” teaching the “learner,” any person with a child or a pet knows that both parties are constantly teaching the other.

To teach the learner, we offer feedback by either **adding** a stimulus to the learner's environment, or **removing** a stimulus. We take these actions to either **increase** the frequency of the behavior, or **decrease** it. That gives us four possibilities, which we can summarize in a grid. Figure 11.12 shows this grid of four cells, along with their conventional names.

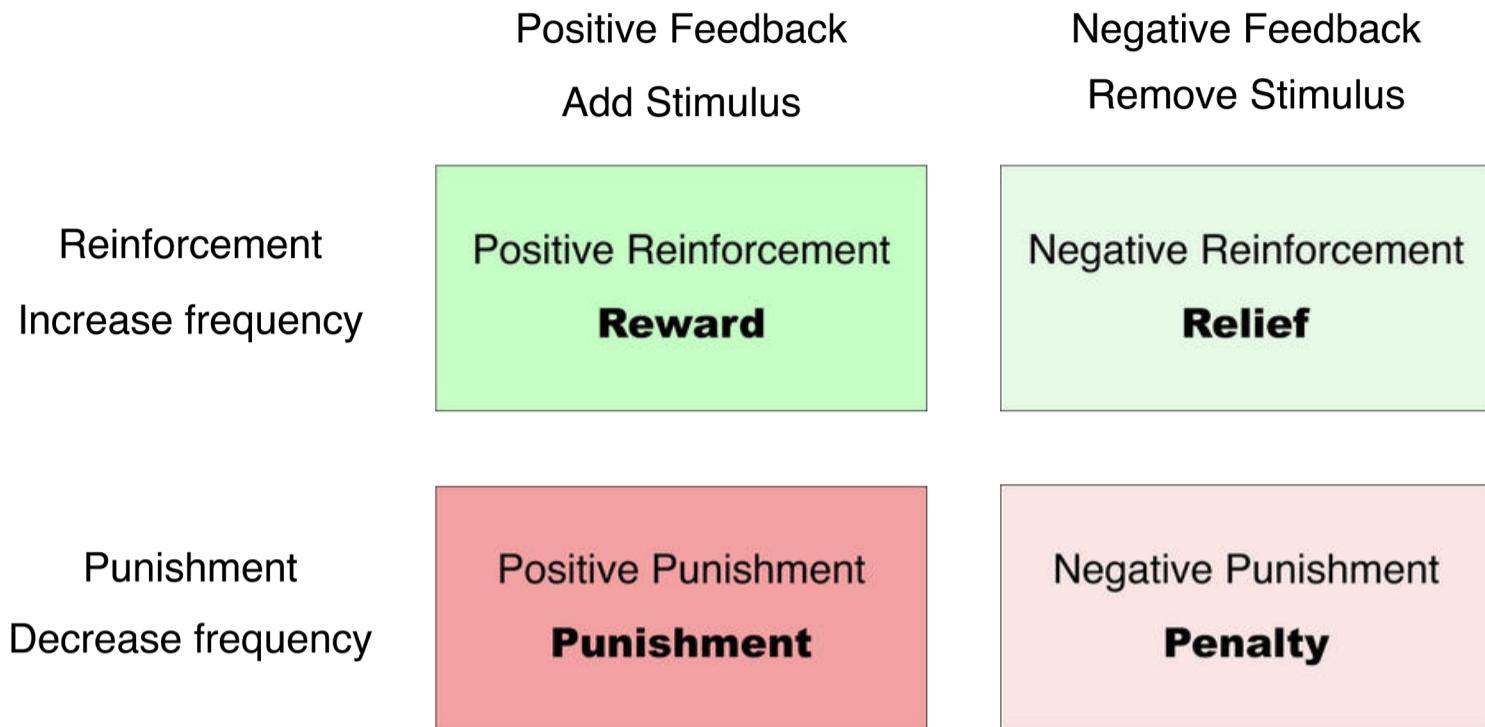


Figure 11.12: The four categories for operant conditioning. When we want to increase the frequency of a behavior, we choose one of the actions in the top row. To decrease the frequency, we choose an action on the bottom row. The left column holds actions that add a stimulus to the environment, and the right columns holds actions that remove a stimulus from the environment (after [DesMaisons12]).

Let's look at some parent-child interactions that demonstrate each case.

When a mother gives her daughter a cookie for cleaning up her room, she's added a stimulus in order to promote more of that room-cleaning behavior. That **reward** is also called **positive reinforcement**. Many animal trainers advocate using only positive reinforcement for training, believing it is both the most effective and humane approach to learning [Dahlo7] [Pryor99].

When a boy who wants ice cream chants, “Please? Please? Please?” over and over, and finally stops only when his father agrees to give him some, the child has removed a stimulus in order to promote more of that giving behavior from his father. The **relief** the child has given to his father constitutes **negative reinforcement**, because something unpleasant has been removed.

When a girl refuses to eat dinner, and her father says she can't have dessert either, the father is removing a stimulus in order to prevent more of that refusing-to-eat behavior. That **penalty** constitutes **negative punishment**, since it's due to the removal of something desirable.

When a mother is in a store with her son and refuses to buy him something he wants, and he throws a tantrum that he knows will upset his mother, her son is adding a stimulus in order to prevent more of the treat-denying behavior. The **punishment** he's dishing out to his mother is **positive punishment**, since it's due to the introduction of something undesirable.

To see how these ideas are used in machine learning, let's consider teaching just a single perceptron, as discussed in Chapter 10. When training this single artificial neuron, we provide feedback only when it makes a mistake, as we attempt to reduce the frequency of this behavior of providing wrong answers. We correct those mistakes by adjusting the weights in the neuron. Since we're introducing changes, we're adding something to the system.

Because we add a stimulus to decrease the frequency of a behavior, this technique falls in the lower-left category of positive punishment.

Many machine-learning training methods use positive punishment, but not all. For example, we'll see in Chapter 26 that reinforcement learning always offers a feedback signal for every action, so it uses both positive reinforcement and positive punishment.

References

- [Allegranza16] Mauro Allegranza, “All men are mortal, Socrates is a man, therefore Socrates is mortal, original quote”, Philosophy Stack Exchange, 2016. <https://philosophy.stackexchange.com/questions/34461/all-men-are-mortal-socrates-is-a-man-therefore-socrates-is-mortal-original>

- [Dahlo07] Cristine Dahl, “Good Dog 101: Easy Lessons to Train Your Dog the Happy, Healthy Way,” Sasquatch Books, 2007.
- [DeMichele16] Thomas DeMichele, “Hume’s Fork Explained”, Fact/ Myth, 2016. <http://factmyth.com/humes-fork-explained/>
- [DesMaisons12] Ted DesMaisons, “A Positive-Minded Primer on Punishment and Reinforcement-with a Buddhist Twist (Part 1 of 2)”, ANIMA Blob, 2012. <http://animalearning.com/2012/12/14/a-positive-minded-primer-on-punishment-and-reinforcement-with-a-buddhist-twist-part-1-of-2/>
- [Domingos12] Pedro Domingos, “A Few Useful Things to Know About Machine Learning”, Communications of the ACM, Volume 55 Issue 10, October 2012. <https://homes.cs.washington.edu/~pedrod/papers/cacm12.pdf>
- [Doyle01] Arthur Conan Doyle, “A Study in Scarlet”, section “About the Author”, House of Stratus, 2001. <http://www.houseofstratus.com/titles/3536-9780755106387-study-in-scarlet>
- [Doyle90] Arthur Conan Doyle, “The Sign of the Four,” 1890. <https://www.gutenberg.org/files/2097/2097-h/2097-h.htm>
- [Doyle92] Arthur Conan Doyle, “The Adventure of the Cardboard Box”, Strand Magazine, 1892. <http://www.gutenberg.org/files/2344/2344-h/2344-h.htm>
- [Fiboni17] Fiboni V.O.F, “Overview of Examples & Types of Syllogisms”, 2017. <https://www.fibonicci.com/logical-reasoning/syllogisms/examples-types/>
- [Fieser11] Jim Fieser, “Philosophy 210: Logic”, 2011. <https://www.utm.edu/staff/jfieser/class/210/11-induction-outline.htm>
- [Garvey16] James Garvey, “The Persuaders: The Hidden Industry That Wants To Change Your Mind”, Icon Books, 2016.

- [Graham16] Jacob N. Graham, “Ancient Greek Philosophy,” The Internet Encyclopedia of Philosophy, 2016. <http://www.iep.utm.edu/greekphi/>
- [Hume77] David Hume, “An Enquiry Concerning Human Understanding,” A. Millar, 1777. <http://www.davidhume.org/texts/ehu.html>
- [Hurley15] Patrick Hurley, “A Concise Introduction to Logic,” 12th edition, 2015. See <http://faculty.bsc.edu/bmyers/Section5.3.htm>
- [Kant81] Immanuel Kant, “Critique of Pure Reason”, 1781. <http://strangebeautiful.com/other-texts/kant-first-critique-cambridge.pdf>
- [Kaplan99] Craig Kaplan, “The Halting Problem,” The Craig Web Experience, 1999. <http://www.cgl.uwaterloo.ca/csk/halt/>
- [Kemerling97] Garth Kemerling, “Categorical Syllogisms”, “The Philosophy Pages”, 1997-2011. <http://www.philosophypages.com/lg/e08a.htm>
- [Pryor99] Karen Pryor, “Don’t Shoot the Dog: The New Art of Teaching and Training Revised Edition”, Bantam, 1999.
- [Skinner51] B.F. Skinner, “How to Teach Animals”, Freeman, 1951.
- [Wikipedia17] Wikipedia authors, “Biological Immortality, 2017. https://en.wikipedia.org/wiki/Biological_immortality
- [Wolpert97] D.H. Wolpert and W.G. Macready, “No Free Lunch Theorems for Optimization”, IEEE Transactions on Evolutionary Computation, 1(1), 1997. <https://ti.arc.nasa.gov/m/profile/dhw/papers/78.pdf>

Chapter 12

Data Preparation

When we train a system with data, our results will only be as accurate as the data we've learned from. We'll see how to process a dataset so that it's ready for efficient and accurate training.

Contents

12.1 Why This Chapter Is Here	433
12.2 Transforming Data.....	433
12.3 Types of Data	436
12.3.1 One-Hot Encoding.....	438
12.4 Basic Data Cleaning	440
12.4.1 Data Cleaning.....	441
12.4.2 Data Cleaning in Practice	442
12.5 Normalizing and Standardizing	443
12.5.1 Normalization	444
12.5.2 Standardization.....	446
12.5.3 Remembering the Transformation.....	447
12.5.4 Types of Transformations	448
12.6 Feature Selection	450
12.7 Dimensionality Reduction	451
12.7.1 Principal Component Analysis (PCA)	452
12.7.2 Standardization and PCA for Images	459
12.8 Transformations	468
12.9 Slice Processing.....	475
12.9.1 Samplewise Processing.....	476
12.9.2 Featurewise Processing.....	477
12.9.3 Elementwise Processing	479
12.10 Cross-Validation Transforms	480
References	486
Image Credits.....	486

12.1 Why This Chapter Is Here

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, damaged hard drives, simulation programs with bugs, or even incomplete transcriptions of paper records. We always need to look at our data and **massage** it, or fix any problems.

A rich body of methods has been developed for just this job. They're referred to as techniques for **data preparation**, or sometimes **data cleansing**, **data cleaning**, or **data scrubbing**. The idea is to pre-process our data *before* learning from it, so that our learner can use the data most efficiently.

This data preparation is essential because most learners are numerical, so the particular way the data is structured can have a strong effect on the information the algorithm can extract from it.

12.2 Transforming Data

When we automate the data cleaning process, we often include some **transformations** that modify the information we're working with. This can involve scaling all the numbers to a given range, or shifting them so their mean is at 0, or even eliminating some superfluous data so that the learner has less work to do.

When we do these things we must always obey a vital principle: When 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 process our training data, we typically modify or even combine the values in ways that are designed to improve the computer's learning efficiency or accuracy. Typically, we examine all of the training

data we've got, decide how to modify it, apply the modifications, and then use that modified data to train our learner. If we later receive new data to evaluate, we *must apply the exact same modifications* to the new data before we give it to our algorithm. This step must not be skipped. Figure 12.1 shows the idea visually.

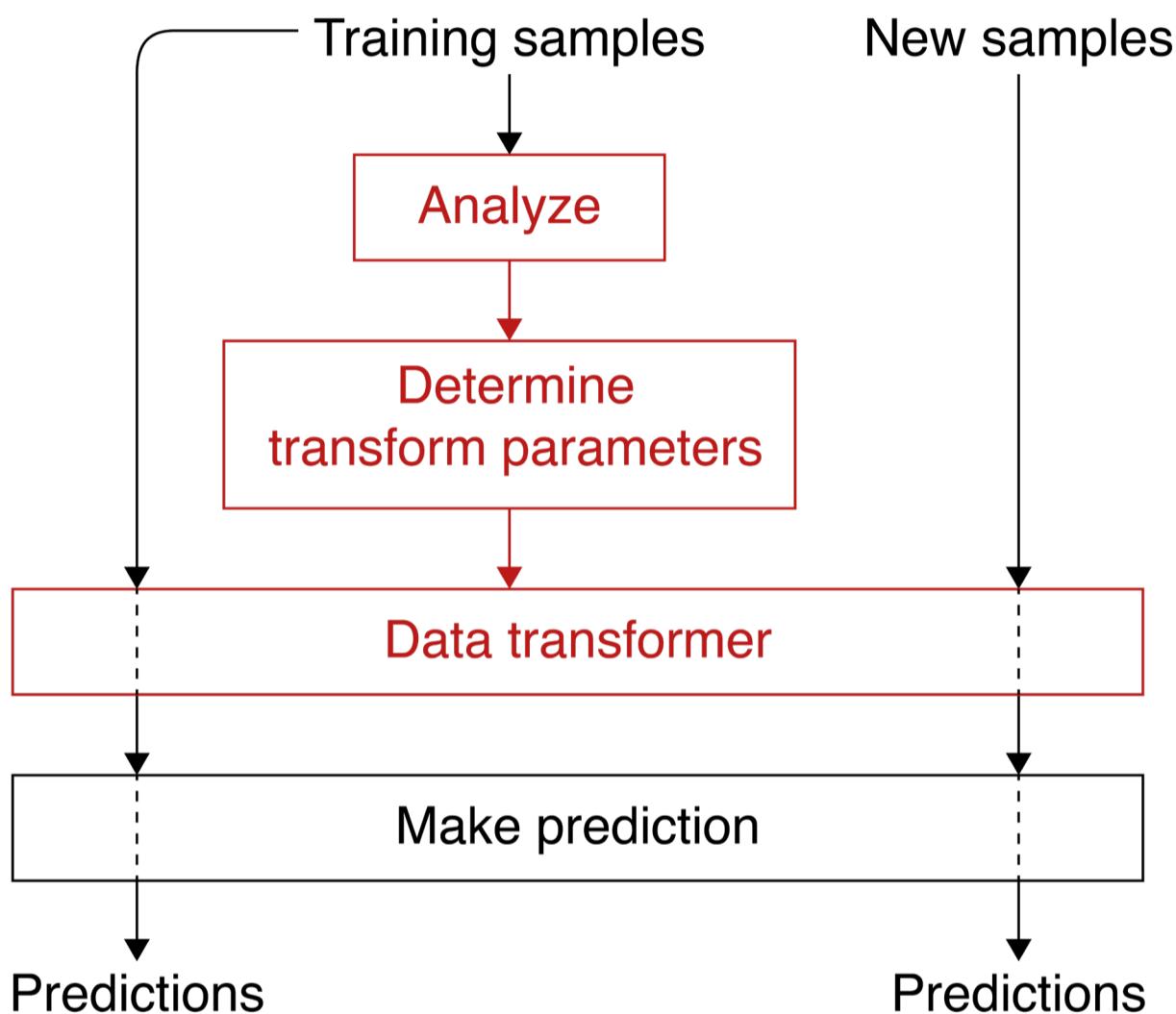


Figure 12.1: The flow of pre-processing for training and evaluating. The training samples are analyzed to determine the parameters for their transformation. From then on, all samples we supply to the network are pre-processed with that same transformation.

The need to re-use 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 could collect a huge number of photos of both animals to use as training data. Both animals have four legs, and a head, and a tail, and so on. What sets them apart are their

distinctive black-and-white markings. To make sure our learner pays attention to these elements, we'll isolate them in each photo, and we'll train with those isolated patches of texture. In other words, this is all that the learner sees. Figure 12.2 shows a couple of these samples.

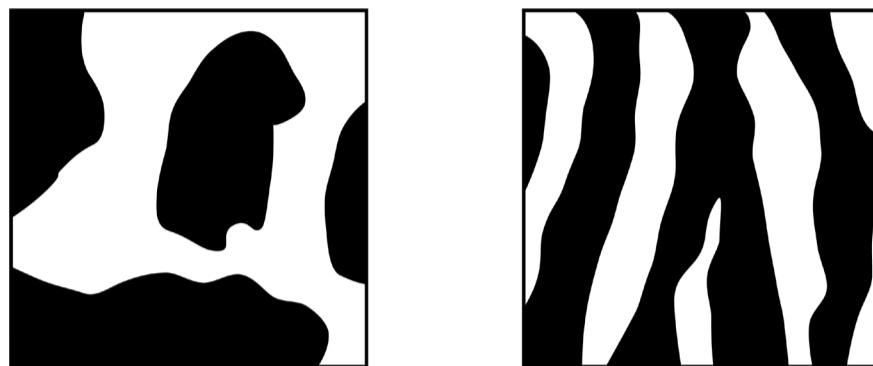


Figure 12.2: Left: A patch of texture from a cow. Right: A patch of texture from a zebra.

Suppose that we've trained our system and deployed it, but we forgot to tell people about this pre-processing step of cropping each image to just the texture. A typical user might then give our system complete pictures of cows and zebras, like those in Figure 12.3, and ask it to identify the animal in each one.



Figure 12.3: Left: A photo of a cow. Right: A photo of a zebra. If we trained our system on the isolated patterns of Figure 12.2, it could be misled by all the extra details in the photos.

As humans, we can pick out the patterns from these photos. But the computer could be misled by the legs, the heads, the ground, and other details, reducing its ability to give us good results.

The difference between the prepared data of Figure 12.2 and the unprepared data of Figure 12.3 could result in a system that performs beautifully on our training data, but completely fails in the real world. To avoid this, all new data like that in Figure 12.3 must be transformed to have the same form as the training data in Figure 12.2.

Forgetting to transform new data in the same way as the training data is an easy mistake to make, and will usually cause our algorithms to under-perform, sometimes to the point of becoming useless.

The rule to remember is this: we determine how to modify our training data, modify it, and *remember how we modified it*. Then any time we deal with more data, we must first *modify that data in the identical way* that the training data was modified before providing it to the algorithm.

12.3 Types of Data

Before we look at specific data transformations, let's consider the types of data we work with, and a common form for representing data that describes categories.

We'll see that a common theme is whether or not we can *sort* a given type of data. Sorting is so useful that if a type of data doesn't have a natural way to be sorted, we'll want to invent a way to make it so.

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 or category, such as "cow" or "zebra." There are two types of categorical data, corresponding 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 it can be sorted. The rainbow colors can be thought of as ordinal, because they have the natural ordering in which they appear in a rainbow, from "red" to "orange" on to "violet." Another example are strings that describe a person at different ages, such as "infant," "teenager," and "elderly." These strings have a natural order, so they can be sorted.

Nominal data is categorical data without a natural ordering. For example, a list of desktop items such as "paper clip," "stapler," and "pencil sharpener" have no natural or built-in ordering. Nor would a list of types of clothing, like "socks," "shirts," "gloves," and "bowler hats."

We can turn nominal data (that is, strings without an order) into ordinal data just by defining an order and sticking to it. We could assert that the order of clothing, for instance, should be from head on down, so our previous example would have the ordering "bowler hats," "shirts," "gloves," and "socks," thereby becoming 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.

We often convert string data, both nominal and ordinal, into numbers so that they're easier to work with in the computer. The easiest way to do this is to just identify all the strings in our database (and any others we expect could be in later data) and assign each string a unique number starting with 0. Many libraries provide built-in routines to create and apply this transformation.

12.3.1 One-Hot Encoding

In machine learning it's sometimes more convenient to work with a list of numbers rather than a single value. For instance, we might build a classifier that looks at a photograph and tells us which of 10 different objects that photo shows. The classifier could output just a single value, such as 3 or 7, telling us which category it thinks is the most likely.

But we often want to find out how likely the other categories were, too. Then we'd like to get back a list with one value for each category, where the size of the value represents the classifier's confidence that the photo belongs in that category. The first entry would indicate category 1, the second entry category 2, and so on. So if it's in category 5, then the fifth entry in the list would be the largest, but the other categories might have some smaller values, suggesting that the classifier thinks those choices shouldn't be completely ruled out.

We can make some of the steps in the process more convenient if the output of the categorizer, and the label we assigned to the sample, are both in the form of lists.

Converting a label like 3 or 7 into the right kind of list is called **one-hot encoding**, referring to only one entry in the list being “hot”, or marked. We sometimes say that we've turned our single number into a **dummy variable**, referring to the whole list. So when we provide category labels to the system during training, we provide this one-hot encoded list, or dummy variable, rather than a single integer.

To get started, if we have categorical data, we convert it to numerical data as above, by assigning each different value a unique number starting with 0.

Now each time we see one of these values, we replace it with a list of 0's. The list is as long as our total number of possible values. Then we place a 1 in the slot corresponding the value we want to represent.

Let's see this in action. Figure 12.4(a) shows the eight colors in the original 1903 box of Crayola Crayons [Crayola16]. Let's suppose these colors appear as strings in our data.

Colors in our data	Assignment of a number to each value	One-hot encoding of each color
red	red → 0	red → [1, 0, 0, 0, 0, 0, 0, 0]
yellow	yellow → 1	yellow → [0, 1, 0, 0, 0, 0, 0, 0]
blue	blue → 2	blue → [0, 0, 1, 0, 0, 0, 0, 0]
green	green → 3	green → [0, 0, 0, 1, 0, 0, 0, 0]
orange	orange → 4	orange → [0, 0, 0, 0, 1, 0, 0, 0]
brown	brown → 5	brown → [0, 0, 0, 0, 0, 1, 0, 0]
purple	purple → 6	purple → [0, 0, 0, 0, 0, 0, 1, 0]
black	black → 7	black → [0, 0, 0, 0, 0, 0, 0, 1]

Figure 12.4: One-hot encoding for the original eight Crayola colors in 1903. Suppose that these are strings in our input data. (a) The original 8 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 8 numbers, all of which are 0 except for a 1 in the position corresponding to that string's value.

We'll assign a value from 0 to 7 to each of these 8 strings (more typically, we'll call a library routine to do this for us), as in Figure 12.4(b). From now on, any time we see one of these strings used in our data, we'll replace it with a list of 8 values, as in Figure 12.4(c). Each value is 0 except for a 1 placed in the index corresponding to the value of the string.

Figure 12.5 shows using one-hot encoding with a set of samples with multiple features. Each sample has two numbers and one string, so we one-hot encode the string and leave the numbers alone.

Original Data	Assignments	One-hot Encoded Data
[3, 1, "socks"]	"bowler hats" → 0	[3, 1, [0, 0, 0, 1]]
[2, 7, "gloves"]	"shirts" → 1	[2, 7, [0, 0, 1, 0]]
[4, 1, "socks"]	"gloves" → 2	[4, 1, [0, 0, 0, 1]]
[1, 3, "bowler hats"]	"socks" → 3	[1, 3, [1, 0, 0, 0]]
[1, 3, "shirts"]		[1, 3, [0, 1, 0, 0]]

Figure 12.5: When a sample has multiple features, we can convert individual features into one-hot form while leaving the others alone. (a) Our input data has 3 features. We'll convert the string to one-hot encoding. (b) The library assigns a unique number to each string, starting at 0. (c) The one-hot encodings of the strings.

Because the one-hot representation is easier for the computer but less convenient for us, we normally do this operation near the end of data preparation, so we don't have to deal with interpreting all those 0's and 1's.

12.4 Basic Data Cleaning

Let's review some simple things to make sure that our data is well cleaned (or well **groomed**), before we actually start applying any kind of wide-scale modifications.

If our data is in textual form, then we want to make sure there are no typographical errors, misspellings, embedded unprintable characters, or other obvious issues that could prevent a clean interpretation. For example, if we have a collection of animal photos along with a text file describing them, we want to make sure that every giraffe is labeled as “giraffe,” and not “girafe” or other typos or variants like “giraffe-very tall”. We’d also want to fix any case errors, like “Giraffe” when we’re

expecting all lower-case. The computer will assign a different number to each different string, so every reference to a giraffe needs to use the identical string.

There's a handful of other common-sense things we should look for.

12.4.1 Data Cleaning

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 a single piece of data is accidentally included multiple times, our learner will interpret it as multiple, different samples that just happen to have the same value, and thus that sample will have more influence than it should.

We also want to make sure we don't have any crazy errors, like missing a decimal point and using a value of 1000 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, when 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 received something else.

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, there are different ways to typeset numbers using scientific notation, and programs can easily misinterpret forms they're not used to. The value 0.007 is commonly printed out in scientific notation form as $7e-3$, but when we use that as input to another program it might be interpreted as $(7 \times e) - 3$, where e is Euler's constant (about 2.7). The result is that the computer thinks we're providing it a value that's a little more than 16, rather than about 0.007. We need to catch these sorts of things before we feed them to our software.

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 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.

We also want to identify any pieces of data that seem dramatically different from all the others. Data points that are significantly out of the norm are called **outliers**. Some of these might be mere typos, like the forgotten decimal point above. Others might be the result of human error, like accidentally copying in part of one data set into another, or forgetting 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, then 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.

12.4.2 Data Cleaning in Practice

Though the steps above may seem straightforward, carrying them out in practice 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.

There are many tools available to help us clean data quickly. Some are standalone, and others are built into machine-learning libraries. There are also commercial services who will clean data for a fee.

It's useful to keep in mind the 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 can make it.

The issues we've discussed in this section are often considered **pre-processing**, involving local fixes to individual elements.

With the small stuff taken care of, we can now turn our attention to the large-scale data transformations that modify our data so we can improve our ability to learn from it.

12.5 Normalizing and Standardizing

We often work with samples whose features span very different numerical ranges.

For instance, suppose we collected data on a herd of African bush elephants. Our data describes each elephant with 4 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 very different ranges of numbers. Speaking roughly, because of the numerical nature of the algorithms we use, the computer will consider the larger numbers to be more important than the smaller ones. But those large numbers are little more than accidental. For instance, if we'd chosen to measure the age in decades rather than hours, the range for that feature would be 0-5 rather than 0-420,000.

In this list, one of our features can take on negative values. Because this is the only source of negative values, this anomaly can also influence how the computer treats those values.

We want all of our data to be roughly comparable, so that our choices of units and other arbitrary decisions don't influence how the system learns from the data.

12.5.1 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.

We’ll be using the word in its statistical sense, where it means scaling the data so that it lies in some specific range. We usually choose the range as either $[-1,1]$ or $[0,1]$, depending on the data and what it means (it wouldn’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. We’ll see examples of this in Chapters 15 and 23.

Figure 12.11 shows a 2D data set 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, again to help us see how the points move. The colors have no other meaning.

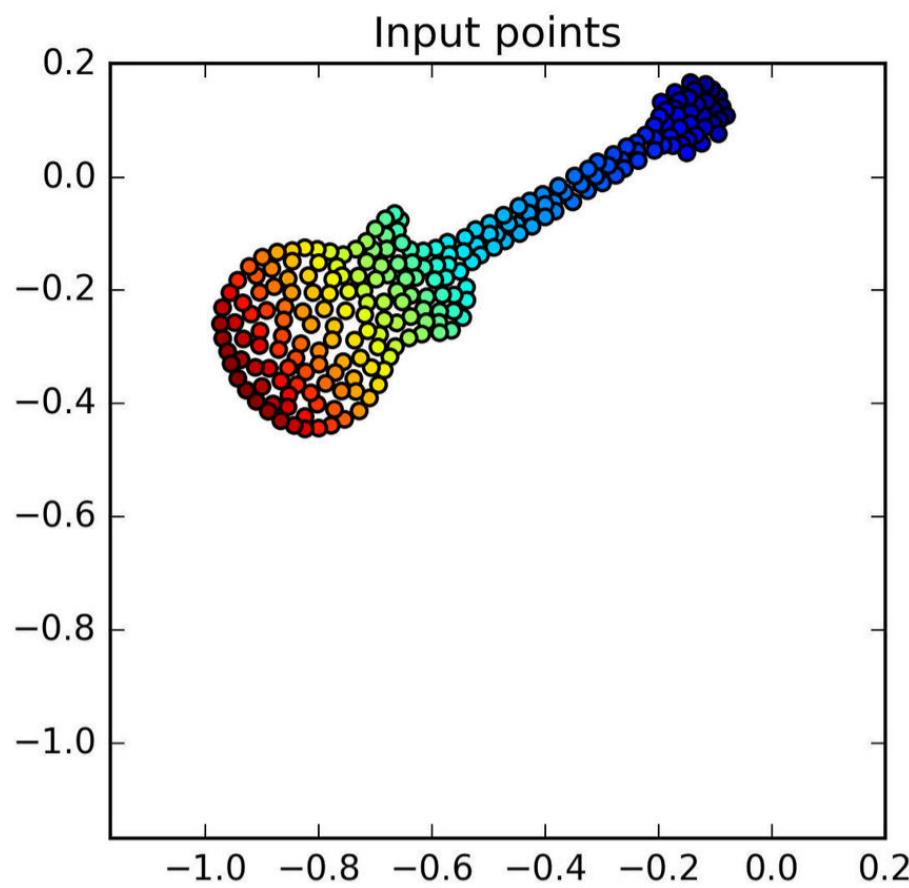


Figure 12.6: A guitar shape made of 232 points. Each point is a sample described by its two features, named X and Y. The colors are there just to help us keep track of the points from one figure to the next, and don't mean anything else.

Typically, these points would be the results of measurements, say the age and weights of some people, or the tempo and volume of a song. To keep things generic, we'll call the two features X and Y.

Figure 12.7 shows the results of normalizing each feature our guitar-shaped data to the range $[-1,1]$ on each axis. That is, the X values are scaled from -1 to 1 , and the Y values are independently scaled from -1 to 1 . The shape formed by this operation has skewed a little bit because it's been stretched vertically more than horizontally.

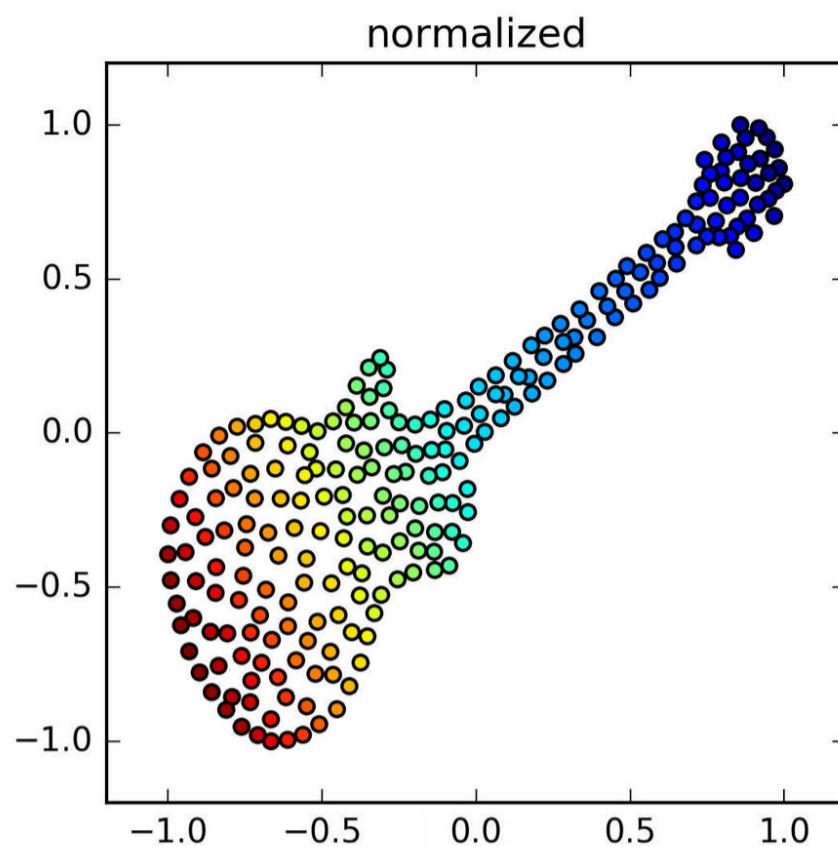


Figure 12.7: The data of Figure 12.6 after normalization to the range $[-1,1]$ on each axis. The skewing of the shape is due to it being stretched more along the Y axis than the X.

12.5.2 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 that feature is zero (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% of the values in that feature will then lie in the range -1 to 1 . In our 2D example, the X values are stretched or compressed horizontally until about 68% 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 there will be points outside of the range $[-1, 1]$, so our results are different than what we get from normalization.

As with normalization, most libraries will offer a routine to standardize any or all our features in one call.

Figure 12.12 shows the application of standardization to our starting data in Figure 12.11.

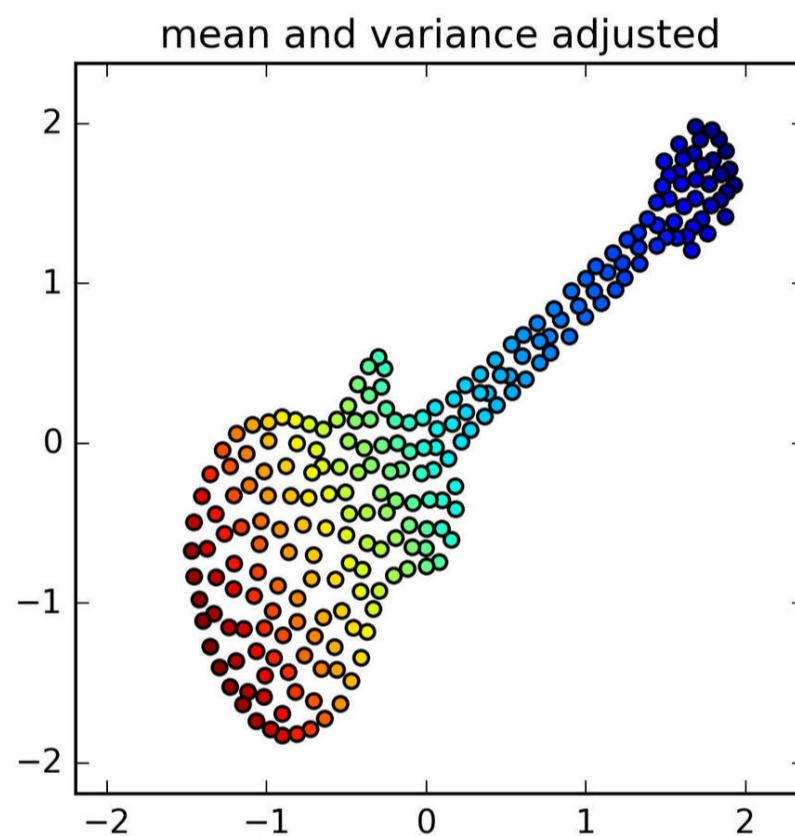


Figure 12.8: When we standardize our data, we move it horizontally and vertically so that the mean value of each feature is 0, and then we stretch or compress it horizontally and vertically so that a standard deviation of that feature (that is, about 68% of the data) lies between -1 and 1 on each axis.

12.5.3 Remembering the Transformation

Both normalization and standardization routines are controlled by parameters that tell them how to do their jobs. Those parameters are found by the library routines that analyze the data before they apply the transformation.

Because it's so important to transform future data with the same operations, these library calls will always give us a way to hang onto those 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 will almost never itself be 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% 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.

12.5.4 Types of Transformations

Some transformations are **univariate**, which mean 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 (like before, the name comes from combining *multi* for many, with *variate* for *variable* or *feature*).

As a simple example, consider a normalization transformation, like the one we saw above. This is a univariate transformer, because it treats each feature as a separate set of data to be manipulated. That is, it scales all the X values to $[0,1]$, and independently scales all the Y values to $[0,1]$. The two sets of features don't interact in any way. In other words, how the X axis gets scaled does not depend at all on the Y values, and vice-versa.

Figure 12.9 shows this ideal visually for a normalizer applied to data with 3 features.

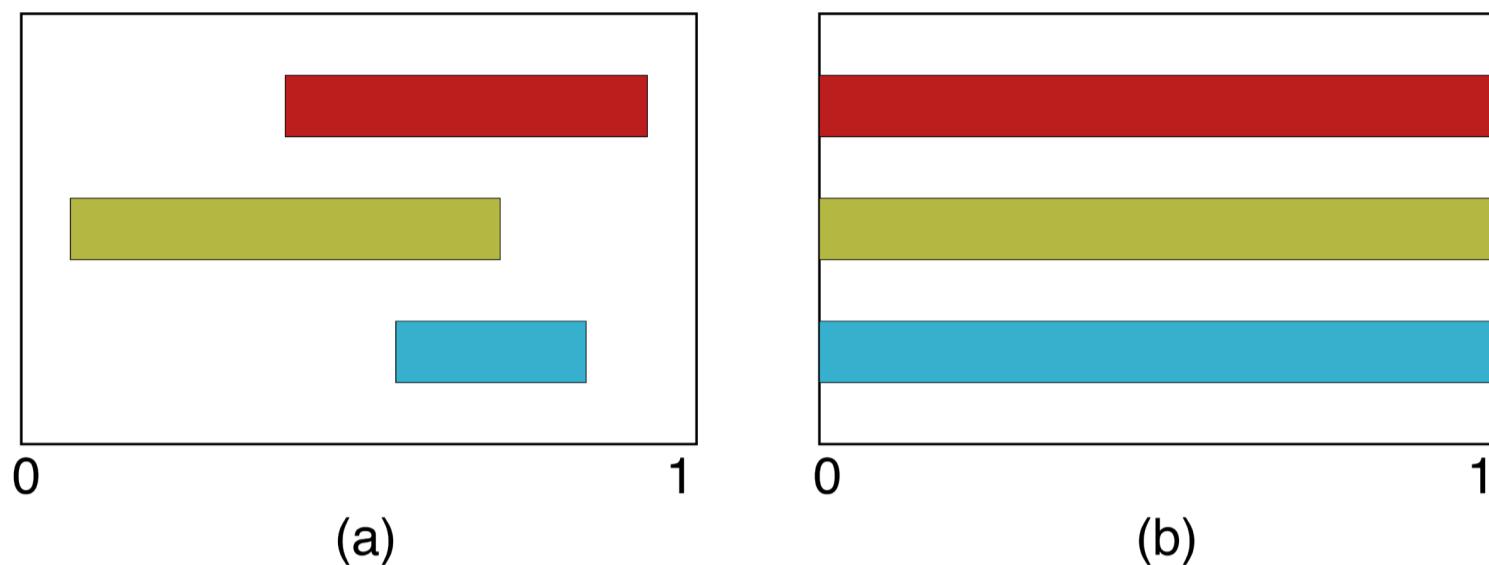


Figure 12.9: When we apply a univariate transformation, each feature is transformed independently of the others. Here we are normalizing the features to the range [0,1]. (a) The starting ranges of three features. (b) Each of the three ranges is independently shifted and stretched to the range [0,1].

By contrast, a multivariate algorithm looks at multiple features at a time and treats them as a group. The most extreme (and probably 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, then we move and stretch them all as a group until they *collectively* fill the range [0,1]. This is illustrated in Figure 12.10.

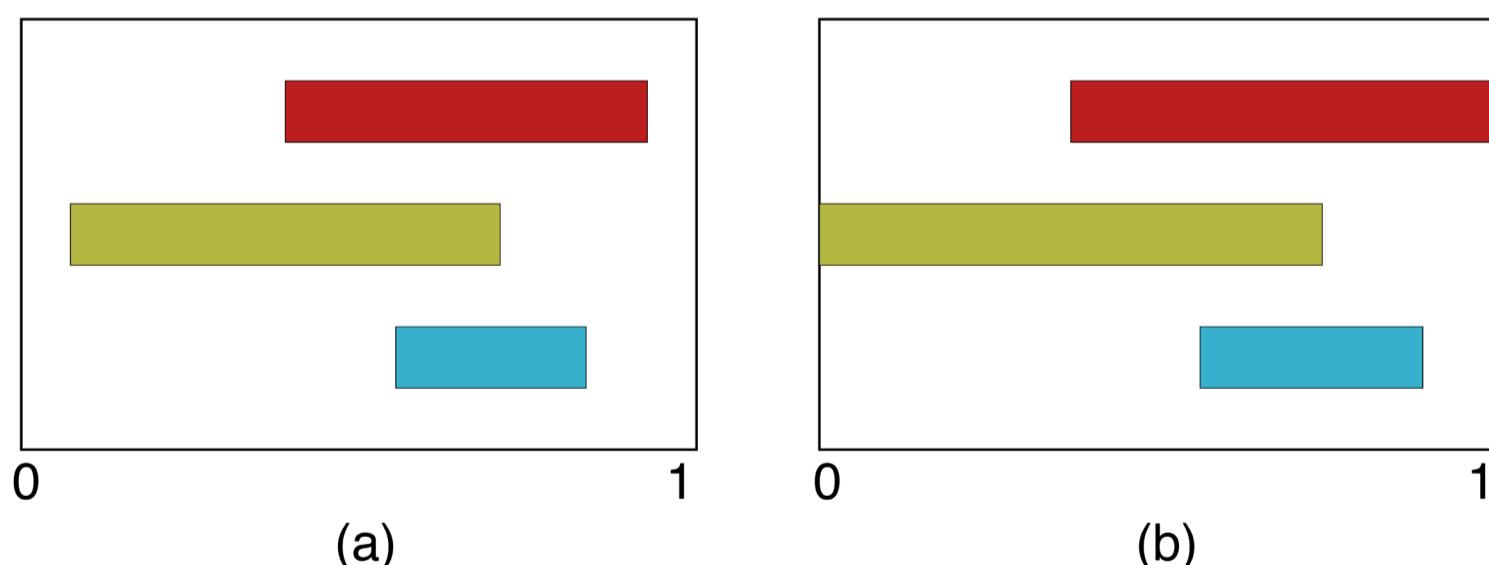


Figure 12.10: When we apply a multivariate transformation, we treat multiple features simultaneously. Here we are again normalizing to the range [0,1]. (a) The starting ranges of three features. (b) The bars are shifted and stretched as a group so that their collective minimum and maximum values span the range [0,1].

Many transformations can be applied in either a univariate or multivariate way. We choose based on our data.

For instance, the univariate version made sense above when we scaled our X and Y samples because they're essentially independent. But suppose our features were temperature measurements made at different times over the course of different days. We'd probably want to scale all the features together, so that as a collection they span the range of temperatures we're working with.

12.6 Feature Selection

It stands to reason that the less data we need to process during training, the faster our training will go. So if there's a way to reduce the amount of work the computer has to do while maintaining the same efficiency and accuracy of learning (or something close), then this would be a good thing.

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 is called **feature selection**, or sometimes **feature filtering**.

Let's consider an example where some data is actually **superfluous**, or unnecessary. 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. Including that data not only isn't going to help the computer identify one elephant from another, it will also slow us down. We ought to just remove that useless field from our data.

We can generalize this idea to removing features that are *almost* useless, or contribute very little, or simply make the least contribution to getting the right answer.

Continuing with our images of elephants, we might create spreadsheet entries for each animal’s height, weight, last known latitude and longitude, trunk length, ear size, and so on. But it might be that for this species the trunk length and ear size are closely correlated. Then we could remove (or filter) either one and still get the benefit of the information they jointly represent.

Many libraries will automatically estimate the impact of removing each field from a database. We can then use that as guidance to simplify our database and speed our learning without sacrificing more performance 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.

12.7 Dimensionality Reduction

Another approach to reducing the size of our dataset is to combine 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 is some of the features in our data might be somewhat **redundant**. We can improve learning performance without losing information by compressing our database to remove this repetition.

As we mentioned earlier, we might be working with a species of elephant for which the trunk length and ear size are *correlated*: when one goes up, so does the other. So if we know either of these measurements, we can make a very good guess at the value of the other. As a result, we could compress our database a little by replacing those two fields with just one field, perhaps containing just one of these values, or perhaps some kind of combination of them. This is common in human physiology, where the *body mass index*, or BMI, is a single number that combines height and weight [CDC17].

Let's look at a tool that automatically determines how to select and combine features to make the smallest impact on our results.

12.7.1 Principal Component Analysis (PCA)

Principal component analysis, or **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. Our presentation follows that of [Dulchi16].

Figure 12.11 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.

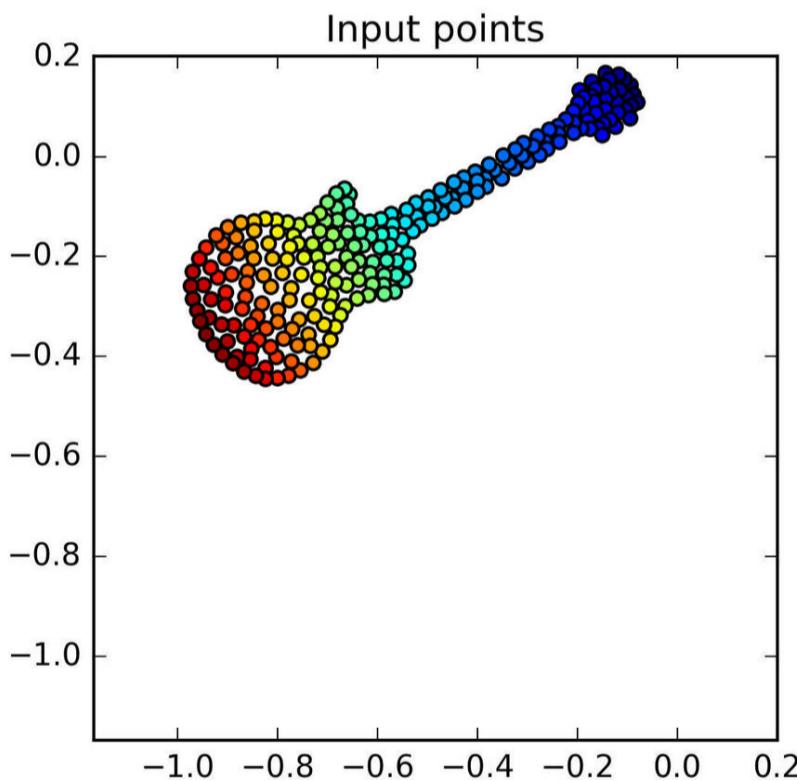


Figure 12.11: The starting data for our discussion of PCA. As before, the colors don't have any meaning, but are there just to help us watch what happens to each data point as it's processed.

Our goal will be to crunch this two-dimensional data down to one-dimensional data. That is, we'll replace each pair of X and Y with a single new number based on both of them, just as the BMI is a single number that combines a person's height and weight. When we use PCA on a real dataset, we might combine many sets of features simultaneously.

We'll start by standardizing the data. Figure 12.12 shows this combination of mean normalization followed by variance normalization, as we saw before.

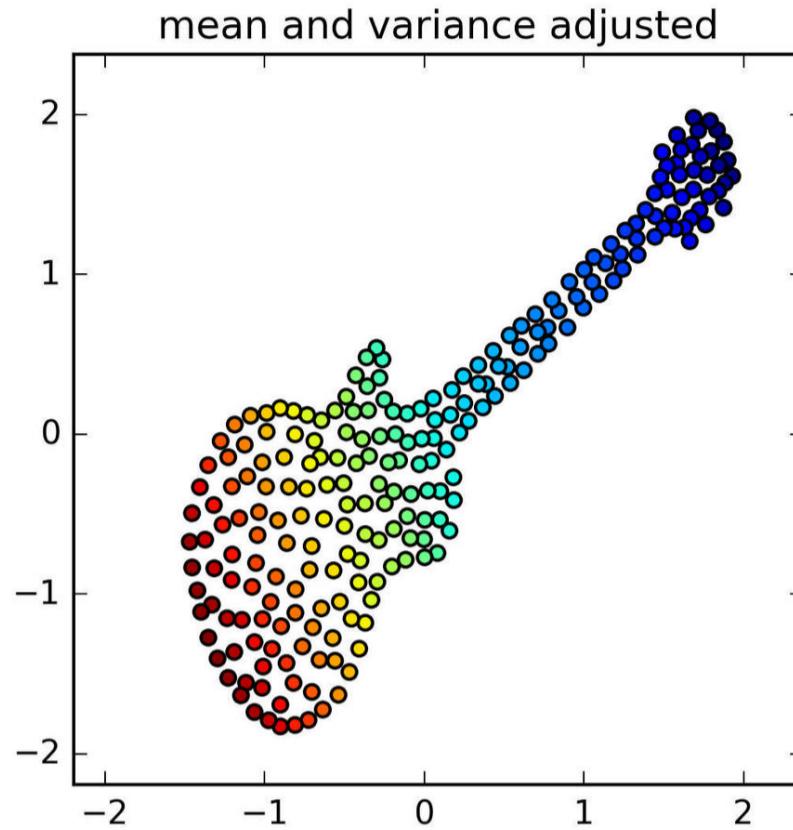


Figure 12.12: Our input data after standardizing.

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. The process of doing this is called **projection**. Because our line is horizontal, we only need to move our points up or down to find their closest point on the projection line.

The results of projecting the data in Figure 12.12 onto a horizontal projection line are shown in Figure 12.13.

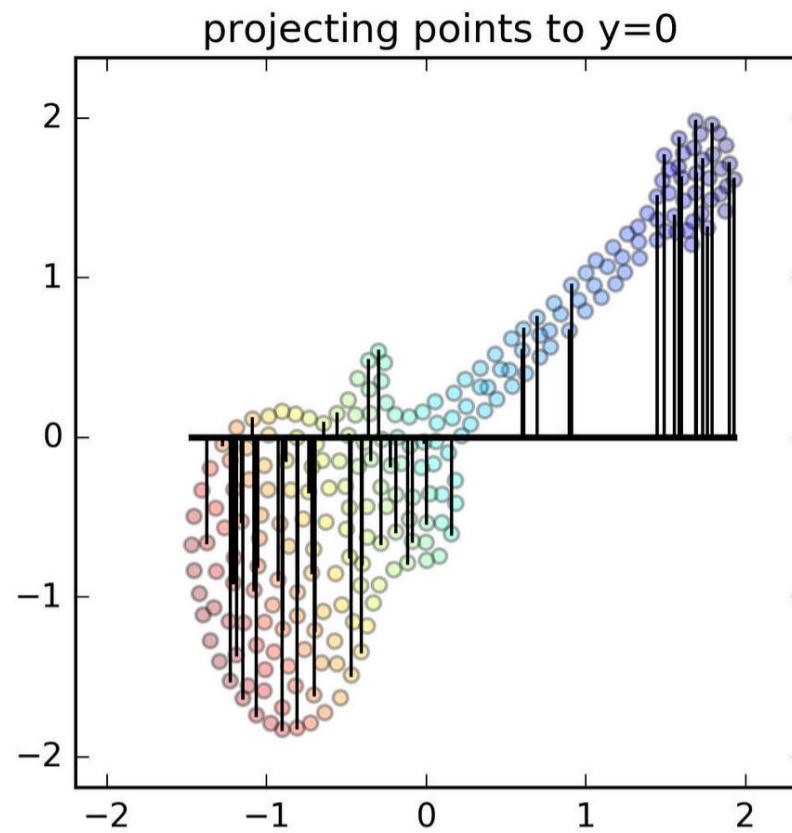


Figure 12.13: We project each data point of the guitar by moving it to its nearest point on the projection line. For clarity, we're showing the path taken by only about 25% of the points.

The results after all the points are processed is in Figure 12.14.

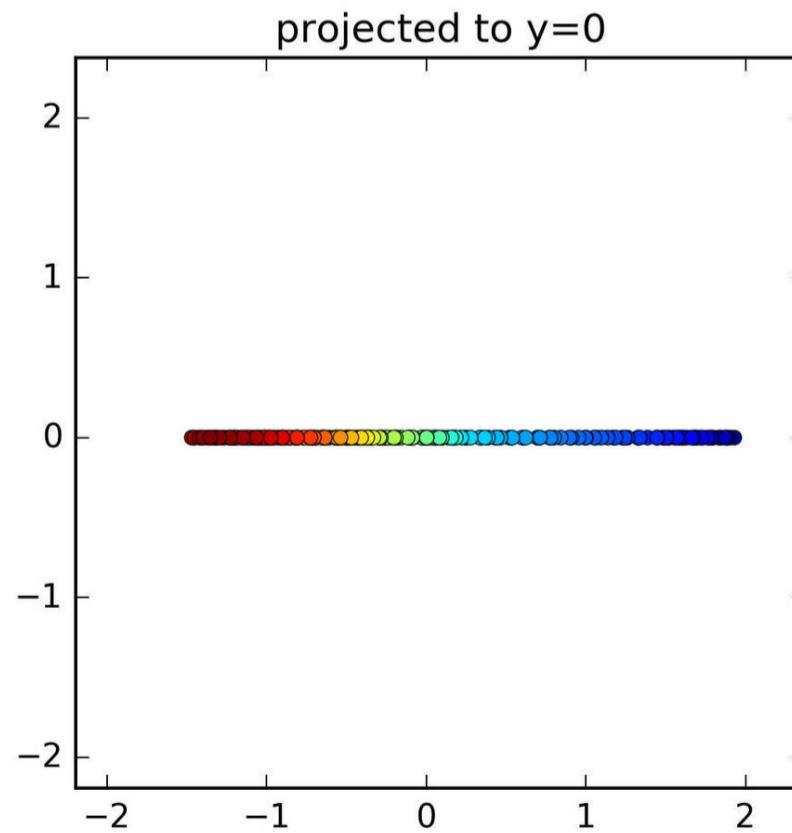


Figure 12.14: The result of Figure 12.13 after all the points have been moved to the projection line. Each point now is described only by its X coordinate, so we now have a one-dimensional dataset.

This is the 1-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 the body-mass index by simply using the weight and ignoring the height.

To improve the situation, we'll 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, will have the largest range of points.

Any library routine that implements PCA will find this line for us automatically. Figure 12.15 shows this line for our guitar data.

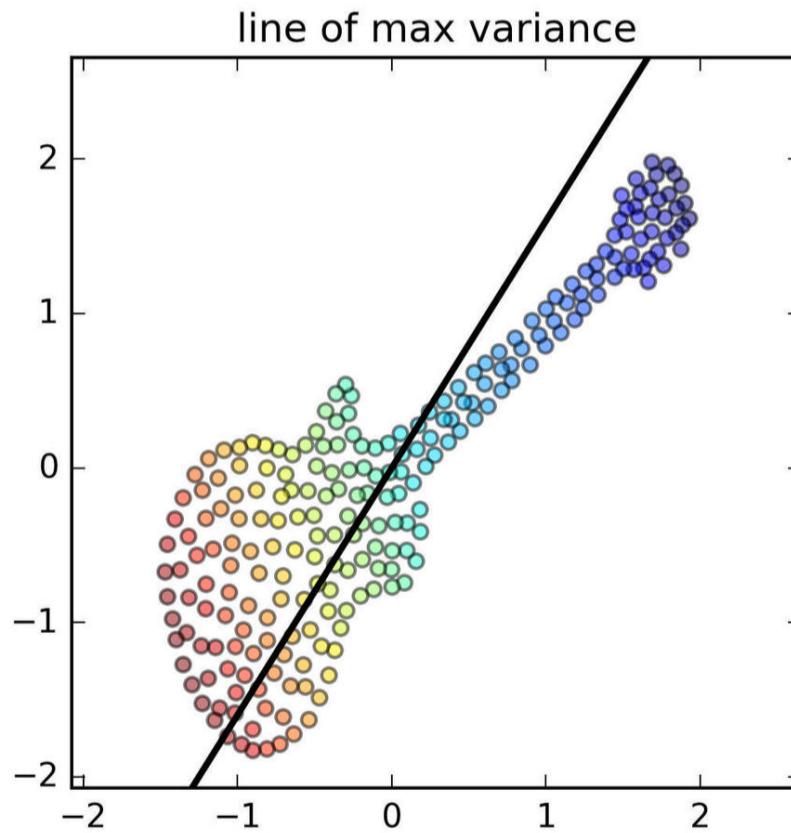


Figure 12.15: The thick black line is the line of maximum variance through our original data. This will be our projection line.

Now we continue just like before. We'll 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 12.16 shows this process.

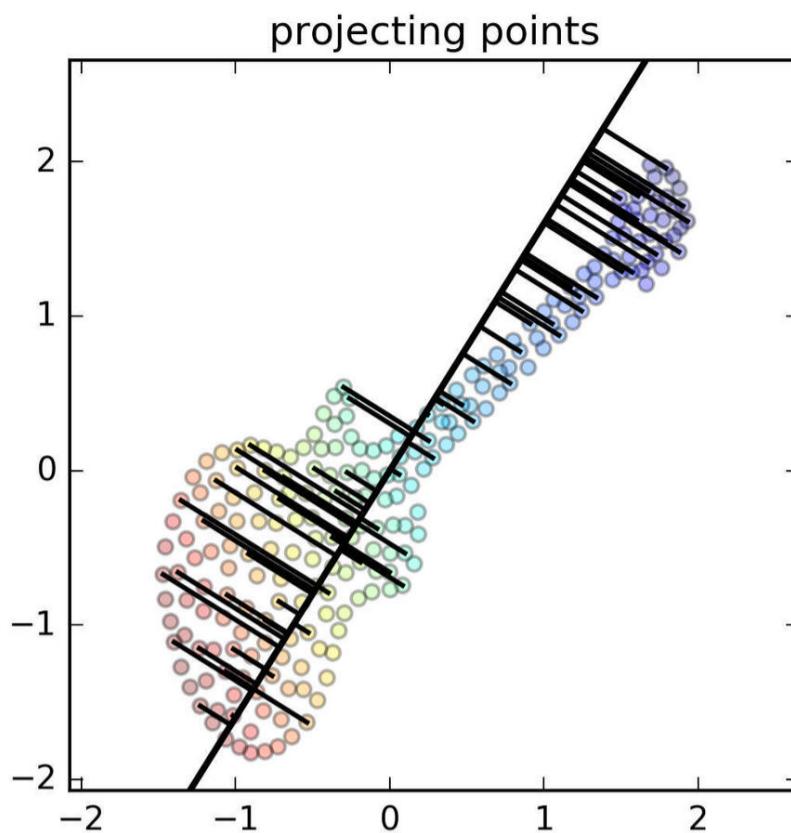


Figure 12.16: We project the guitar data onto the projection line by moving each point to its closest point on the line. For clarity, we're showing the path taken by only about 25% of the points.

The projected points are shown in Figure 12.17. Note that they all lie on the line of maximum variance that we found in Figure 12.15.

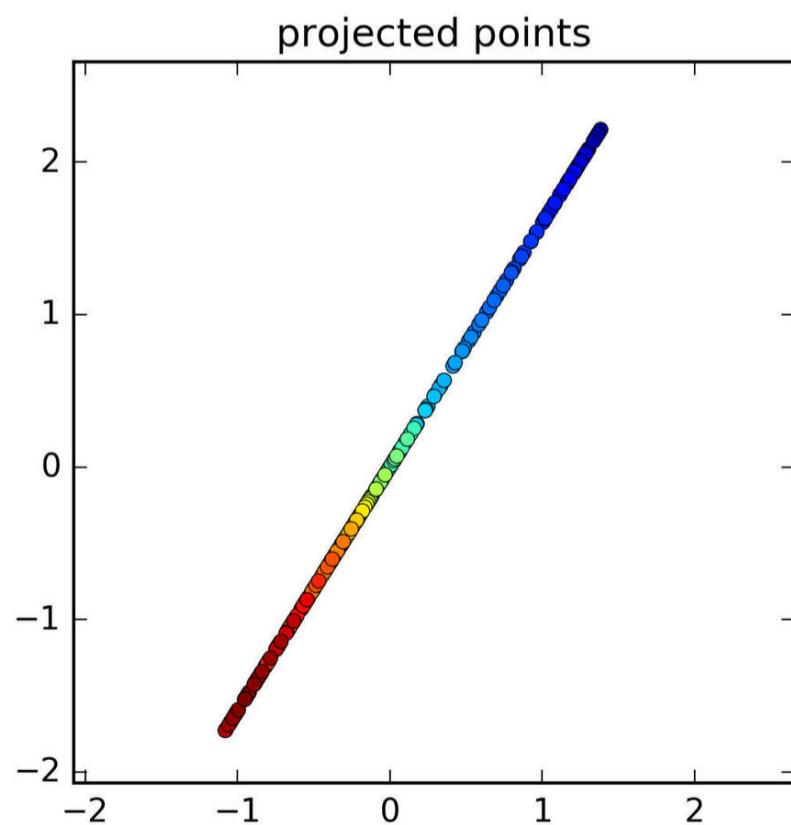


Figure 12.17: The points of the guitar data set projected onto the line of maximum variance.

These points lie on a line, but they're still 2D. To fully reduce them to 1D, we can rotate them until they lie on the X axis, as shown in Figure 12.18. 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.

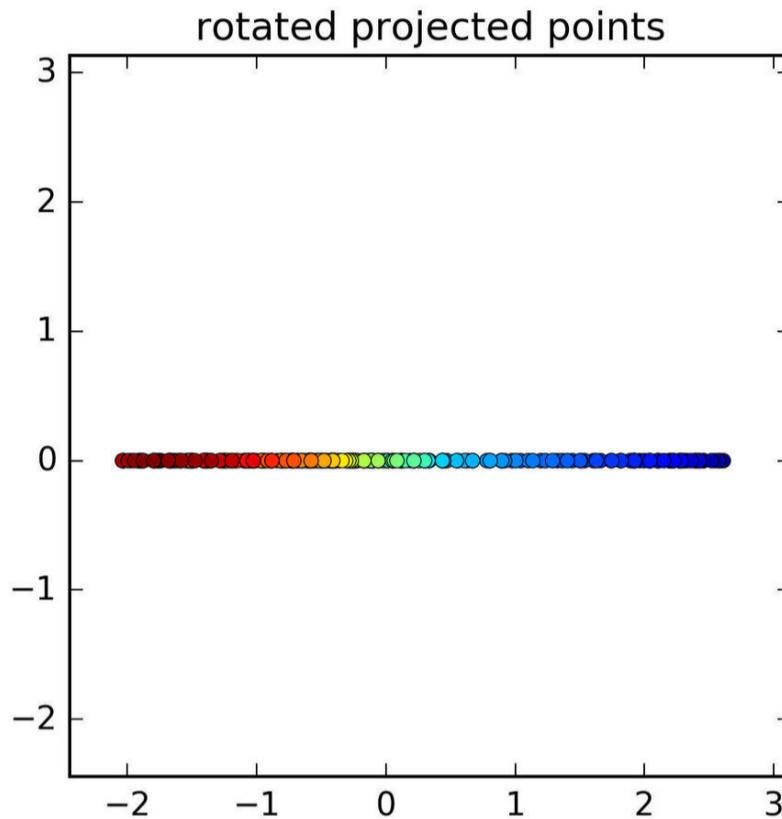


Figure 12.18: Rotating the points of Figure 12.17 into a horizontal position. Every point is now described only by its X value, so we've turned our 2D data into 1D data. Note that we didn't simply eliminate one dimension, because both the original X and Y values contributed to this one-dimensional data.

Although the straight line of points in Figure 12.18 looks like the line of points in Figure 12.14, 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.

All of these steps are carried out automatically by a machine learning library when we call its PCA routine.

The beauty of this new data is that every point's single value (its X location) is a combination of the 2D data it started with. We've reduced the dimensionality of our dataset by one dimension, but we've done so while retaining as much information as we could. Our learning algorithms now only have to process 1D data rather than 2D, so they'll learn faster.

Of course, we've thrown some information away, so we could suffer from reduced learning speed or accuracy. 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 had 3D data, we could imagine placing a plane in the middle of the cloud of samples, and projecting our data down onto the plane. The library's job would be to find the best orientation of that plane. This would take our data from 3D to 2D. If we wanted, we could use the same technique as above and imagine a line through the volume of points, taking us from 3D down to 1D.

In practice we can use this technique in problems with any number of dimensions, where we might reduce the dimensionality of the data by tens or more.

The critical questions for this kind of algorithm are how many dimensions we should try to compress, which ones should be combined, and how they get combined. We usually use the letter k to stand for the number of dimensions in our data after PCA has done its work. So in our guitar example, k was 1.

In that sense, we can call k a parameter of the algorithm, though we usually 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 kind of unfortunate, so it's important to pay attention to the context when we see references to k for one purpose or another.

Compressing too little means our training and evaluation steps will 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** that we'll cover in Chapter 15.

As always, whatever PCA transformations we use to compress our training data must also be used in the same way for all future data.

12.7.2 Standardization and PCA for Images

Images are an important and special kind of data. Let's apply standardization and PCA to a set of images.

A color image has 5 values at each point: X, Y, red, green, and blue, so we'd say the dataset is 5-dimensional, or 5D. For simplicity, we'll stick to monochrome images here. Each pixel has just 3 values, X, Y, and gray, giving us a 3D dataset.

In this context, we don't think of the X and Y values as features like the grayscale values, because we don't want to change the image dimensions. So we think of our pixels as features that have one value (a level of gray), and X and Y are just indices that we use to name the samples.

We'll think of each pixel as a feature. So if our images are 100 pixels wide and 50 pixels high, each image has $100 \times 50 = 5000$ features. There is a lot of spatial information in these features, meaning that pixels that are near to one another in any direction often will share some kind of relationship. But to keep things simple, for now we'll ignore that spatial information. Instead, we'll just turn our 2D image into a new 1D list. We'll start with the first row of the image, append the second row, then the third, and so on as in Figure 12.19. When we're done we'll have a list with one entry for each pixel in the image.

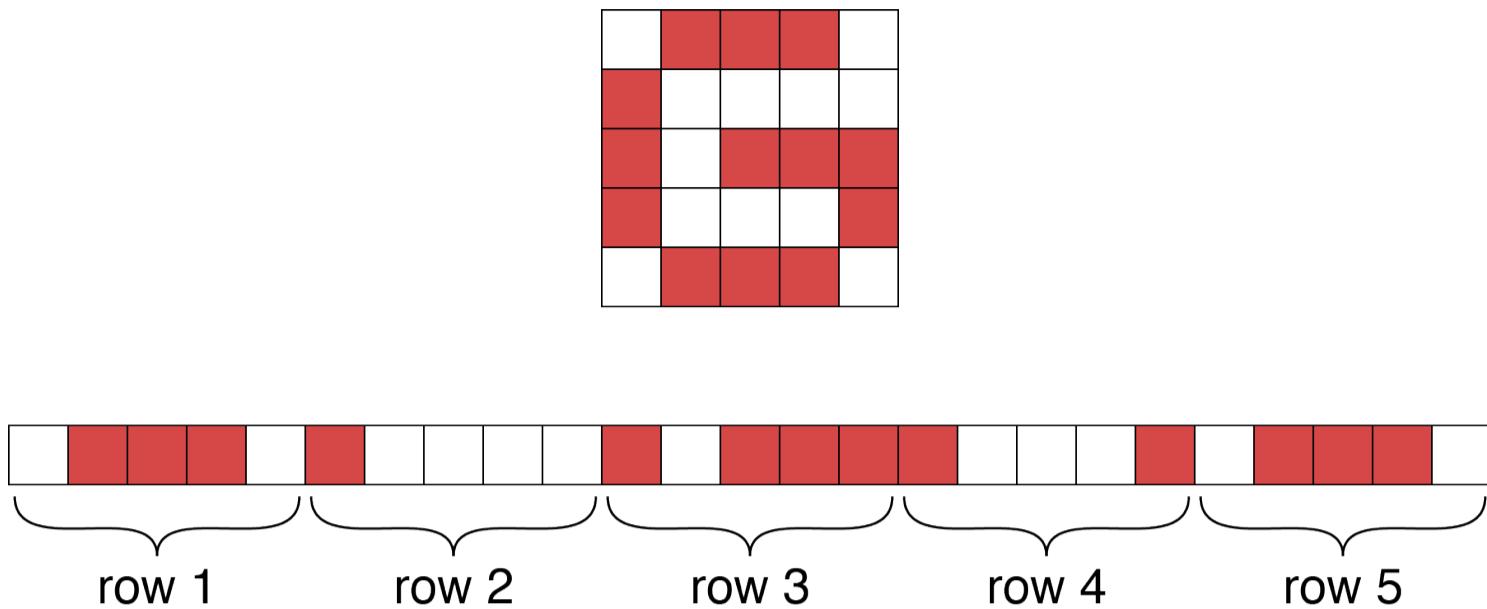


Figure 12.19: Turning an image into a list. Top: Our input image. Bottom: The list formed by laying one row after the other, from top to bottom.

As we saw above, when we use a tool like PCA we want our features to have a mean of 0 and a standard deviation of 1. So let's process our pixels that way [Turk91]. The top rows of Figure 12.20 show five flattened images, each of size 5 by 5. The bottom rows show those same images after this processing. In effect, we consider each column one at a time, process the pixels in that column, and then save those new values.

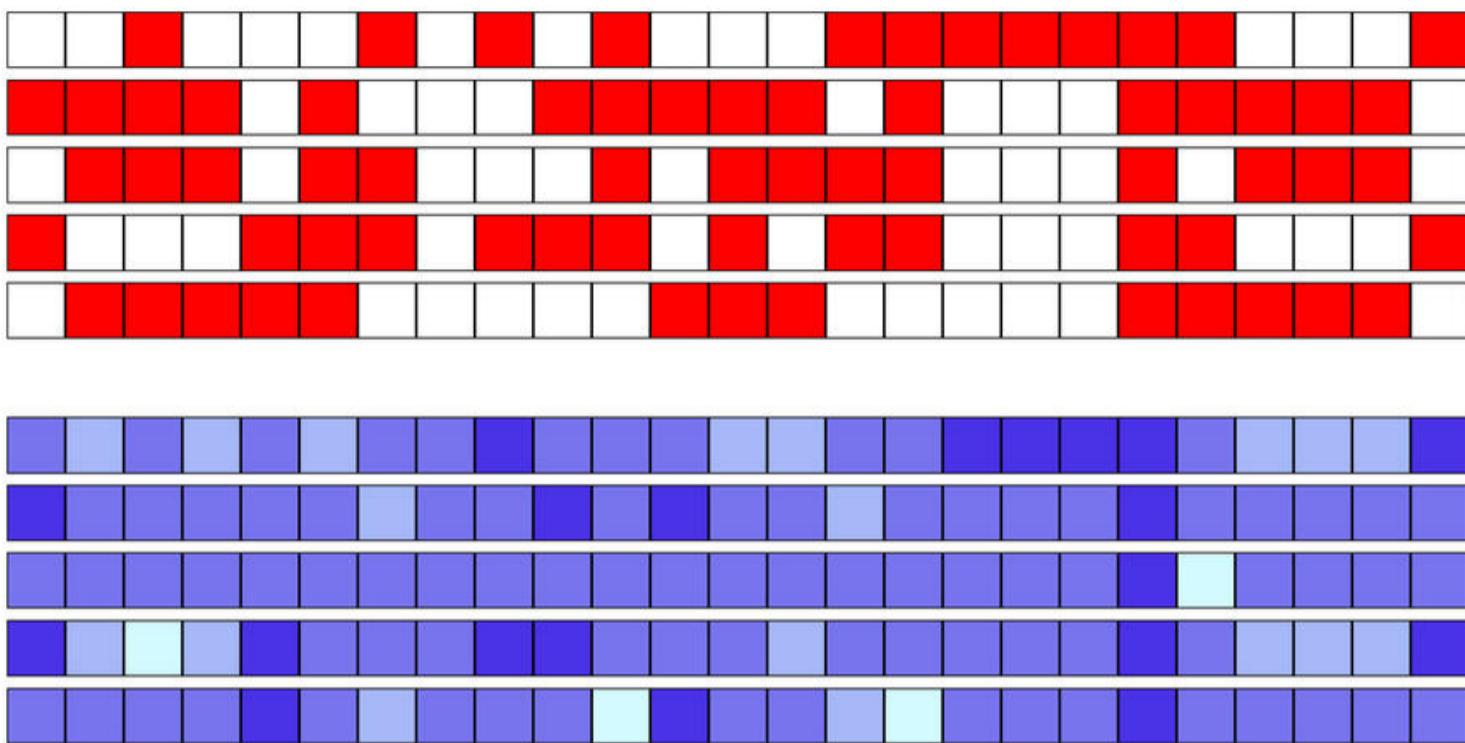


Figure 12.20: Processing our image lists on a per-feature (or per-pixel) basis. The top five rows show flattened out binary images for the letters A, B, G, M, and S. We examine each column one at a time, and adjust it to have a mean of 0 and standard deviation of 1. The resulting pixels are shown in the bottom five rows, scaled so the smallest value is light blue, and the largest value is dark blue.

It may seem odd that we’re processing each pixel independently, as that means we’re ignoring all of the spatial information between pixels (in later chapters we’ll see other approaches that do use that information). But for now, our mental model is that each image is simply a big list of independent feature values, as if they were measurements of temperature, wind speed, humidity, etc.

Let’s put our discussion above into action. We’ll begin with the 6 pictures of Huskies shown in Figure 12.21. These were aligned by hand so that the eyes and nose were 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 12.21: Our starting set of Huskies.

A database of six dogs isn't a lot of training data, so we'll enlarge our database by running through our set of six images in random order over and over. Each time through, we'll make a copy, and then randomly shift it horizontally and vertically up to 10% on each axis, rotate it up to 5° clockwise or counter-clockwise, and maybe flip it left to right. Then we'll add that transformed image to our training set. Figure 12.22 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 4000 dog images.



Figure 12.22: Each row shows a set of new images created by shifting, rotating, and perhaps horizontally flipping each of our input images. We used this process to make our training database of 4000 images.

Since we'd like to run PCA on these images, our first step is to standardize them. This means we'll analyze the same pixel in each of our 4000 images, and adjust the collection to have zero mean and unit variance, as we saw in Figure 12.20. We'll standardize all 4000 images. The standardized versions of our first 6 generated dogs are shown in Figure 12.23.



Figure 12.23: Our first six Huskies after standardization.

In our earlier discussion of PCA we projected 2D data onto a single 1D line. Our Husky images are 64 by 45, so each one has $64 \times 45 = 2880$ features. PCA will project these to as many directions as we ask, generating a new, projected image each time.

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 different weights, will best reconstruct the input images. Each of the projections found by PCA is technically known as an **eigenvector**, from the German *eigen* meaning “same,” 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 12.24 shows our 12 **eigendogs**.

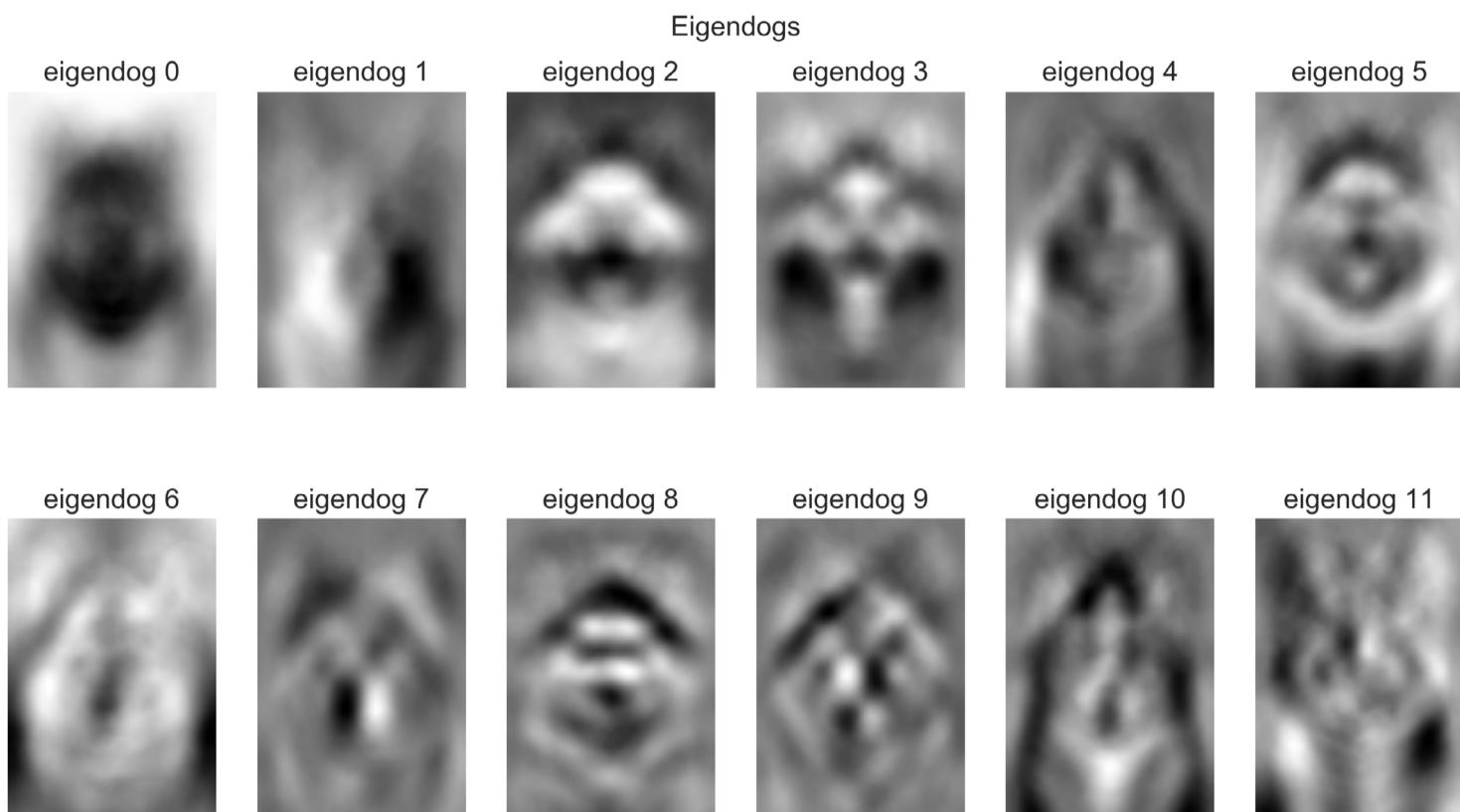


Figure 12.24: The 12 eigendogs produced by PCA.

Looking at the eigendogs tells us a lot about how PCA is analyzing our images. The first eigendog is a big black smudge roughly where most of the dogs appear in the image. The second eigendog seems to be capturing some of the left-right shading differences. Each eigendog seems to be a little more complex than the ones before, capturing different details between the images.

Let's now see how well we can recover our original images by combining the eigendogs with different weights. Figure 12.25 shows the best weights that PCA could find for each input image. We created the reconstructed dogs by scaling each eigendog image from Figure 12.24 with its corresponding weight, and then adding the results together.

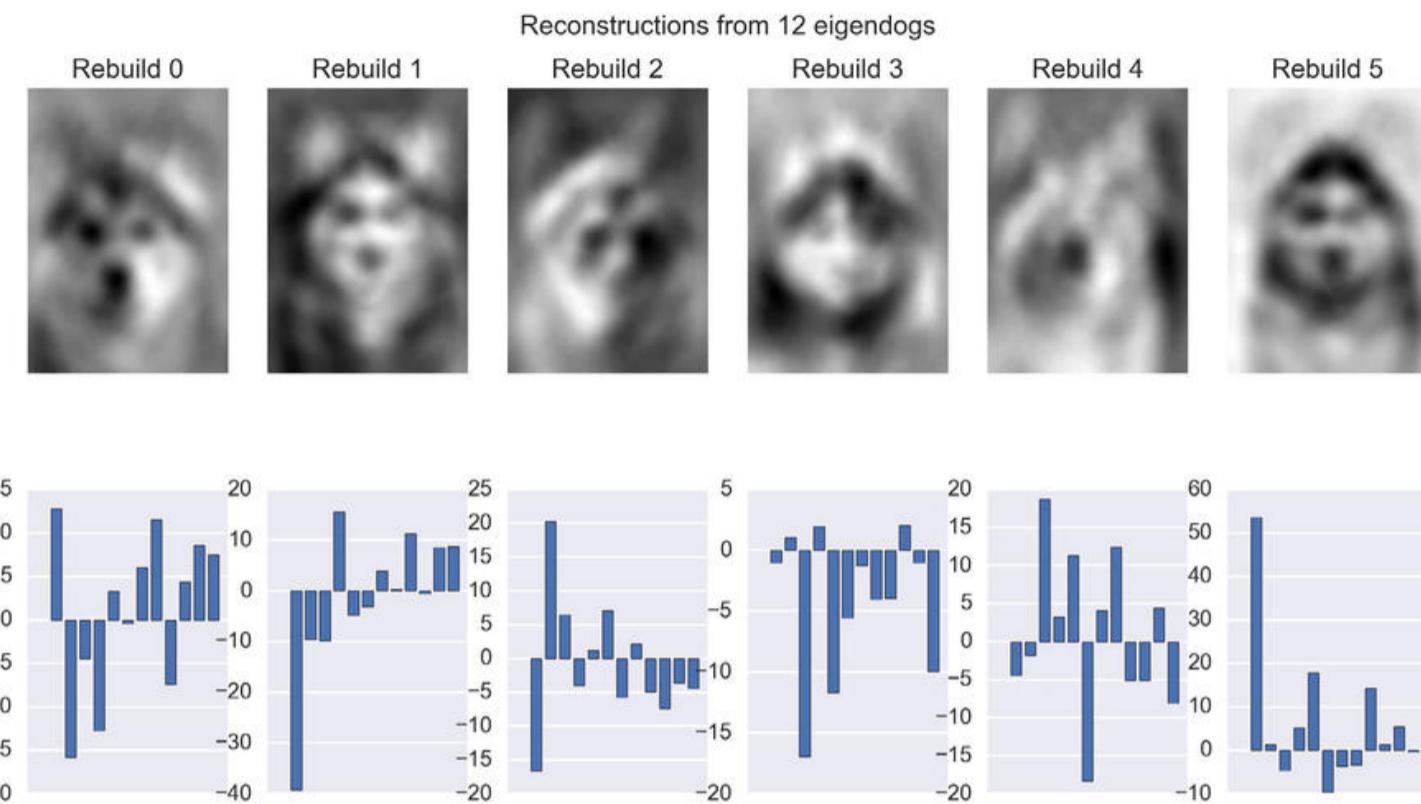


Figure 12.25: Reconstructing our original inputs from a set of 12 eigendogs. Top row: The reconstructed dog. Bottom row: The weights applied to the eigendogs of Figure 12.24 to build the image directly above. Notice that the vertical scales on the weights are not all the same.

Figure 12.25 is not great. We've asked PCA to represent all 4000 images in our training set with just 12 pictures. It did its best, but some of these don't look much like dogs.

Let's try using 100 eigendogs. The first 12 eigendog images look just like those in Figure 12.24, but then they get more complicated and detailed. The results of reconstructing our first set of 6 dogs are shown in Figure 12.26.

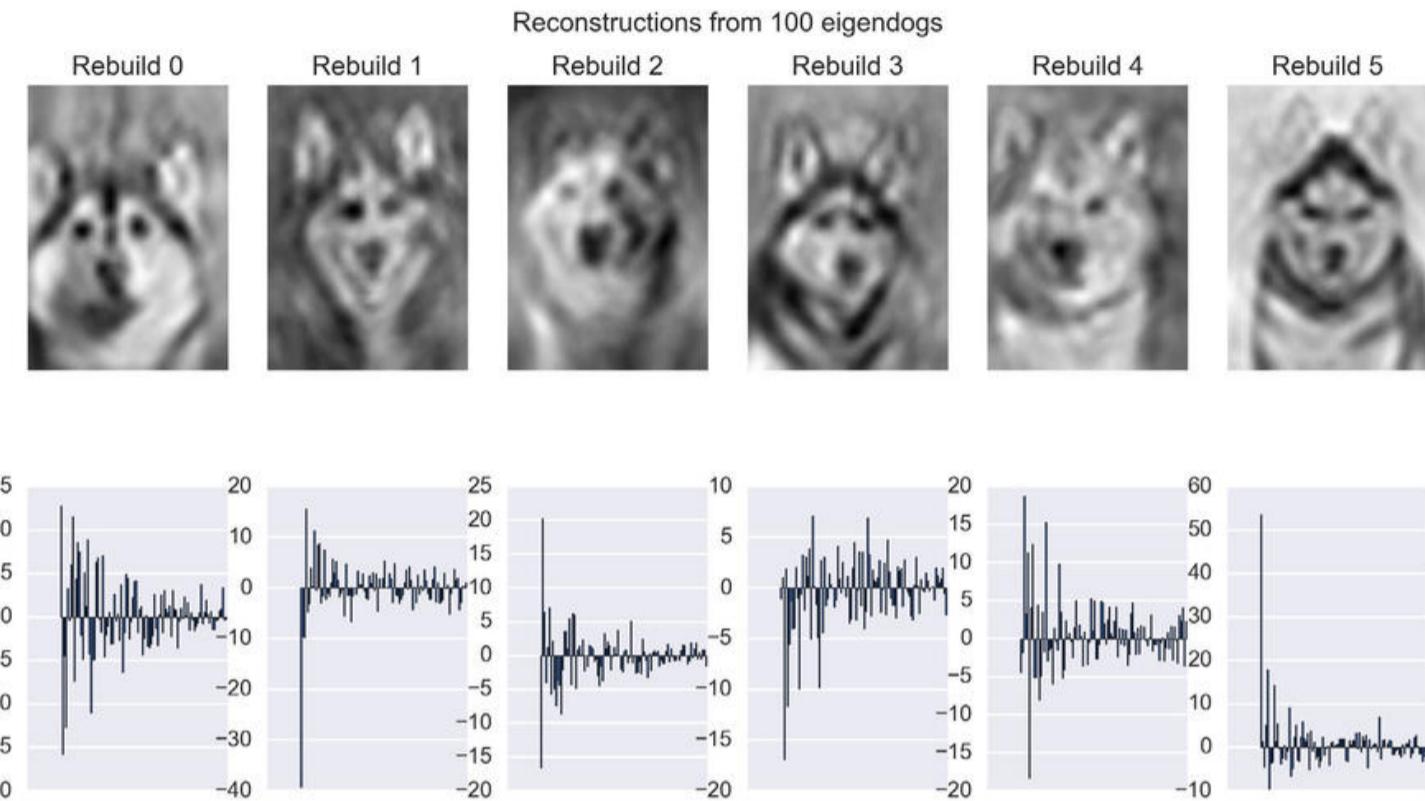


Figure 12.26: Reconstructing our original inputs from a set of 100 eigendogs. Top row: The reconstructed dog. Bottom row: The weights applied to the eigendogs. Notice that the vertical scales are not all the same.

That's better! They're starting to look like dogs, though a couple of them don't have any obvious ears, which is a pretty important part of a dog's face.

Let's crank up our number of eigendogs to 500 and try again. Figure 12.27 shows the results.

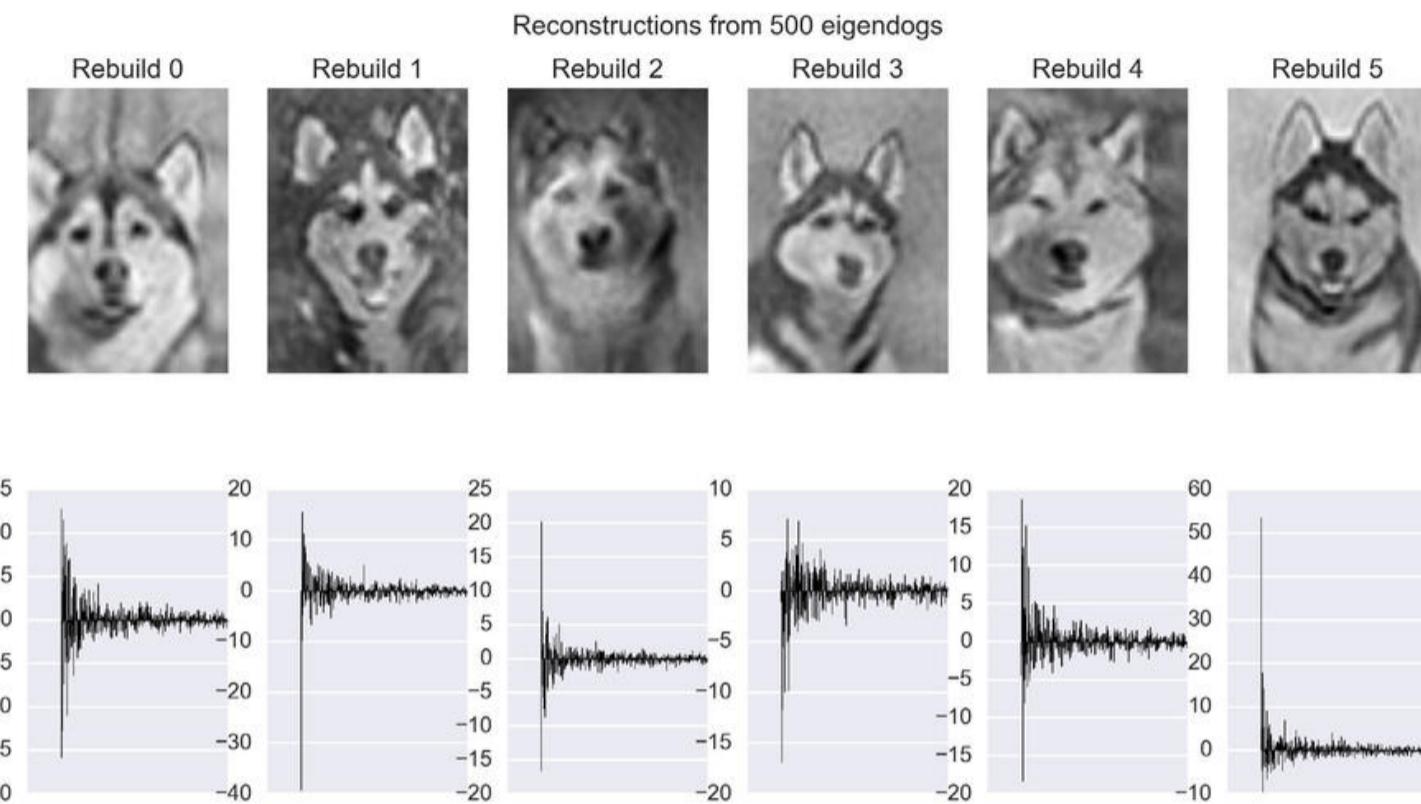


Figure 12.27: Reconstructing our original inputs from a set of 500 eigendogs. Top row: The reconstructed dog. Bottom row: The weights applied to the eigendogs. Notice that the fourth set has a different vertical scale.

That's looking pretty great. These are all easily recognized as the 6 standardized dogs in Figure 12.23. There's a little bit of noise, but starting with 500 images and a weight for each one we've done a fine job of matching these images. There's nothing special about these first 6 images. If we look at any of the 4000 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 becoming a little sharper and less noisy.

Notice that in each reconstruction, 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 for us right now is not that we can make images that look just like the starting set, but rather that we can use the eigendogs to help a categorizer classify dogs for us.

For example, rather than training our categorizer on all 2880 pixels from each image, we could train it on just its 100 or 500 weights. The categorizer never sees a full image. 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 the weights, and the computer gives us back a category based on those values. This can save a lot of computation, which translates to a savings in time.

12.8 Transformations

Let's look more closely at the steps involved in computing and applying transformations. We'll also look at how we'll sometimes want to undo those steps, so we can more easily compare our results to our original data.

Let's 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 the middle of winter, so we expect temperatures both above and below freezing (0° Celsius).

So 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 12.28.

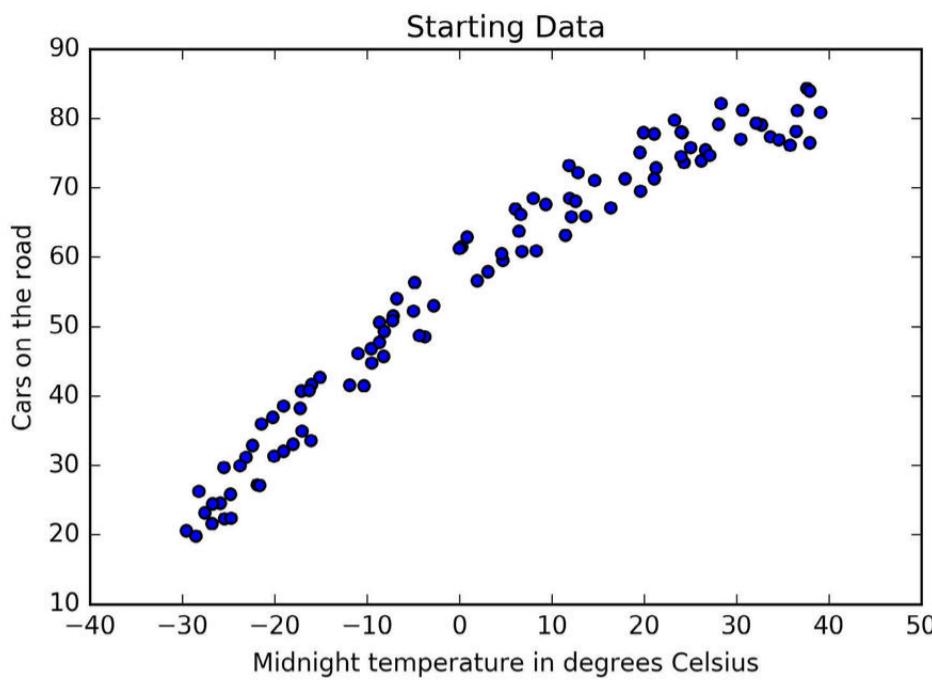


Figure 12.28: Each midnight we measure the temperature, and then the following morning we measure the number of cars on the road between 7 and 8 AM.

We'd like to give this data to a machine-learning system that will learn the connection between temperature and traffic density. This is a regression problem, where we'll feed in a sample consisting of one feature, describing the temperature in degrees, and we get back a real number telling us the number of cars on the road.

Let's suppose that the regression algorithm we're using works best when its input data is scaled to the range $[0,1]$. So we'll normalize the data to $[0,1]$ on both axes, as in Figure 12.29.

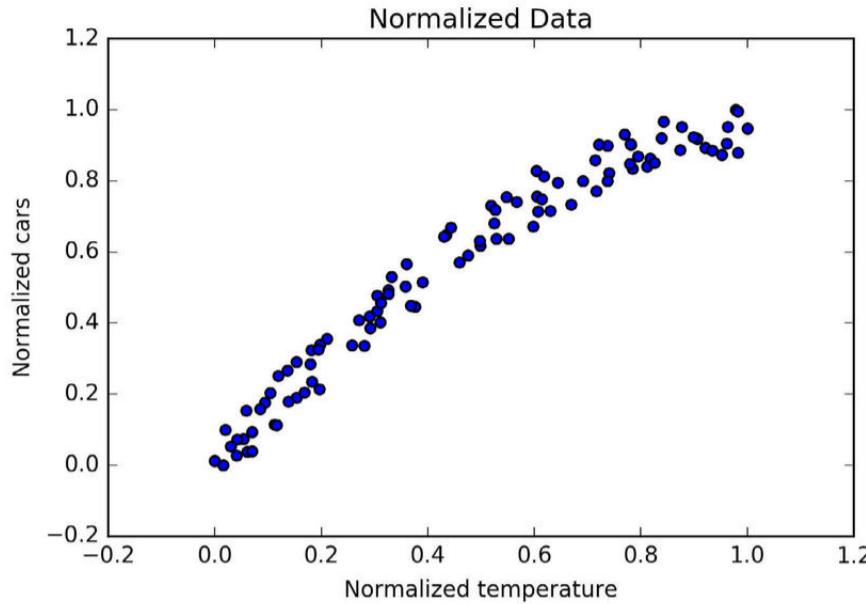


Figure 12.29: Normalizing both ranges to $[0,1]$ makes the data more amenable for training.

This looks just like Figure 12.28, 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, we'll use a kind of 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 is an object that is capable of performing this transformation (also called a **mapping**).

Second, we'll give that object our input data to analyze. It finds the smallest and largest values, and uses them to create the transformation that will shift and scale our input data to the range $[0,1]$. Since we want to scale the cars as well as the temperature, we'll need a transformer object for each axis. So far, nothing has changed in any of our data.

Figure 12.30 shows the idea.

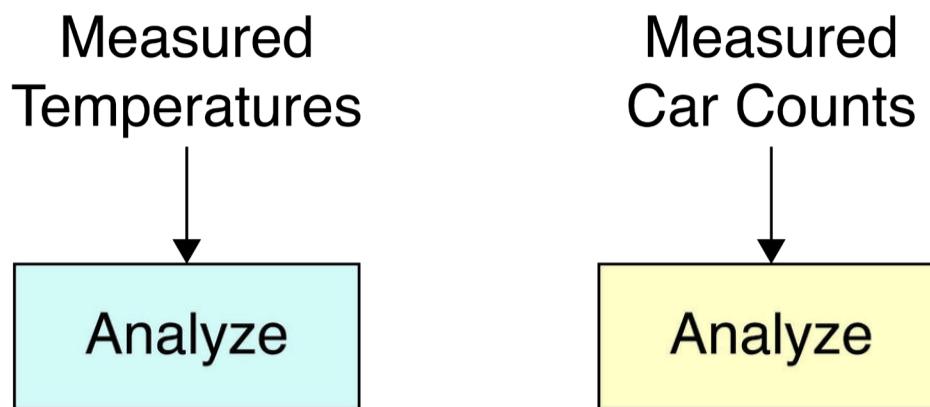


Figure 12.30: Building transformation objects. Left: The temperature data is fed to a transformation object. That object analyzes the data to find its minimum and maximum values, and it saves those internally. The data is unchanged. Right: We also build a transformer for the car counts.

The third step is to give our data to the transform object again, but this time we tell it to apply the transformation it's already computed. When it's done, it returns a new set of data that has been transformed to the range [0,1]. Figure 12.31 shows the idea.

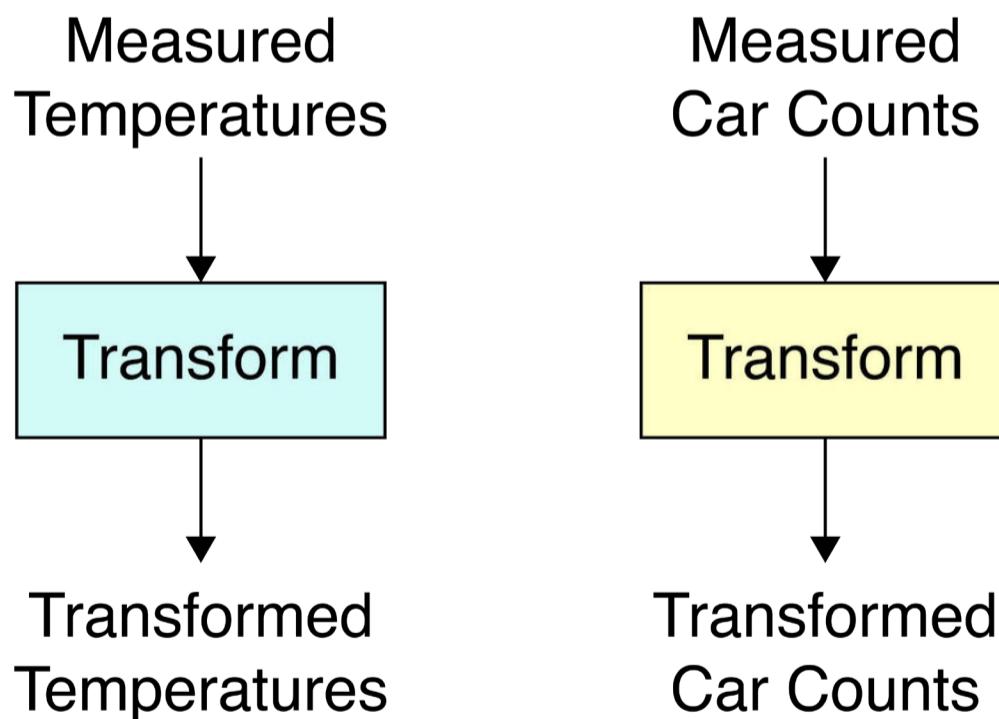


Figure 12.31: Each feature is modified by the transformation we previously computed for it. The output of the transformations goes into our learning system.

Now we're ready to learn. We give our transformed data to our learning algorithm, and we let it figure out the relationship between the inputs and the outputs, as shown schematically in Figure 12.32.

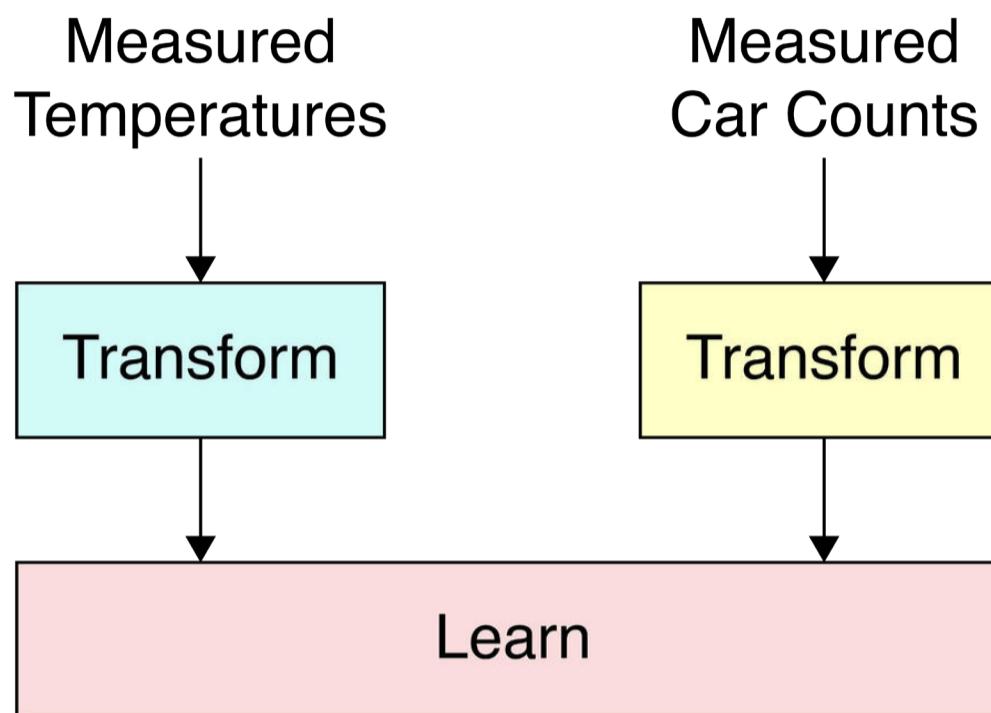


Figure 12.32: A schematic view of learning from our transformed features and targets.

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.

What should happen? We can't just feed -10 into our system, because it's expecting a number from 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 that had better turn into `0.29` again. If it becomes any other value, our system will think it's warmer or colder than it actually is.

And 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. So if `-10` turned into `0.29` during training, any new input of `-10` will turn into `0.29` during deployment as well.

Let's suppose the system determines an output of out 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 will come with a routine to **invert**, or **undo** its transformation, providing us with an **inverse transformation**. In this case, it inverts the normalizing transformation we built it to apply. Since during training the object transformed `39` cars into the normalized value `0.32`, the inverse transformation will turn the normalized value `0.32` back into `39` cars. This is the value we print out to the city manager. Figure 12.33 shows these steps.

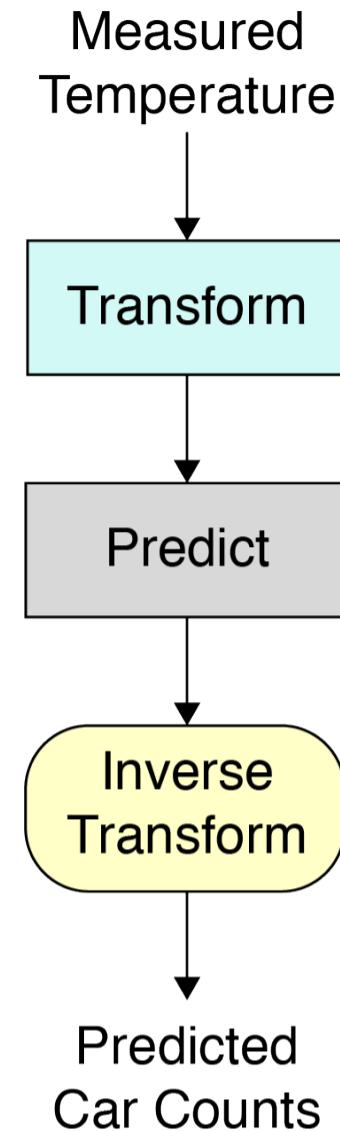


Figure 12.33: When we feed a new temperature to our system, we transform it using the transformation we computed for our temperature data, turning it into a number from 0 to 1. The value that comes out is then run through the inverse of the transformation we computed for the car data, turning it from a scaled number into a number of cars.

Once again we see that *we must always transform our input data using the original transformation that we used for the training data*.

One thing that can go apparently wrong here is if we get new samples that are outside of the original input range. Suppose we get a surprisingly cold temperature reading one night of -50 , which is far below the minimum value in our original data. The result is that the transformed value will be 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 will transform to a value greater than 1 , which is again outside of $[0,1]$.

Both situations are fine. Our desire for [0,1] was to make training go as efficiently as possible, and also to keep numerical issues in check. Once the system is trained, then we can give it any values we want as input, and it will do its best to calculate a corresponding output.

12.9 Slice Processing

In our traffic example above we had two features per sample (the temperature and traffic density). But we can imagine a richer data set where every sample has tens or thousands of features.

How do we pre-process such complicated datasets? What pieces of the data do we pull out, and in what form, to create and apply our transformations?

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 each discussion we'll assume our data is arranged in a 2D grid. Each row is a single sample, and each element in that row is a feature. So if we look at a column of the grid, we're looking at all the values taken by that feature. Figure 12.34 shows the setup.

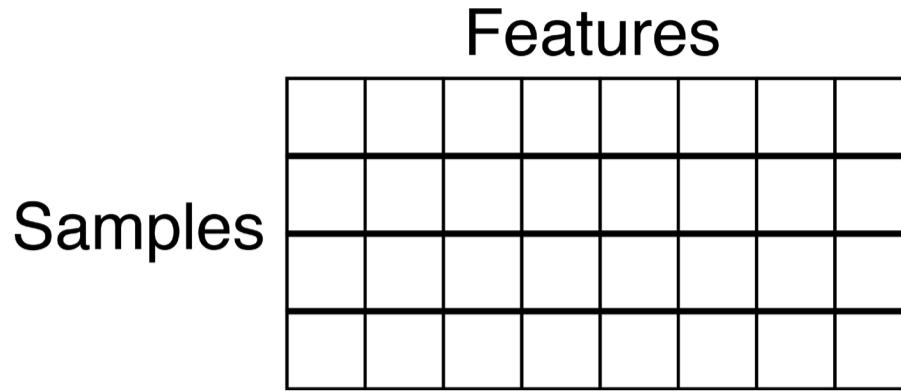


Figure 12.34: Our database for the coming discussion is a 2D grid. Each row contains multiple features, which make up the columns.

12.9.1 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, say a person speaking into a cell phone. The features in each sample are the amplitude of the audio at successive moments, as in Figure 12.35.

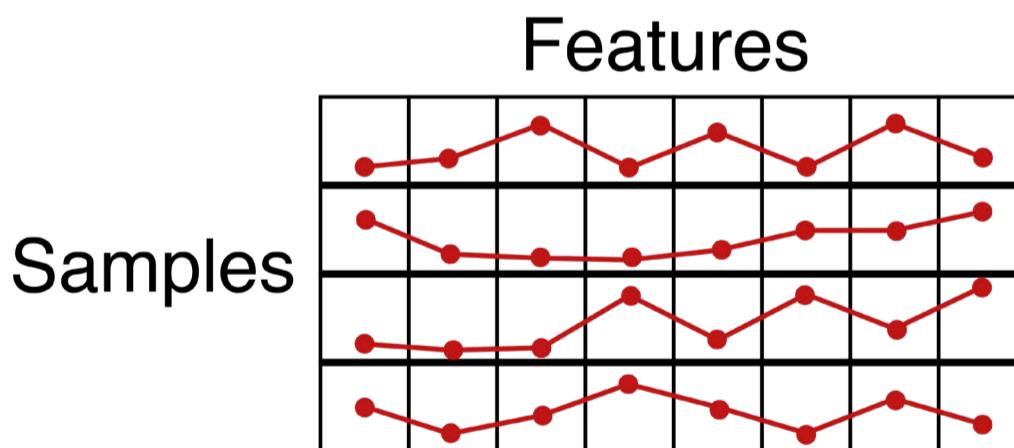


Figure 12.35: Each sample consists of a series of measurements of a short audio waveform. Each feature gives us an instantaneous measurement of the volume of the sound at that moment.

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. This samplewise processing is shown in Figure 12.36.

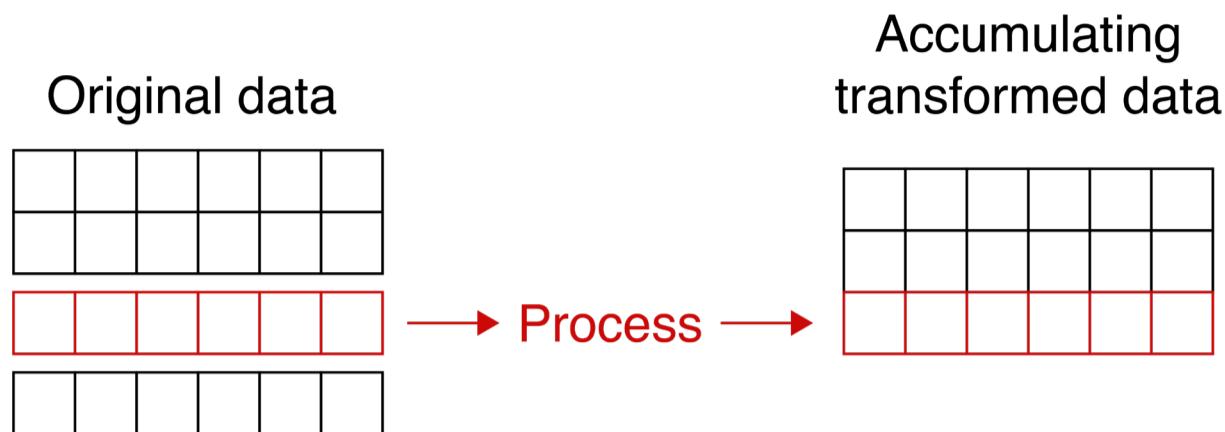


Figure 12.36: When we process data samplewise, each sample (or row of our database) is processed independently of the others. Here we’re working our way down the original dataset top to bottom, transforming one sample at a time and then appending it to a new dataset.

When we process data samplewise, we think of each row (or sample) as the thing we want to adjust. In turn we select each row, analyze it, and transform it. We can either then over-write the original data, or save the result in a new dataset.

This kind of processing makes sense for data like images and audio, where we think of all the features in one sample as closely related to one another, using the same scale.

12.9.2 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, wind speed, humidity, and cloud cover. This gives us four features per sample, as in Figure 12.37.

	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 12.37: Each night we measure four different aspects of the weather. When we process these featurewise, we analyze each column independently. Here we find the minimum and maximum of each column (shown in square brackets) and use those values to transform the inputs into a new set of values between 0 and 1.

It wouldn'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 also all the values for wind speed, temperature, and so on. In other words, we'll modify each feature in turn, as in Figure 12.38.

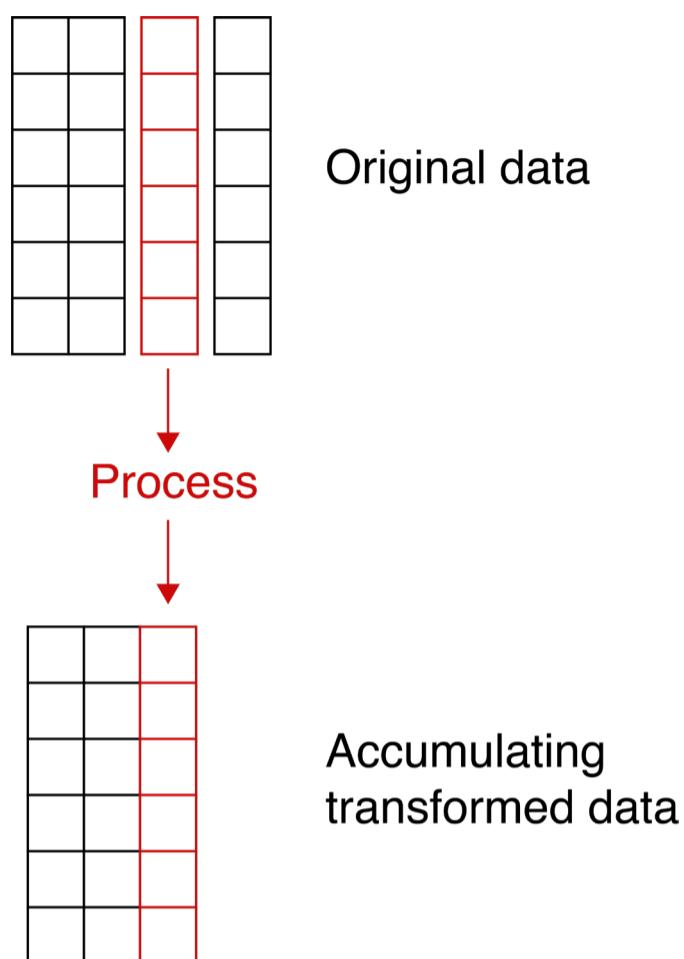


Figure 12.38: When we process data featurewise, each feature (or column of our dataset) is processed independently of the others. Here we’re working our way across the original dataset left to right, transforming one feature at a time and then appending it to a new dataset.

When we process data featurewise, we think of each column (or feature) as the thing we want to adjust. In turn we select each column, analyze it, and transform it. When we process data featurewise, each column of feature values is sometimes called a **fibre**.

12.9.3 Elementwise Processing

The elementwise approach treats each element in the grid of Figure 12.34 as an independent entity, and applies the same transformation every element in the grid.

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 4 members, and contains the heights of each of the four 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]. So we apply an elementwise scaling operation and divide every pixel value in the entire input by 255, giving us data from 0 to 1.

12.10 Cross-Validation Transforms

We've seen that the right way to process data is to build the transformation from the training set, 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 into it, affecting the transformation. This means that we won't transform the data the way we intend. Worse, we'll see that this leakage can lead to the system getting an unfair advantage when it evaluates our test data, reducing the apparent error. We might 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 it can come in different forms, many of which are 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 for us, so we don't have to write our own code to do it. But we'll look under the hood to see how a seemingly reasonable approach would lead to information leakage. We'll then see how to fix it.

Seeing this in action will help us get just that much better at preventing, spotting, and fixing information leakage in our own systems, and any custom code that we write.

The new element in this discussion is due to how cross-validation starts each step with just a piece of the overall data set. So unlike previous examples, we won't be analyzing the whole data set and then building a transformation from it. Instead, we'll be analyzing just a piece of the data set. We'll see that this change demands that we take extra care.

Recall that in cross-validation we pull out one **fold** (or section) of the training set. Then we build a new learner and train it with the resulting data. When we're done training, we evaluate the learner using the pulled-out 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 based on that specific set of data. 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 12.39 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 to the samples (turning them red). 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. So each time through the loop we remove a fold, train on the remaining samples, then test the validation fold and create a score.

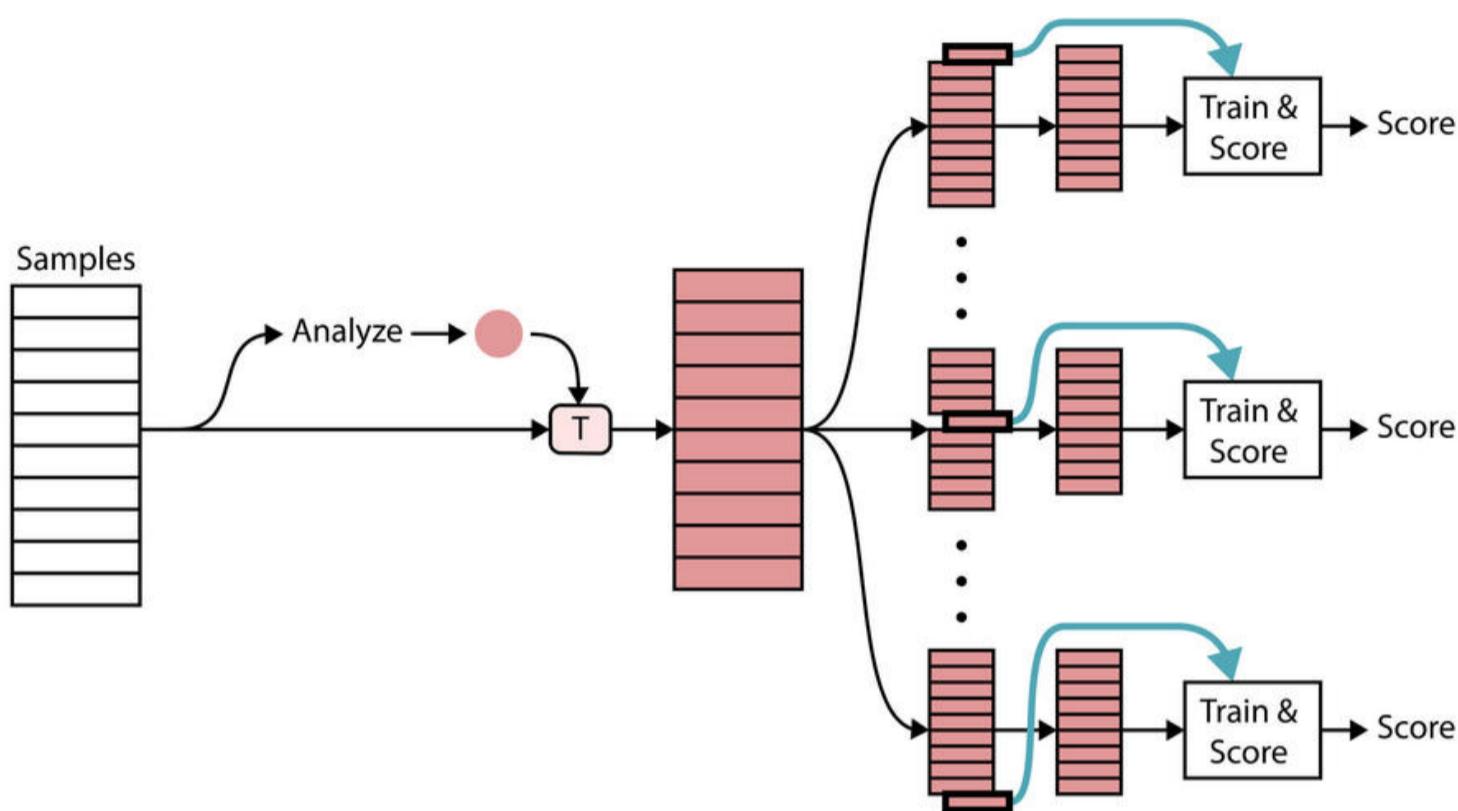


Figure 12.39: A **wrong** way to do cross-validation. We do not want to compute the transformation before we do cross-validation. The red circle represents the transformation. It's applied to the data by the red box marked with a T.

The problem here is that when we analyzed the input data to build the transformation, we include 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 to scale all the features in the training set, as a group, to the range 0-1. Stated with the terms we saw earlier, we're going to perform a multivariate featurewise scaling to the range [0,1]. 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 12.40 shows the range of data contained in each of the five folds. We're going to analyze all the folds and built our transformation from that.

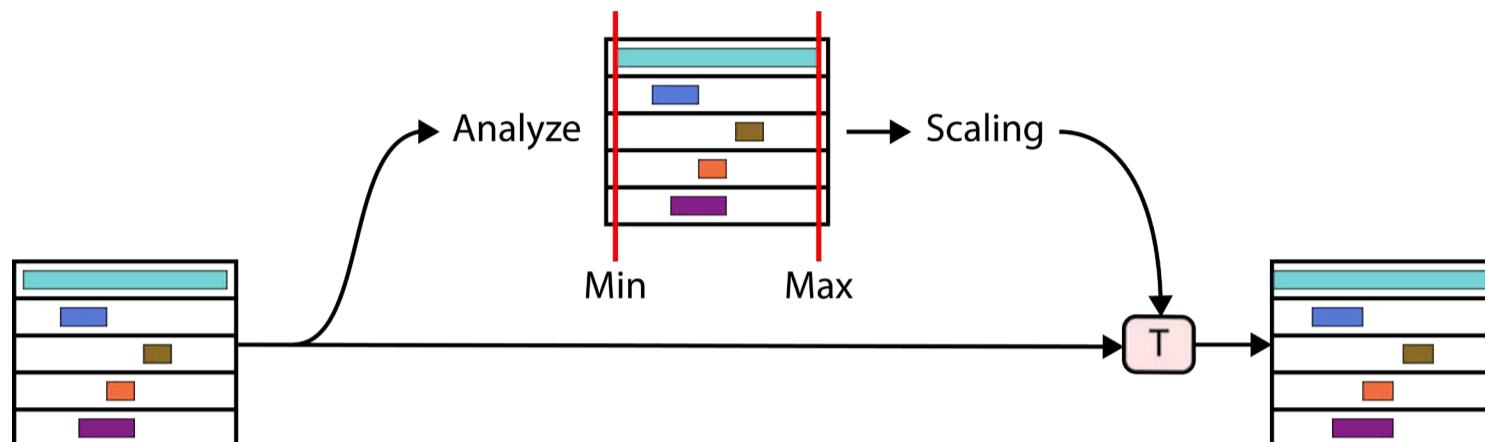


Figure 12.40: A **wrong** way to transform our data for cross-validation is to transform everything at once before the loop.

In Figure 12.40, our data set is shown at the left, split into 5 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.

If we weren't doing cross-validation, then the step of Figure 12.40 would be entirely reasonable, because it's just analyzing and scaling all of our data. To see the problem that comes up, let's use this transformed data for cross-validation.

Our input data is the stack of five transformed folds at the far right of Figure 12.40. We'll start by extracting the first fold and setting it aside, train with the rest of the data, and then validate. But we've done something bad here, because *our training data's transformation was influenced by the data in the fold*.

This is a violation of our basic principle that we create the transform using only the values in the training data. 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 do this is to remove the fold data from the samples, *then* build the transformation from the data remaining, and apply that transformation to both the training and fold data. Figure 12.41 shows this visually.

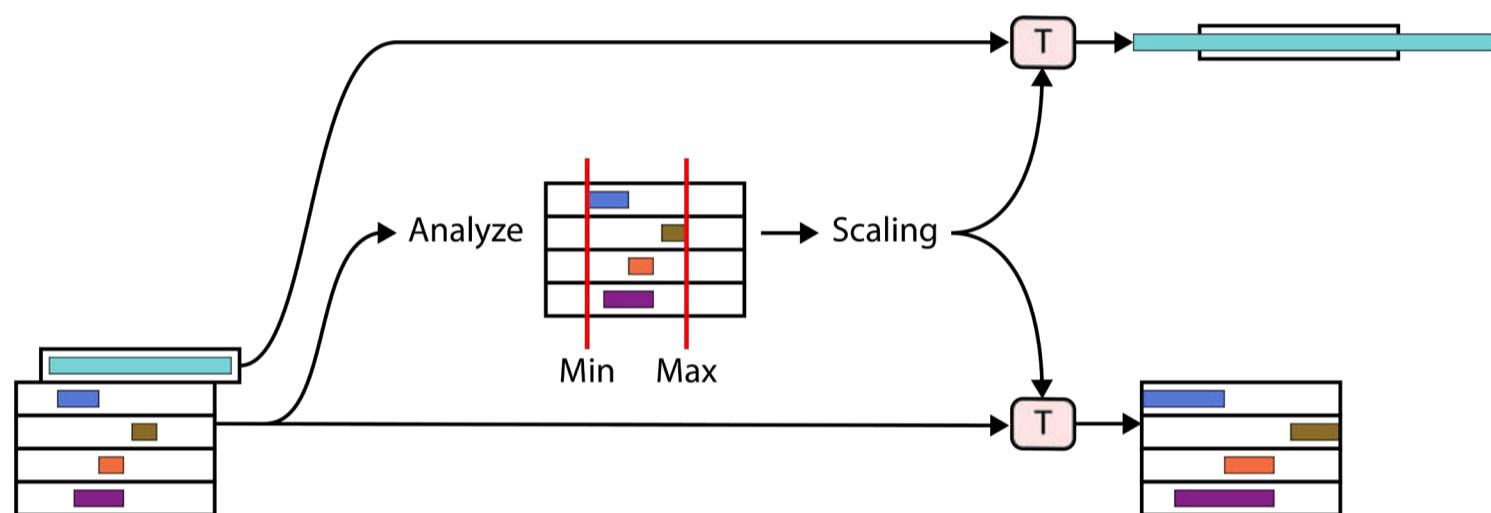


Figure 12.41: The proper way to transform our data for cross-validation is to first remove the fold samples, and then compute the transformation from the data that remains. Now we can apply that transformation to both the training set and the fold data. Note that here the fold data ends up way outside 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 move this scheme into the loop, and compute a new transformation for every training set. Figure 12.42 shows the right way to proceed.

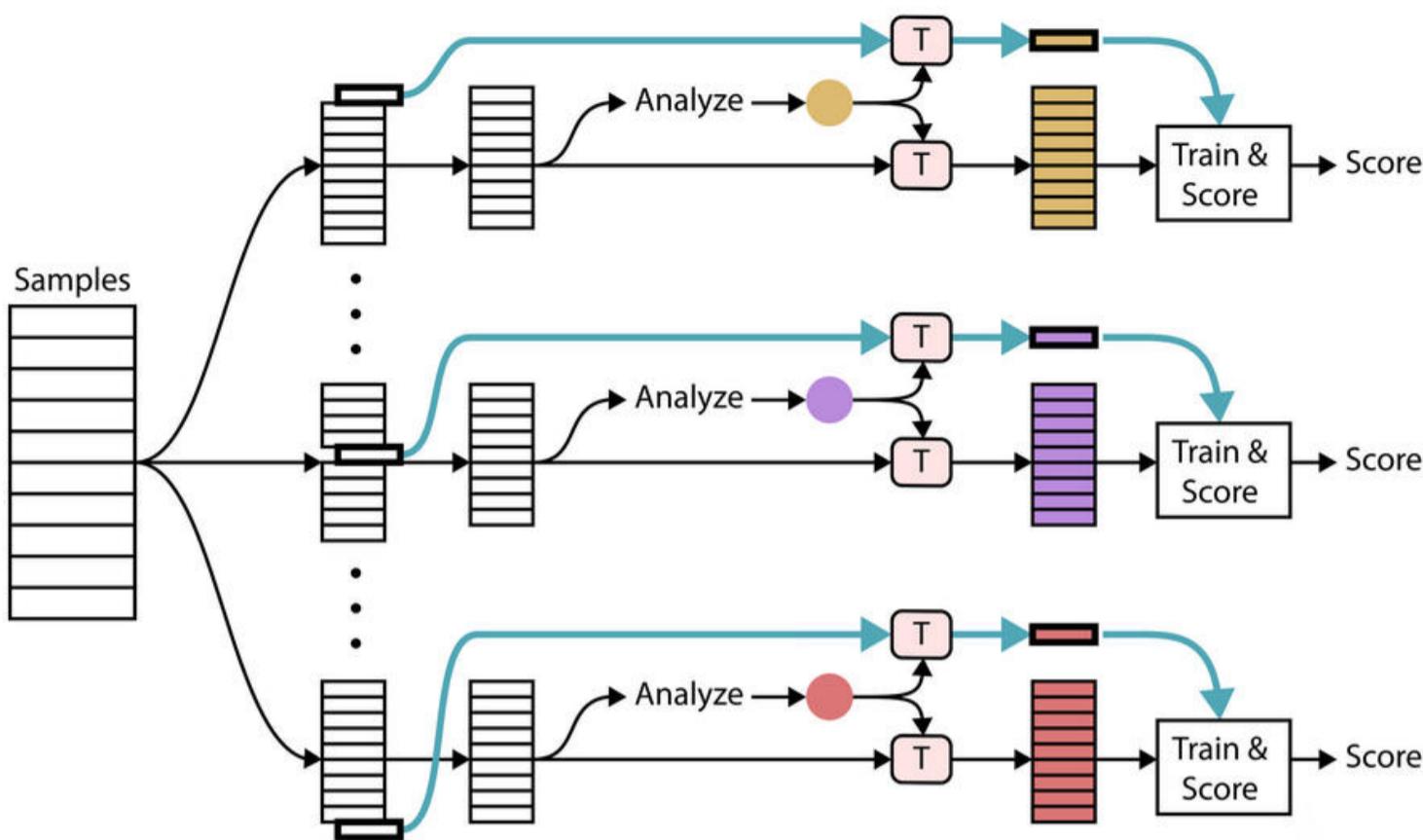


Figure 12.42: The proper way to do cross-validation. For each fold we want to use as a validation set, we analyze the starting samples with that fold removed, and then apply the resulting transform to both the training set and the validation set in the fold. The different colors are meant to suggest 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, 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.

References

- [CDC17] Centers for Disease Control and Prevention, “Body Mass Index (BMI)”, 2017. <https://www.cdc.gov/healthyweight/assessing/bmi/>
- [Crayola16] “What were the original eight (8) colors in the 1903 box of Crayola Crayons?”, 2016. <http://www.crayola.com/faq/your-history/what-were-the-original-eight-8-colors-in-the-1903-box-of-crayola-crayons/>
- [Dulchi16] Paul Dulchi, “Principal Component Analysis”, CS229 Machine Learning Course Notes #10. <http://ufldl.stanford.edu/wiki/index.php/PCA>
- [Krizhevsky09] Alex Krizhevsky, “Learning Multiple Layers of Features from Tiny Images”, University of Toronto, 2009. <http://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>
- [McCormick14] Chris McCormick, “Deep Learning Tutorial - PCA and Whitening”, Blog post, 2014. <http://mccormickml.com/2014/06/03/deep-learning-tutorial-pca-and-whitening/>
- [Turk91] Matthew Turk and Alex Pentland, “Eigenfaces for Recognition”, Journal of Cognitive Neuroscience, Volume 3, Number 1, 1991. <http://www.face-rec.org/algorithms/pca/jcn.pdf>

Image Credits

Figure 12.3, Cow

<https://pixabay.com/en/cow-field-normande-800306>

Figure 12.3, Zebra

<https://pixabay.com/en/zebra-chapman-steppe-zebra-1975794>

Figure 12.21, Figure 12.23, Husky

<https://pixabay.com/en/husky-sled-dogs-adamczak-1105338>

Figure 12.21, Figure 12.23, Husky

<https://pixabay.com/en/husky-dog-outdoor-siberian-breed-1328899>

Figure 12.21, Figure 12.23, Husky

<https://pixabay.com/en/dog-husky-sled-dog-animal-2016708>

Figure 12.21, Figure 12.23, Husky

<https://pixabay.com/en/dog-husky-friend-2332240>

Figure 12.21, Figure 12.23, Husky

<https://pixabay.com/en/green-grass-playground-nature-2562252>

Figure 12.21, Figure 12.23, Husky

<https://pixabay.com/en/husky-dog-siberian-husky-sled-dog-2671006>

Chapter 13

Classifiers

A survey of a variety of popular supervised and unsupervised algorithms for classifying data into different categories.

Contents

13.1 Why This Chapter Is Here	490
13.2 Types of Classifiers	491
13.3 k-Nearest Neighbors (KNN).....	493
13.4 Support Vector Machines (SVMs).....	502
13.5 Decision Trees.....	512
13.5.1 Building Trees.....	519
13.5.2 Splitting Nodes.....	525
13.5.3 Controlling Overfitting	528
13.6 Naïve Bayes.....	529
13.7 Discussion.....	536
References	538

13.1 Why This Chapter Is Here

Classifying images, sounds, and other data is a major application of machine learning. In Chapter 7 we talked about the foundations that make classification work. But there's more to classification than just building and training a generic algorithm.

As we've seen, understanding our data is a key step in designing and training a good machine learning system. It's always worth our time to look over our data and develop a feeling for it, so that we can match the design of our system to the task we want it to perform.

For example, suppose someone gives us aerial photographs of different farms, and they want us to classify the fields based on the crops that are growing there.

How many different crops should our classifier be able to distinguish? 5? 25? Until we look at the data, we have no way of knowing. Using too few categories will limit the value of our answers, but using too many will slow everything down and might even introduce useless distinctions (say between ripe and not-ripe fields of the same crop). Before we start building, we need to look at the data and get a sense for about how many different categories we're going to want to detect and distinguish.

A great way to explore the data is to use one of the many tools that implement specific classification algorithms. In some cases, these tools might even be all we need to build a complete solution to our classification problem. In other cases, we can use them for running quick experiments, and then use what we learn to design a deep-learning classifier that is well-tuned to our problem.

Because these tools are such a useful first step in coming to grips with new data, we'll look at some of the most popular ones here. These tools wouldn't usually be considered deep learning algorithms themselves, but they're an important part of the process of developing a complete solution for a particular problem.

We'll illustrate our classifiers using 2D data and usually only 2 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.

Using modern libraries, most of these algorithms can be applied to our own data with just one or two lines of code.

13.2 Types of Classifiers

Before we get going, we'll break down the world of classifiers into two main approaches.

In one approach, the classifier attempts to characterize the data by fitting a set of **parameters** to it, much as we could fit a straight line to data by setting the parameters of that line. This is naturally enough called the **parametric** approach.

The parametric approach begins with some kind of description of a classification mechanism. In other words, it begins with an assumption about the form of the boundary that it's going to look for. It might assume that boundary will be a straight line, or a 10-dimensional floppy sheet, or a sphere with a bulge at one end. It will then look for the best values for the parameters that describe that shape so it best matches the input data.

We could say that a parametric classifier begins with a hypothesis that the data can be classified using a structure of a specific type, and the job of learning is to find the best parameters for that structure. In the language of Chapter 4 we could say that this hypothesis is a Bayesian prior.

The other type of classifier is called **non-parametric**, which is a little unimaginative but still descriptive. The non-parametric approach doesn't begin with a hypothesis. Instead, it simply collects data and builds an algorithm from it as it goes. The algorithm still has a known form, of course, but its structure, and what variables it employs, depend on the training data.

A non-parametric classifier will usually save most or all of the data as it comes in, and tries to find an organization for it that will facilitate categorizing new data when it arrives.

Using our ideas from Chapter 11, we can say that a parametric approach is deductive, since it starts with a hypothesis and looks for data to satisfy it. In the same way, the non-parametric approach is inductive, since it starts by collecting data, and then tries to draw inferences from the patterns it can find.

The speed and memory tradeoffs of these two approaches are complimentary.

The parametric classifier doesn't need much memory, since it's just storing the parameters that describe its model. Calculating these parameters can take a lot of effort, so training can run slowly. When our model is deployed, classifying new data is fast, because we typically only need to evaluate some function that uses the parameters and the new feature values.

On the other hand, because non-parametric classifiers often save much or all of the sample data, their memory consumption goes up as we train. The more training samples it sees and stores, the more memory the classifier will consume. Because it's doing very little work for each sample, training goes very quickly. When our model is deployed,

classifying new data is slow, because the algorithm will grind through the potentially huge database of stored samples to work out the best classification for the new sample.

Generally speaking, non-parametric systems are attractive when our training set is small, we need to train quickly, and we don't need to evaluate a lot of new samples. The parametric approach is better when our training set is large, and we don't mind paying for slow training times in return for fast classification on new data.

When we're exploring a data set, it's not uncommon to switch among classifiers as we run experiments to refine our understanding of the data.

We'll now look at some non-parametric classifiers, and then move to parametric algorithms.

13.3 k-Nearest Neighbors (KNN)

We'll begin with a non-parametric algorithm called **k-Nearest Neighbors**, or **kNN**. Here, 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 generally considered 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 very different techniques. The key difference is that k-means clustering learns from unlabeled data, while kNN works with labeled data. In other words, they fall into the categories 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 categorized.

The central idea of how kNN categorizes a new sample is appealingly geometric, and demonstrated in Figure 13.1.

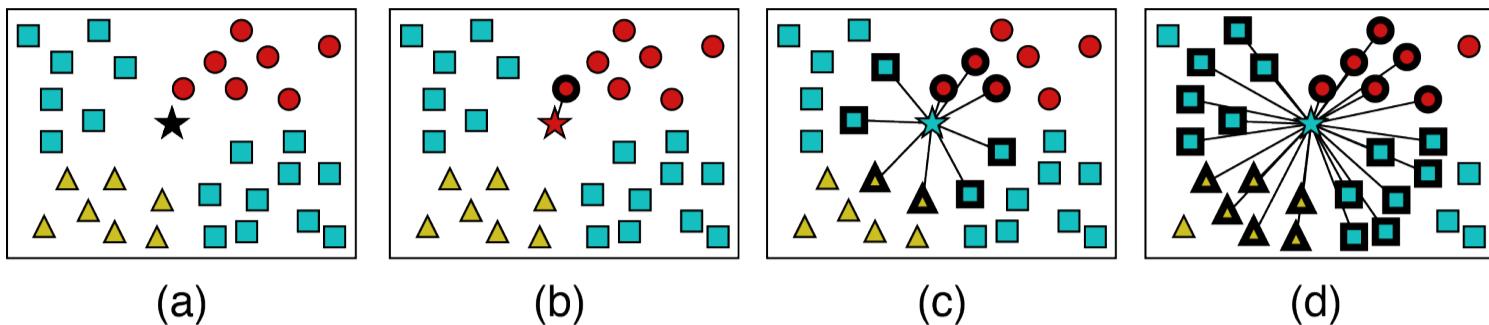


Figure 13.1: To find the class for a new sample, we find the most popular of its k neighbors. (a) A new sample (the star) surrounded by samples of 3 classes. (b) When $k=1$, the 1 nearest neighbor is round, so the category for the star sample is round. (c) When $k=9$, we find 3 round samples, 4 squares, and 2 triangles, so the star is assigned the square category. (d) When $k=25$, we find 6 round samples, 13 squares, and 6 triangles, so the star is again assigned to the square category.

In Figure 13.1(a) we have a sample point (a star) amid a bunch of other samples that represent three categories (round, square, and triangle). To determine a category for our new sample, we look at the k nearest samples (**neighbors**), and we let them vote. Whichever category is most popular becomes the new sample's category. We show which samples are considered for different values of k by drawing a circle that encloses them. In Figure 13.1(b) we've set k to 1, meaning we want to use the class of the nearest sample. In this case it's a circle, so this new sample will be categorized as round. In Figure 13.1(c) we've set k to 9, so we look at the 9 nearest points. Here we find 3 circles, 4 squares, and 2 triangles. Because there are more squares than any other class, the black star is categorized as square. In Figure 13.1(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 and a value of k . It then finds the closest k samples to the new sample, and lets them “vote” on the appropriate category. Whichever category has the most votes wins, and that becomes the category for the new sample.

There are various ways of breaking ties and handling exceptional cases, but that's the basic idea.

Note that k-nearest neighbors 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.

K-nearest neighbors is attractive because it's simple, and training 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. At some point, using large amounts of memory can by itself slow down the algorithm. 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 ways to speed this up, but it still remains a slow method. For applications where classification speed is important, like real-time systems and web sites, 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 very 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 simultaneously and 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 13.2 we show a "smile" dataset of 2D data that falls into two categories.

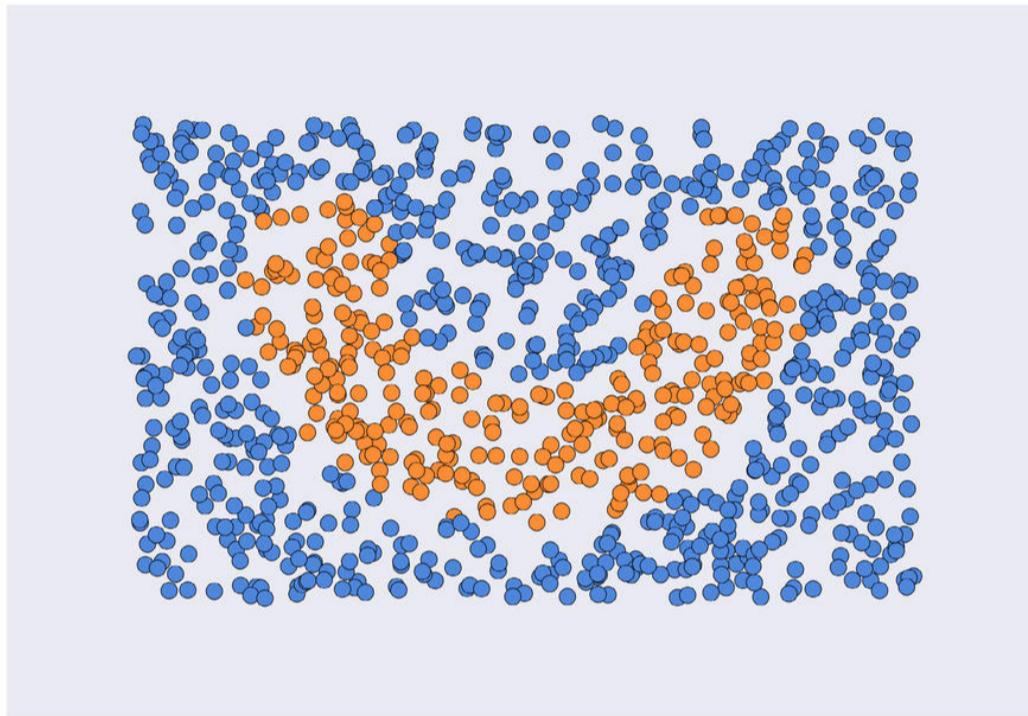


Figure 13.2: A "smile" dataset of 2D points. There are two categories, blue and orange.

Using kNN with different values of k gives us the results in Figure 13.3.

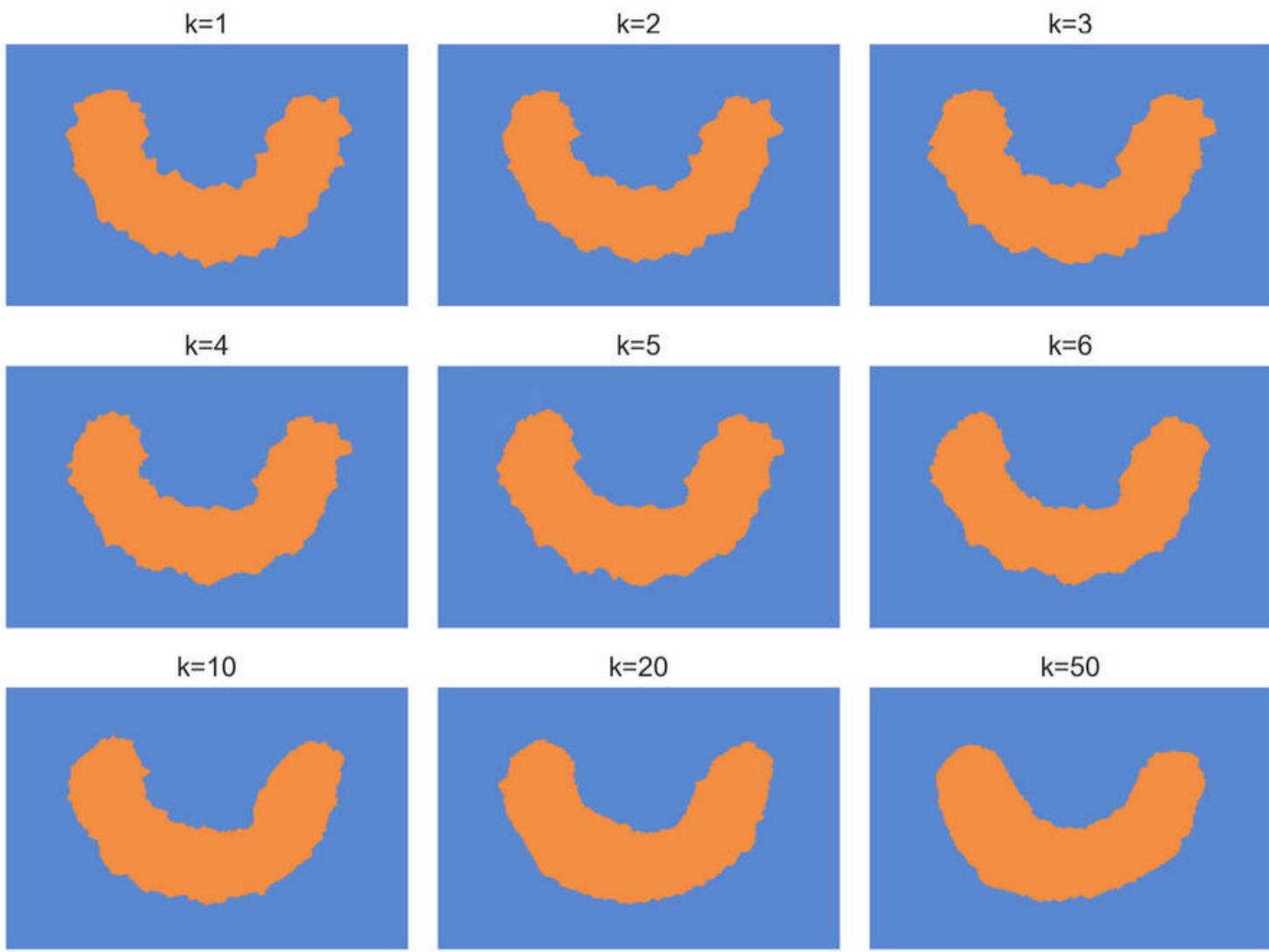


Figure 13.3: Classifying all the points in the rectangle using kNN for different values of k . Notice how rough the edges are for low values of k , and how they smooth out as k increases. The k values in the top two rows go up by 1 with each image. The bottom row uses much bigger increments.

With a small number of neighbors the classification boundaries are pretty rough. As we use more neighbors, they smooth 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 13.4 shows a noisy version of Figure 13.2.

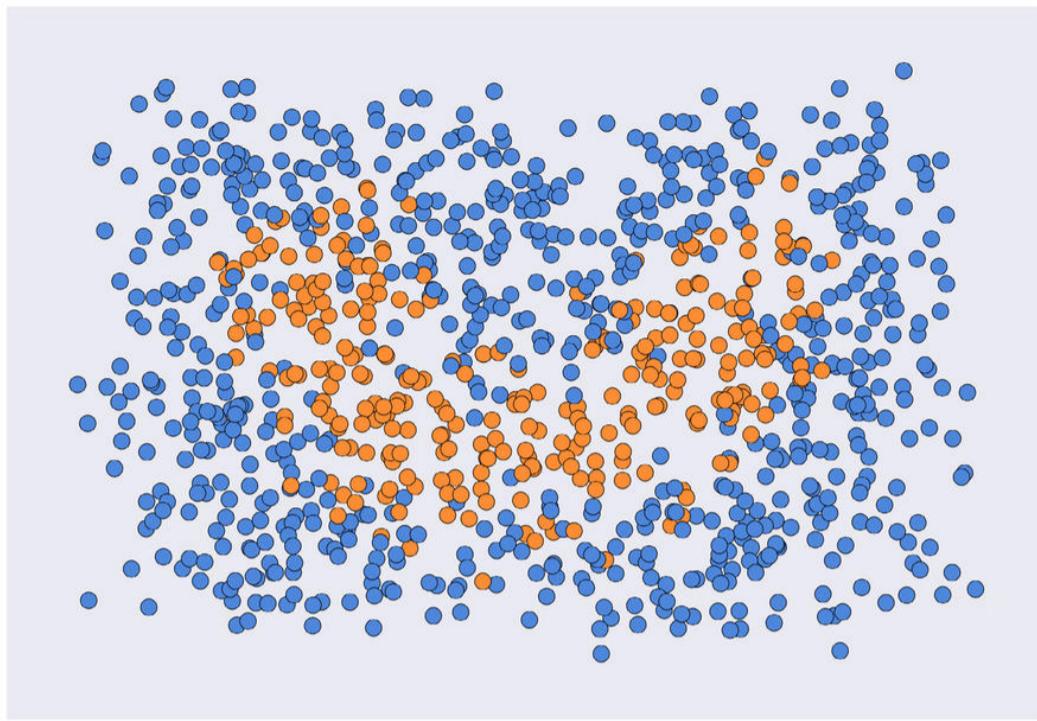


Figure 13.4: A noisy version of the smile dataset. We're using the same number of points as in Figure 13.2, but we've added some 2D noise to jitter the locations of the samples after they were assigned to the blue and orange categories.

The results for different values of k are shown in Figure 13.5.

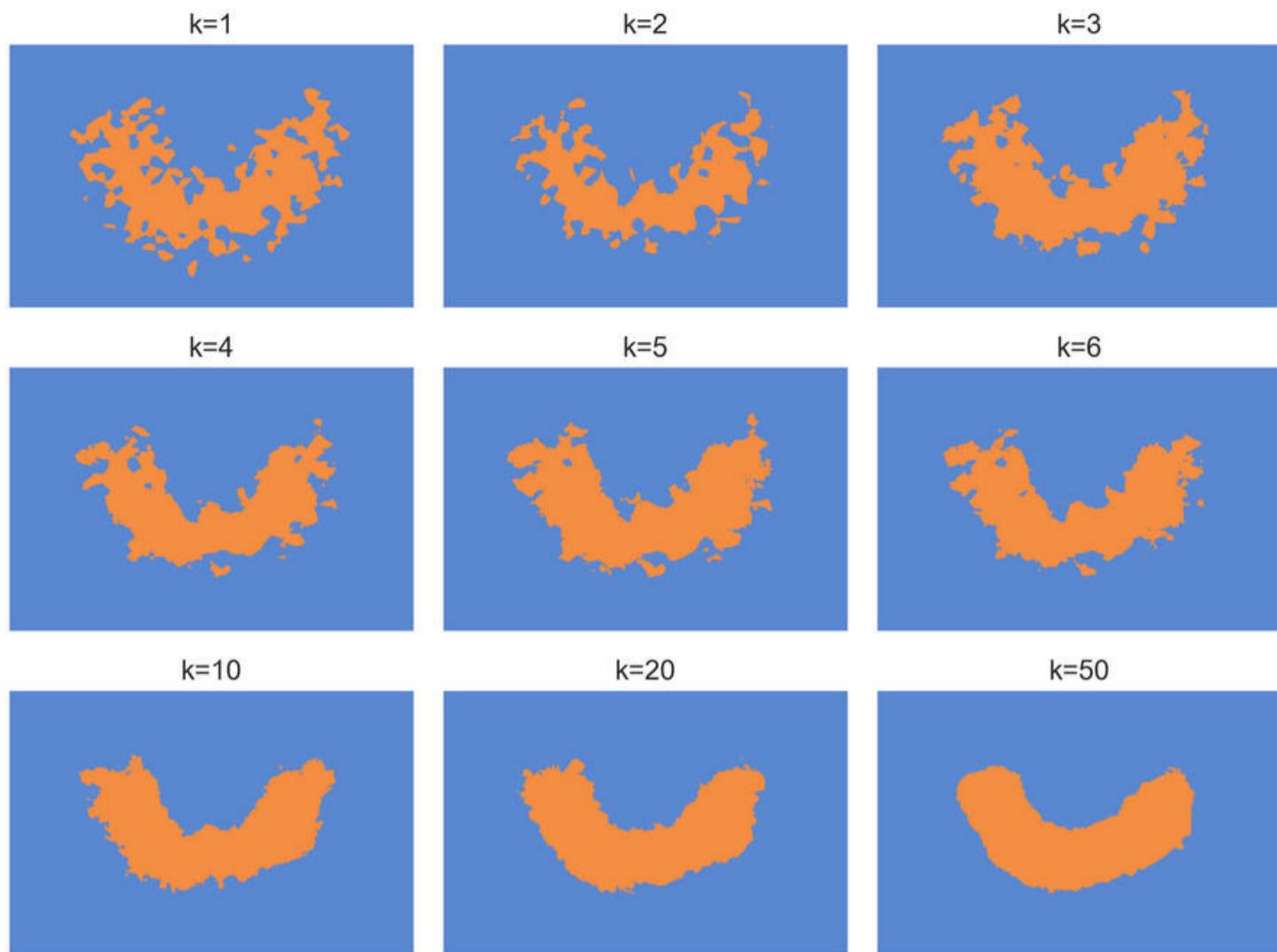


Figure 13.5: Using kNN to assign a category to points in the plane. Note how much the algorithm overfits for low values of k . As the number of neighbors increases, the edges smooth out considerably.

We can see that in the presence of noise, small values of k lead to overfitting. In this example, we had to get k up to 50 to smooth out the boundaries.

A nice feature of kNN comes from the fact that it doesn't explicitly represent the boundaries between categories. This means it can handle any kind of boundaries, or any distribution of points. To see this, let's add some eyes to our smile, creating three disconnected sets of one category. The resulting noisy data is in Figure 13.6.

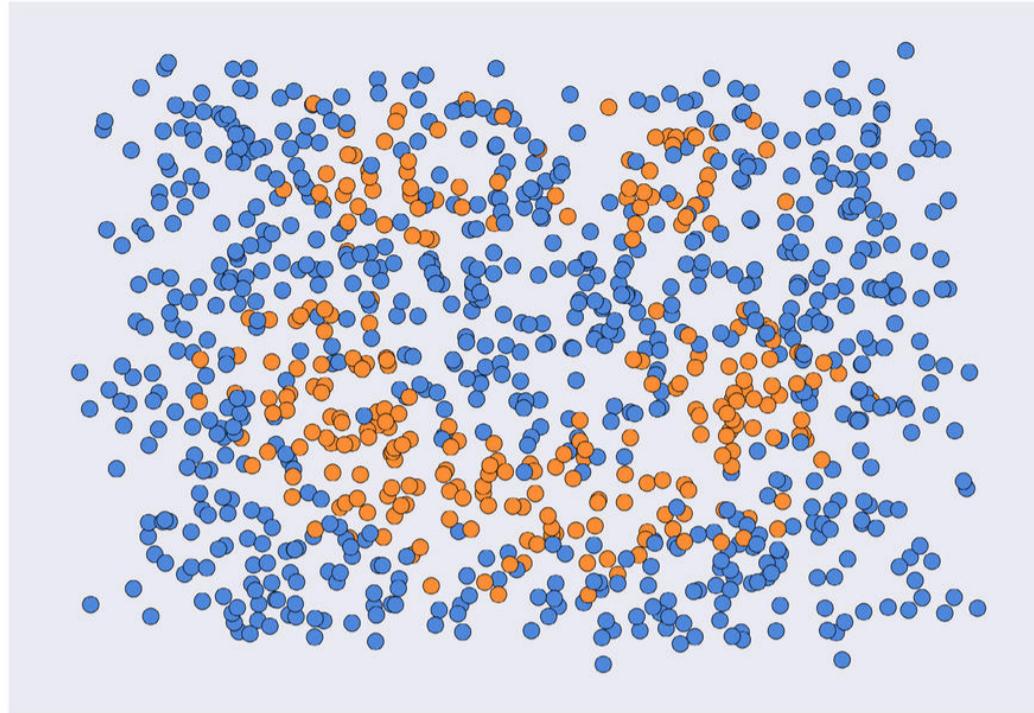


Figure 13.6: Adding two eyes to our smile database, then adding noise to jitter the sample locations.

The resulting grid for different values of k is shown in Figure 13.7.

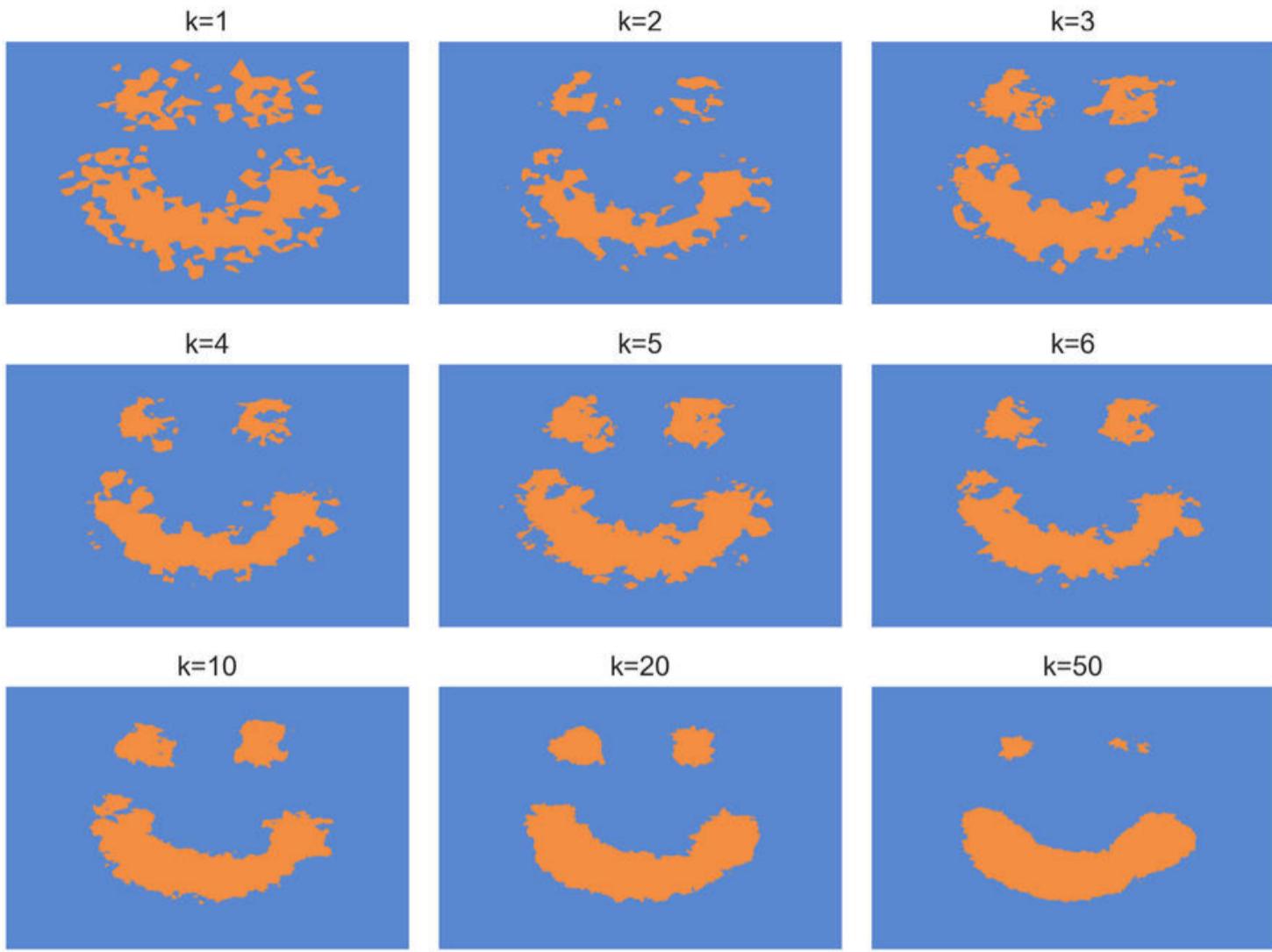


Figure 13.7: kNN doesn't create boundaries between clusters of samples, so it works even when a class is broken up into pieces. As expected, the algorithm overfits at low values of k . But notice that when $k=50$, the eyes are disappearing. That's because in the neighborhood of the eyes, when we test for a large number of neighbors, the blue points will almost always dominate over orange. If we increased k further the eyes would disappear completely.

In this example, a value of about 20 for k looks best. When k gets up to 50, the small number of samples that represent the eyes start to lose on every vote, and the eyes are eroding.

So too small a value of k can give us ragged edges and overfitting, but too large a value of k can start to erode small features. As is so often the case, finding the best parameters 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.

13.4 Support Vector Machines (SVMs)

Our next categorizer takes a very different approach. Like some categorizers we've seen before, this one will attempt to find an explicit boundary between different sets of samples. As usual, we'll use 2D data and just 2 categories, but the technique can be easily applied to far more features and categories. Our discussion is inspired by the presentation in [VanderPlas16].

Let's begin with two blobs of points, one for each of two classes, shown in Figure 13.8.



Figure 13.8: Our starting dataset consists of two blobs of 2D samples.

We'd like to find a boundary between these clusters. To keep things simple, we'll use a straight line. There are lots of lines that will split the two groups. Three candidates are shown in Figure 13.9.

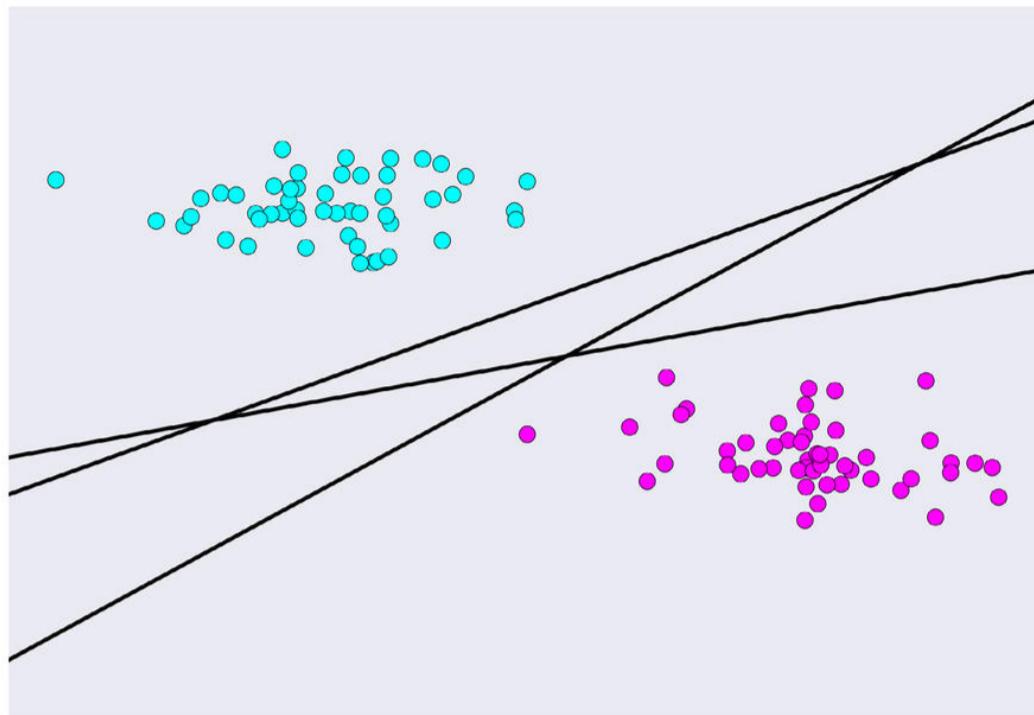


Figure 13.9: Three of the infinite number of lines that can separate our two clusters of samples. Which line is best?

Which of these lines should we pick? One way to think about this is to imagine new data that might come in. Given how distinct these clusters are, it's likely that new points will land pretty close to one of these clumps. But those new points might be on the edge or a bit beyond one of these clusters.

This leads us to prefer a line that is as far away as possible from both clusters. That way any of these new points have their best chance of landing in the proper cluster.

To see how far each line is from the clusters, we can find the nearest sample to each line, and then draw a symmetrical boundary of that size around each line. Figure 13.10 shows the idea.

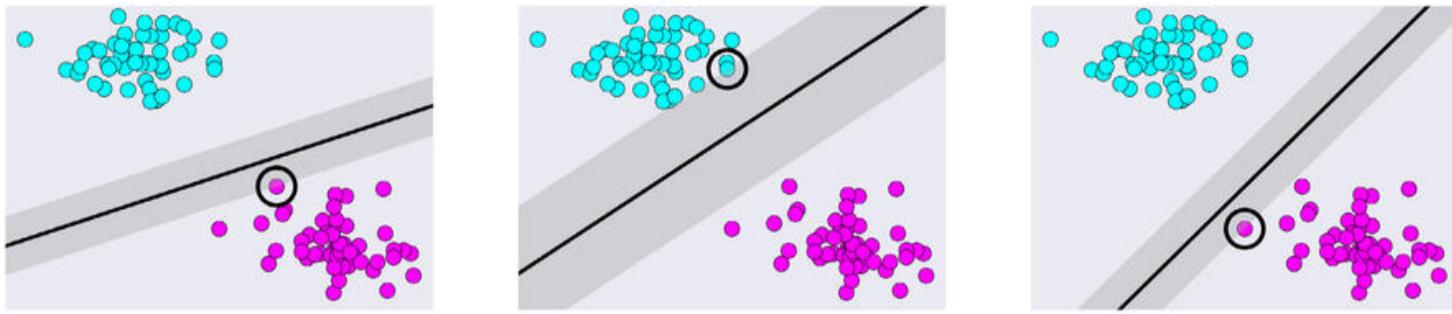


Figure 13.10: We can assign a quality to each line by finding the distance from that line to the nearest data point. Here we've drawn a gray zone symmetrically around each line, at that distance, along with a circle around the point that's closest to the line. The best line of the three is the one in the middle, because it will be the least likely to make an error if a new point comes along that drifts into the zone between the clusters.

The line in the middle of Figure 13.10 is the best of these three. It's definitely better than the one on the far right, for example, because a point just a bit above and to the left of the red cluster in the bottom right and over the line would be classified as blue, even though it's closer to the red clump than the blue one.

The algorithm called the **support vector machine**, or **SVM**, uses this idea [Steinwart08]. An SVM finds the line that is farthest from all the points in both clusters (the word “machine” here can be thought of as a fanciful synonym of “algorithm.”) This best line is shown in Figure 13.11.

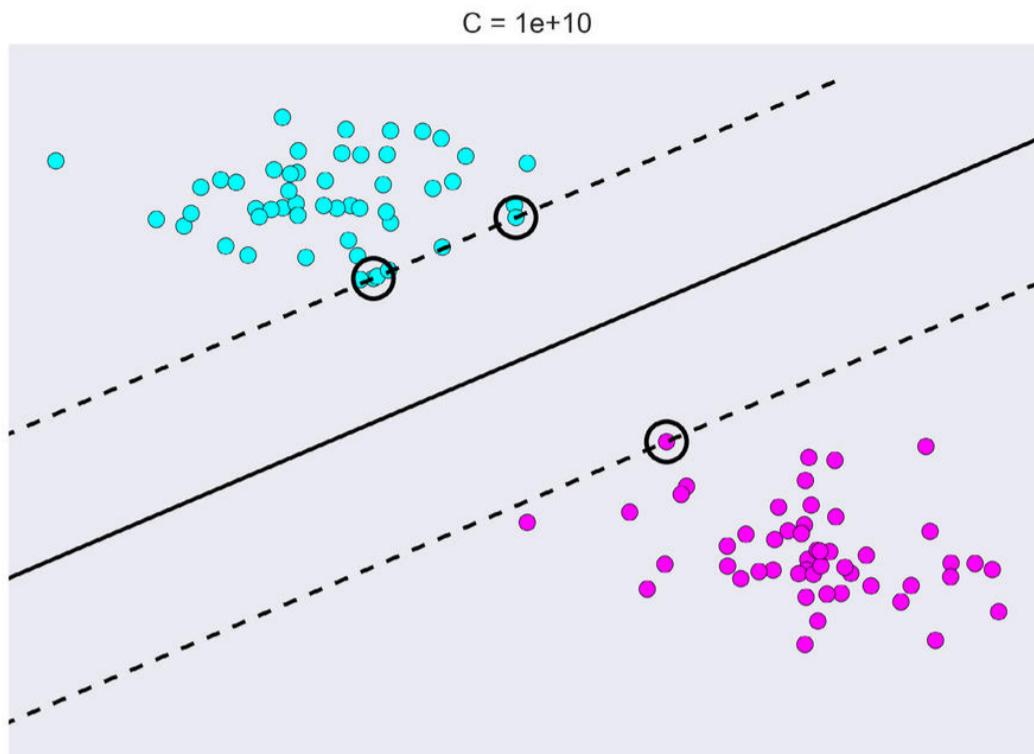


Figure 13.11: The SVM algorithm finds the line that has the greatest distance from all the samples. That line is shown in black. The highlighted circles are called the support vectors, and they define both the line and the zone around it, here shown with dashes. The value of C in the title above the figure is explained below in the text.

In Figure 13.11 the highlighted samples are called the **support vectors** (here, the word “vector” can be thought of as meaning “sample”). The algorithm’s first job is to locate these circled points. Once it found them, the algorithm then found the solid line near the middle of the figure. This line is the farthest from every sample in each set.

The distance from the solid line to the dashed lines that pass through the support vectors is called the **margin**. So we can re-phrase the idea by saying that the SVM algorithm finds the line with the largest margin.

What if the data is noisy, and the blobs overlap, as in Figure 13.12? 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 13.12: A new set of data where the blobs overlap. What's the best straight line we can draw in this case?

The SVM algorithm gives us control over a parameter that's conventionally called simply C . This controls how strict the algorithm is about letting points into the region between the margins. We can think of C meaning something like "clearance." The larger the clearance, the more the algorithm demands an empty zone around the line. The smaller the clearance, the more points can appear in a zone around the line.

The algorithm is sensitive to the value of C , and the data we're using. Because of this, we frequently need to search for the best setting using trial and error. In practice, that usually means trying out lots of values and evaluating them with cross-validation.

Figure 13.13 shows our overlapping data with a C value of 100,000 (or $1e+05$ in scientific notation).

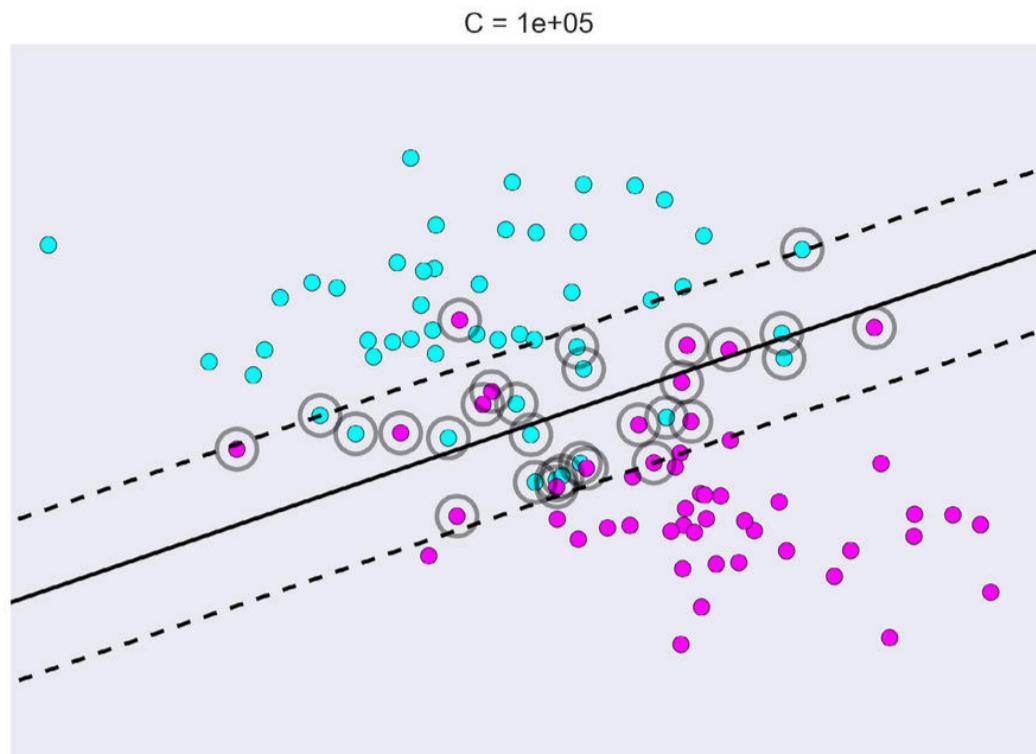


Figure 13.13: The value of C tells SVM how sensitive to be to points that can “intrude” into the zone around the line that’s fit to the data. The smaller the value of C , the more points are allowed. Here we set C to 100,000 (or $1e+05$).

Let’s drop C down to 0.01. Figure 13.14 shows that this lets many more points into the region around the line.

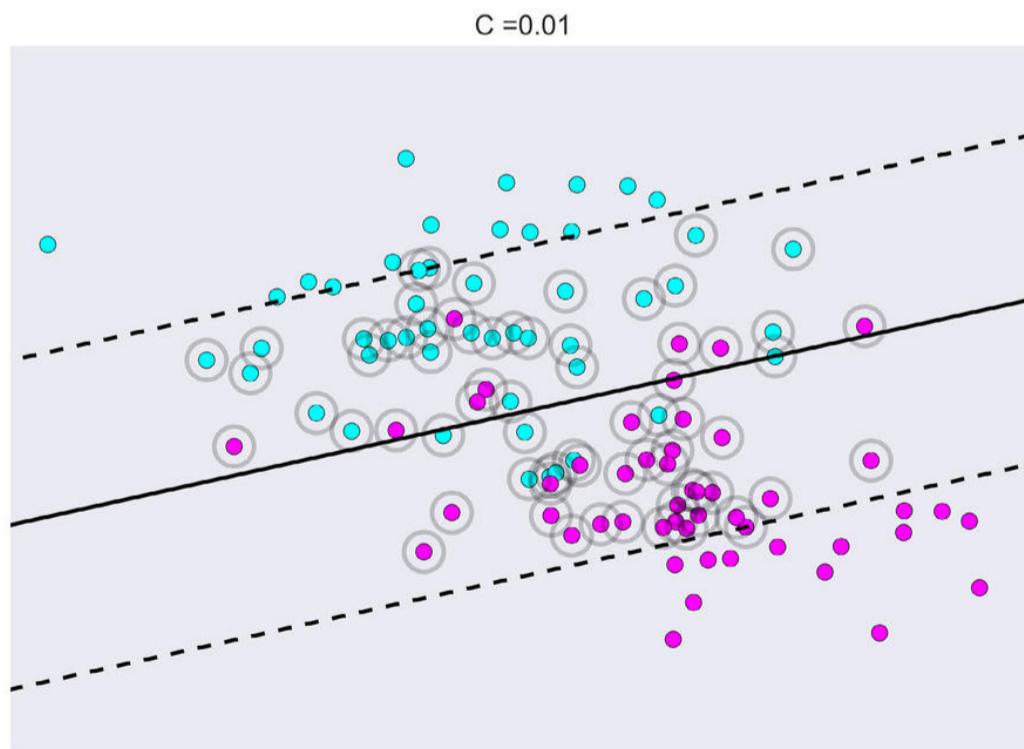


Figure 13.14: Lowering C to 0.01 lets in many more points. As a result of having more support vectors to work from, the line that’s fit is more horizontal than in Figure 13.13.

The lines in Figure 13.14 and Figure 13.13 are different. Which one we prefer depends on what we want from our classifier. If we think the best boundary will come from the details near the zone where the points overlap, we'll want a large 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'd want a smaller value of C to include more of those farther-away points.

The SVM algorithm has a trick up its sleeve. Suppose that we have the data of Figure 13.15, where there's a blob of samples of one category surrounded by a ring of samples of another. There's no way we can draw a line to separate these two sets.

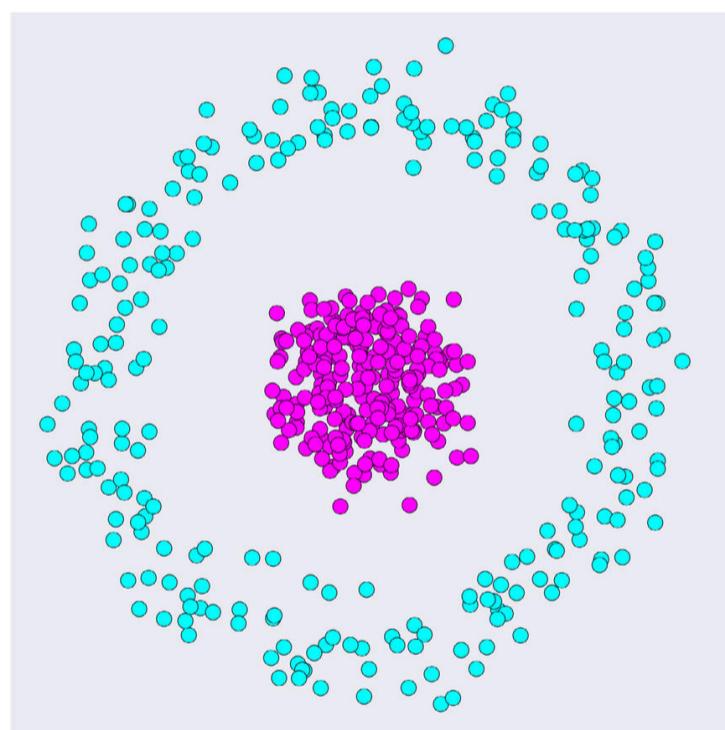


Figure 13.15: This set of samples is a challenge for basic SVM, since there's no straight line that can separate them.

Here comes the clever part. What if we add a third dimension to each point by elevating it by an amount based on that point's distance from the center of the rings? Figure 13.16 shows the idea.

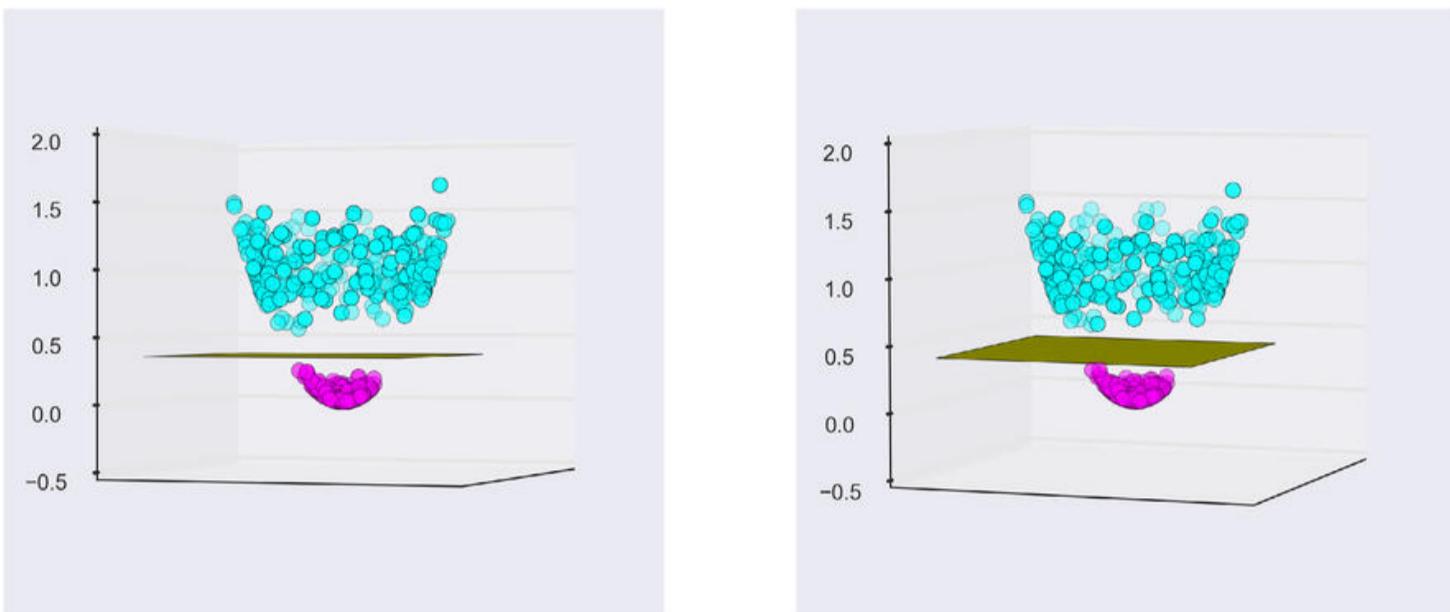


Figure 13.16: If we push each point in Figure 13.15 upwards by an amount based on its distance from the center of the pink blob, we get two distinct clouds of points. It's now easy to place a plane between them. Just as a line is the linear element in the plane, a plane is the linear element in space, so it's the sort of thing SVM can find. These two images are two views of the same data, showing the points and the plane between them.

As we can see in Figure 13.16, 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 13.17 highlights the support vectors for the plane between the two clusters of points.

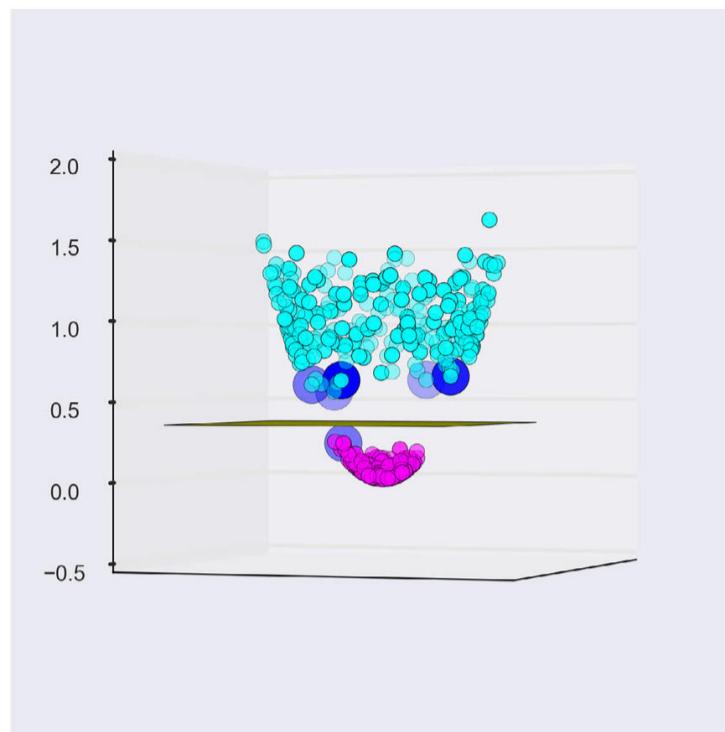


Figure 13.17: The support vectors for the plane.

Now all points above the plane can be placed into one category, and all those below into the other.

If we highlight the support vectors we found from Figure 13.17 in our original 2D plot, we get Figure 13.18.

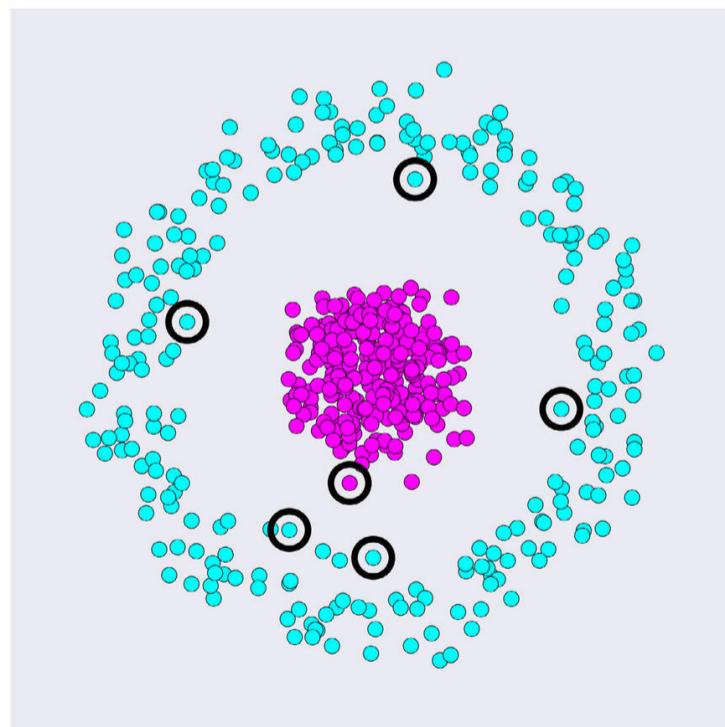


Figure 13.18: Looking down on Figure 13.17.

If we include the boundary created by the plane, we get Figure 13.19.

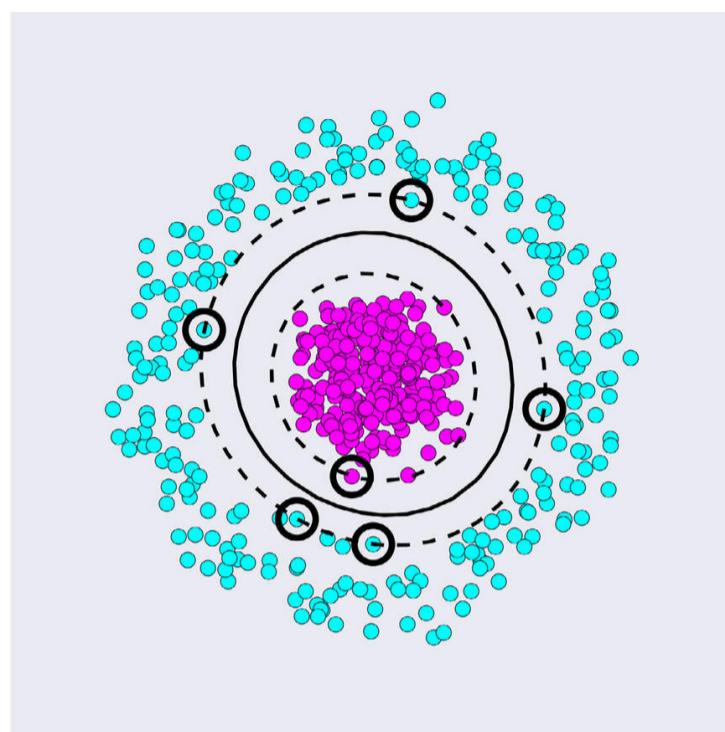


Figure 13.19: The intersection of the curved shape we built from our data and the plane we placed in 3D is shown by the solid line. Here we show the support vectors and the margin they create.

In this case we found the right way to modify our data basically by looking at it, and then coming up with a good 3D transformation of the data that let us split it apart. We don't want to have to go through that process manually every time. When the data has many dimensions we might not even be able to visualize it well enough to even guess at a good transformation.

The beauty of the approach we've described here is that the search can be automated in a clever way. The SVM algorithm can effectively try many different transformations and find the dividing line or surface for each one, eventually choosing the one that works best.

But even that would be unattractive, because the operation of adding dimensions to the data and then finding the support vectors in that expanded dataset can take a lot of time, and the bigger our dataset, the more time it will take.

The surprise that makes this search practical is that there's a way to write the mathematics that speeds things up significantly.

The technique involves modifying a piece of math called the **kernel**, which forms the heart of the algorithm. Mathematicians sometimes honor a particularly clever or neat idea with the complimentary term "trick" (like the "bias trick" we saw in Chapter 10). In this case, rewriting the SVM math to handle this extra work efficiently is called the **kernel trick** [Bishop06]. The kernel trick lets the algorithm find the distances between transformed points without actually transforming them! That's a neat trick, and a major efficiency boost.

The kernel trick is used automatically by all major libraries, so we don't even have to ask for it [Raschka15].

13.5 Decision Trees

Let's consider another non-parametric classification method. Like kNN, this saves up the training samples as they come in, but it builds a structure around them. When it comes time to evaluate a new sample, we use that structure to come up with an answer quickly.

We can illustrate the idea 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) asks a series of yes/no questions.

If the guesser can correctly identify the target in 20 questions or less, they win. One reason the game endures is that it's fun to narrow down the enormous number of possible people, places, and things down to one specific instance with such a small number of simple questions.

A typical structure is shown in Figure 13.20.

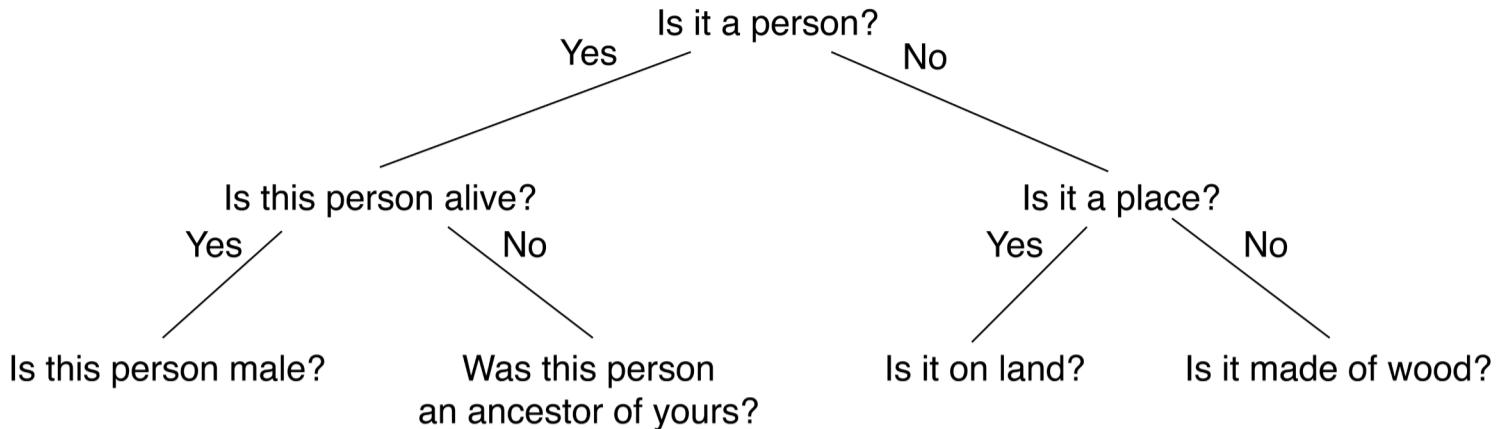


Figure 13.20: A tree for playing “20 questions.” Note that after each decision there are exactly two choices, one each for “yes” and “no.”

We call a structure like Figure 13.20 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** 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 requires that.

We 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 13.21 shows a tree with these labels.

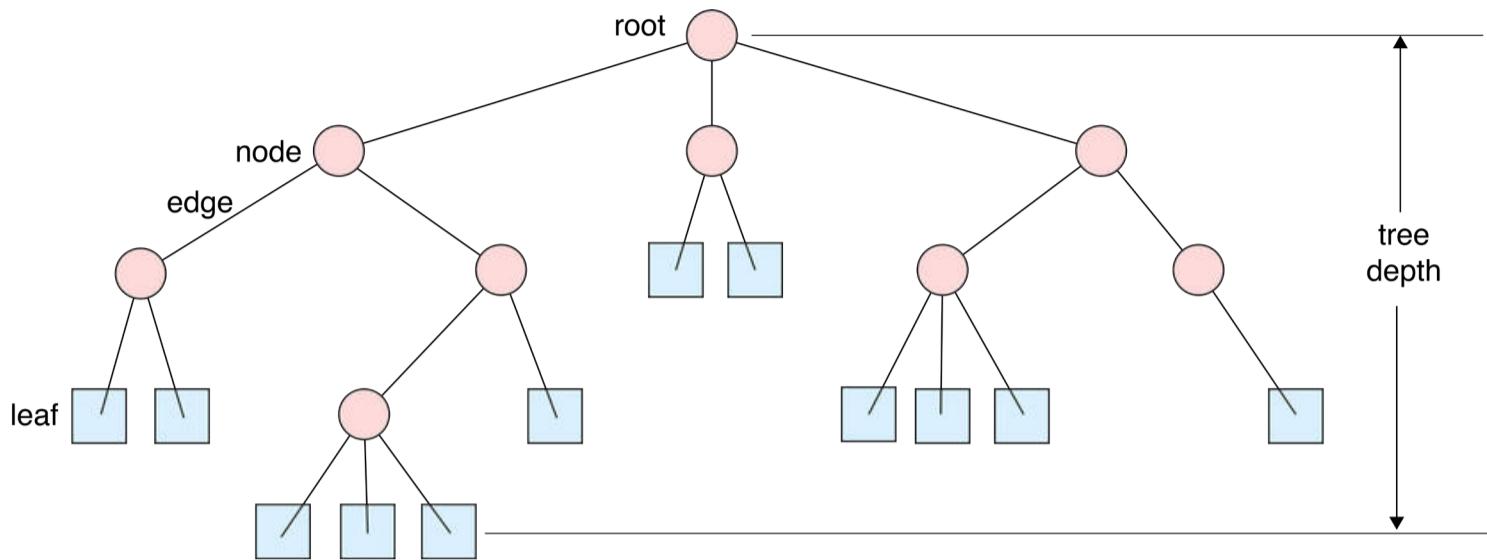


Figure 13.21: Some terminology for a tree. Each circle is a node. The node at the top has a special name: the root. The lines that join nodes are called edges. The tree’s depth is the number of nodes from its root to its farthest child. Here, the depth is 4.

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 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 **branch** or **sub-tree**. Nodes that share the same immediate parent are called **siblings**.

The family-tree language usually doesn’t go farther than that, so we rarely see references to things like a “great-grand-nephew-twice-removed” node.

Figure 13.22 shows some of these ideas graphically.

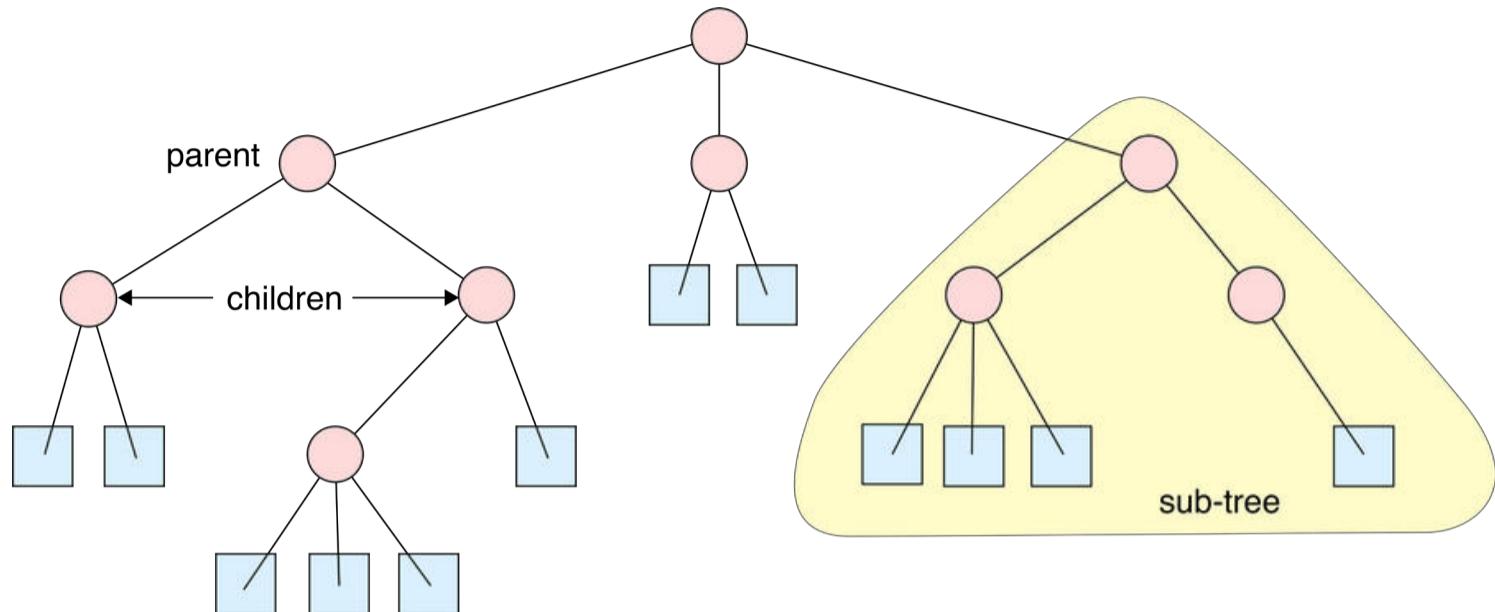


Figure 13.22: Some more terms applied to trees. A node is a parent if there are nodes immediately below it. Those nodes are called children of that parent. A subtree is a piece of the tree that we want to focus on.

An interesting quality of the *20 Questions* tree is that it is **binary**: every parent node has exactly two children. This is a particularly easy kind of tree for a computer to build and use. 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 in Figure 13.23(a), and the corresponding binary tree is shown in Figure 13.23(b). 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.

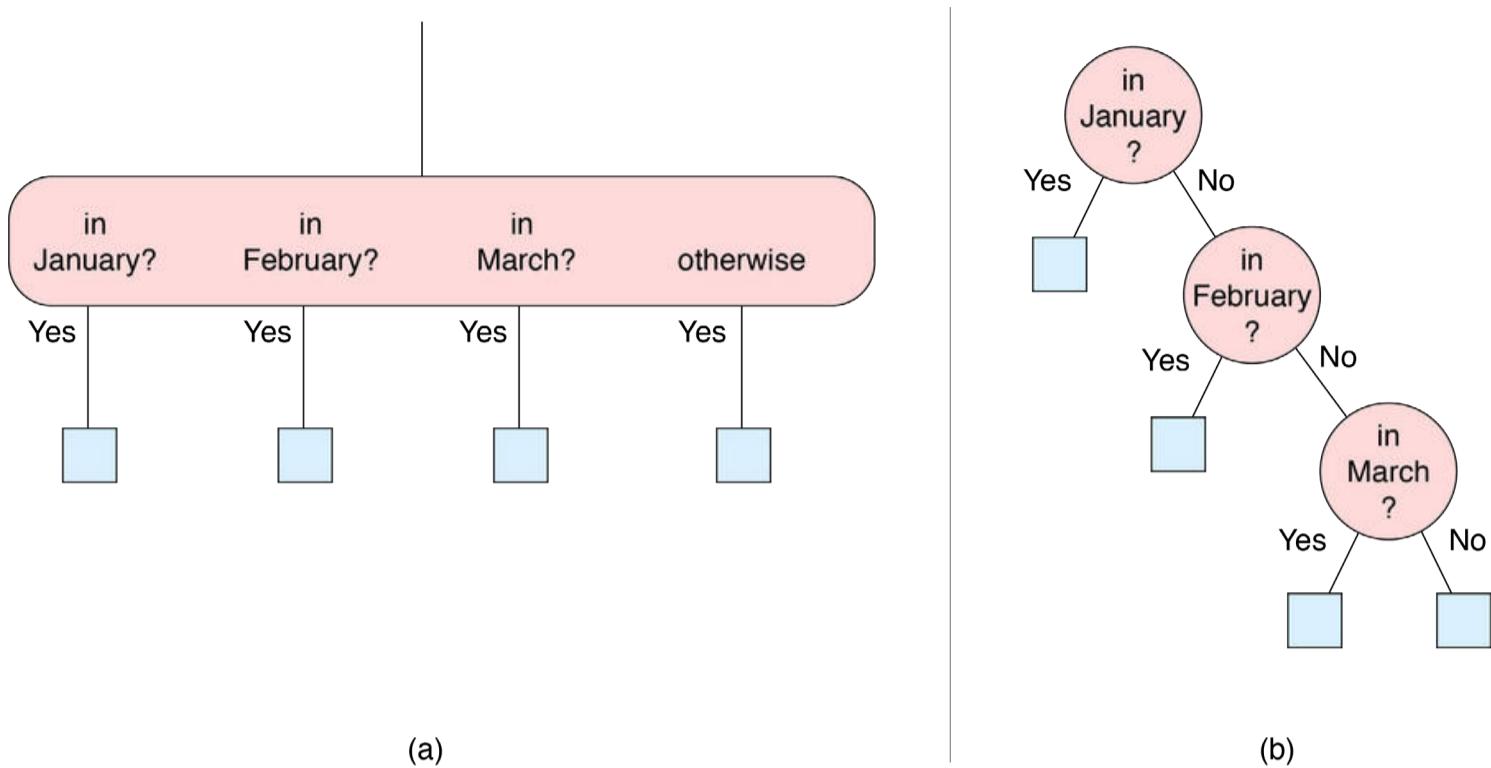


Figure 13.23: We can imagine a tree where there are many children after a node, each with its own test. We can always turn this kind of bushy tree into a binary tree that has just one yes/no question at every node.

Of course, we've been building up all of these tree ideas and terms in order to use them. The technique of **decision trees** uses this general idea to categorize data. 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 out continuous variables, as in regression problems. These are called **continuous variable decision trees**.

We'll stick with the categorical versions here. An example of such a tree is shown in Figure 13.24.

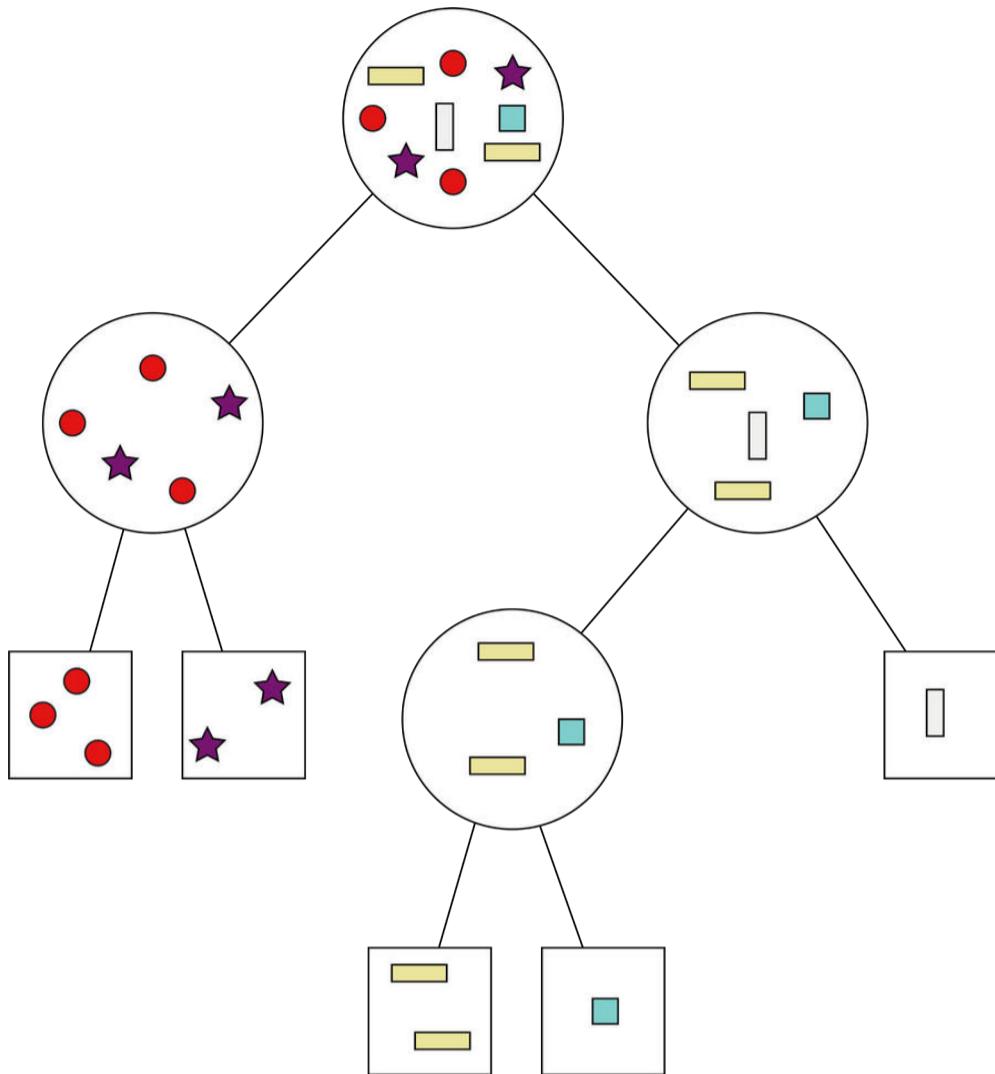


Figure 13.24: A categorical decision tree. From the many samples at the top, each with its own category, we apply a test at each node until we produce leaves that have only one category of samples each.

The general idea is that we build the tree during training. Let's look at an idealized version of the tree building and evaluation process.

The root and all parent nodes contain tests based on the features of a sample. The leaf nodes contain the training samples themselves. When a new sample arrives, we start at the root and descend, following the branches based on the tests evaluated on the features of this sample, as in Figure 13.23.

When we reach a leaf node, we test to see if the new training sample has the same category as all the other samples in that leaf. If so, we add the sample to the leaf and we're done. Otherwise, we **split** the node, and come up with some kind of test based on the features that will let us distinguish between this sample and the previous samples.

in the node. That test gets saved with the node, and we create at two children, moving each sample into the appropriate child, as in Figure 13.25.

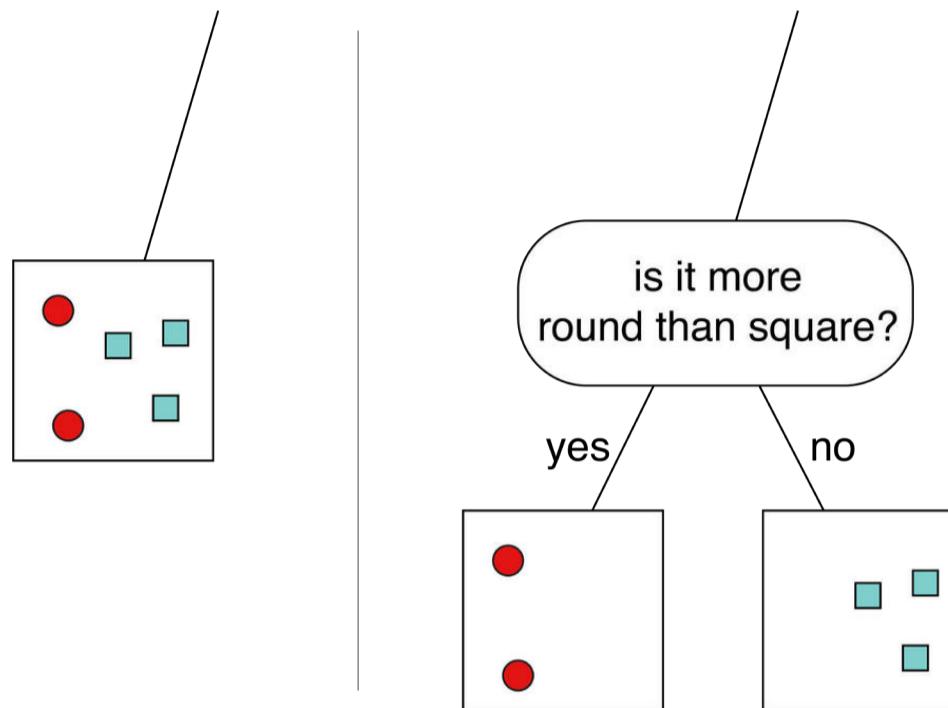


Figure 13.25: Splitting a node. Left, we see a leaf node that has both round and square samples. Right, we decide to turn that leaf into a regular node by replacing it with a test. The test creates two leaf nodes, each of which has more uniform contents than the leaf did before it was split.

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 branches based on the sample's results when we apply its features to the test at each node. When we land in a leaf, we report that the sample belongs to the category of the objects in that leaf.

This was an idealized process. In practice, our tests can be imperfect and our leaves might contain a mixture of objects of different categories, because for efficiency or memory reasons we chose not to split it any more. For example, if we land in a node that contains samples that are 80% from category A and 20% from category B, we might report that the new sample has an 80% chance of being in A, and 20% chance of being in B. If we have to report just one category, we might report that it's A 80% of the time, and B the other 20%.

We call this a **greedy** algorithm. There's no global strategy that helps us find the smallest or most efficient tree. Instead, we split nodes on the fly using just the information that we have on-hand at that moment.

13.5.1 Building Trees

Let's look at the tree-building process for a couple of examples.

The data in Figure 13.26 shows a cleanly-separated set of data representing 2 classes.

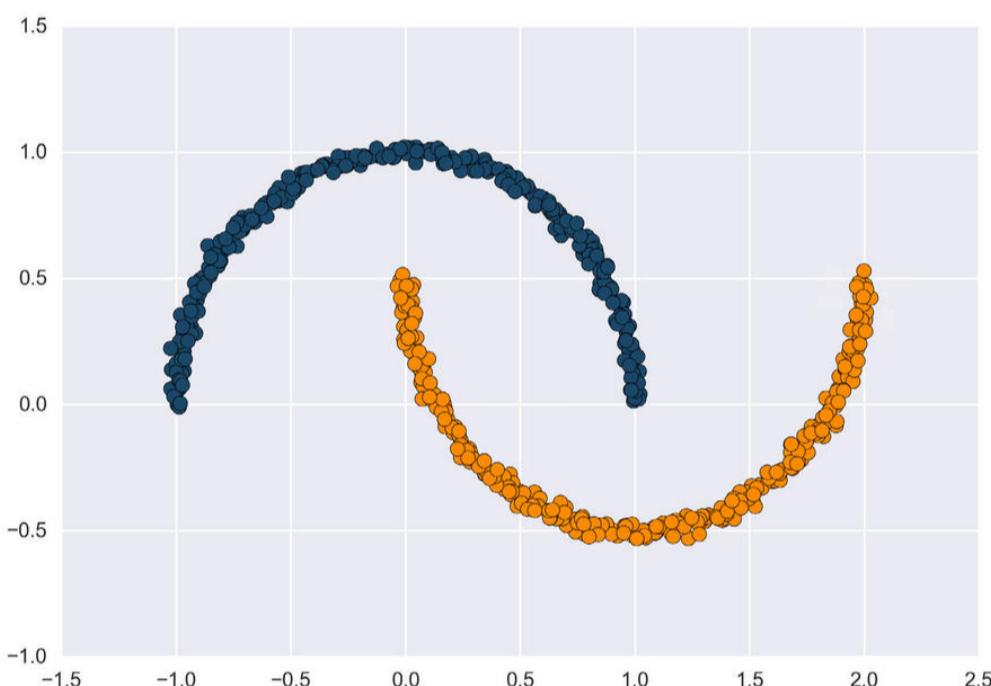


Figure 13.26: Our 600 starting data points for building a decision tree. These 2D points represent two classes, blue and orange.

Each step in building a tree 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. So we often think about the size of our tree in terms of the number of leaves it contains. The tree-building process is shown in Figure 13.27 for increasingly large numbers of leaves.

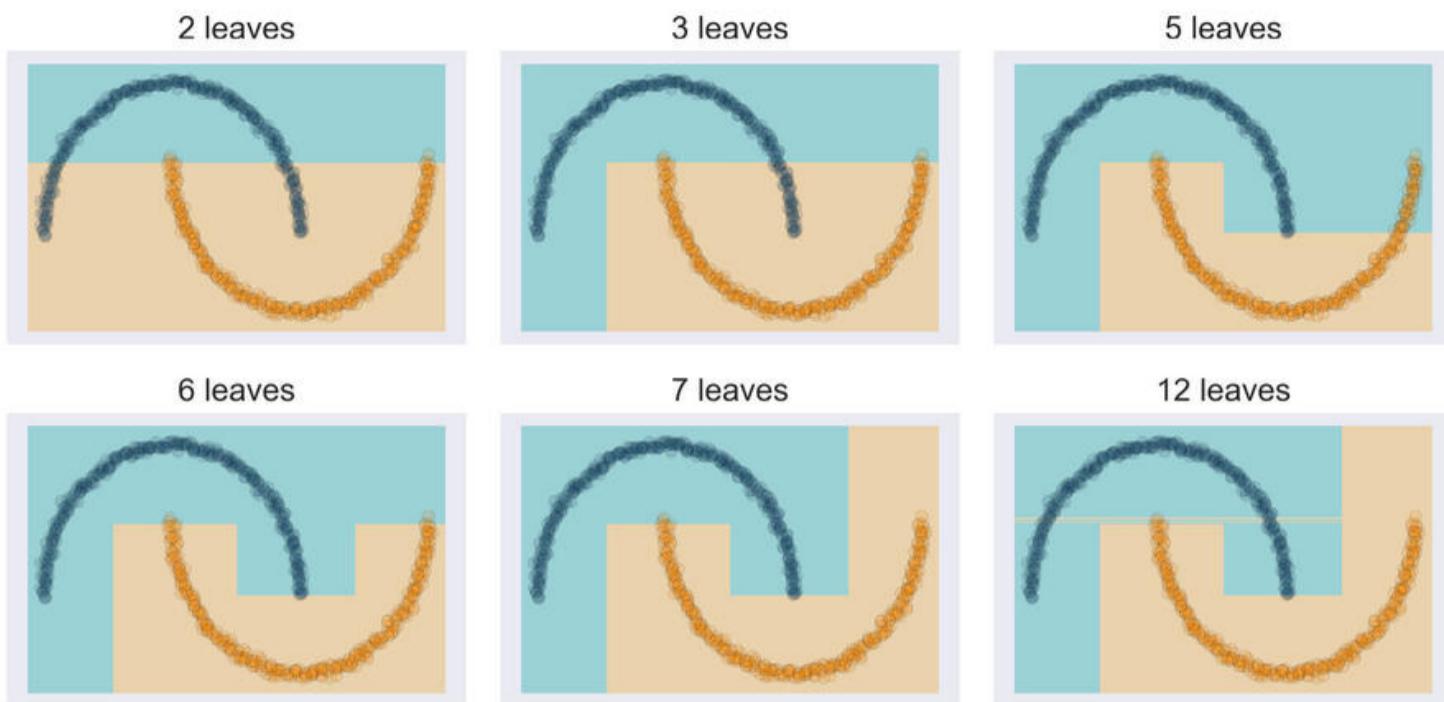


Figure 13.27: Building a decision tree for our data in Figure 13.26. Notice how the tree starts with big chunks and moves on to smaller and more precise regions.

In this case our cells are rectangles (as is usual), and our splits are vertical or horizontal lines that cut a rectangle in two. Notice how the regions gradually refine as they fit the data.

This tree needs only 10 leaves. The final tree and the original data are shown together in Figure 13.28. This tree fits the data perfectly.

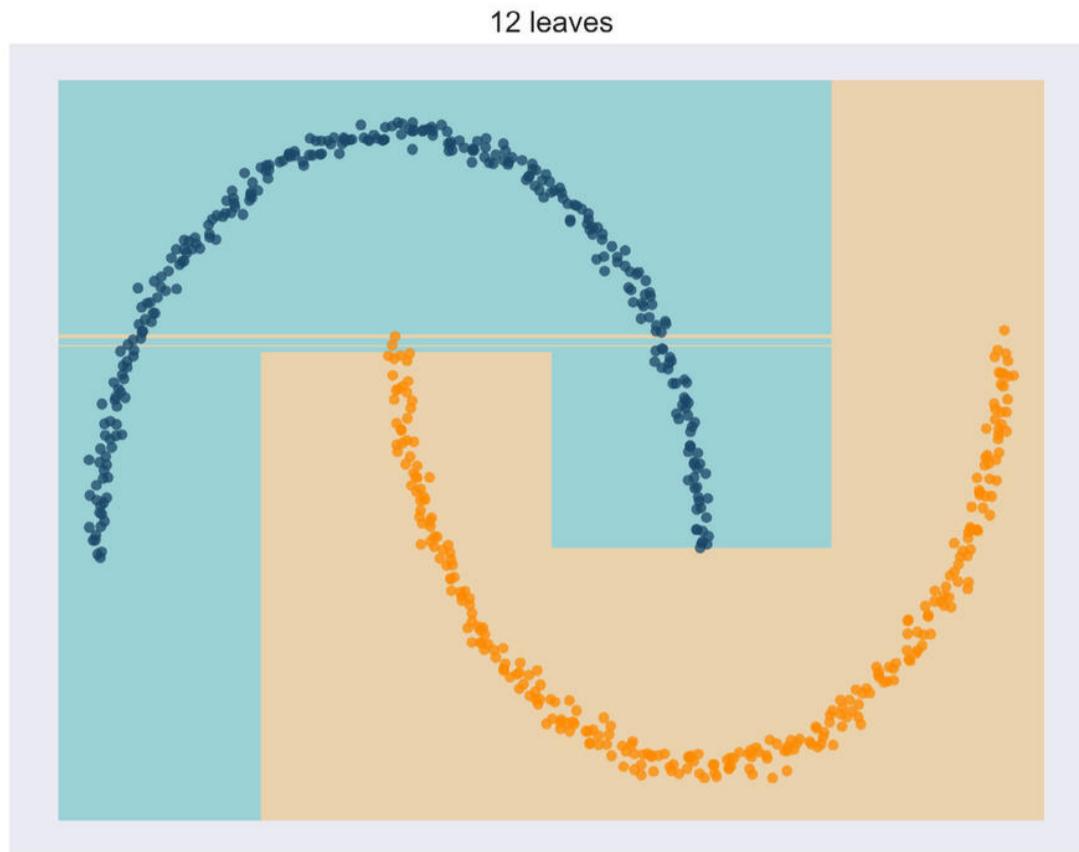


Figure 13.28: Our final tree with 10 leaves. Note the two horizontal, very thin rectangles. They enclose the centers of three blue samples at the top of the left side of the arc, while slipping between the orange dots. Thus all points are correctly classified.

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 influences the tree's shape. To see this, in Figure 13.29 we ran the same algorithm as for Figure 13.28 two times, but in each case we used a different, randomly-chosen 70% of the input data.

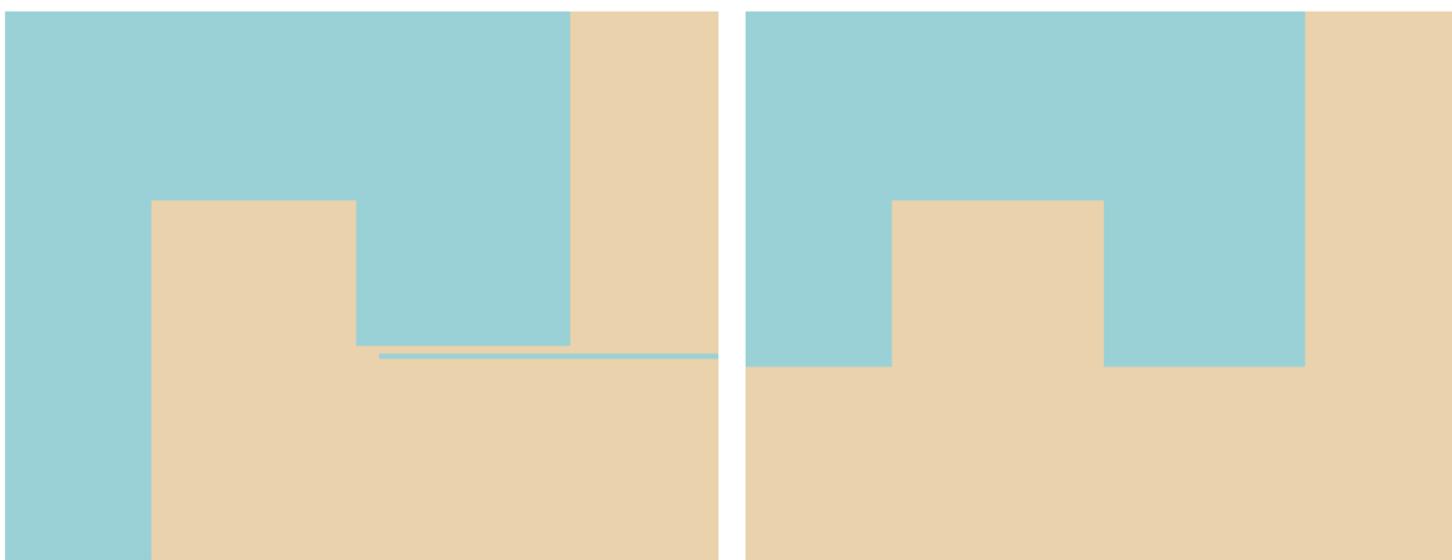


Figure 13.29: Decision trees are very sensitive to their inputs. (a) We randomly chose 70% of the original samples and fit a tree. (b) The same process but for a different randomly-selected 70% of the original samples.

These two decision trees are similar, but definitely not identical. This problem is much more pronounced when the data isn't so easily separated, so let's look at an example of that now.

Figure 13.30 shows another pair of half-moons, but this time with lots of noise added to the samples after they've had their categories assigned. The two categories no longer have a clean boundary.

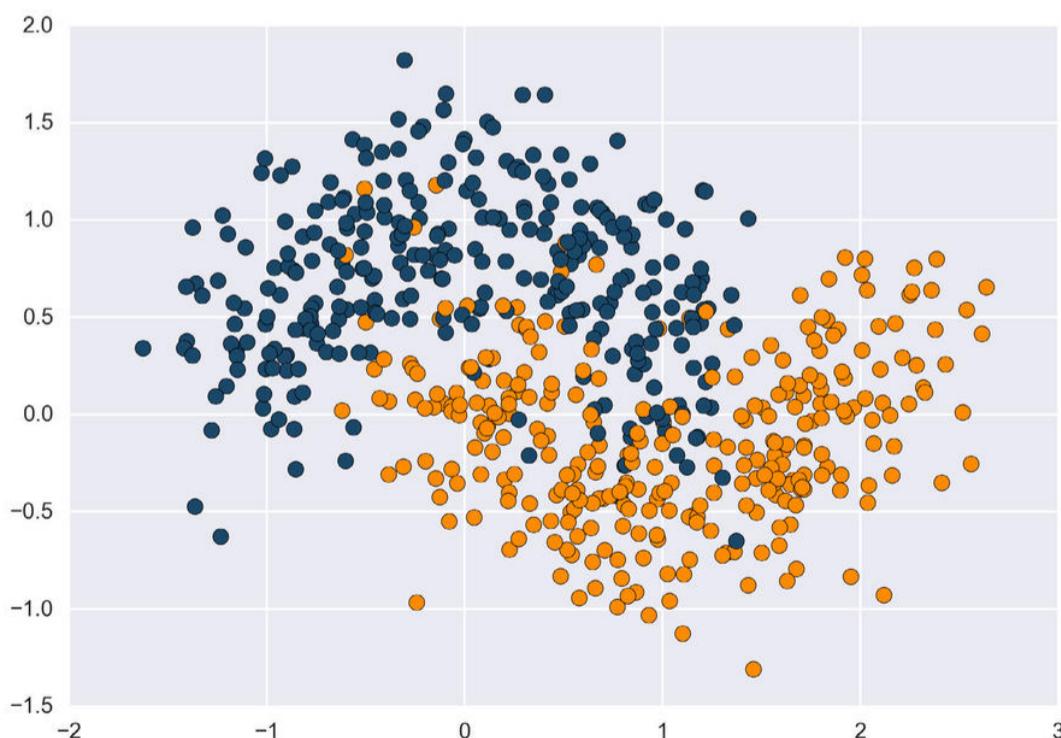


Figure 13.30: A noisy set of 600 samples for building decision trees.

Fitting a tree to this data starts out with big regions, but it rapidly turns into a very complicated set of tiny boxes as the algorithm splits up nodes this way and that to match the noisy data. Figure 13.31 shows the result. In this case, it required 100 leaves for the tree to correctly classify the points.

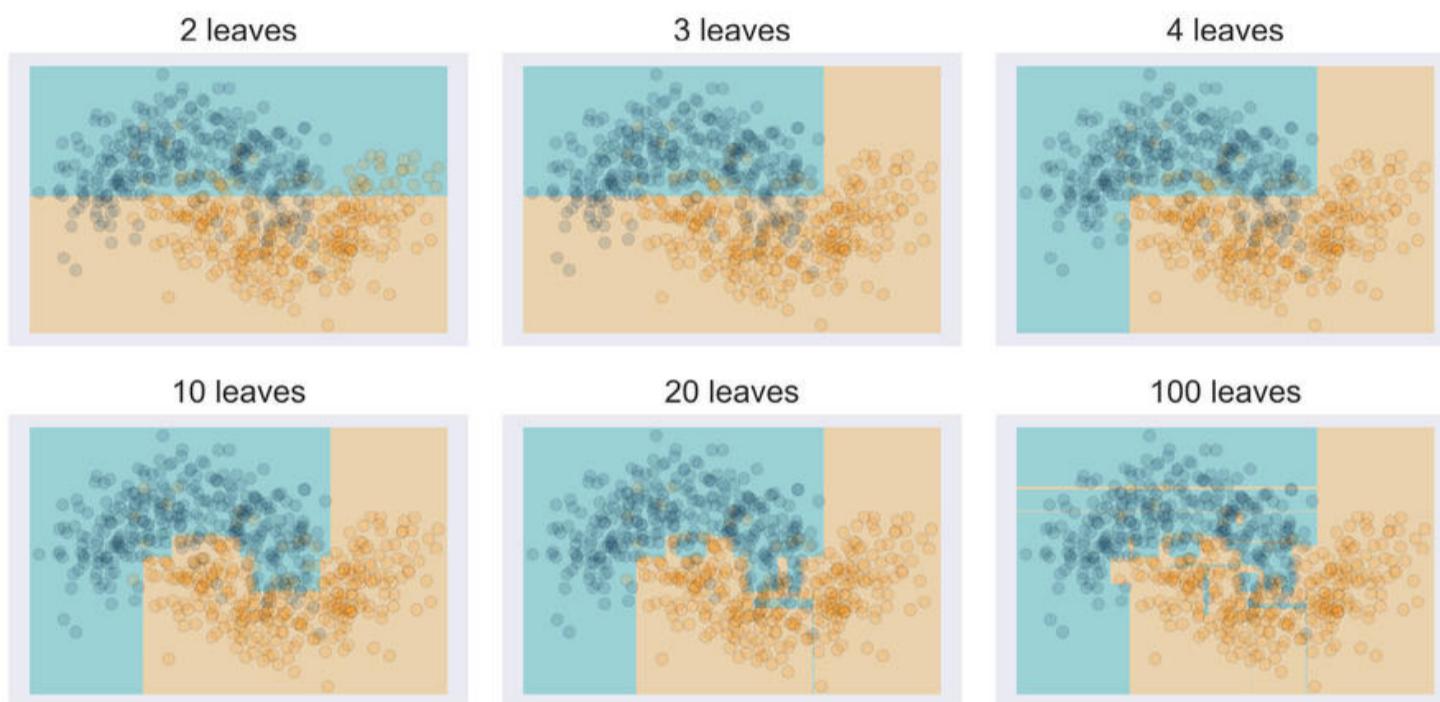


Figure 13.31: The tree-building process. Note that the second row uses large numbers of leaves.

Figure 13.32 shows a close-up of the final tree and the original data.



Figure 13.32: 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'd 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 now repeat our process of building trees using a random 70% of the data. Figure 13.33 shows the results.

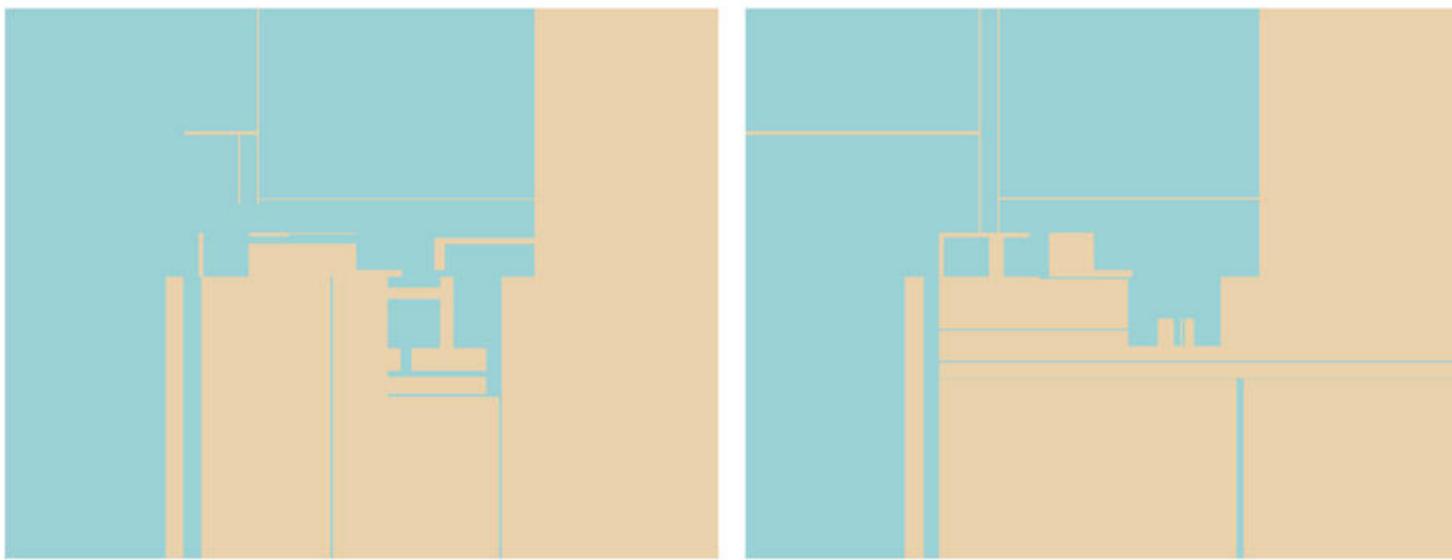


Figure 13.33: A couple of trees built with different sets of 70% of the original samples.

There are similarities, but these trees are significantly different, and there are lots of little bits that exist only to categorize a few samples. This is overfitting in action.

Although this may look pretty bad for the decision tree method, we'll see in Chapter 14 that by combining many very simple decision trees into a group, or **ensemble**, we can create classifiers that perform very well, and do it efficiently.

13.5.2 Splitting Nodes

Before we leave decision trees, let's return briefly to the node-splitting process, since many libraries offer us a choice of splitting algorithms to choose from.

There 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 its **purity**. This is a number that tells us how uniform the samples in that node are. If they're all of the same category, then the node is completely pure. The more samples we have of other categories, 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 might test the value of just one feature, and ignore the others. We might look at groups of features and test on some aggregated values from them. We're free to choose a completely different test, based on different features, at every node. This gives us a huge variety of possible tests to comb through.

Figure 13.34 shows a node containing a mix of objects of different categories. We'd like to get all the reddish objects in one child, and all the bluish ones in another. Rather than test on color, we try using a simpler test based on the radius of each circle. The figure shows the result of splitting on the radius using three different values.

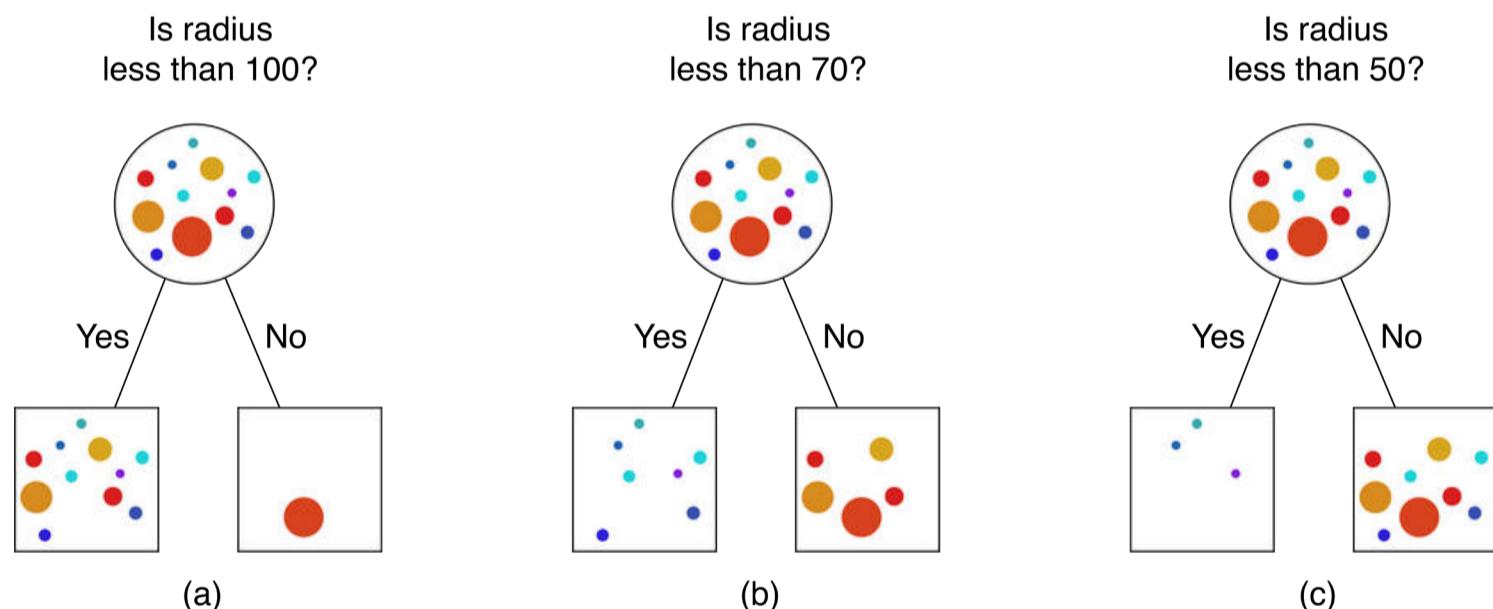


Figure 13.34: We'd like to split the top node, gathering all the red-like objects in one child and all the blue-like objects in the other. Here we're using the radius as the feature to split on. (a) Using a radius of 100, all the circles pass the test except for the largest red one. (b) Using a radius of 70 puts the red-like circles in one child, and the blue-like circles in the other. (c) Using a radius of 50 leaves a few blue children in one child, with all the red objects and some blue in the other. The value of 70 produces the most pure results.

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. So 70 is the best value to use if we choose to split on the radius feature. If we use this test for this node, we'll remember both which feature we're splitting on (the radius) and what value to test for (70).

Since we could potentially split the node using any feature in the samples, we need some way to evaluate those tests so we can pick the one that produces the best results, which here means the children that are purest.

Let's look at two popular ways to test the quality of a split. Each approach involves measuring some characteristic of the children cells that result from the split we're considering.

The **Information Gain** measure, also called **IG**, computes the entropy of all the children cells, adds them together, and compares that result to the entropy in the parent cell. Recall from Chapter 6 that the entropy is a measure of complexity, or how many bits it would take to communicate some piece of information. When a cell is pure, it has very low entropy. As the cell contains more samples from multiple classes, the purity goes down and the entropy goes up.

So if a node's children are each purer than the parent, they will have a lower total entropy. 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 misclassification of a sample. For example, suppose a leaf has 10 samples of category A and 90 samples of category B. If a new sample ends up at that leaf and we need to report a single class, and we reported A, there'd be a 10% chance that we were wrong. Gini impurity measures those errors at each leaf, and after trying multiple splits, it 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 each option and pick the one that works the best.

13.5.3 Controlling Overfitting

As we mentioned above, decision trees want to classify every sample correctly, so they have a strong tendency to overfit.

As we saw in Figure 13.27 and Figure 13.31, 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.

So one popular strategy is to limit the tree's depth while it's building. We simply don't split any nodes that are more than a given number of steps from the root.

A different but related strategy says that we'll never split a node that has less than a certain number of samples, no matter how mixed they are.

Another approach is to reduce the size of the tree after it's made, in a process called **pruning**. This works by trimming leaves. 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 that leaf and put its samples back into its parent. If we remove all the children of a node, then it becomes a leaf itself, and a candidate for further pruning. Pruning the tree makes it shallower, which offers the additional benefit of also making it faster when categorizing new data.

Because depth limiting and pruning simplify the tree in different ways, they will usually give us different results.

13.6 Naïve Bayes

Let's look at a parametric classifier that is often used when we need quick results, even if they're not the most accurate.

This classifier works very fast because it begins with assumptions about the data. The classifier 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 which then becomes our new prior.

But what if we just commit to the prior ahead of time, and then do our best to make it work?

The **Naïve Bayes** classifier takes that approach. It's called "naïve" because the assumptions we make in our prior are not based on the contents of our data. That is, we're making an uninformed, or "naïve," characterization of our data. We just *assume* that the data has a certain structure.

If we're right, great, we'll get good results. The less well the data matches this assumption, the worse the results will be. Naïve Bayes is popular because this assumption turns out to be correct often enough that it's worth taking a look.

The interesting thing is that we never even check to see if our assumption was justified. We just plow ahead as if we were certain.

In one of the more common forms of naïve 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 assert that they will look like a Gaussian curve.

If our features really do follow Gaussian distributions, then this assumption will be a good fit. The amazing thing about naïve Bayes is that this assumption seems to work well far more frequently than we'd probably expect.

Let's see it in action.

We'll start easy, with data that *does* satisfy the prior. Figure 13.35 shows a dataset that was created by drawing samples from two Gaussian distributions.

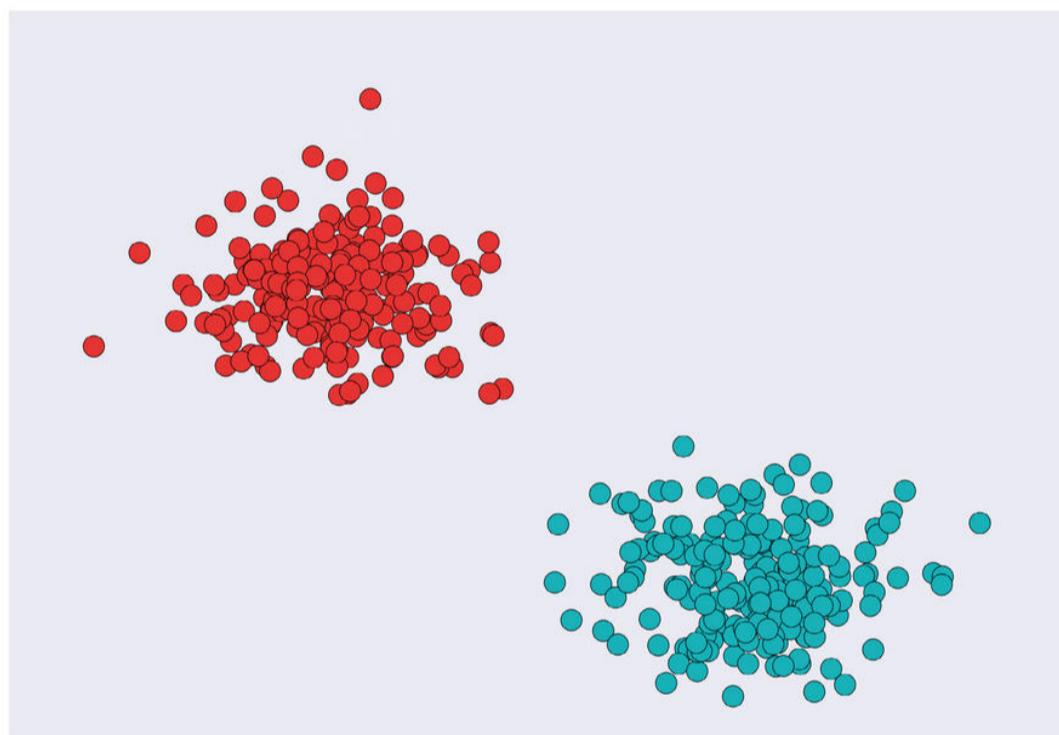


Figure 13.35: A set of 2D data for training with naïve Bayes. There are two classes, red and blue.

When we give this data to a naïve 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 13.36.

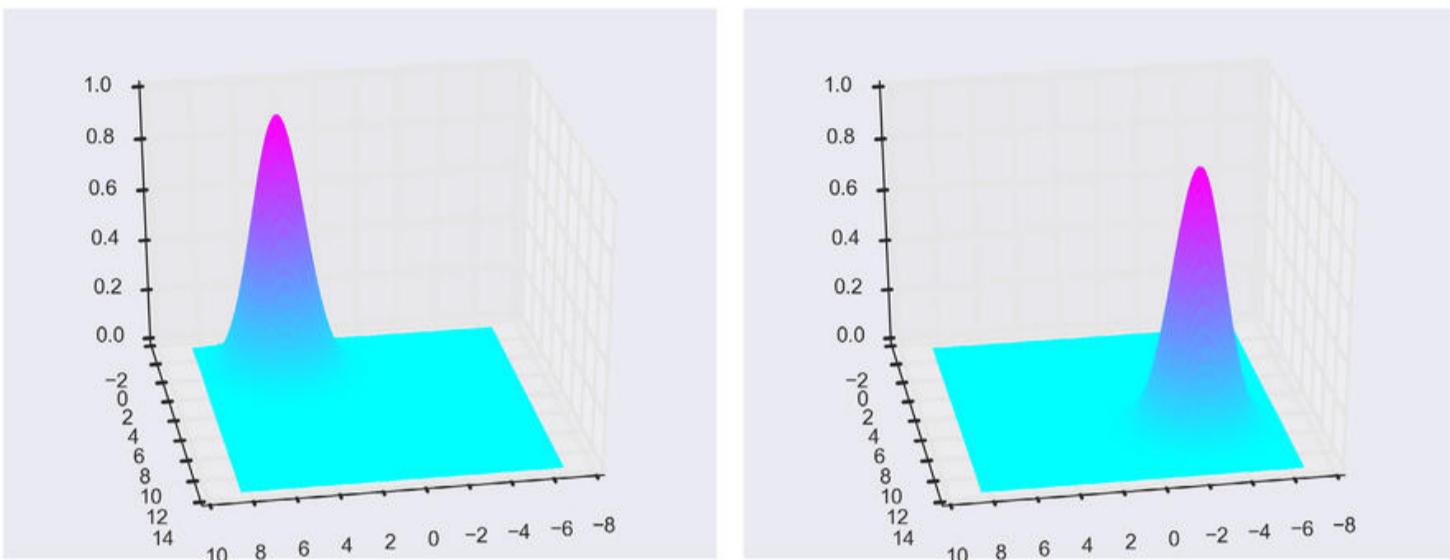


Figure 13.36: Naïve Bayes fits a Gaussian to each of the X and Y features of each class. Left: The Gaussian for the red class. Right: The Gaussian for the blue class.

If we overlay the Gaussian blobs and the points and look directly down, as in Figure 13.37, 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 naïve Bayes classifier was expecting.

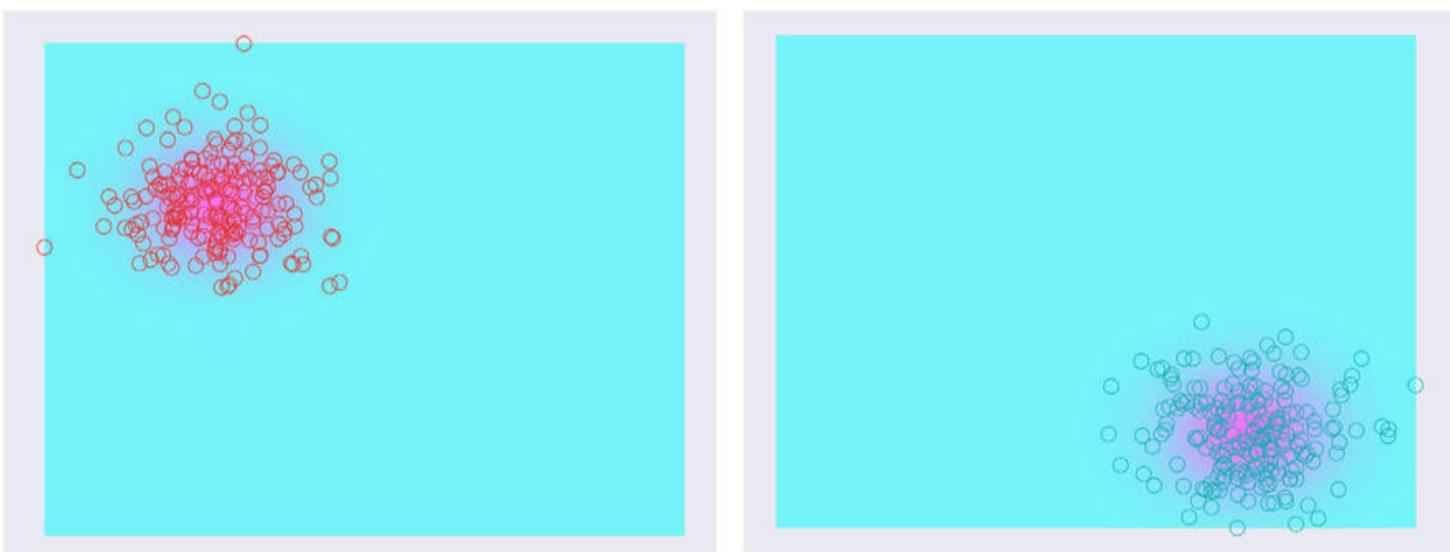


Figure 13.37: Our entire training set overlaid on the Gaussians of Figure 13.36. The naïve Bayes classifier found good matches to the distributions that these points were originally drawn from.

To see how well the classifier will work in practice, let's split the training data, putting a randomly-selected 70% of the points into a training set, and the rest into a test set. We'll train with this new training set, and then draw the test set on top of the Gaussians, giving us Figure 13.38.

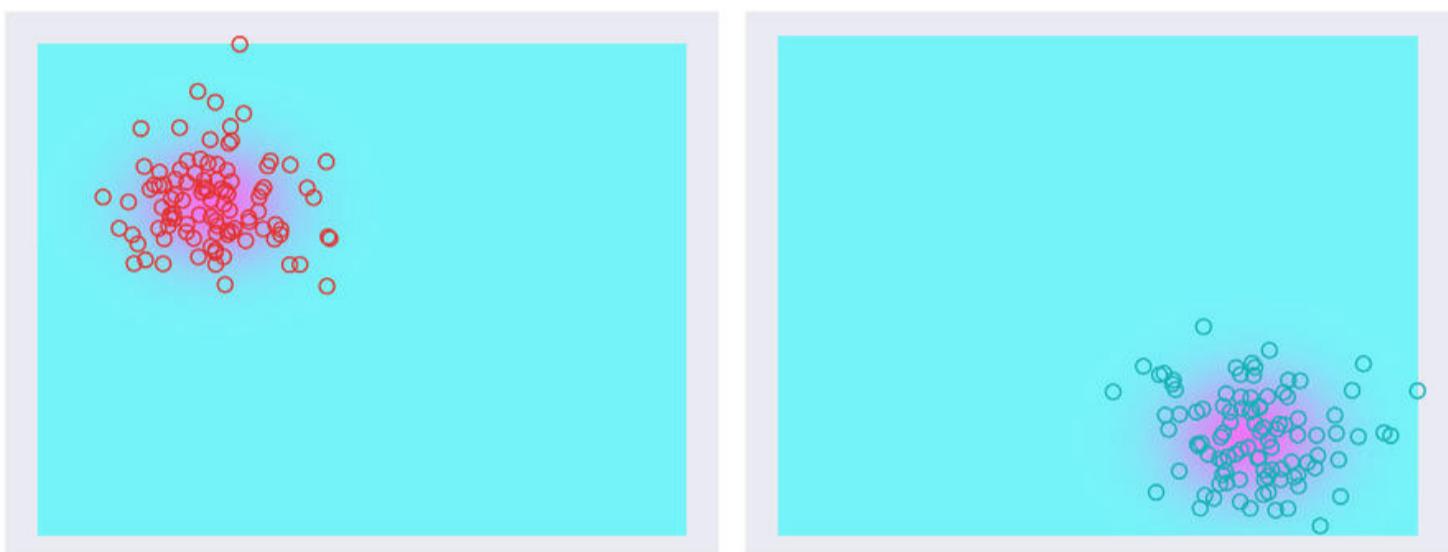


Figure 13.38: The test data after training with 70% of our starting data. The predictions are very good.

In Figure 13.38 we drew all the points that were classified as belonging to the first set on the left, and all those in the second set on the right, maintaining their original colors. We can see that all of the test samples were correctly classified.

Now let's try an example where we *don't* satisfy the prior that all features of all samples follow Gaussian distributions.

Figure 13.39 shows our new starting data of two noisy half-moons.

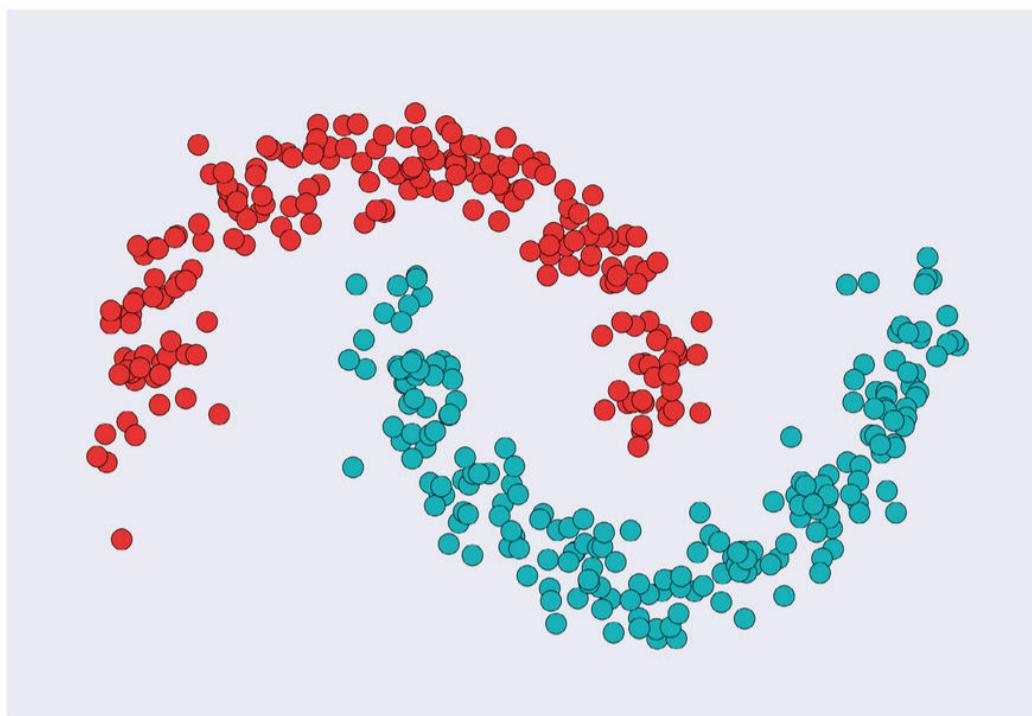


Figure 13.39: Some noisy half-moon data. The red and blue classes are not drawn from Gaussian distributions, so we'd expect naïve Bayes to match them poorly.

When we give these samples to the naïve 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 13.40.

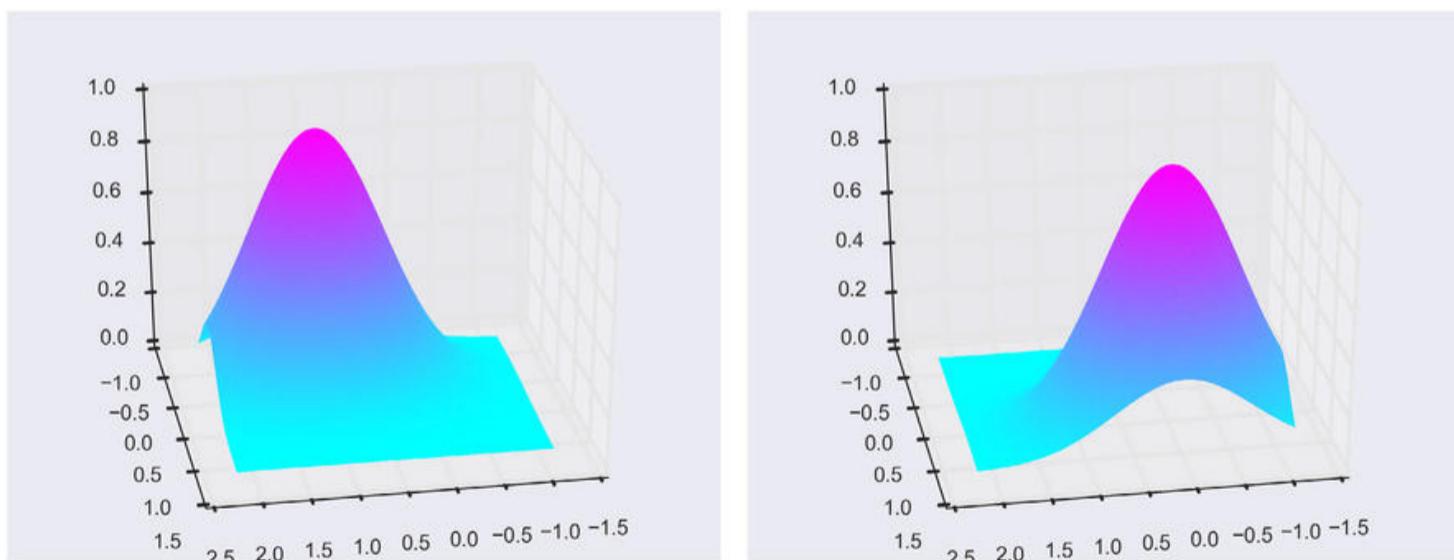


Figure 13.40: When our Gaussian-based naïve Bayes algorithm matches our half-moons data of Figure 13.39, it assumes the values are Gaussians, because it always assumes that. These are the best Gaussians it could fit to the data.

Of course these will not be a good match to our data, because they don't satisfy the assumptions. Overlaying the data on the Gaussians in Figure 13.41 shows that the matches aren't abysmal, but they're pretty far off.

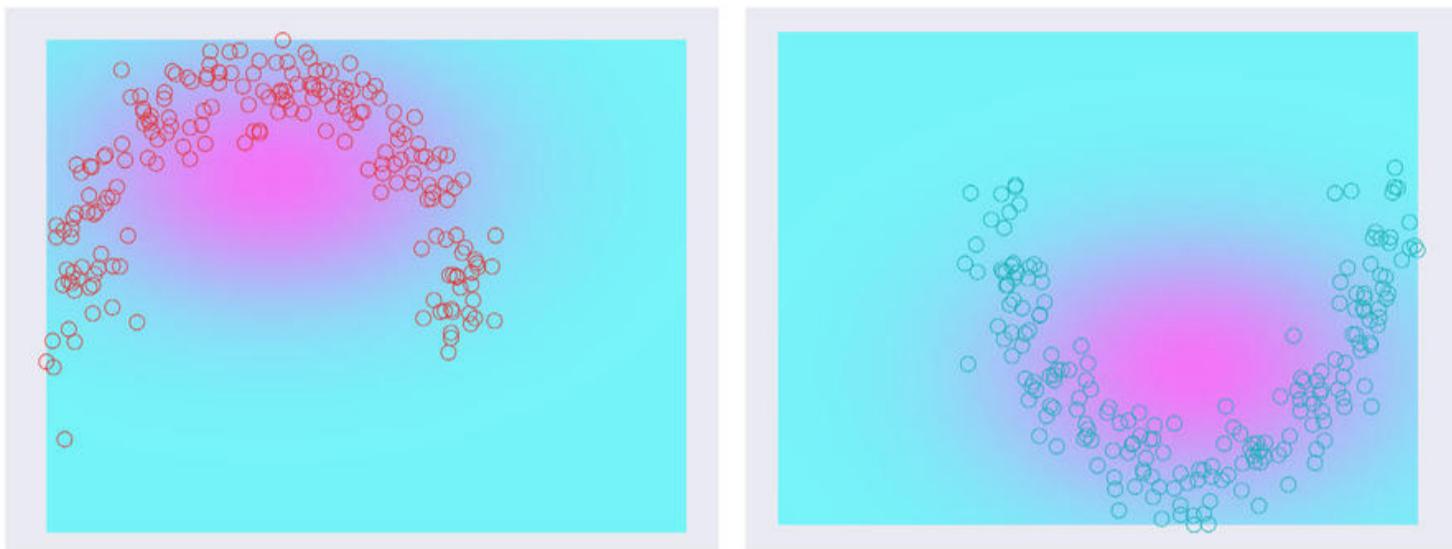


Figure 13.41: Our training data from Figure 13.39 overlaid on the Gaussians of Figure 13.40.

Like before, we'll now split the half-moons into training and test sets, train on the 70%, and look at the predictions. In the left image of Figure 13.42 we can see all the points that were assigned to the red class. As we would hope, this has most of the red points. But some of the red points are missing, and some of the blue points have snuck in, because their value from the first Gaussian is higher than the second.

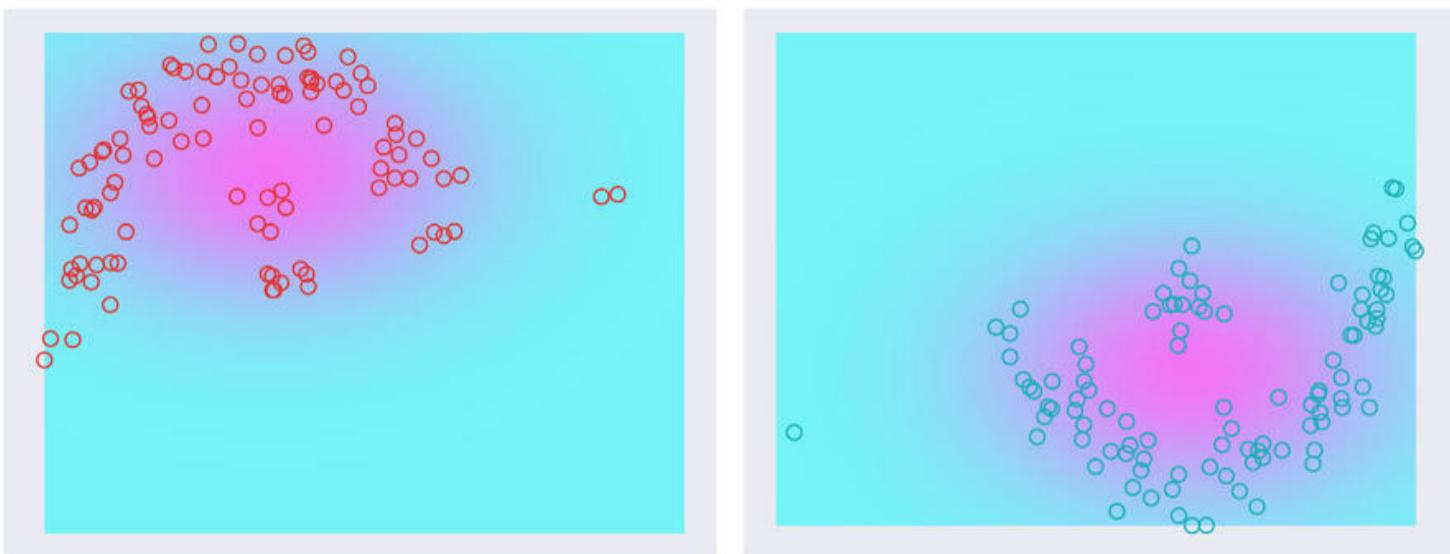


Figure 13.42: After training with a subset of the noisy half-moons, these are the predicted classes for the test data. Left: Most of the red samples got placed into the red class, but some red samples were let out, and some blue samples were incorrectly classified as red. Right: Most of the blue samples were correctly classified as blue, but not all of them, and some red samples were misclassified as blue.

On the right of Figure 13.42 we can see that most of the blue points got assigned to the blue class, but some of the red points snuck in.

We shouldn't be too surprised that there were misclassifications, because our data did not follow the assumptions made by the naïve Bayes prior.

What is amazing is how well it did. In general, naïve Bayes does a surprisingly 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.

Because naïve Bayes is so fast and straightforward, it's common to include it as one of the classifiers we look at 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.

13.7 Discussion

We've looked at four popular classification algorithms. Most machine-learning libraries offer all of these, along with many others.

If we're just exploring our data, then it makes sense to try each algorithm that seems like it could produce insight into our data's structure.

Let's look very briefly at the pros and cons of these four classifiers.

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 the algorithm can consume huge gulps of memory, because it's saving every training sample. If the training set is larger than the available memory, the operating system will typically need to start using virtual memory to save the samples, which can slow the algorithm down significantly.

Support vector machines are slower to train than kNNs, but they're much faster when predicting. Once trained, they don't need much memory, since they only store the boundaries. And they can use the kernel trick to find classification boundaries that are much more complicated than the straight lines (and flat planes, and higher-dimensional flat surfaces) that SVM natively produces. On the other hand, the training time grows with the size of the training set. And the quality of the results are very sensitive to the "clearance" 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, but given the slow training times for SVM, this can end up taking a long time.

Decision trees are fast to train, and they’re also fast when making predictions. They can handle weird boundaries between classes, though at the cost of a deeper tree. They have a huge downside due to their appetite for overfitting (though as we mentioned, we will address that later by using collections of small trees, so all is not lost). Decision trees have a huge appeal in practice because we can *explain* their results in ways that make sense. Other classification algorithms make decisions based on mathematical or algorithmic processes that may be hard to share with people who just want to understand the results. Decision trees have the advantage of being very clear: the result we got was because of this chain of decisions, which we can explicitly state. Sometimes people use decision trees, even when the results are inferior to other classifiers, because the results are easier to comprehend by humans.

Naïve 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 to tune, which means we don’t have to do extensive searching with cross-validation, as with SVM’s C parameters. If the classes we’re working with are well separated, then the naïve Bayes prior often produces good results. The algorithm works particularly well when our data has many features, because in high-dimensional spaces samples are farther apart, producing the kind of separation that plays to the strengths of naïve Bayes [VanderPlas16].

In practice, we often try naïve Bayes first, because it’s fast to train and predict and it 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.

We’ll see later that deep (that is, multi-layer) neural nets can also excel at classification. Nevertheless, the algorithms we’ve seen here are frequently used in everyday practice because they’re well understood, and we can usually comprehend (though sometimes with some effort) why they produce the results they do.

References

- [Bishop06] Christopher M. Bishop, “Pattern Recognition and Machine Learning”, Springer, 2006.
- [Raschka15] Sebastian Raschka, “Python Machine Learning”, Packt Publishing, 2015.
- [Steinwart08] Ingo Steinwart and Andreas Christmann, “Support Vector Machines”, Springer, 2008.
- [VanderPlas16] Jake VanderPlas, “Python Data Science Handbook”, O’Reilly, 2016.

Chapter 14

Ensembles

Individual learners are usually imperfect, and can sometimes give us incorrect results. But by combining many learners, the collection can out-perform its pieces.

Contents

14.1 Why This Chapter Is Here	541
14.2 Ensembles	542
14.3 Voting.....	543
14.4 Bagging.....	544
14.5 Random Forests.....	547
14.6 ExtraTrees	549
14.7 Boosting.....	549
References	561

14.1 Why This Chapter Is Here

Anyone can make mistakes, and that includes learning 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. In that case, we can run our data through multiple learners, each trained a little differently, and let them vote to determine a final answer.

This isn't a new idea. The Apollo spacecraft of the 1960's carried one computer in the command module, which orbited the moon, and one computer in the lunar module, which landed there. Neither space-craft had a backup computer. Each of these computers was built with integrated circuits, which were relatively new at the time. To protect against failures due to this new technology, every board was duplicated twice, producing three copies in all. All three systems always ran in synchrony, a technique called "Triple Modular Redundancy." The output of each module was decided by majority vote [Ceruzzi15].

In machine learning, groups of learners are called **ensembles**. Like the Apollo computers, the general idea is that if we get multiple predictions from independent algorithms, then the most popular prediction is likely to be the right one.

Ensembles sometimes can provide a boost in efficiency. In some situations, we can build an ensemble out of many small and fast learners, and get back results that are as good as those produced by big and complex learners, but in less time.

14.2 Ensembles

Making decisions is hard, whether the job is done by human or machine. In some human societies, we deal with individual imperfections in decision-making by aggregating the opinions of many people. Laws are passed by senates, political decisions are made by boards, and even individual leaders can be 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 by using the consensus of multiple, independent people.

While this doesn't guarantee good decisions, it does help avoid problems caused by any one person's idiosyncratic biases.

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, over-representations, or any other kind of systemic error, those errors will be baked into the learner as well. If we teach a learning algorithm with data that points in a certain direction, the results we get back will also point in that direction.

This can have profound implications in the real world. When we use machine learning to evaluate loans for homes or businesses, determine admissions to colleges, or pre-screen job applicants, any unfairness or bias in our training data will cause similar unfairness and bias in the decisions the system makes. Even benign prejudices will influence the results.

One way to avoid this problem is to train multiple learners using different training sets from different sources. Of course, any systematic bias in all the training sets will still influence the results, but if we strive to use diverse training sets that are really representative of the types of decisions we seek, we may be able to avoid accidentally biasing our outputs.

When we have trained a bunch of learners on these different datasets, we will usually ask each one to evaluate each new input. Then we let the learners **vote** to determine a final result.

This approach is popular among scientists who run complex models of natural phenomena, such as the weather, the climate, or the formation of the solar system. A key task when designing and running any simulation is to be aware of what assumptions are built into different analytical models, and how those assumptions can bias their results. By combining lots of simulations, each built on a different model with different assumptions, we hope that the biases will cancel each other out and the most correct result will emerge. This is why, for example, many climate forecasts are the result of running multiple, independent forecasting models (sometimes at different scales) and then aggregating their results [NRC98]. Weather forecasts often come from running the same model many times but with slightly different starting conditions, and selecting the most frequently-predicted outcomes. This latter technique is called **ensemble forecasting** [Met17].

14.3 Voting

When we have multiple results from multiple learners, we need to somehow come up with a single, final answer.

The typical way to do this is to use **plurality voting** [VotingTerms16]. Put simply, each learner casts a single vote for its prediction, and whatever prediction receives the most votes is the winner (ties are often broken by either randomly selecting one of the tied entries, or re-voting).

Plurality voting is easy to understand and implement, and it often does a good job of giving us a sensible result.

But plurality voting is not perfect. If we use it in a social context, it's even more problematic, and it can cause some surprising and unwelcome results.

But when working with multiple learners, plurality voting is by far the most popular approach for determining their output because it is simple, fast, and usually produces acceptable results.

In this chapter, we'll use **weighted plurality voting**. In this variation of plurality voting, every vote gets a certain **weight**, which is just a number. Suppose that all of our options begin with a score of 0, and someone casts a ballot for category A with a weight of +4, so A's score is now 4. We can also cast ballots with negative weights, so if someone else casts a ballot for A with a score of -1.5, A's score would then be 2.5. Whoever has the largest (that is, the most positive) total after adding up all the ballots is the winner.

14.4 Bagging

A popular type of ensemble is built entirely out of decision trees. These trees are fast for both training and predicting, so it's not a burden to use big collections of them. And decision trees are popular for both categorization and regression. To keep the following discussion specific, we'll focus on the categorization variety.

There are two popular techniques for bundling decision trees into larger groups. Each of these produces an ensemble that can significantly outperform the individual decision trees they're made of.

The core idea is to build a collection of decision trees that are trained with slightly different training sets. This means that they will be similar to each other, but not exact duplicates. When a new sample arrives for prediction, we give that sample to every tree in the ensemble. The predictions from all the trees are collected, and the most popular prediction is the output of the ensemble.

Our first ensemble construction technique that we'll look at is called **bagging**, which is a portmanteau of **bootstrap aggregating**. As we might guess, this uses the bootstrap idea we saw in Chapter 2. There, we saw how to use bootstrapping to evaluate the quality of some

statistical value computed by using multiple subsets of the population we're interested in. In this case, we'll create many small sets built from a training set, and use those to train our collection of decision trees.

Starting with our original training set of samples, we build our multiple new sets, or **bootstraps**, by picking just a few items from the original, using sampling with replacement. This means that it's possible we'll pick the same sample more than once. Figure 14.1 shows the idea. Remember that each sample comes with its labeled category, so we can train with it.

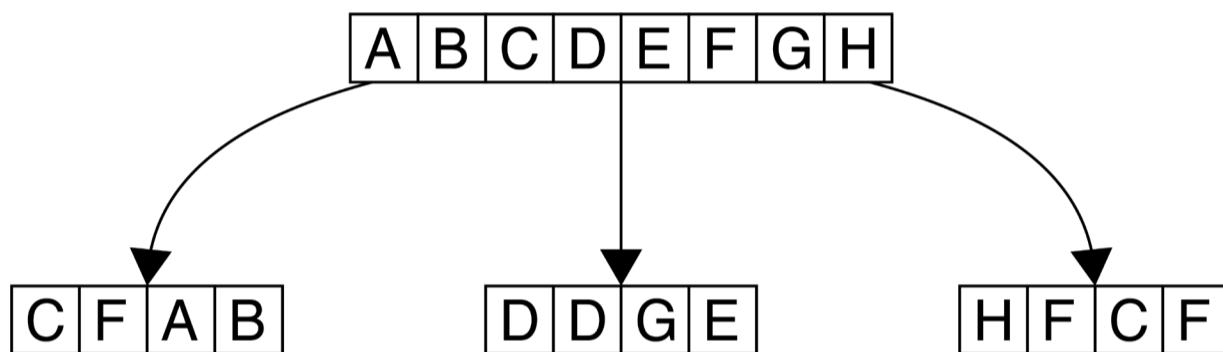


Figure 14.1: Creating bootstraps from a set of samples. At top, a set of 8 samples. The letters are to help us identify which samples are which, and are not the labels (we assume that each sample has an associated label, which we're not showing explicitly here). By extracting samples from this set, we can make many new sets of 4 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.

We create one decision tree for each bootstrap, and we train the tree on the samples in the bootstrap. Taking those trees together gives us our ensemble.

When training is done and we're evaluating a new sample, we give it to all the decision trees in the ensemble, producing one category prediction from each tree.

For regression problems, the final result is the average of all the predictions. For classification problems, we treat the predicted classes as votes in a plurality election, producing either a winner or a tie. Figure 14.2 shows the process graphically.

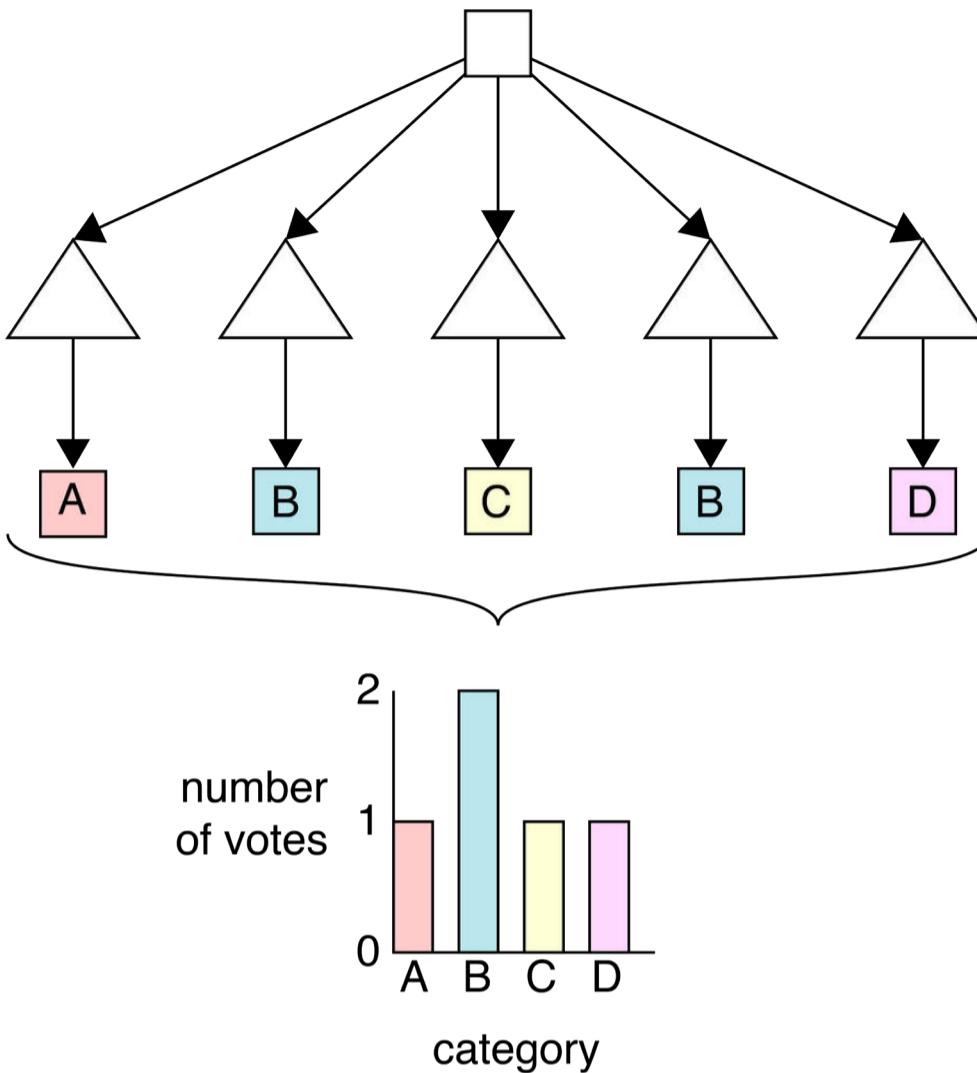


Figure 14.2: A new piece of data (top) arrives at our ensemble. Because it was trained by bagging, each decision tree is a little different. The data is given to each of our decision trees (shown as triangles), which each produce a predicted category (A, B, C, or D). We run a plurality-voting election with those categories (fourth row), where the most popular category is B, so that category wins and is the output of the ensemble.

We only need to specify two parameters to run this algorithm. The first is to choose how many samples should be used in each bootstrap.

Second, we need to choose how many trees we want to build. Analysis has shown that adding more classifiers makes a system a better predictor, but after some point adding yet more classifiers just makes it slower and not much better. 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 there are categories of data [Bonab16], though we can use cross-validation to search for the best number of trees for any given dataset.

14.5 Random Forests

We can improve on bagging and create even better ensembles of decision trees. The trick is to change how we build our trees during training.

As we saw in Chapter 13, 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 stick to splitting on just one feature, the problem becomes how to choose which feature we want to use, and what value of that feature to test for. We can compare different choices of features, and different splitting points, using the measurements we saw in Chapter 13, such as information gain or the Gini impurity.

Let's look at this process from the same point of view that led us to use bootstrapping and bagging above. This will give us yet another level of variation from one decision tree to the next, helping us get a more diverse range of opinions when it comes to voting on the best category to assign to a new sample.

At every node, when it's time to find a splitting test, we normally look at all of the features to decide which one is the best one to split on. But instead of looking at all of the features, let's instead consider just some random subset of them that we'll choose without replacement. In other words, we select some of the features, and find the best one from among that list. We don't even consider splits based on the features we're ignoring.

Later, when we come to split another node, we again choose a brand-new subset of features, and again determine our new split using only those. This technique is called **feature bagging**. The idea is shown in Figure 14.3.

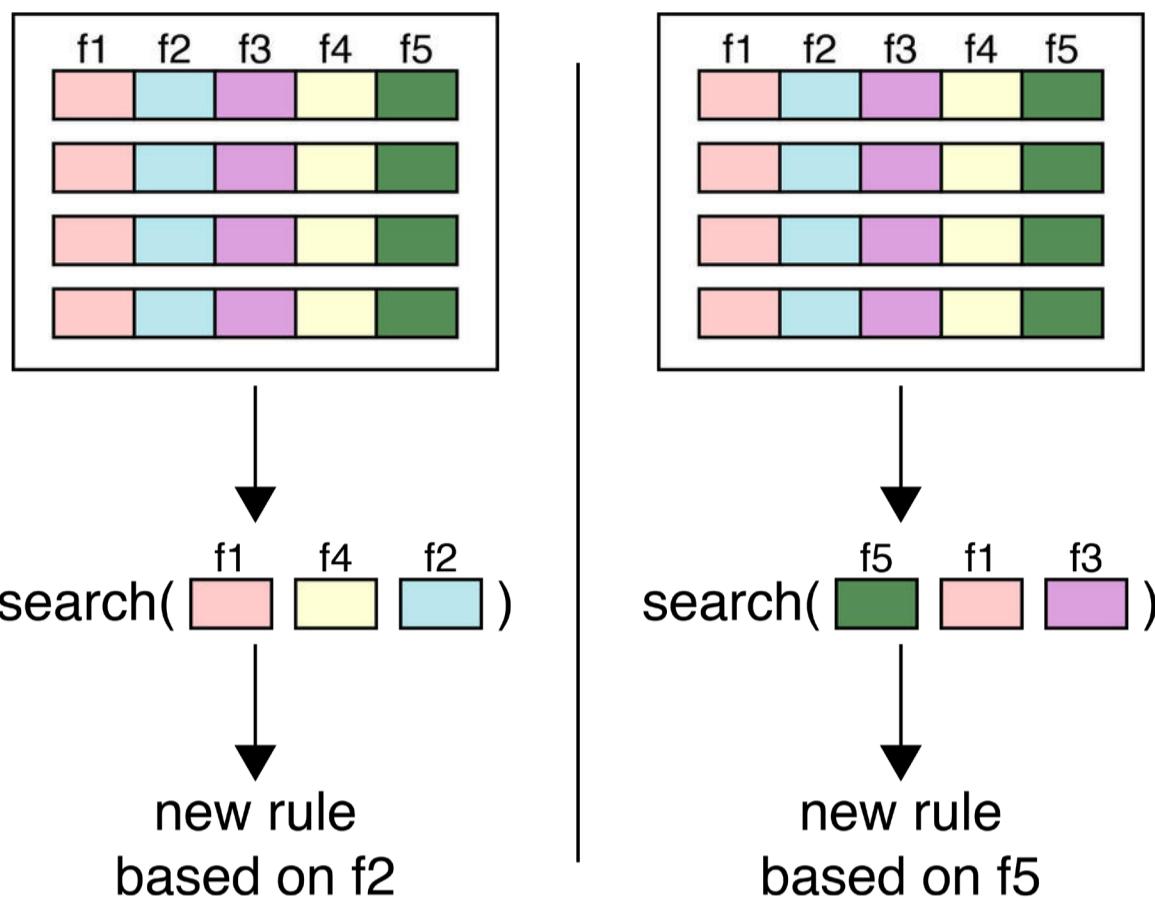


Figure 14.3: Determining which feature to use when splitting a node by feature bagging. Left: At top is a set of 4 samples, each with 5 features. The middle row shows 3 features chosen at random. We search for the best of these three features to determine how to split this node. The best choice is feature 2. Right: The same process at another node. Here we randomly select another set of 3 features, and search those for the best feature to split on.

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 the random selections of samples we extract for each bootstrap, and the number of trees to build. We also have to specify what fraction of the features to consider at each node. This is often a percentage, but many libraries offer a variety of mathematical formulas based on the number of features.

14.6 ExtraTrees

We can add yet one more randomization step to our process of making ensembles of decision trees.

When we normally 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 evaluate what's "best" using a measure like information gain.

But 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 **ExtraTrees**.

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 bit of accuracy in order to reduce overfitting.

14.7 Boosting

Let's now look at a technique called **boosting** that is applicable not only to decision trees, but to any kind of learning system [Schapire12]. We'll discuss boosting in terms of binary classifiers, or systems that identify their inputs as belonging to one of two classes.

Boosting lets us combine a large number of small, fast, and inaccurate learners into a single accurate learner.

Let's suppose our data contains samples of just two categories. A completely random binary categorizer would assign one of these two categories arbitrarily, so if the samples are evenly split in the training

set, we'd have a 50-50 chance of any sample being correctly labeled. We call this **random labeling**, or that the odds of it getting the right answer are simply up to **chance**.

What if a binary classifier does just a tiny bit better than chance? Maybe it has a lousy decision boundary. Figure 14.4 shows a set of data of two categories, a binary classifier that is no better than chance, and a binary classifier that is just a tiny bit better than chance.

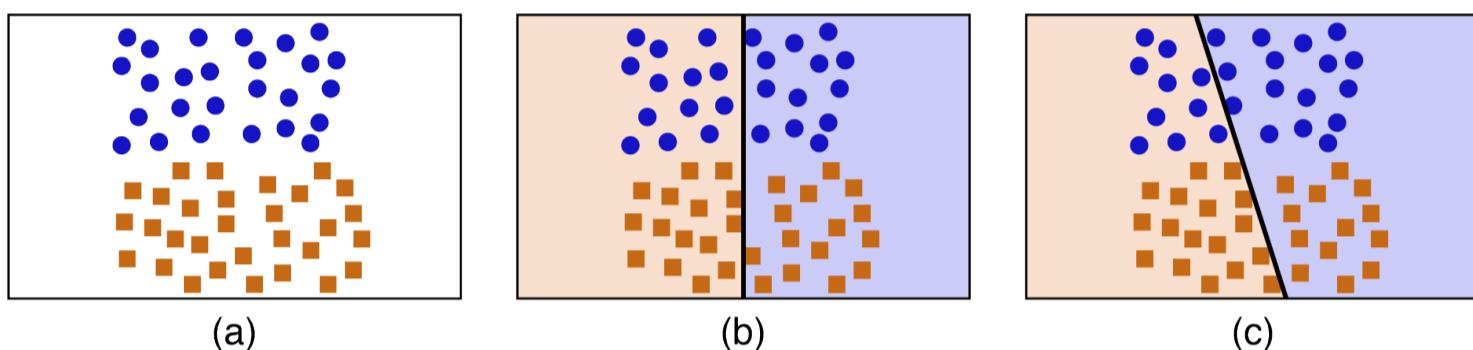


Figure 14.4: Some bad binary classifiers. (a) Our training data. The best split would be a horizontal line across the middle of the box. (b) A terrible binary classifier. This is no better than chance. (c) A binary classifier that is only a little bit better than chance. It is a terrible classifier, but thanks to the small tilt in the boundary line, it is slightly less terrible than the classifier in part (b).

The learner in Figure 14.4(b) is no better than chance, with half of each class getting incorrectly classified. This would be a useless classifier. The learner in Figure 14.4(c) is just slightly less useless than the classifier in part (b) because the slight tilt in the boundary line means it will by work assign the categories just a little bit better than random.

We call the classifier in Figure 14.4(c) a **weak learner**. A weak learner in this situation is any classifier that is even the slightest bit accurate. Literally, it only needs to be a little bit better than chance, or assigning the categories randomly.

A weak learner is just as useful to us even if it does *worse* than chance. That's because we have only two categories. So if a classifier is below chance (that is, it assigns the wrong category more frequently than the right one), then we can just swap the output classes, and now it's doing better than chance, rather than worse.

In contrast to weak learners, a **strong learner** is a good classifier that gets the correct label most of the time. The stronger the learner, the better its percentage of being right.

Weak classifiers are easy to find.

Perhaps the most commonly-used weak classifier is a decision tree that is 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**. But 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.

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 weak ones are common because they're usually faster.

Let's see how boosting works with an example.

Figure 14.5 shows a training set of samples that belong to two different classes.

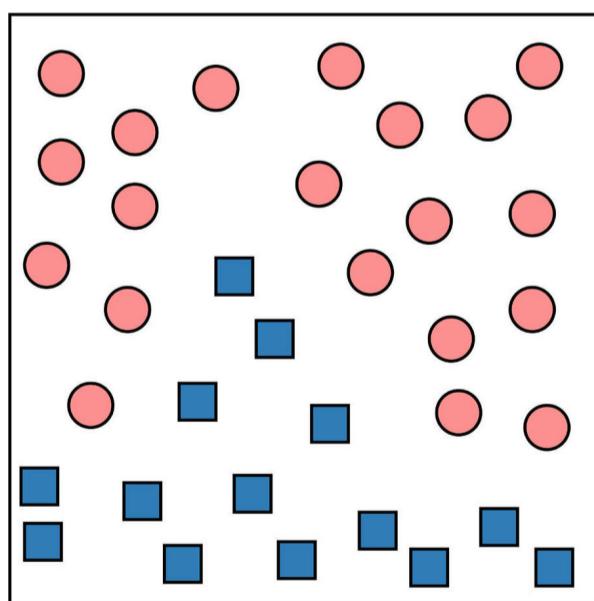
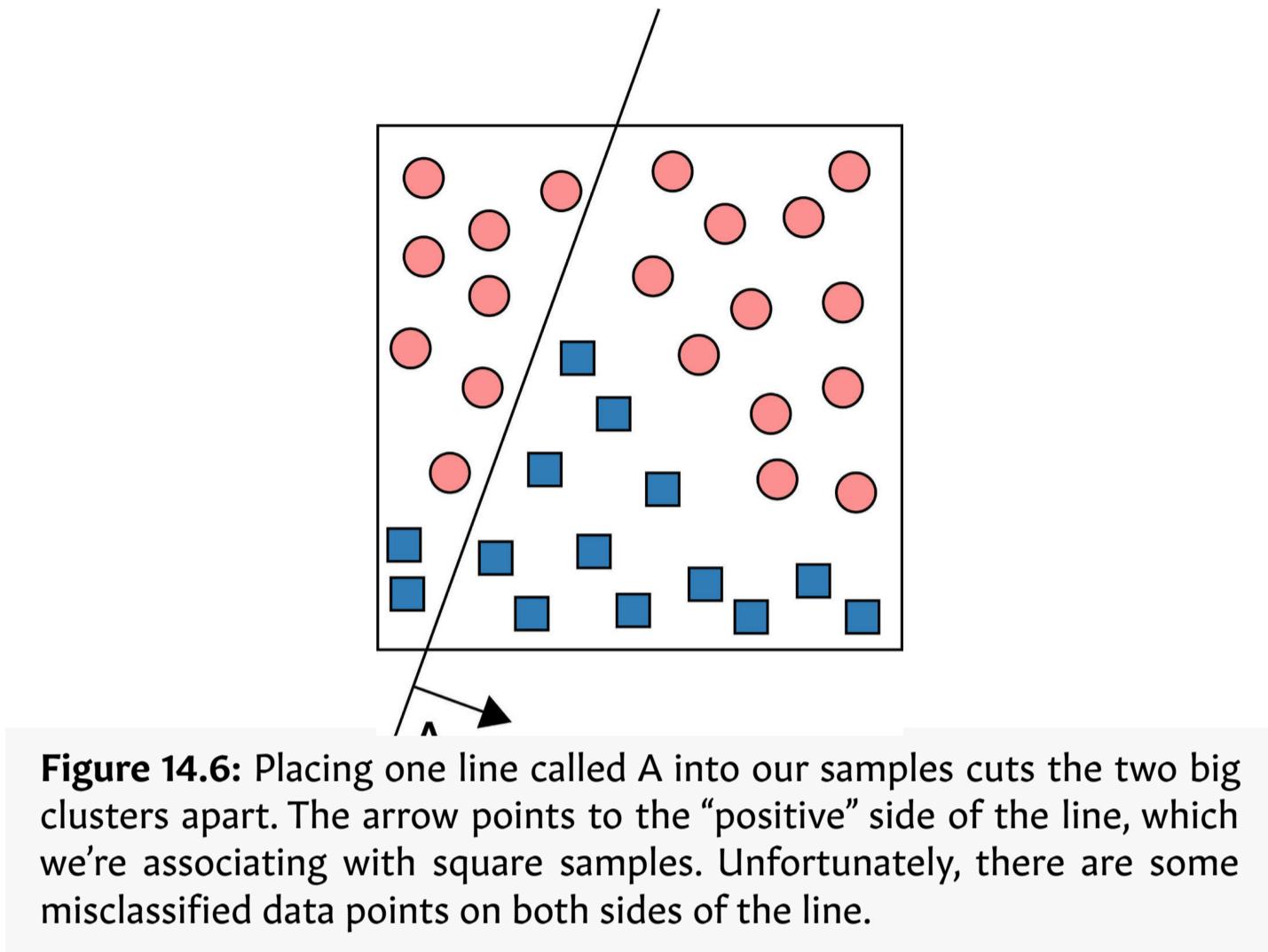


Figure 14.5: A collection of samples we're going to classify using boosting.

What might be some good classifiers for this data? A fast and easy to understand classifier is a single perceptron, which we discussed in Chapter 10. This would draw a straight line through this 2D dataset. A set of 2D samples is called **linearly separable** if we can draw a straight line between them. We can see that no straight line is going to split up this data, because the round regions surround the square region in the middle on both sides.

Even though no single straight line can separate this data, multiple straight lines can, so let's use straight lines as our weak classifiers. Remember, these classifiers don't have to do a great job on the whole dataset. They only have to do a bit better than chance. Figure 14.6 shows one such line, which we'll call A. Everything on the side of A pointed to by the arrow will be classified as square, and everything on the other side will be classified as round. This line does a good job of splitting the round group in the upper left from the rest of the data, line pointed to by the arrow) but it misclassifies the small cluster of square samples in the lower left.



To use boosting, we’ll want to add more lines (that is, additional weak learners) so that every region formed by the lines contains samples of a single color. After adding two more of these straight-line classifiers, we might end up with Figure 14.7.

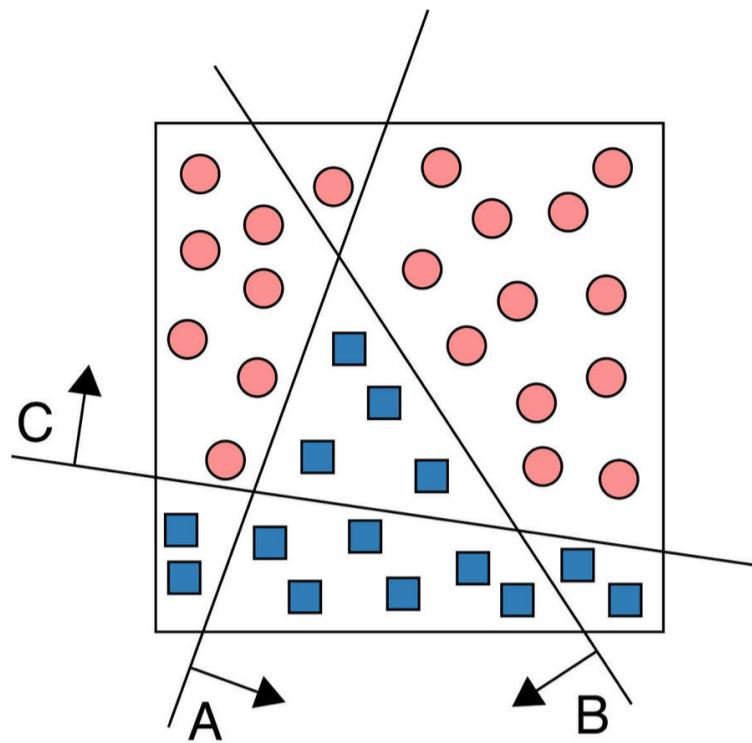


Figure 14.7: Two more lines added to Figure 14.6. Now each region formed by these three lines is made up of samples of only one class.

Although Figure 14.7 has only three lines, or separators, they create seven non-overlapping regions. Three of these regions are inhabited exclusively by round samples, and the other four by square samples.

Each of the lines A, B, and C is a weak learner, returning results that are far from perfect. Now we'll combine them to do a far better job.

To make the discussion easier, we'll name the three learners after their associated line: A, B, and C. Each of the seven regions they create is on the positive side of one or more lines. We'll also simplify the diagram to make it symmetrical, because our focus now is on how we combine regions, and not their particular shapes. The result is shown in Figure 14.8.

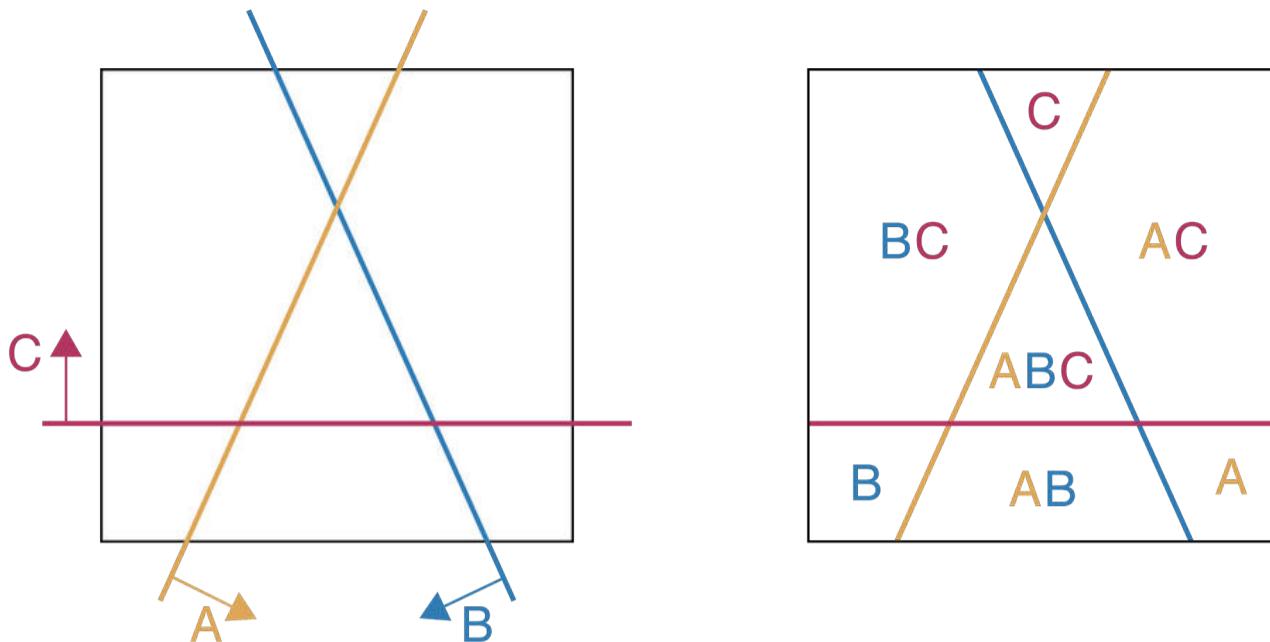


Figure 14.8: On the left, we show three lines named A, B, and C. On the right, each region is marked with the names of the learners that put that region on the positive side of their respective lines.

Rather than have each learner report a category (round or square) we'll set it up to report a value associated with that learner. So a sample is on the positive side of a learner's line, it reports its value, otherwise it reports 0.

What values should we use? To demonstrate how the process works, we'll assign +1 to all three learners. We'll see later that these values will be determined automatically for us by the algorithm.

So with that convention, if a sample is on the positive side of a learner's line (that is, the side pointed to by the arrow in Figure 14.8), then that learner will return a value of +1. Otherwise, it returns 0.

Let's take the region at the top, marked C in Figure 14.8. It's on the positive side of learner C, earning it a score of 1. But it's on the negative side of both A and B, each of which thus contribute 0, for a total of 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 14.9

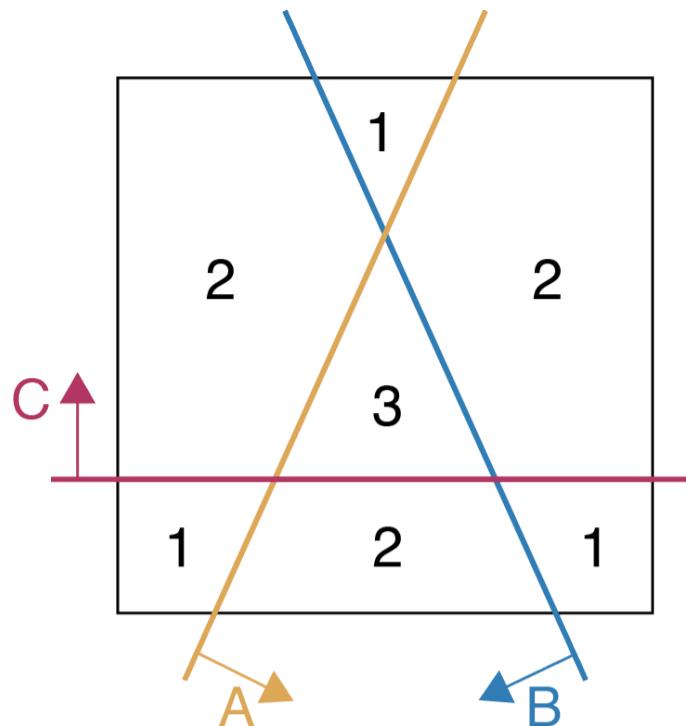


Figure 14.9: The composite score for each of the seven regions. Each letter in Figure 14.8 earns that region 1.

To turn this information into a working classifier, we need to find the right weights for each region, and the right threshold for assigning categories. Earlier we assigned values of 1 to each learner, so those were their weights in the voting, but many other values could work as well. In Figure 14.10 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'll use the weights (1.0, 1.5, -2) for A, B, and C respectively.

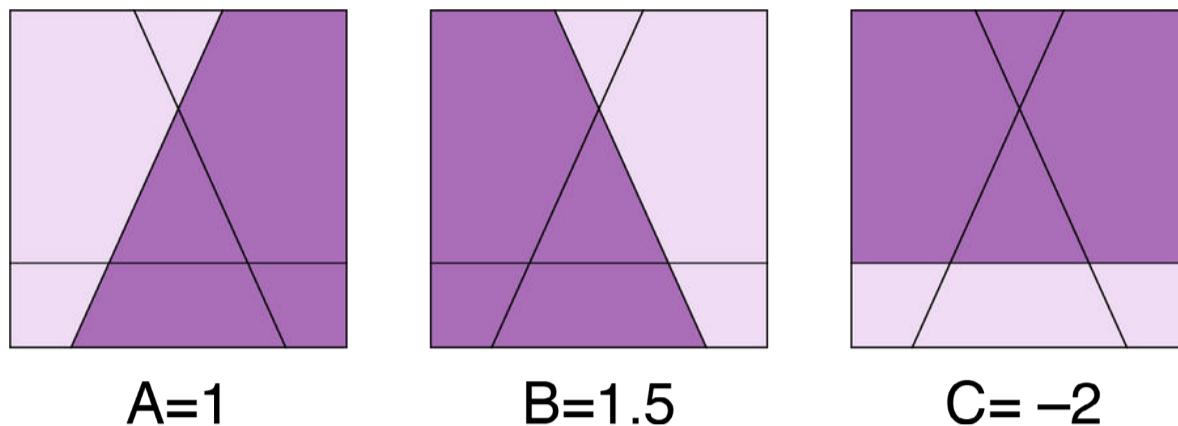


Figure 14.10: We assign a numerical value to each region that is classified as positive by each learner. The dark regions for each line get the weight associated with that line.

The sums of all of these scores are shown in Figure 14.11. We've correctly classified our data!

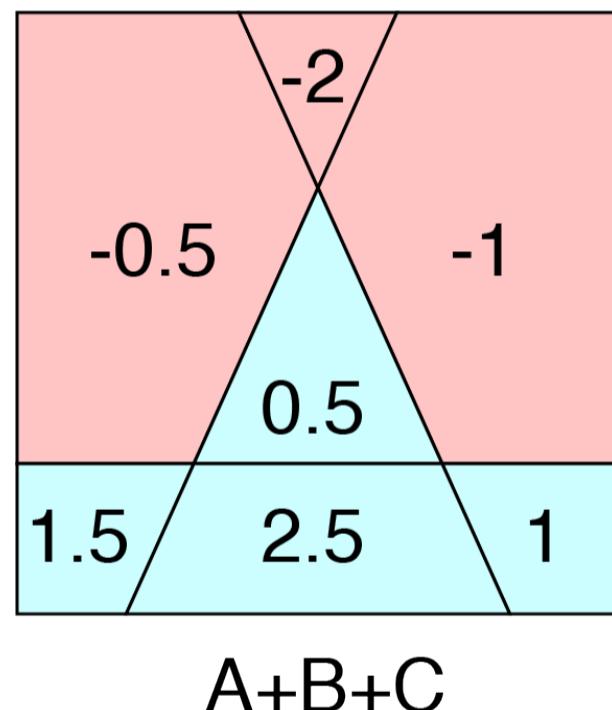


Figure 14.11: Adding up the scores from Figure 14.10. The blue cells are the only ones with positive values. These are the ones that we want from Figure 14.7.

Let's look at another example. Figure 14.12 shows a new set of data.

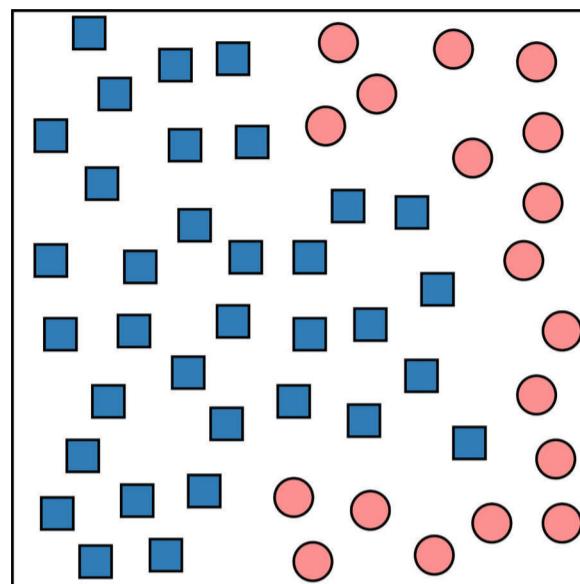


Figure 14.12: A set of data we'd like to classify using boosting.

For this data, we'll try using four learners. Figure 14.13 shows the four weak learners that the boosting algorithm might find in order to partition this data.

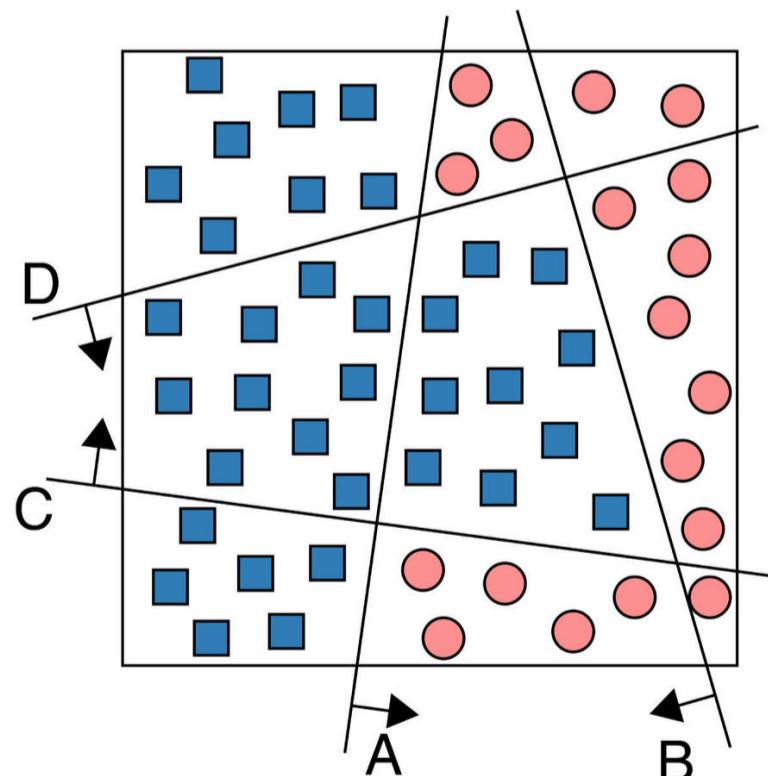


Figure 14.13: Four lines that let us classify the data of Figure 14.12.

As before, we'll manually assign weights to these learners. This time we'll use -8 , 2 , 3 , and 4 for learners A, B, C, and D respectively. Figure 14.14 shows which regions will have those weights added to their overall score. Light colored regions implicitly receive a value of 0 . We've again simplified the geometry of our diagram, to keep our attention on how we work with and combine the regions, and not their particular shapes.

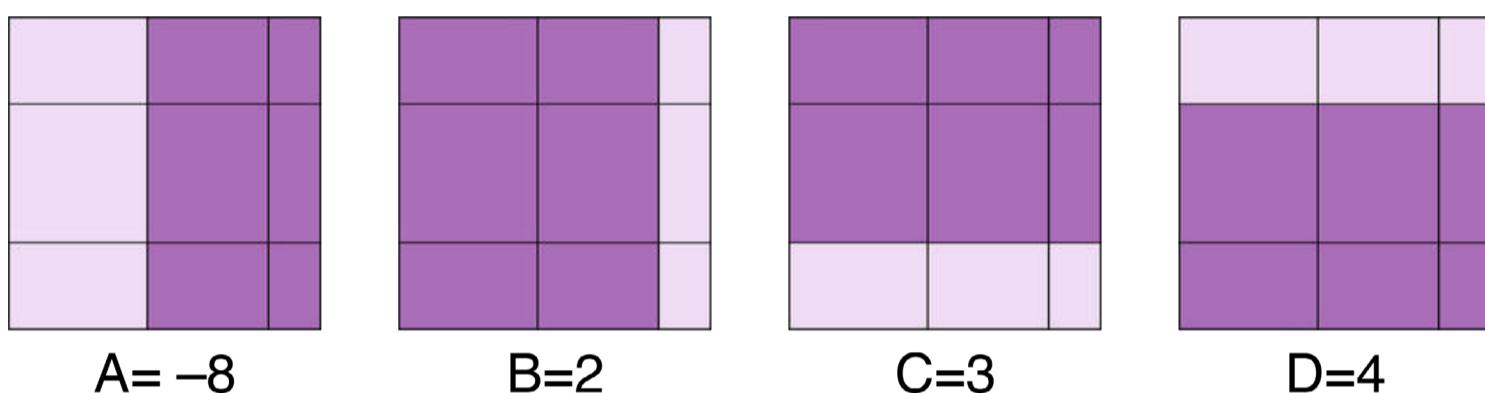


Figure 14.14: The regions corresponding to each learner.

Figure 14.15 shows the sums of the contributions for each region. We've found a way to combine the four learners to create a region that correctly classifies the points in Figure 14.13.

5	-3	-5
9	1	-1
6	-2	-4

$A + B + C + D$

Figure 14.15: The sums of the scores of each region from Figure 14.14. Positive regions are shown in blue, and they correctly classify the points in Figure 14.13.

The beauty of boosting is that it combines classifiers that are simple and fast, but lousy, into a single great classifier. It even figures out which simple classifiers to use, and what weights they should have when voting.

The only thing we have to pick is how many classifiers we want. In boosting, as in the other ensemble algorithms we've looked at, a rule of thumb is to start with about as many classifiers as there are categories available for them to determine [Bonab16]. That means that our examples above started on the high side, since we used 3 or 4 classifiers for only 2 categories. But as in so many things in machine learning, the best value is found by trial and error, often using cross-validation to evaluate each possibility.

Boosting made its first appearance as part of an algorithm called **Adaboost** [Freund97] [Schapire13]. 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 before (these are trees that have only a root node and immediate children). Decision stumps are lousy classifiers, but they're better than chance, which is all we need. In fact, the lines we've been using in our examples could be represented as decision stumps, so we've already seen that they can adapt to different training sets.

Each weak learner produced by the boosting algorithm is also frequently called a **hypothesis**. The idea is that the weak learner embodies a conjecture, or hypothesis, such as “all samples in category A are on the positive side of this line.” We sometimes say that boosting helps us turn weak hypotheses into strong ones.

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 (like our examples above) [Fumerao8] [Kak16]. This is partly why it is extremely popular and successful with decision tree classifiers.

References

- [Bonab16] R. Bonab, Hamed; Can, Fazli (2016). A Theoretical Framework on the Ideal Number of Classifiers for Online Ensembles in Data Streams. Conference on Information and Knowledge Management (CIKM), 2016.
- [Ceruzzi15] Paul Ceruzzi, “Apollo Guidance Computer and the First Silicon Chips,” Smithsonian National Air and Space Museum, Space History Department, 2015. <https://airandspace.si.edu/stories/editorial/apollo-guidance-computer-and-first-silicon-chips>
- [Freund97] Y. Freund and R.E. Schapire, “A Decision-theoretic Generalization of On-line Learning and an Application to Boosting”, Journal of Computer and System Sciences, 55 (1), pp. 119–139. 1997.
- [Fumerao08] Giorgio Fumera, Roli Fabio, Serrau Alessandra, “A Theoretical Analysis of Bagging as a Linear Combination of Classifiers”, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 30, Number 7, pp. 1293-9. 2008.
- [Kak16] Avinash Kak, “Decision Trees: How to Construct Them and How to Use Them for Classifying New Data”, Purdue University, 2016. <https://engineering.purdue.edu/kak/Tutorials/DecisionTreeClassifiers.pdf>
- [Met17] United Kingdom Met Office, “What is an ensemble forecast?”, Met Office Research, Weather science blog, 2017. <https://www.metoffice.gov.uk/research/weather/ensemble-forecasting/what-is-an-ensemble-forecast>
- [NRC98] National Research Council, “The Atmospheric Sciences: Entering the Twenty-First Century”, The National Academies Press, 1998. <https://www.nap.edu/read/6021/chapter/9> (page 178)

[Schapire12] Robert E. Schapire, Yoav Freund, “Boosting Foundations and Algorithms”, MIT Press, 2012.

[Schapire13] Robert E. Schapire, “Explaining Adaboost”, in “Empirical Inference: Festschrift in Honor of Vladimir N. Vapnik”, Springer-Verlag, 2013. <http://rob.schapire.net/papers/explaining-adaboost.pdf>

[VotingTerms16] RangeVoting.org, “Glossary of Voting-Related Terms”, 2016. <http://rangevoting.org/Glossary.html>



Chapter 15

Scikit-learn

The scikit-learn library is a free, open-source project that provides tools and algorithms for machine learning. We'll survey some of its most important and useful offerings.

Contents

15.1 Why This Chapter Is Here	566
15.2 Introduction.....	567
15.3 Python Conventions.....	569
15.4 Estimators	574
15.4.1 Creation	575
15.4.2 Learning with fit().....	576
15.4.3 Predicting with predict()	578
15.4.4 decision_function(), predict_proba()	581
15.5 Clustering	582
15.6 Transformations	587
15.6.1 Inverse Transformations.....	594
15.7 Data Refinement	598
15.8 Ensembles	601
15.9 Automation	605
15.9.1 Cross-validation.....	606
15.9.2 Hyperparameter Searching	610
15.9.3 Exhaustive Grid Search	614
15.9.4 Random Grid Search	625
15.9.5 Pipelines.....	626
15.9.6 The Decision Boundary.....	641
15.9.7 Pipelined Transformations	643

15.10 Datasets	647
15.11 Utilities.....	650
15.12 Wrapping Up.....	652
References	653

15.1 Why This Chapter Is Here

Machine learning is a practical field. Although this book is focused on concepts, there's nothing quite like putting something into practice to transform ideas into understanding.

Implementing an idea makes us face decisions that we might have otherwise brushed off as easy or unimportant, and can reveal misunderstandings and holes in our knowledge. In a field like machine learning, where so much is still an art, and we make decisions based on intuition and what we've learned from previous mistakes and successes, experience is invaluable.

How do we gain that experience? A time-honored approach is to write our own code from scratch, and then use that code to build and run machine learning systems. Writing our own code is interesting, educational, and it can be rewarding. But it's also a lot of work. And when it comes to big algorithms, it can be a mountain of work, much of which is about the art of writing good programs, and not about machine learning at all.

Alternatively, we can use routines in an existing library. A wide variety of libraries are readily available, for free, in diverse languages and systems. These libraries have often been designed by experts, stress-tested by thousands of users, and tuned, debugged, and optimized over many years.

If writing low-level code from scratch is appealing to you, there are many guides to that path [Müller-Guido16] [Raschka15] [VanderPlas16]. In this book, we take the path that uses existing libraries. We still have to confront a wide range of decisions, but we sidestep the low-level programming details that, while essential, can also be distracting.

In this chapter we'll put our ideas into practice using a popular Python-based toolkit for machine learning called **scikit-learn** (pronounced “sy'-kit-lern”).

We’re using scikit-learn because it’s widely used, well supported and documented, stable, open-source, and fast.

To keep the subject matter of this book useful to people who aren’t keen on Python, we’ll restrict our use of scikit-learn to this chapter, and Chapters 23 and 24 where we’ll use it to build deep learning systems along with another free library. Nothing else in this book depends on familiarity with this library or Python.

Even if you have no interest in scikit-learn, we suggest that at least skimming this chapter will help you cement some of the ideas we’ve been discussing.

We’ll be using scikit-learn version 0.19, released in August of 2017 [scikit-learn17e]. Versions released after that are likely to be compatible with the code we’ll be using. Downloading and installing the library is usually easy on most systems. See the download instructions on the library’s home page for detailed instructions [scikit-learn17f].

A pleasant way to use Python is to work interactively with a Python interpreter. The free Jupyter system provides a simple and flexible interface to interpreted Python that runs inside of any major browser [Jupyter17]. Jupyter notebooks containing running versions of all the programs in this chapter are available for free download on GitHub at <https://github.com/blueberrymusic>.

15.2 Introduction

The Python library **scikit-learn** is a free library of common machine-learning algorithms.

To use it well requires some familiarity with the Python language and the NumPy library. NumPy (pronounced “num’-pie”) is a Python library focused on manipulating and calculating with numerical data. This nesting of dependencies is shown in Figure 15.1.

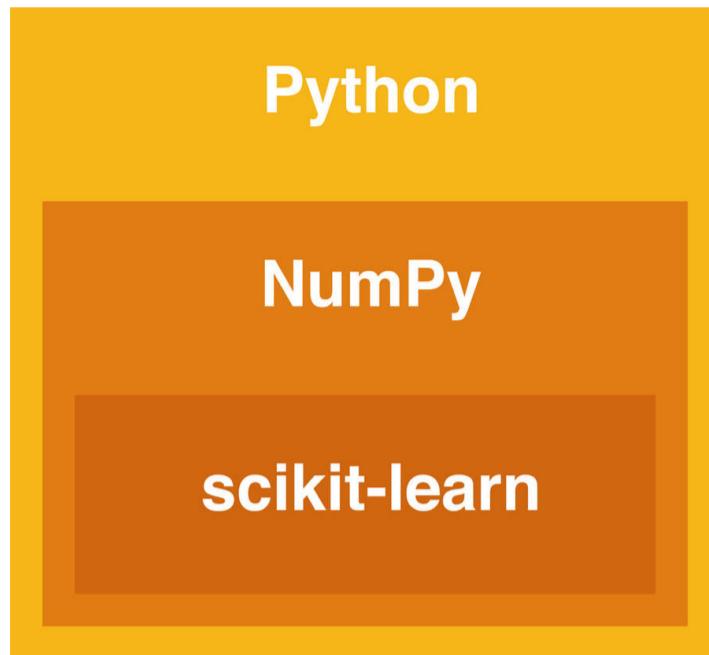


Figure 15.1: The scikit-learn library uses routines from the numerical calculation library NumPy, which in turn depends upon Python.

There are several other scientific “kits” in Python, each called “scikit-something.” For example, scikit-image is a popular system for image processing. These kits are free and open-source projects produced and maintained by serious developers who largely pursue these efforts on their free time as a public service. Another widely-used library is named SciPy (pronounced “sie’-pie”), and it provides a wealth of routines for scientific and engineering work.

Getting deep into Python, NumPy, or scikit-learn, is not an afternoon project. Each of these entities is big, and has a lot to offer. On the other hand, a weekend of work on any of them can provide a solid start.

If you’re new to any of these topics, the References section offers some resources for getting started. If you ever see a term or routine that seems unfamiliar, fountains of documentation on all of these topics are just an internet search away. In this chapter, our focus is on scikit-learn, so we’ll assume that you are familiar enough with Python and NumPy to at least follow along.

Even though this chapter is focused on scikit-learn, we won’t go deep into the library. Like any substantial piece of software, scikit-learn has its own conventions, defaults, assumptions, limitations, shortcuts,

best practices, common pitfalls, easily-committed mistakes, and so on. And each routine within the library has its own specific details that are important to know in order to get the best results. To give these topics proper attention would require a book of its own, and indeed such books exist [Müller-Guido16].

Our goal in this chapter is to give a sense of the library and how it's used. For details on any routine or technique, consult the online documentation, reference books, or even some of the helpful web pages and blog posts online.

15.3 Python Conventions

Scikit-learn is *big*. The library offers around 400 routines, which range from small utilities to large machine-learning algorithms bundled up into a single library call. These routines are extensively documented on the library's website [scikit-learn17b]. As of this writing, the current release (version 0.19) groups the library's routines into a whopping 35 categories.

Since we're taking a very general overview here, we'll invent our own smaller set of categories to organize our discussion: **Estimators**, **Clustering**, **Transformations**, **Data Refinement**, **Ensembles**, **Automation**, **Datasets**, and **Utilities**.

Many elements of scikit-learn are arranged like interlocking pieces, and some are designed to be used in particular combinations. We'll see some examples of those combinations later when we demonstrate specific algorithms.

Scikit-learn is not directed to neural networks, which are the hallmark of deep learning. Rather, these tools are intended for both standalone data science, and as utilities for neural networks, helping us explore, understand, and modify our data. Many deep learning projects begin with a thorough exploration of the data carried out with scikit-learn.

This chapter inevitably has code listings, but we have tried to keep these small and focused. Full code listings are great resources for learning the details of a system, and how to manage issues like structuring data and calling the right operations in the proper sequence. On the other hand, long blocks of code are boring to read. And real projects usually spend considerable time and effort on little details that are particular to that specific program, and don't aid our understanding of the general principles.

So while we will build real systems to do real work, we'll keep these project-specific steps to a minimum. We'll typically build up our projects one step at a time. So we'll show and discuss each step, but then consider it part of the program and not repeat that code every time as the program grows. In a few instances, we will bring all the code together in one big listing at the end, but usually we leave that complete summary to the Jupyter notebooks that accompany the chapter.

That means that most of the code examples in this chapter are just fragments, and not complete programs. The general thinking behind this approach is that one can always build up a little test program around the fragment to explore what it does in detail.

Whenever we use scikit-learn, we must **import** it first, since Python doesn't load it automatically. When we import scikit-learn, it goes by the shorthand name **sklearn**. This name is just the top-level name for the library. The routines themselves are organized into a set of **modules**, which we also need to import.

There are two popular ways to import a module. The first is to import the whole module. Then we can name anything in that module in our code by prefixing it with the module's name, followed by a period. Listing 15.1 shows this approach, where we create an object named `Ridge`, which comes from the scikit-learn module named `linear_model`.

```
from sklearn import linear_model

ridge_estimator = linear_model.Ridge()
```

Listing 15.1: We can import the whole module `linear_model` and then use it as a prefix to name the Ridge object. Here we make a Ridge object with no arguments and save it in the variable `ridge_estimator`.

The other approach is to import only what we need from the module, and then we can use that object without referring to its module in the code, as in Listing 15.2.

```
from sklearn.linear_model import Ridge

ridge_estimator = Ridge()
```

Listing 15.2: We can import just the Ridge object from `linear_model`, and then we don't need to name the module when we use the object. The result of this code is identical to the result of Listing 15.1.

Generally speaking, it's usually easier during development to import the whole module, since then we can try out different objects from that module without constantly finding and changing the `import` statement. When we're all done, we often go back and clean things up to import only what we're actually using, since that's considered a bit cleaner. In practice, both importing the whole module and importing just specific pieces are common, and often even mixed in the same file.

We will be using NumPy a lot, so we'll frequently import it as well. Conventionally, when we import NumPy we use the abbreviation `np`. This is just a widely-used convention, and not a rule.

We also frequently include the `math` module to take advantage of the wide variety of little math routines it offers. The convention is not to rename this library, so we refer to it in our code as `math`.

Another common library is the `matplotlib` graphic library. This too has a conventional name. The module we usually use from this library is called `pyplot`, and the tradition is to call this the word `plt`, or “plot” without the o.

Finally, the Seaborn library offers some additional graphics routines, and also modifies the visuals produced by `matplotlib` to look more attractive [Waskom17]. When we import Seaborn, it is conventional to call it `sns`. In this chapter our only use of Seaborn will be to import it and tell it to replace `matplotlib`’s default aesthetics with its own. We do this by calling `sns.set()`, and we usually place that call on the same line as the `import` statement, separated by a semicolon.

Listing 15.3 shows typical `import` statements for these other libraries.

```
import numpy as np
import math
import matplotlib.pyplot as plt
import seaborn as sns ; sns.set()
```

Listing 15.3: This set of `import` statements will be our starting point in almost every project.

To find the right module to import for each scikit-learn routine we want to use, we can refer to scikit-learn’s online API Reference [scikit-learn17a]. That reference also contains a complete breakdown of everything in scikit-learn, though often in very terse language. The API Reference is great when we treat it as just that, a reference when we want to remember what something’s called or how to use it. But because of its brevity, it’s less useful for explanations. That information is best found from one of the books or sites listed in the References section.

The people who manage scikit-learn have very high standards for including new routines, so additions are rare. The library is updated when there are important bug fixes and other improvements, and it’s occasionally re-organized. As a result, some routines become marked as “deprecated”, which means that they are planned for removal, but

are left in for a while so people can update their code. Deprecated routines are often subsumed into more general library features, or simply re-organized into a different module.

Many of the routines in scikit-learn are made available to us through **objects**, in the object-oriented sense. For example, if we want to analyze some data in a particular way, we usually create an instance of an object designed for that kind of analysis. Then we get things done by calling that object's methods, instructing the object to do things like analyze a set of data, calculate statistics on the data, process it in some way, and predict values for new data.

Thanks to Python's built-in garbage collection, we don't have to worry about recycling or disposing of our objects when we're through with them. Python will automatically reclaim memory for us when it's safe to do so.

Using objects in this way works very well in practice, since it lets us keep our focus on what we want to do, and less on the mechanics.

A good way to learn a huge library like scikit-learn is to casually look up each routine or object the first time we see it used, often while typing in code from a reference, or studying someone else's program. A quick look at the options and defaults can help flesh out our sense of what the routine is able to do. Later, while working on another project, something may ring a bell, or otherwise draw us back to this routine or object.

Then we can look at the documentation again and read it more closely. In this way we can gradually learn a bit more about the breadth and depth of each feature as we find ourselves using it, which is much easier than trying to memorize everything about every object and routine the first time we encounter it.

Almost all object and routines in `sklearn` accept multiple arguments. Many of these are optional and take on carefully-chosen default values if we don't refer to them. For simplicity, in this chapter we'll usually pass only the mandatory arguments, and leave all the optional arguments at their defaults.

15.4 Estimators

The code in this section is in the Jupyter notebook
`Scikit-Learn-Notebook-1-Estimators.ipynb`

We use scikit-learn's **estimators** to carry out supervised learning. We create an estimator object, and then give it our labeled training data to learn from. We can later give our object a new sample it hasn't seen before, and it will do its best to predict the correct label.

All estimators have a core set of common routines and usage patterns.

As we mentioned earlier, each estimator is an individual *object* in the object-oriented programming sense. The object knows how to accept data, learn from it, and then describe new data it hasn't seen before, maintaining everything it needs to remember in its own instance variables. So the general process is to first *create* the object, and then send it *messages* (that is, call its built-in routines, or methods) causing it to take actions. Some of those actions only modify the internal state of the estimator object, while others compute results that are returned to us.

Figure 15.2 shows what we'll be aiming for in this section. We'll start with a bunch of points, and use a scikit-learn routine to find the best straight line through them.

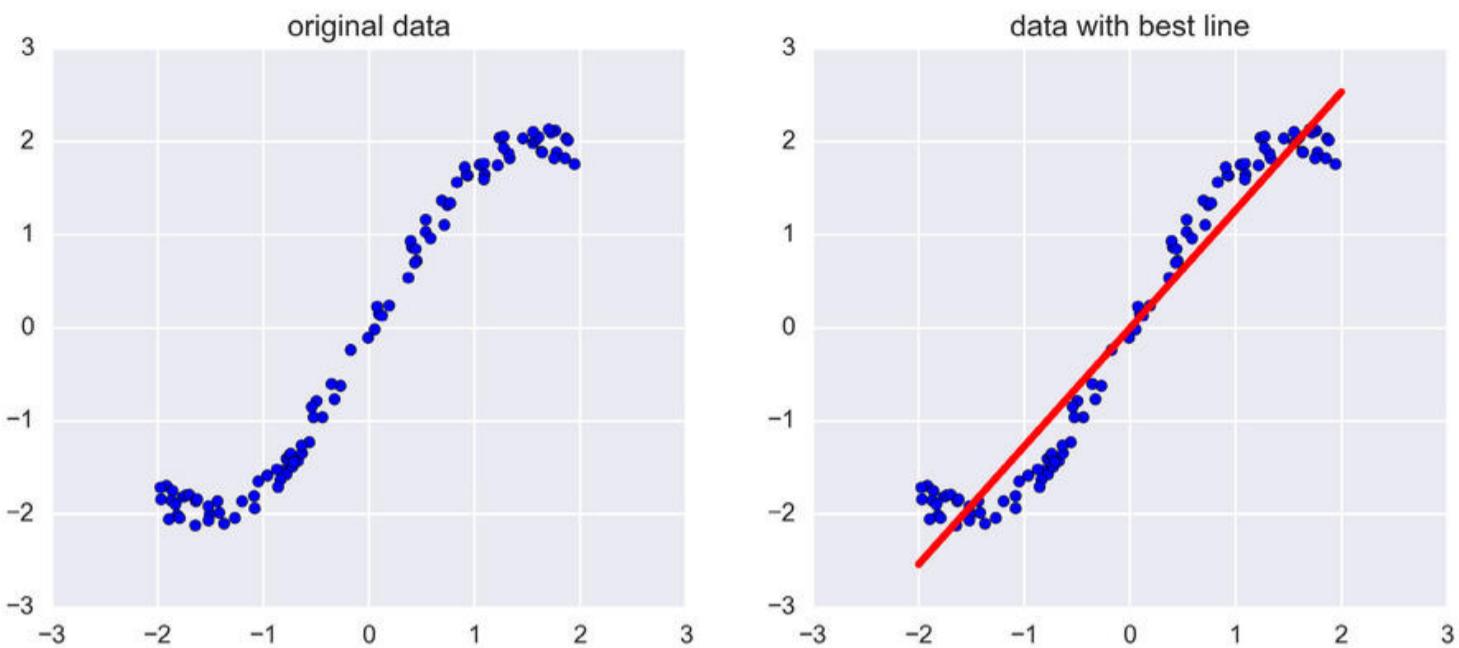


Figure 15.2: A straight line fit by a scikit-learn estimator.

15.4.1 Creation

The first step is to *create*, or *instantiate*, the **estimator object**, often just called the **estimator**. As an example, let's instantiate an estimator called `Ridge` from the `linear_model` module. The `Ridge` object is a versatile estimator for doing this sort of regression task, and it includes built-in regularization. We just name the object (prefixing it with its module name), along with any arguments we want or need to provide. Since we're sticking with all the defaults, we won't pass any arguments. Listing 15.4 shows the code.

```
from sklearn import linear_model
ridge_estimator = linear_model.Ridge()
```

Listing 15.4: Creating a Ridge estimator object.

After this assignment, the variable `ridge_estimator` refers to an object that implements the Ridge regression algorithm. When we use the simplest version of the estimator with 2D data, as we're doing here, this finds the best straight line that goes through the data.

All estimators have two routines named `fit()` and `predict()`. We use these respectively to teach our estimator, and to have it evaluate new data for us.

15.4.2 Learning with `fit()`

The first routine we'll look at is called `fit()`, and it is provided by every estimator. It takes two mandatory arguments, containing the samples that the estimator will learn from and the values (or targets) associated with them. The samples are usually arranged as a big numerical NumPy grid (called an **array**, even when there are many dimensions), where each row contains one sample.

Let's look at one example of generating data in the right format, so we can get a feeling for the process. We'll create the data shown on the left of Figure 15.2.

In Listing 15.5, we start by seeding Numpy's pseudo-random number generator, so we'll get back the same values every time. This lets us re-run the code and still generate the same data. Then we set the number of points we want to make in `num_pts`. We'll make our data based on a piece of a sine wave (a nice curve that's built into the `math` library), but we'll add a little noise to each value. Then we run a loop, appending point x and y values to the arrays `x_vals` and `y_vals`. The array `x_vals` contains our samples, and `y_vals` contains the target value for each corresponding sample.

```

np.random.seed(42)
num_pts = 100
noise_range = 0.2
x_vals = []
y_vals = []
(x_left, x_right) = (-2, 2)
for i in range(num_pts):
    x = np.random.uniform(x_left, x_right)
    y = np.random.uniform(-noise_range, noise_range) + \
        (2*math.sin(x))
    x_vals.append(x)
    y_vals.append(y)

```

Listing 15.5: Making data for training.

The `x_vals` variable holds a list. But the `Ridge` estimator (like many others) wants to see its input data in the form of a 2D grid, where each entry is a list of features on its own row. Since we have only one feature, we can use Numpy's `reshape()` routine to turn `x_vals` into a column. Listing 15.6 shows this step.

```
x_column = np.reshape(x_vals, [len(x_vals), 1])
```

Listing 15.6: Reshaping our `x_vals` data into a column.

Now that we have our data in the form expected by our `Ridge` object, we hand the samples over and ask it to find a straight line through the points. The first argument gives the samples and the second argument gives the labels associated with those samples. In this case, the first argument contains the X coordinate of each point, and the second contains the Y coordinate. We could turn the second argument into a column as well if we wanted, but `fit()` is happy to accept this argument as a 1D list.

```
ridge_estimator.fit(x_column, y_vals)
```

Listing 15.7: We train an estimator by calling its `fit()` method with the training data as an argument.

The `fit()` routine analyzes the input data and saves its results internally in the `Ridge` object. We can think of `fit()` as meaning, “analyze this data, and save the results so that you can answer future questions on this and related data.”

Conceptually, we can think of `fit()` like a tailor who starts with a bolt of fabric, and then upon meeting and measuring a customer, proceeds to cut, sew, and *fit* a custom set of clothes for that person. In the same way, `fit()` calculates data that is customized to this particular input data, and saves that inside the estimator. If we call this object’s `fit()` again with different set of training samples, it will start over and build a new set of internal data to fit that new input. This all happens inside of the object, so the `fit()` routine doesn’t return anything new (for convenience, it does return a reference to the object it was called on. So in our case, it just returns the same `ridge_estimator` we just used to call `fit()` itself).

This means that we can make multiple estimator objects and keep them all around at the same time. We might train them all on the same data and then compare their results, or we might make many instances of the same estimator and train it with different data sets. The key thing is that each object is independent of the others, containing the parameters needed to answer questions about the data it was given.

Our program so far puts together everything from Listing 15.4 to Listing 15.7.

15.4.3 Predicting with `predict()`

Once we’ve trained our estimator, we can ask it to evaluate new samples with `predict()`. This takes at least one argument, containing the new samples that we want the estimator to assign values to. The routine returns the information that describes the new data. Usually, this is a NumPy array that contains either one number or one class per sample.

Let's get the left and right ends of the line that our estimator fit to our data. We'll give the estimator the position of the left end of the data (which we set above to -2), and get back the corresponding y value. We'll do the same for the right endpoint. Listing 15.8 shows the calls.

```
y_left = ridge_estimator.predict(x_left)
y_right = ridge_estimator.predict(x_right)
```

Listing 15.8: Getting the y values of the straight line at two values of x .

Let's put this all together. We'll import the stuff we need, make the data, make the estimator, fit the data, and get the left and right Y values of the line through the data. We'll just combine Listing 15.4 through Listing 15.8. To keep things short, we'll replace the data-generating step in Listing 15.5 with a comment.

Listing 15.9 shows the code.

```

import numpy as np
import math
from sklearn.linear_model import Ridge

# In place of these comments, make the data.
# Save the samples in column form
# as x_column, and the target values as y_vals

# Make our estimator
ridge_estimator = Ridge()

# Call fit(), get best straight-line fit to our data
ridge_estimator.fit(x_column, y_vals)

# Find y values at left and right ends of the data
y_left = ridge_estimator.predict(x_left)
y_right = ridge_estimator.predict(x_right)

```

Listing 15.9: Using the Ridge object to fit some 2D data.

Figure 15.3 repeats Figure 15.2, showing the result of Listing 15.9, after we add a few lines of code to create the plots. We just draw all the data points, and then draw a red line from the point (x_{left} , y_{left}) to the point (x_{right} , y_{right}), where we set x_{left} and y_{left} ourselves when we made the data.

We leave the plotting details to the code in the notebooks. Information about using `matplotlib` can be found in the library's own documentation online, or in online or print references [VanderPlas16].

Now that we have the endpoints for this line, we can find the estimated Y value for any X by just plugging that X value into the line's equation.

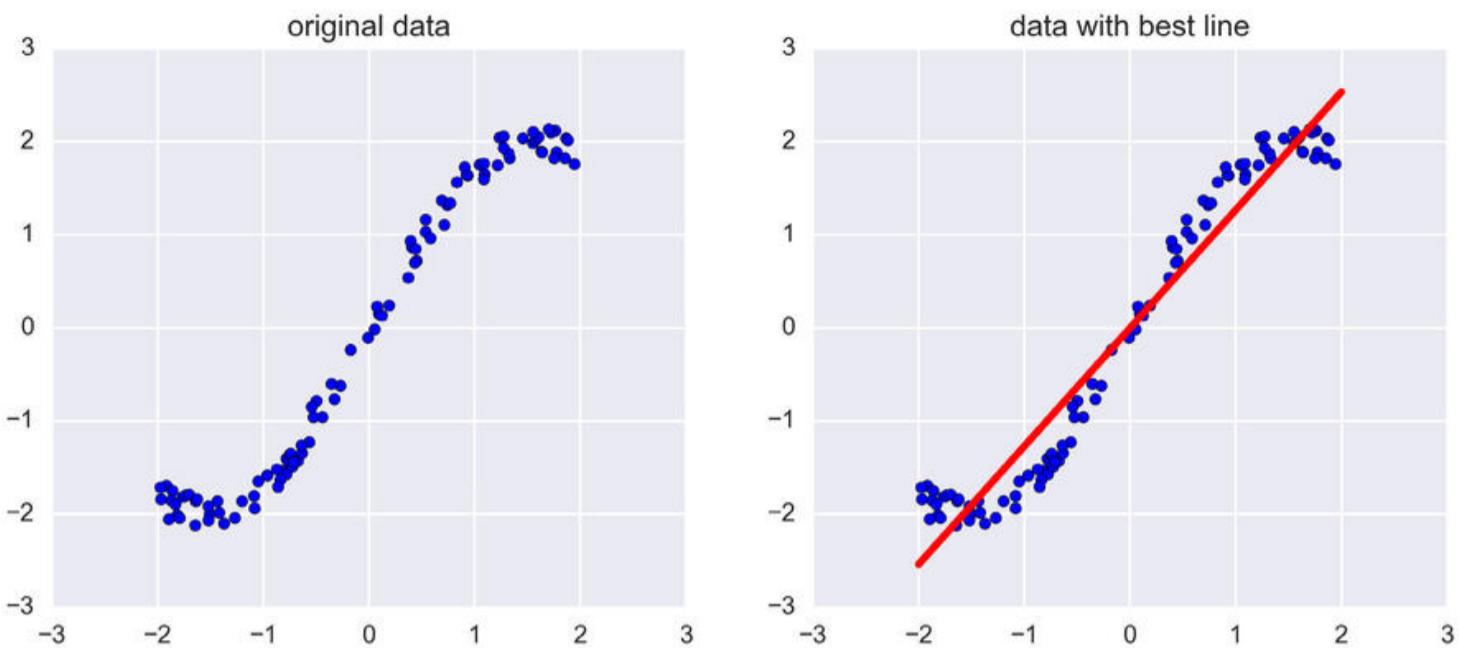


Figure 15.3: A straight line fit by Ridge to 2D data, using the code of Listing 15.9.

15.4.4 `decision_function()`, `predict_proba()`

We just saw how to use an estimator to solve a regression problem.

The procedure is very similar for classification problems, which use classification estimators.

For classifiers, the `predict()` function gives us a single category as a result. But sometimes we want to know how “close” a sample came to being categorized into each of the other classes. For example, a classifier might decide that a given photo has a 49% chance of being a tiger, but a 48% chance of being a leopard (with the other 3% being other types of cats). The `predict()` function would tell us the photo is a leopard, but we might want to know that the tiger was a very close second.

For these situations where we care about the probabilities of our input with respect to all of the possible classes, many categorizers offer some additional options.

The routine `decision_function()` takes a set of samples as input, and returns a “confidence” score for every class and every input, where larger values represent more confidence. Note that it’s possible for many categories to have large scores.

By contrast, the routine `predict_proba()` returns a *probability* for each class. Unlike the output of `decision_function()`, all of the probabilities for each sample always add up to 1. This often makes the output of `predict_proba()` easier to understand at a glance than that from `decision_function()`. It also means we can use the output of `predict_proba()` as a probability distribution function, which is sometimes convenient if we’re doing additional analysis of the results.

Though all classifiers provide `fit()`, not all of them also offer either or both of these other routines. As always, the API documentation for each classifier tells us what it supports.

15.5 Clustering

The code in this section is in the Jupyter notebook
`Scikit-Learn-Notebook-2-Clusters.ipynb`

Clustering is an unsupervised learning technique, where we provide the algorithm with a bunch of samples, and it does its best to group similar samples together.

Our input data will be a big collection of 2D points with no other information.

There are lots of clustering options in scikit-learn. Let’s use the **k-means** algorithm, which works a lot like our overview of clustering in Chapter 7. The value of k tells the algorithm how many clusters to build (though the routine calls that argument `n_clusters`, for “number of clusters,” rather than the shorter and more cryptic single letter “ k ”).

We start by creating a `KMeans` object. To get access to that, we need to import it from its module, `sklearn.cluster`. When we make the object, we can tell it how many clusters we're going to want it to build once we give it data. This argument, named `n_clusters`, defaults to 8 if we don't give it a value of our own. Listing 15.10 shows these steps, including passing a value for the number of clusters.

```
from sklearn.cluster import KMeans

num_clusters = 5
kMeans = KMeans(n_clusters=num_clusters)
```

Listing 15.10: Importing `KMeans` and then creating an instance.

Clustering algorithms are often used with data that has many dimensions (or features), perhaps dozens or hundreds. But when we create our clustering object we don't have to tell it how many features we'll be using, because it will get that information from the data itself when we provide that later via a call to `fit()`.

As usual, let's use 2 dimensions (that is, 2 features per sample) so we can draw pictures of our data and the results. To create our dataset, we'll make 7 random Gaussian blobs, and pick points at random from each one. Figure 15.4 shows the result. We also show the data color-coded by the blob that generated each sample, but that's strictly for our human eyes. The computer only sees a list of X and Y values.

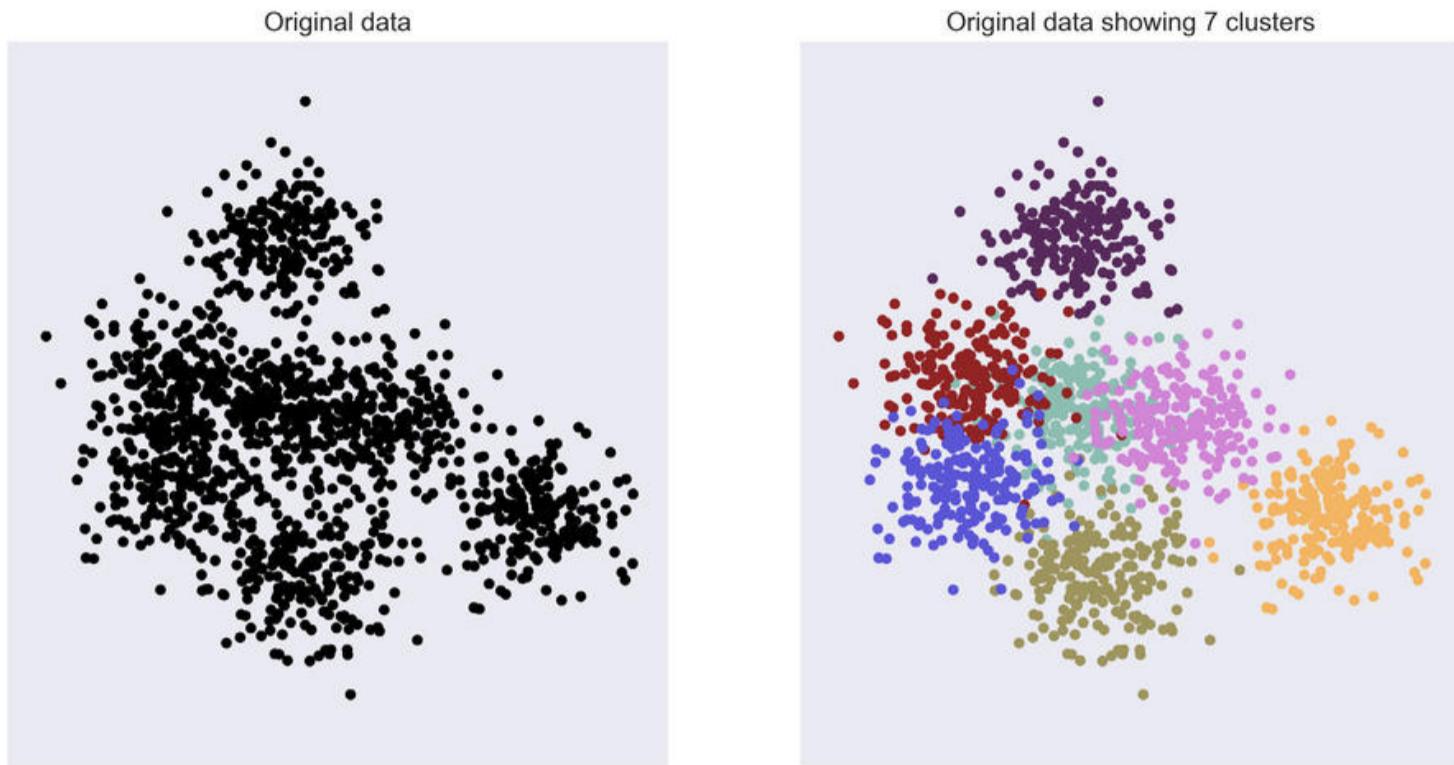


Figure 15.4: The data we'll use for clustering. Left: The data the algorithm will get. Right: A cheat sheet that shows us which of the 7 Gaussian blobs was used to generate each sample.

Remember, we're just giving it the black dots from Figure 15.4, so the algorithm knows nothing about these points except for their location. As before, we'll shape our data as a NumPy array that is a big column, where each row has two dimensions, or features: the X and Y values of that row's point. We'll call that data `XY_points`. To build the clusters from this data, we only have to hand it to our object's `fit()` routine. Listing 15.11 shows the code.

```
kMeans.fit(XY_points)
```

Listing 15.11: We give our data to `kMeans` and it will work out the number of clusters we requested when we made the object.

As soon as `fit()` returns, the clustering is done. To visualize how it broke up our starting data, we'll run through the same data again, and ask for the predicted cluster for each sample. We'll use the `predict()` method with the same data we used for fitting in Listing 15.12.

```
predictions = kMeans.predict(XY_points)
```

Listing 15.12: We can find the predicted cluster for our original data just by handing it to `predict()`.

The variable `predictions` is a NumPy array that's shaped as a column. That is, it has one row for every point in the input, and each row has one value: an integer telling us which cluster the routine has assigned to the corresponding point in our `XY_points` input. For example, the fifth row of `XY_points` holds the X and Y values of a point, and the fifth row of `predictions` holds an integer telling us which cluster that point was assigned to.

Let's run this process for a bunch of different numbers of clusters. Leaving out the plotting code, it looks like Listing 15.13.

```
for num_clusters in range(2, 10):
    kMeans = KMeans(n_clusters=num_clusters)
    kMeans.fit(XY_points)
    predictions = kMeans.predict(XY_points)
    # plot the XY_points and predictions
```

Listing 15.13: Trying out a range of different numbers of clusters.

The results are shown in Figure 15.5.

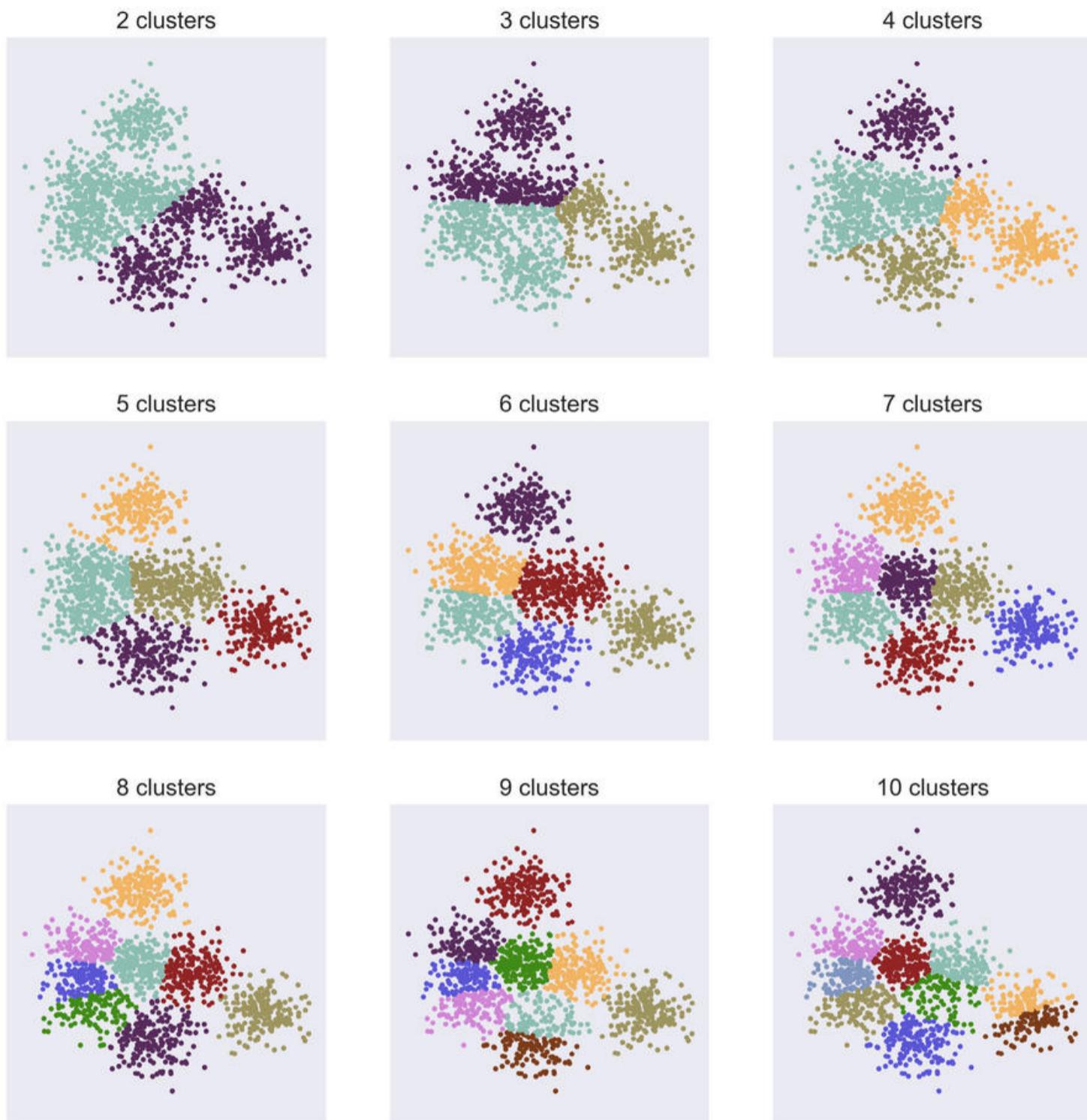


Figure 15.5: The KMeans algorithm clustering our starting data from Figure 15.4 into different numbers of clusters. Each cluster is shown in a different color. Recall we generated our data using 7 blobs.

Visually, things start to look reasonable starting at about 5 clusters. Because we used overlapping Gaussians to create our dataset, the groups are not all cleanly distinguished. But starting around 5 clusters we can see that the topmost and rightmost chunks get identified well, with the main mass getting subdivided into smaller regions. Starting at around 9 clusters those outer chunks seem to be getting split up

themselves. So somewhere between 5 and 8 seems like a good choice, which is a satisfying result knowing that we started with 7 overlapping Gaussians.

Scikit-learn offers multiple clustering routines because they approach the problem in different ways, and produce different kinds of clusters.

Clustering algorithms are a great way to get a feeling for the spatial distribution of our data. A downside of clustering is that we have to pick the number of clusters. There are algorithms that try to pick the best number of clusters for us [Kassambara17], but often we still have to look at the results and decide if they make sense.

15.6 Transformations

The code in this section is in the Jupyter notebook
Scikit-Learn-Notebook-3-Transformations.ipynb

As we discussed in Chapter 12, much of the time we'll want to **transform**, or modify, our data before we use it. We want to make sure that the data fits the expectations of the algorithms we'll give it to. For instance, many techniques work best if they receive data in which every feature is centered at 0, and scaled to the range $[-1, 1]$.

Scikit-learn offers a wide range of objects, called **transformers**, that carry out many different kinds of data transformations. Each routine accepts a NumPy array holding sample data, and returns a transformed array.

We usually choose which transformer to apply based on the estimator we will ultimately give our data to. Each estimator that wants its data in a specific form gives that information in its documentation, but it's up to us to explicitly carry out any transformations they suggest or require.

Recall from Chapter 12 that we learn our transformation from the training data, but it's essential that we then apply the same transformation to all further data. To help us carry this out, each transformer offers distinct routines for creating the transformation object, analyzing data to find the parameters for its transformation, and applying that transformation.

We create the object as we've created other objects above, by calling its creation routine with any arguments we want to pass.

To find the parameters for a specific transformation, we call the `fit()` method, just as we do for an estimator. We give `fit()` our data, and the object analyzes it to determine the parameters for the transformation performed by that object. Then we actually transform data with the `transform()` method, which takes a set of data, and returns the transformed version. Then any time we have data for the trained model, whether it's the original training data, or validation data, or even new data arriving after deployment, we'll first run it through this object's `transform()` method.

Notice how nicely this encapsulates the necessity of retaining the transformation. Our object learns what it needs to do just one time, up front, when we call `fit()`, and then it will apply that same transformation to any data we hand it from then on.

Figure 15.6 illustrates the idea.

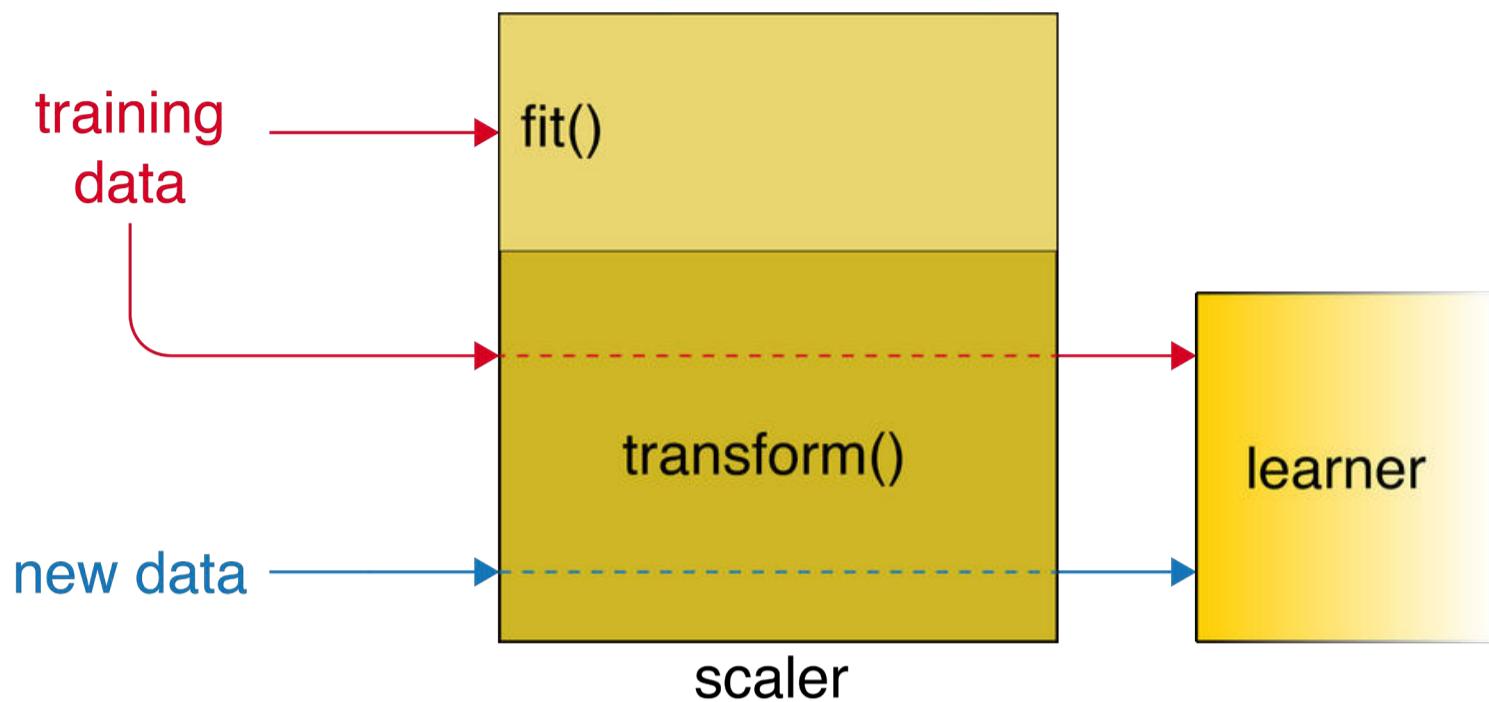


Figure 15.6: Using a scaling transformer to modify our data before training. After creating a scaling transformer, we give it the training data by calling `fit()`. The scaler analyzes the data and determines the scaling transformation. Then we transform the training data with `transform()` and train our estimator with it. From then on, all new data we want to use also goes through the scaling object's `transform()` routine.

Let's see this in action. We'll use a transformer with a very obvious visual effect: each feature is scaled to the range [0, 1]. As usual, we'll use 2D data, so there are two features to be scaled: X and Y.

We'll use a scikit-learn transformer called the `MinMaxScaler`, which was designed for just this kind of task. We can give it data with any number of features, and by default each feature will be independently transformed to the range [0,1].

Let's start with the data in Figure 15.7.

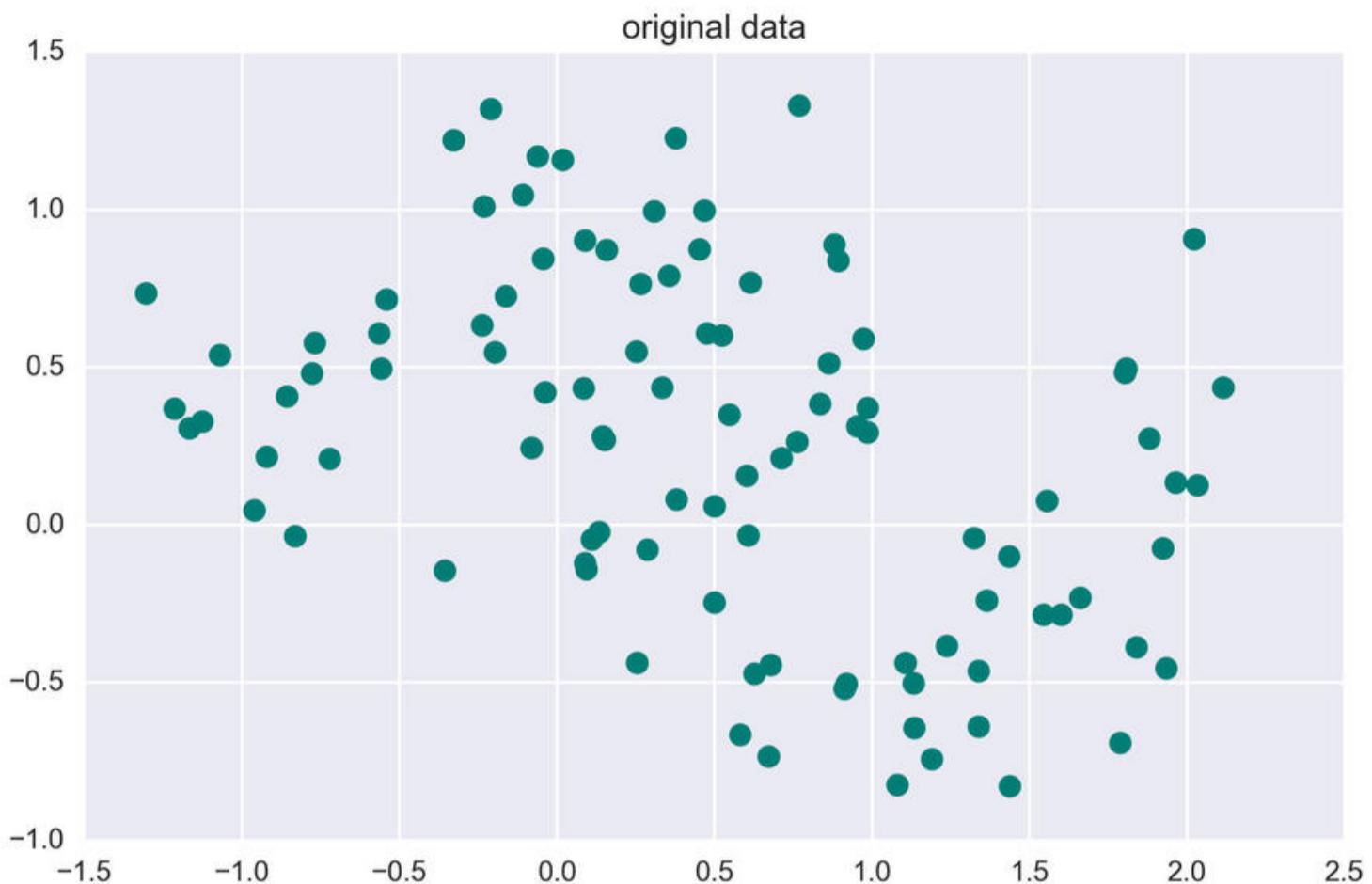


Figure 15.7: The starting two-dimensional (or two-feature) data for scaling. Note that the X values are in the range of about $[-1.5, 2.5]$ and the Y values are in the range of about $[-1, 1.5]$.

To use the scaler, as usual we have to first import the proper module from `sklearn`. In this case, the documentation tells us that it's `sklearn.preprocessing`. Our first step is to create the scaler, and save it in a variable, in Listing 15.14.

```
from sklearn.preprocessing import MinMaxScaler
mm_scaler = MinMaxScaler()
```

Listing 15.14: Creating a `MinMaxScaler` object to scale all of our features at once.

Now that we have our object, let's have it analyze our training data and work out the transformation by calling `fit()` with our training data. As before, we'll arrange our data in tabular form, where each

row contains all the features for one sample. So our data will have two entries per row (one each for the point's X and Y values). Listing 15.15 shows the call.

```
mm_scaler.fit(training_samples)
```

Listing 15.15: When we call `fit()`, our `MinMaxScaler` will work out the transformation to scale each feature to [0,1].

Now we're ready to transform our data. We just call `transform()` on our set of samples, and save the transformed values. We can then hand these to our estimator for training in Listing 15.16.

```
transformed_training_samples = \
    mm_scaler.transform(training_samples)
```

Listing 15.16: We call `transform()` on any set of data to scale it using the transformation determined when we called `fit()`.

The result of transforming our training data is shown in Figure 15.8.

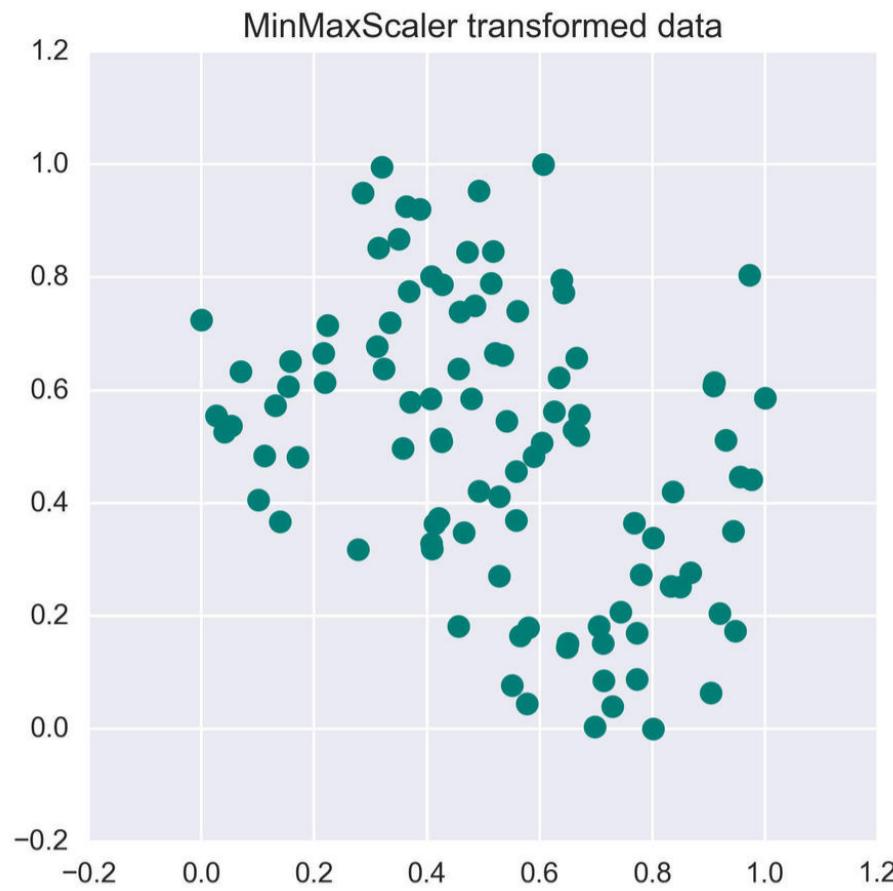


Figure 15.8: Our data from Figure 15.7 transformed by our `MinMaxScaler` so that each feature is independently scaled to [0,1].

We can see that our data now spans the range [0,1] in both X and Y.

Let's suppose that we now want to evaluate the quality of our estimator using some test, or validation, data. We know that before we give this to our estimator, we have to transform it first in the same way that we transformed the training data. We'll do that in Listing 15.17.

```
transformed_test_samples = \
    mm_scaler.transform(test_samples)
```

Listing 15.17: We call `transform()` on our new data to transform it before using it.

Figure 15.9 shows some test data, and its transformed result.

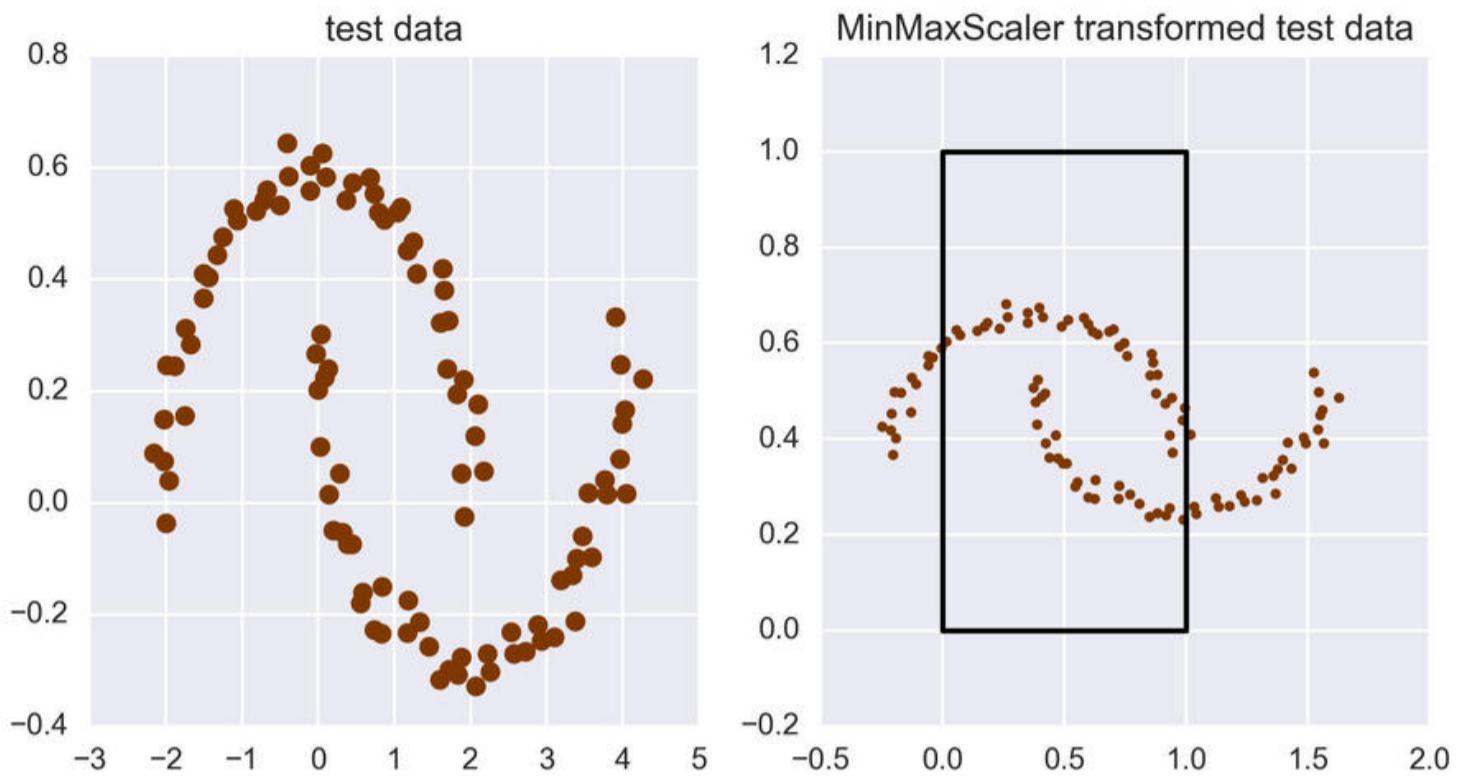


Figure 15.9: Transforming new data. Left: The test data. Right: The test data after transformation with the `MinMaxScaler` we computed for the training data in Figure 15.7. The black rectangle shows the box from (0,0) to (1,1).

Notice that the transformed data is *not* compressed and scaled to the range [0,1] in X or in Y, as we can see from the box from (0,0) to (1,1) in the figure. This is exactly what we'd expect from new data that has larger values than were found in the training data. In this case, the new test data X values are in the range of about [-3, 4], which is much larger than the training data's range of [-1.5, 2.5]. So the transformed X values will fall outside the [0,1] range. The Y range of the test data is about [-0.4, 0.6], which is much less than the training data's range of about [-1, 1.5]. Thus the Y values only use up a small piece of the range [0,1] in Y.

So although the transformation shifted and compressed the X values, it didn't do it enough for this larger data, and the transformed X values fell outside the X range [0,1]. In the same way, the system shifted and compressed the Y range, but this time by too much for this test data, causing the results to use only a small part of the Y range [0,1]. Although our estimator works best for data in the range [-1,1], it will still work well for data that's “close” to that range. The exact meaning

of “close” will vary from one estimator to another, so it pays to check, but it’s doubtful that a few data values out near 1.5 would cause the estimator to produce meaningless results.

15.6.1 Inverse Transformations

We can use `predict()` to get an estimator’s output for some data. Remember that these results are based on the data which we fed into the estimator, which we’d transformed first. Whether we’re looking at training data, test data, or deployment data, it all went through the transformation.

In Chapter 12 we looked at a regression problem that asked us to predict the number of cars on the street given the temperature the previous midnight. We transformed the input data, so the value predicted by the estimator was also transformed. We saw that we had to **undo**, or **invert**, the transform so that it represented a number of cars, rather than a value from 0 to 1.

That process is typical. If we want to compare the results that come out of an estimator with our original, un-transformed data, we have to somehow *undo* the transformation. In our scenario above with the `MinMaxScaler`, we want to un-stretch the samples in the exact opposite ways that we originally stretched them.

Every scikit-learn transformer provides a method called `inverse_transform()` for precisely this purpose, where “inverse” means “opposite.” So this routine applies the opposite of whatever `transform()` did. Figure 15.10 shows the idea, where we’re showing a generic “learner” rather than a specific estimator.

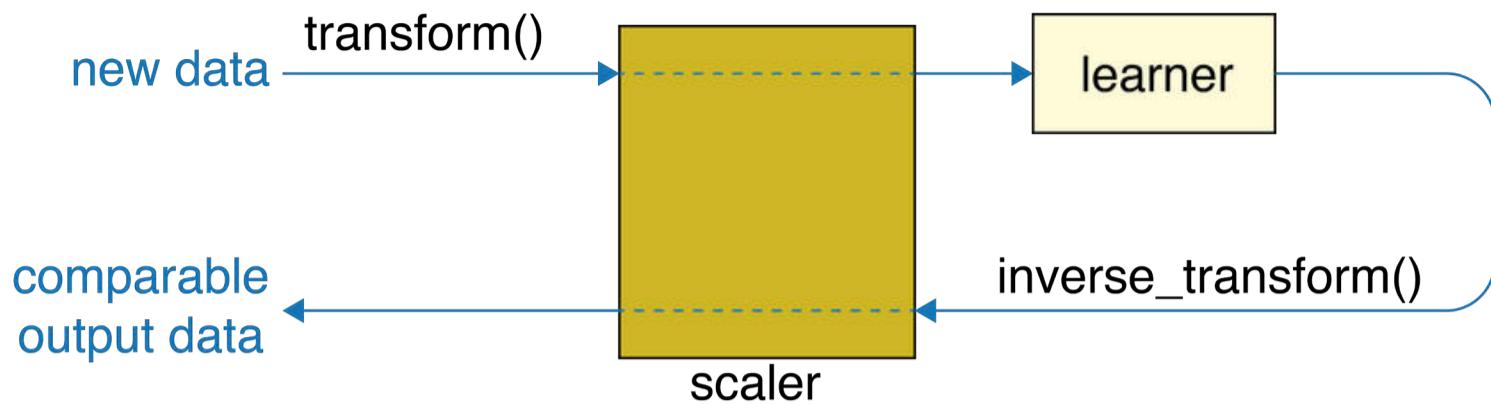


Figure 15.10: The output of the learner is based on the transformed values it receives. To convert that data back to the original form (in this case, the original range), we call the transformer's `inverse_transform()` method on that output data. If we took the learner out of this loop, the values in the lower-left would be identical to those in the upper-left.

For example, we can feed the samples on the right side of Figure 15.9 to `inverse_transform()`, and we'd get back the samples on the left. Listing 15.18 shows the steps.

```
recovered_test_samples = \
    mm_scaler.inverse_transform(transformed_test_samples)
```

Listing 15.18: We call `inverse_transform()` to undo the transformation applied by this object.

Let's see this in action. On the left of Figure 15.11 we see our starting data, which is similar to data we used earlier. The data has an X range of about [0,6] and a Y range of about [-2, 2]. After setting up a new `MinMaxScaler` and calling `fit()` with this data, and then running it through `transform()`, we get the version on the right, where both ranges are now [0,1].

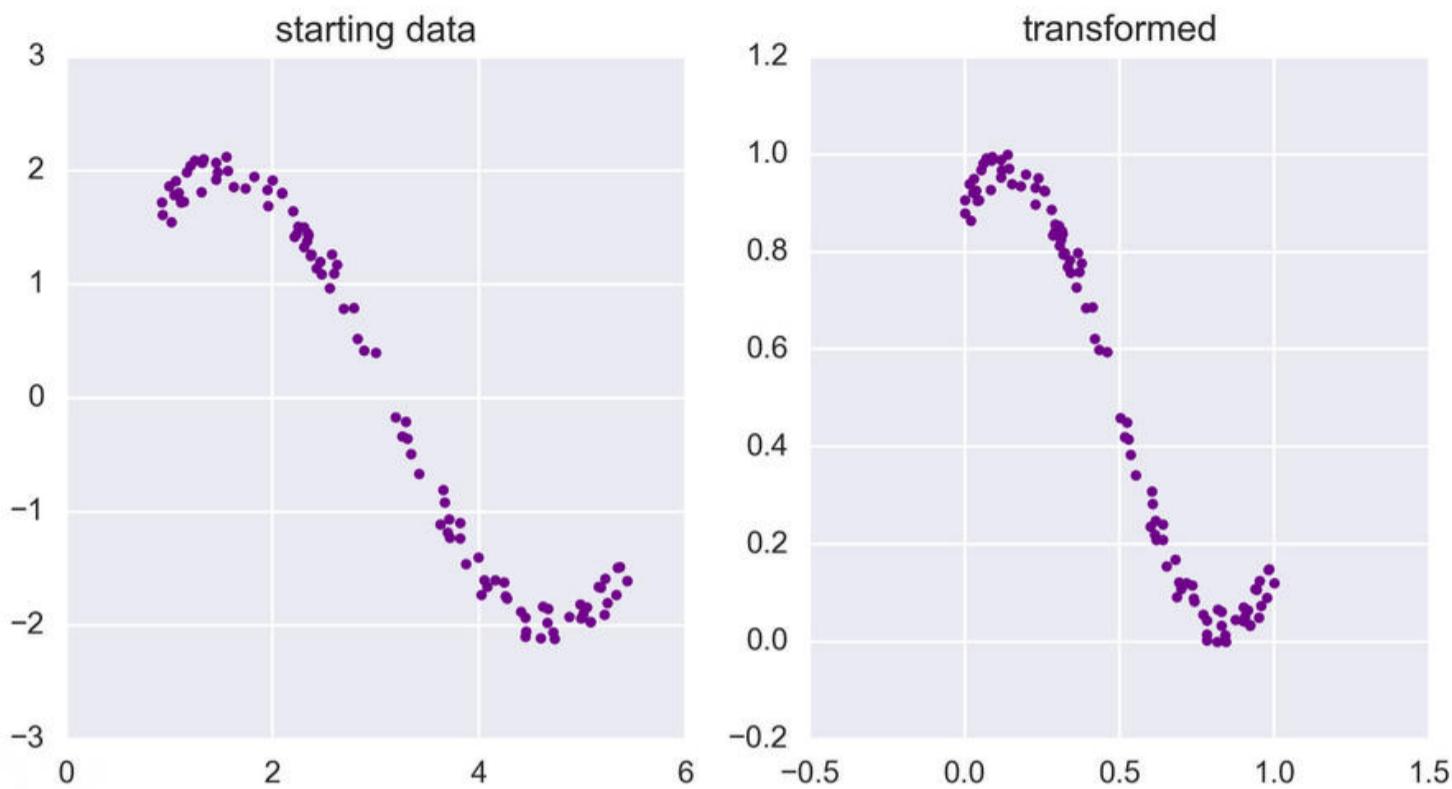


Figure 15.11: Left: Some test data. Right: After setting up a new `MinMaxScaler` object with this data, and then transforming it, both X and Y are in the range [0,1].

Let's now fit a line to our transformed data. We'll use the `Ridge` estimator we saw earlier.

The leftmost image of Figure 15.12 shows the line that our `Ridge` estimator fit to the transformed data, along with the transformed data itself. In the middle image we show that line, along with the *original*, non-transformed input data. It's a terrible match! That's because the line was fit to the transformed data, not the original.

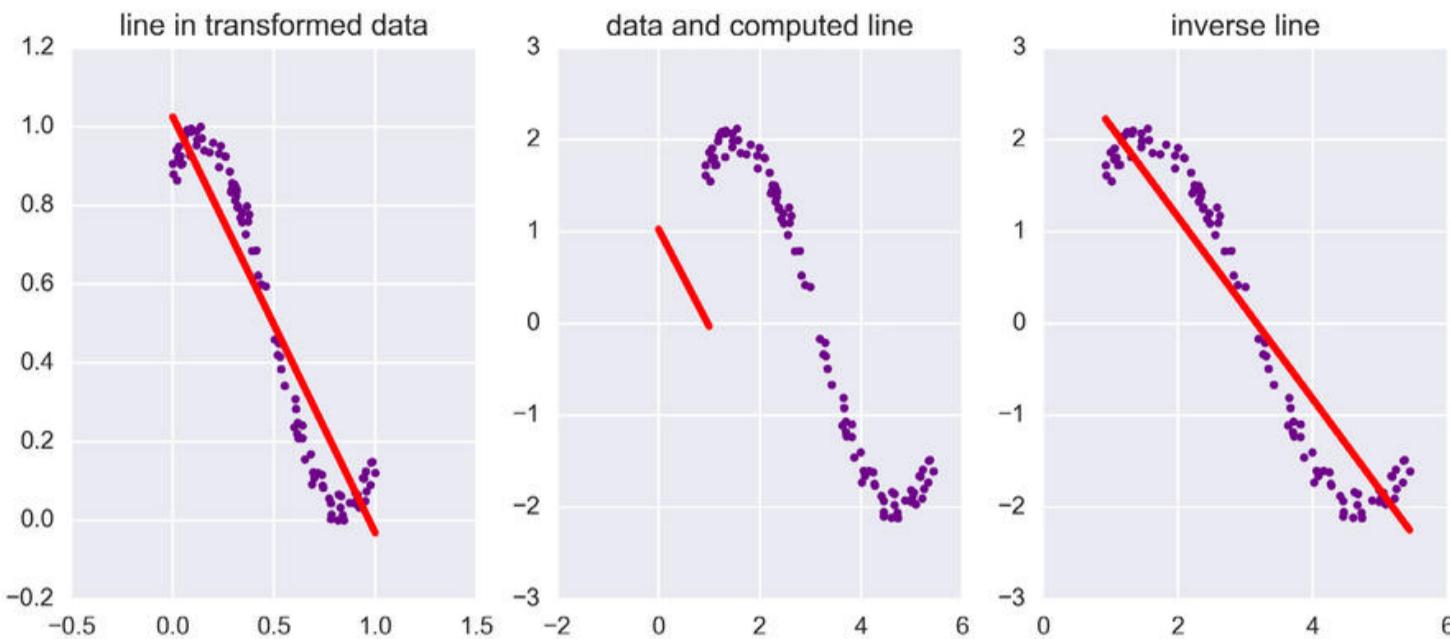


Figure 15.12: Fitting a line to transformed data. Left: Our transformed data, where both X and Y are in the range [0,1], and the line that our Ridge object fit to that data. Middle: Our *original* data (note the different ranges on the axes), along with the line that we found on the left. The line doesn't match at all, because it was fit to the transformed data. Right: When we use `inverse_transform()` on the line endpoints, our `MinMaxScaler` will apply the opposite, or inverse, of the transformation it applied to the data. Now our inverse-transformed line correctly matches our starting data.

To get the line to match up with our input data, we need to run it through the `inverse_transform()` method of the `MinMaxScaler` that we used. To do this, we'll treat the endpoints of the line as two samples, and inverse-transform those two points.

The original data, along with the line after this inverse transformation, are shown on the rightmost image of Figure 15.12.

15.7 Data Refinement

The code in this section is in the Jupyter notebook
Scikit-Learn-Notebook-4-PCA.ipynb

Sometimes we have too much data.

Maybe some of the features in our data are redundant. For example, if our data records deliveries from a pizzeria, we might have a field for the size of each pizza and the size of the box it should be placed into. It's probably the case that the box size can be predicted from the pizza's size, and vice-versa.

The routines that perform **data refinement** are designed to locate and remove such redundancies from our data either by removing some features altogether, or by making new features out of combinations of others. Another class of routines, such as those related to the Principal Components Analysis (PCA) method we saw in Chapter 12, are also able to compress data that's related but not entirely redundant. These trade off some loss of information for a simpler database by combining features.

Let's see an example of data compression, from 3D to 2D. Figure 15.13 shows a dataset of points drawn from a 3D blob shaped like a bulging ellipse. The X range is around $[-0.8, 0.8]$, the Y range is around $[-0.3, 0.3]$ and the Z range is about $[-0.7, 0.7]$.

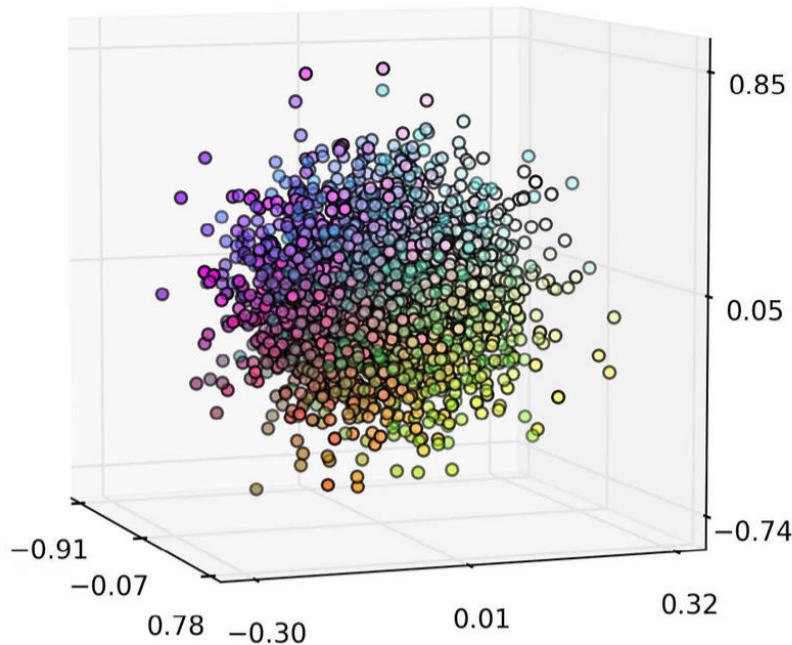


Figure 15.13: Our starting 3D blob. The smallest range is in the Y values. The colors are just to help us keep track of where points are located.

Let’s reduce this 3D blob down to just 2 features (that is, we’ll compress it into 2D data). We’ll use PCA for this, as discussed in Chapter 12. Leaving out the code to build the blob and draw the plots, the core steps are only three: create the `PCA` object (and tell it how many dimensions we want), call `fit()` so it can determine which features to remove, and call `transform()` to apply the transformation. Listing 15.19 shows the code. Here we’re also asking PCA to “whiten” the data, which can help PCA produce its best results..

```
from sklearn.decomposition import PCA
# make blob_data, the 3D data forming the "blob"
pca = PCA(n_components=2, whiten=True)
pca.fit(blob_data)
reduced_blob = pca.transform(blob_data)
```

Listing 15.19: To apply PCA, we first make the PCA object. Here we tell it to reduce our data down to 2 features, and to whiten each feature along the way. Then we call `fit()` with our 3D data so the PCA algorithm can determine how to reduce it. Finally, we call `transform()` to apply the transformation and get back our reduced, 2D data.

The result is shown in Figure 15.14. Each data point is now described by only two values, rather than three. In other words, the result of PCA is that our data has gone from being three-dimensional to two-dimensional.

Note that it wasn't simply squashed along one dimension. As we discussed in Chapter 12, the algorithm placed a plane through the 3D blob so that it captured as much variance in the data as possible, and then projected each point onto that plane.

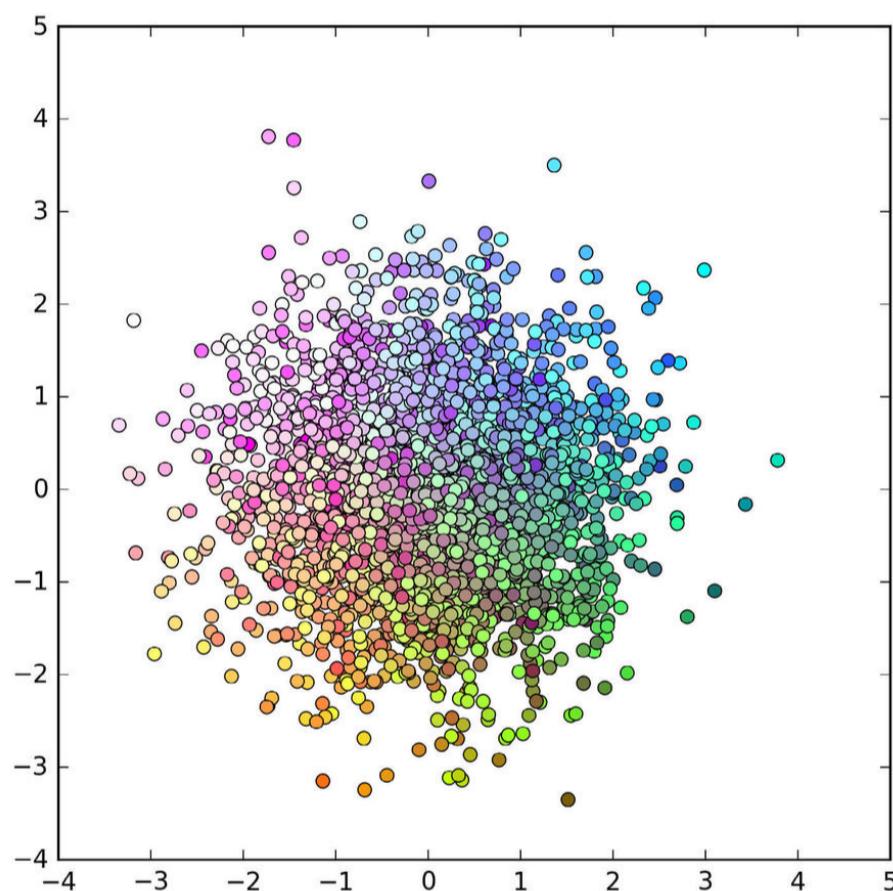


Figure 15.14: Our starting blob of Figure 15.13 after running it through PCA. We requested PCA to save reduce our original 3 features into only 2 while retaining as much information as possible.

15.8 Ensembles

The code in this section is in the Jupyter notebook
Scikit-Learn-Notebook-5-Ensembles.ipynb

Sometimes it's useful to create a bunch of similar but slightly-different estimators, and let them all come up with their own predictions. Then we can use some kind of policy (usually voting) to choose the "best" prediction, and return that as the group's best result.

As we saw in Chapter 13, such collections of estimators are called **ensembles**. Let's see how to build and use ensembles in scikit-learn.

The system is set up so that we can treat an ensemble just like any other estimator. That is, we wrap up all the individual estimators into one big estimator, and use that directly. Scikit-learn takes care of all the internal details for us.

So just like any other estimator, our first step in using an ensemble is to create an ensemble object. Then we call its `fit()` method to give it data to analyze. Finally, we can call its `predict()` method to evaluate new data. We don't have to concern ourselves with the fact that there are multiple estimators hiding inside the estimator object we're using.

Some of the ensembles in scikit-learn will make these collections out of any kind of estimator, while others are limited to just classifiers, just regression algorithms, or even just specific instances of algorithms.

Let's build an ensemble to do classification. We'll use a data set of 1000 points, formed into 5 spiral arms, one for each of 5 classes. This starting data is shown in Figure 15.15.

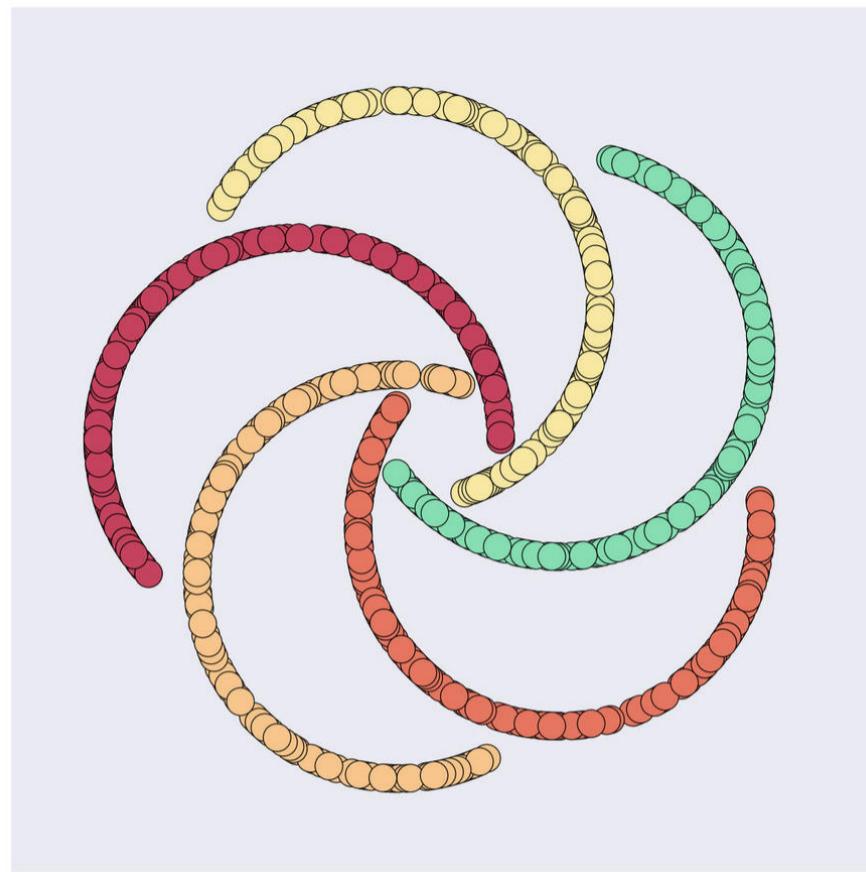


Figure 15.15: Our starting data for ensemble classification. Each of the five arms contains 200 points of a different class.

We'll use 2/3 of these points, randomly selected, as our training data, and the remaining 1/3 as test data.

We'll use a general-purpose ensemble maker which can build a group of classifiers from almost any specific classifier algorithm. The ensemble maker is called `AdaBoostClassifier`. The word `Boost` comes from the fact that internally it uses the technique of **boosting**, which we saw in Chapter 13.

When we create the ensemble with this object, we tell it which algorithm it should build multiple copies of. We'll use the `RidgeClassifier` classifier, which is the classifier version of the Ridge regression algorithm we used above. Listing 15.20 shows the steps.

```
from sklearn.linear_model import RidgeClassifier
from sklearn.ensemble import AdaBoostClassifier
ridge_ensemble = \
    AdaBoostClassifier(RidgeClassifier(), \
        algorithm='SAMME')
```

Listing 15.20: Creating an ensemble of `RidgeClassifier` objects using the `AdaBoostClassifier` ensemble object. The `algorithm` argument needs to be set to `SAMME` for this classifier (this is not a typo of the word “same”), as explained in the documentation.

The documentation for `AdaBoostClassifier` explains that it has two different modes, depending on what methods are supported by the classifier it’s building a collection of. In the case of the `RidgeClassifier`, we need to specify the `SAMME` algorithm (note the two M’s).

We can control how many classifiers go into our ensemble using the optional `n_estimators` argument. For this demonstration we’ll leave it at the default value of 50 estimators. We’ll also leave all the other arguments (which include the learning rate) at their defaults.

Now that our collection is made, using this ensemble of estimators is just like using one of them. We train the ensemble (and all the estimators inside of it) with samples we pass into it using `fit()`. Since we’ve build this ensemble for supervised learning of categories, the `fit()` routine takes in both the samples and their labels. Listing 15.21 shows the call to our ensemble’s `fit()` routine.

```
ridge_ensemble.fit(training_samples, training_labels)
```

Listing 15.21: Fitting our ensemble object.

Finally, we can get new values out by asking the ensemble to predict them using `predict()`, as in Listing 15.22.

```
predicted_classes = ridge_ensemble.predict(new_samples)
```

Listing 15.22: Using our ensemble object to make new predictions.

Internally, ensembles usually pick the final output by using some kind of voting algorithm, where the most frequently predicted category becomes the final result.

Figure 15.16 shows the classification of our test data from our ensemble of 50 Ridge classifiers. These results aren't terribly encouraging.

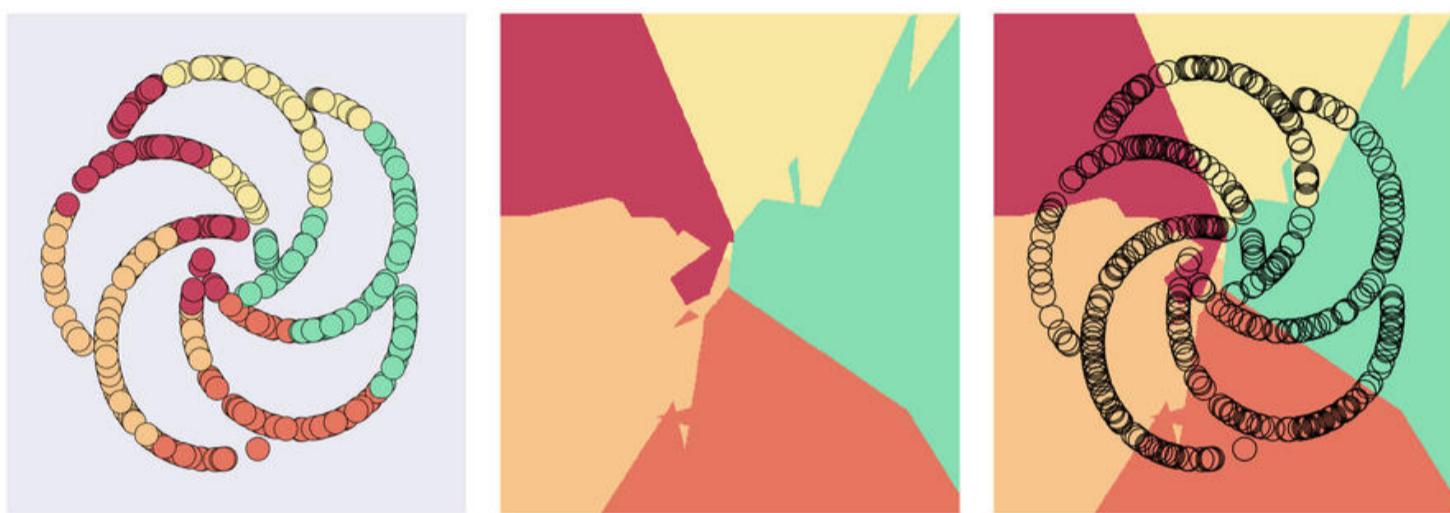


Figure 15.16: The results of our ensemble of 50 Ridge classifiers. Left: The test data, color-coded by the category each point was assigned to. Middle: The regions created by our ensemble. Right: Overlaying the spiral points on the regions.

The trouble is that we're trying to fit 50 straight lines to this data in a way that will let them correctly categorize our swirling data.

We could try to improve these results by experimenting with the number of estimators, the learning rate, or the arguments to the estimators.

Another approach would be to try another estimator. Let's use decision trees. The only changes we need to make are to import the necessary module, and then tell `AdaBoostClassifier()` to build the ensemble out of these classifiers, as in Listing 15.23.

```
from sklearn.tree import DecisionTreeClassifier
tree_ensemble = \
    AdaBoostClassifier(DecisionTreeClassifier())
```

Listing 15.23: Building an ensemble out of decision trees. For decision trees, we can leave the `algorithm` argument at its default value.

Now we use `fit()` and `predict()` just as before. Figure 15.17 shows the results of classifying with 50 decision trees. For this data, and using all the defaults, decision trees turned in a nearly perfect performance!

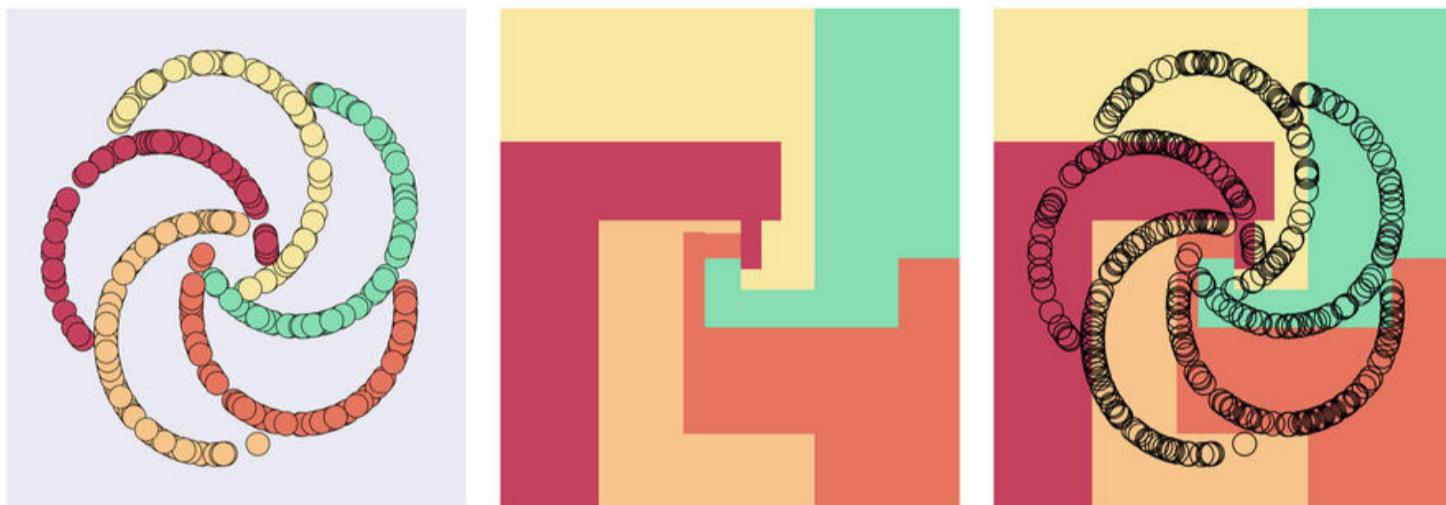


Figure 15.17: The results of an ensemble made of 50 decision tree classifiers. Left: The test data, color-coded by the category each point was assigned to. Middle: The regions created by our ensemble. Right: Overlaying the spiral points on the regions.

15.9 Automation

The code in this section is in the Jupyter notebook
`Scikit-Learn-Notebook-6-Automation.ipynb`

Machine learning systems have lots of parameters, and often lots of hyperparameters, too. The distinction is that the system learns its parameters from the data, while we set the hyperparameters. Typical hyperparameters include the learning rate, the number of clusters in a clustering algorithm, the number of estimators in an ensemble, and the amount of regularization to apply.

We often want to try out many values of these hyperparameters to find the combination that gives us the best performance for a given system and data. If we can't find the very best results, we'd at least like to try a few combinations and use the ones that do the best.

If we want to automate this search process, we need two basic pieces. The first automates the selection of hyperparameters, choosing the values that get used to build and train a learner. The second piece evaluates that learner and assigns a score to its performance. We can then choose the combination that resulted in the best score.

Scikit-learn offers tools for both of these steps. We think of them as **automation** tools, since they handle this repetitive process for us.

15.9.1 Cross-validation

Most learning algorithms have multiple parameters and hyperparameters that control how quickly and how well they learn. Finding the best combination of these values can be difficult, because they depend on the nature of the data we're training with.

As we saw in Chapter 8, we can use **cross-validation** to determine the quality of a model on a given set of data. This technique is particularly attractive when we don't have a big training set, because it doesn't require us to remove a permanent validation set from the training data. Instead, we break up the training into equal-sized pieces called **folds**, and then train and evaluate our model independently many times, using the data in one of the folds as our validation set. The performance of the model is typically reported as the average performance of these multiple versions [scikit-learn17c].

Scikit-learn lets us automate this process. We hand a routine our estimator, our data, and the number of folds we want it to use, and it carries out the whole process for us, reporting the average score when it's done. Many scikit-learn estimators have specialized versions that carry out cross-validation. These are consistently named by appending the routine's usual name with the letters `cv` at the end. They're all listed in the API documentation [scikit-learn17a].

For example, let's suppose we want to use the Ridge classifier we used earlier to learn categories from a set of labeled training data. To evaluate its performance, we'll use cross-validation.

The Ridge classifier is implemented as an object called `RidgeClassifier`. Following the convention we just described, the version of `RidgeClassifier` object with cross-variance built in is called `RidgeClassifierCV`. So our first step is just to make one of these objects.

To carry out cross-validation on this estimator we need only call its `score()` method with our training data and labels. This one call does the whole cross-validation process for us from start to finish. The `score()` routine takes care of chopping up the data into folds, building and evaluating a `RidgeClassifier` for each version of the training data, and then returning the average of the scores. Listing 15.24 shows the steps.

```
from sklearn.linear_model import RidgeClassifierCV

ridge_classifier_cv = RidgeClassifierCV()
mean_accuracy = ridge_classifier_cv.score(
    training_samples, training_labels)
```

Listing 15.24: To run the `RidgeClassifier` object through cross-validation, we first create an instance of the `RidgeClassifierCV` object. Then we call its `score()` routine with our training data and labels. The result is the average accuracy value from running through the training and validation process for each fold.

We can pass a variety of optional parameters into our `RidgeClassifierCV` object when we create it. Perhaps the most important parameter is the number of folds we want to use (here we left it at the default value of 8).

`RidgeClassifierCV` uses a reasonable fold-making algorithm that just breaks up the data into equal pieces. This often works pretty well, but scikit-learn offers several alternatives, called **cross-validation generators**, that can sometimes do better. For instance, the `StratifiedKFold` cross-validation generator pays attention to the data when it creates the folds, and tries to distribute the number of

instances in each class equally. In other words, it tries to make sure each fold has roughly the same makeup as the entire dataset, which is always a good idea.

To use this approach, we first create a `StratifiedKFold` object, and tell it how many folds to use (that's the value of `K` in the object's name). The name of this argument for this object is unfortunately not named something like "number of folds," but is instead `n_splits`.

The `StratifiedKFold` object doesn't need our data when we create it, because the cross-validator will send the data to it for us when it uses this object to build the folds. We just provide our cross-validation stratifier object as the value of an argument named `cv` to the cross-validation object `RidgeClassifierCV`, and the rest happens automatically. Listing 15.25 shows the code.

```
from sklearn.model_selection import StratifiedKFold
strat_fold = StratifiedKFold(n_splits=10)
ridge_classifier_cv = RidgeClassifierCV(cv=strat_fold)
```

Listing 15.25: To use a stratifier of our own choice, we have to first create it, which means we also have to import it. Here we import the `StratifiedKFold` object and tell it to use 10 folds. We pass that to our `RidgeClassifierCV` object through the argument `cv`.

Let's look at this process visually. The scikit-learn model of cross-validation with a classifier is summarized in Figure 15.18. The input to the system consists of the labeled training data and the number of folds, along with the model constructor (that is, the routine that creates the classifier) and its parameters (which we've been leaving at their defaults in our examples). The routine then loops through each fold, removing it from the data, training on what remains, and evaluating the result on the extracted fold. The resulting scores are produced as output, or are averaged together to present just a single value. In this figure, and those to follow, a wavy box with arrows represents a loop. In this case, it's the loop that runs through the process above once for each fold.

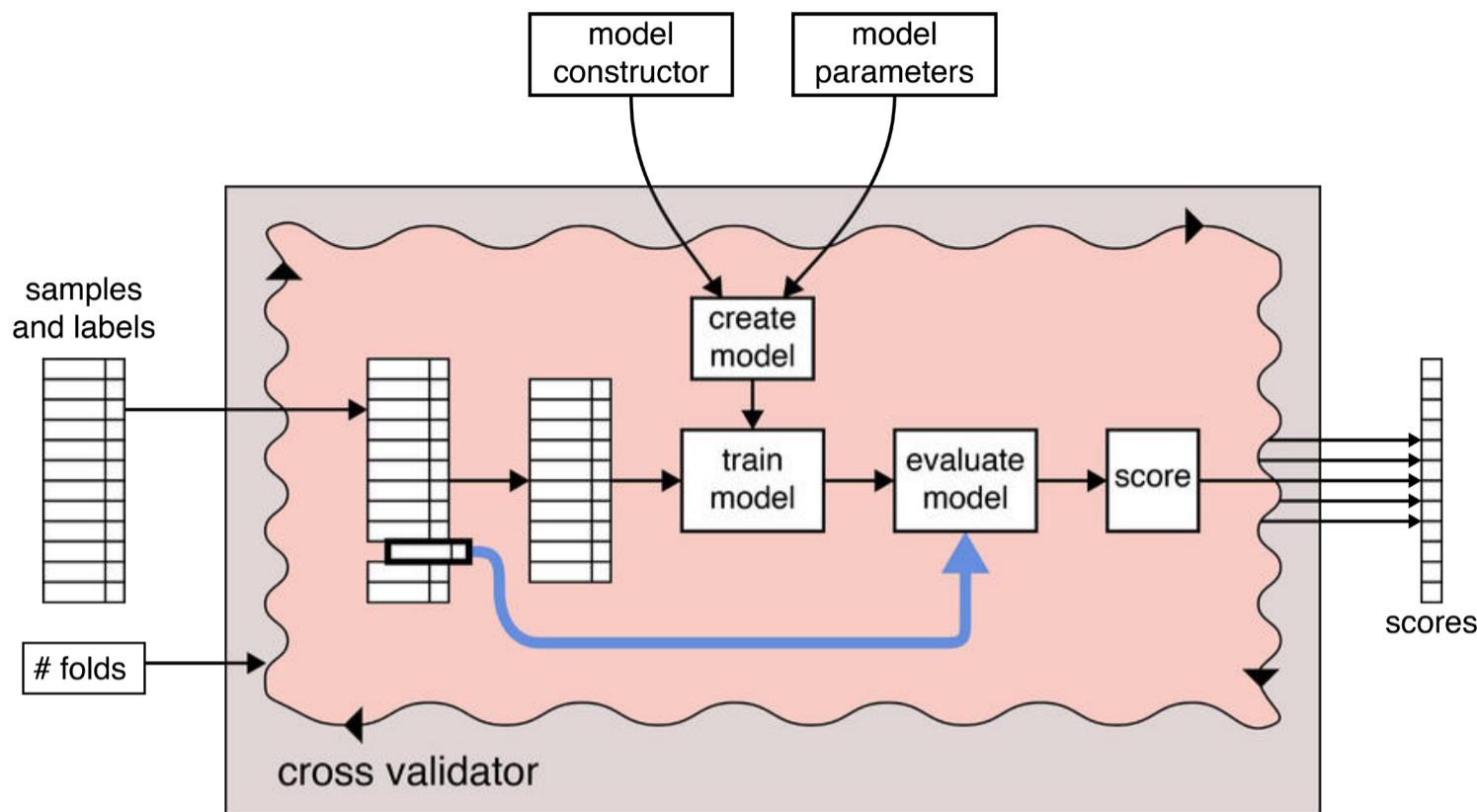


Figure 15.18: A visualization of cross-validation with a classifier. In scikit-learn, the cross-validation routine runs a loop, here indicated by the wavy rectangle with arrows. Each time through the loop, one fold is extracted from the training data. The remaining samples are used to train a model, and then we evaluate the result on the validation fold, producing a score. The result is one score for each fold. We either present those scores, or their average, as output.

Figure 15.18 is missing any transformations. That is, we're not scaling or standardizing our data, or running feature compression on it, or any of the other transformations that we've seen, which can help our models learn.

To include these transformations in cross-validation we have to be careful. If we just drop a transformation into the inner loop of Figure 15.18 without thinking things through, we can risk information leakage. We saw in Chapter 8 that this can give us a distorted view of the performance of our model.

To include transformations properly we can use another scikit-learn tool called a **pipeline**. A good way to understand pipelines is to see how they're used when we search for hyperparameters, which is an important topic on its own.

So we'll look at searching and pipelines next, and then we'll come back to cross-validation and see how to use a pipeline to include transformations.

15.9.2 Hyperparameter Searching

We often distinguish between **parameters**, which are usually adjusted by the algorithms themselves in response to data, and **hyperparameters**, which we generally set by hand.

For example, suppose we want to run a clustering algorithm on a data-set. The cluster sizes and centers are parameters that are learned from the data, and maintained “inside” the algorithm. In contrast, the number of clusters to use is set by us “outside” of the algorithm, and we call such values hyperparameters.

Finding the best values for all the hyperparameters in a learning system is often a challenging task. There may be many values to tinker with, and they might influence one another. So using an automated approach to searching for them can save us a lot of time and hassle.

We could even generalize the idea of searching for hyperparameter values to searching for a choice of algorithm. For example, we might want to try out several different clustering algorithms, looking for the one that does the best job with our data.

To make this search easier, scikit-learn provides a bunch of routines that automate these types of searches for parameters, hyperparameters, and algorithms.

We identify the types of things we want to search over and the values we want them to try out, and the criteria on which to judge each result. Our search process then runs through every combinations of values, ultimately reporting the set that produced the best results.

Note that although scikit-learn makes it easy to set up and run this search, it's not any faster than if we were to do it by hand (except for the typing time). It just methodically grinds through the values to be searched, then builds, trains, and measures the resulting algorithm, over and over again.

There are two popular ways to approach this kind of search: the **regular grid** and the **random search**.

The regular grid tests every combination of the parameters. The word “grid” is in its name because we can visualize all the combinations it tries as points on a grid. A 2D example is shown in Figure 15.19.

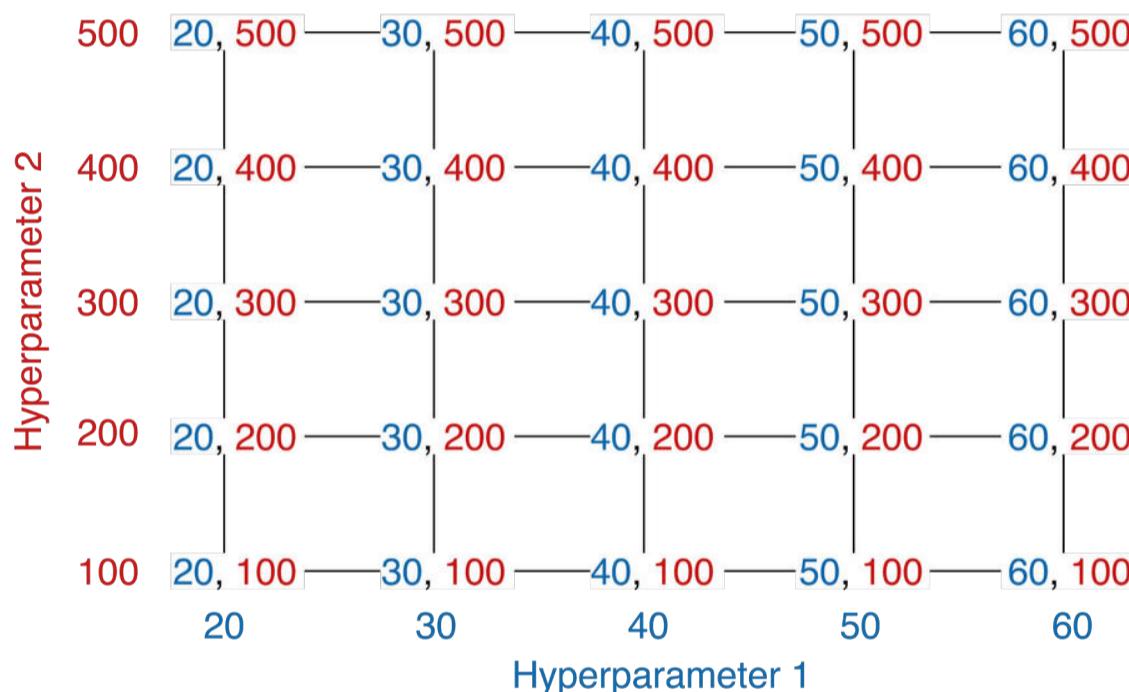


Figure 15.19: A regular grid search using two hyperparameters, each with 5 values. For every combination of the two values, we build, train, and test a new system. There are 25 combinations to be tried.

In this example, we have 5 values for each of two hyperparameters, giving us $5 \times 5 = 25$ combinations to examine.

As we include more parameters to be searched, or more values for each one, the size of this grid, and thus the number of training/measuring steps we have to carry out, will grow correspondingly. This means the whole thing will require more and more time to run. Eventually it could require more time than we have available. For example, if we

have 4 hyperparameters to search, and 6 values for each one, that's $6 \times 6 \times 6 \times 6 = 1,296$ different combinations. If it takes an hour to train and test each model, that search would take 54 days of non-stop computing!

It would be great to cut away parts of the grid and save time, but that's a risky step because we can't know beforehand what combination of variables might turn up the best results.

A grid search usually proceeds methodically, working its way through all the possible combinations in a predictable, fixed order, such as left to right, and top to bottom.

A faster but less informative alternative is to search these combinations **randomly**, rather than in some fixed order. The algorithm picks a random combination of hyperparameter values that hasn't been tried yet, trains and measures the resulting model, and then picks another untried random combination, and so on. We can think of many different conditions to control when this process ought to stop. For example, the algorithm could stop when it's searched every combination, or it's tried a certain number of combinations that we've specified, or it's run longer than an amount of time that we've specified, or we simply get tired of waiting and stop it manually. Then it returns the best combination it found. Figure 15.20 illustrates the idea.

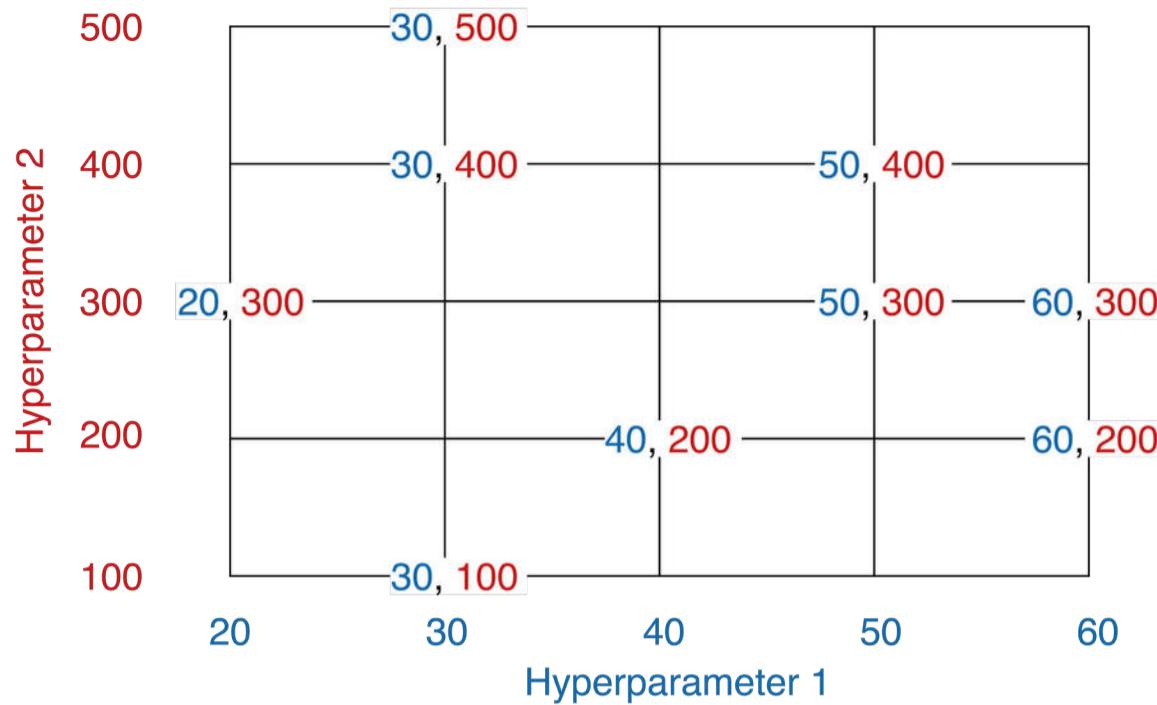


Figure 15.20: A random search doesn't test every combination in sequence, but tests them in random order. This can give us an overall feeling for where the big scores are located without exhaustively trying every combination first. Here, we ran through 8 steps of picking a random combination, then building and measuring the resulting model.

The advantage of this approach over the regular grid is that we can watch the results as they come in, and perhaps get an idea for where we can focus our search. Then we can stop and start again in the neighborhood that looks promising. For example, suppose that in Figure 15.20 the combination (40, 200) performed much better than any of the other combinations. As a result, we might decide to run a new close-up search in that area, perhaps using a regular grid, as in Figure 15.21.

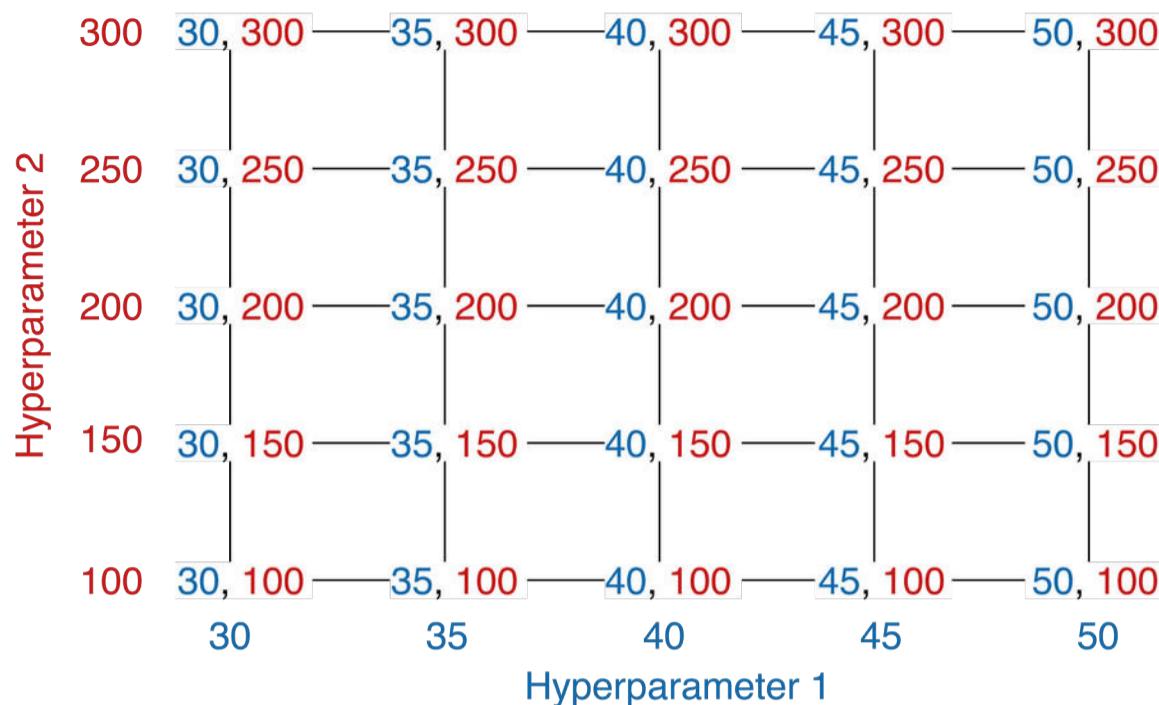


Figure 15.21: When we think we've found the neighborhood of the best values, we can run a new search that zooms in on that region. Here we look around the region where hyperparameter 1 has a value of 40, and hyperparameter 2 is 200.

Let's look at how to use scikit-learn to run a regular, methodical grid search first. We'll see that we set up and call the randomized version in almost an identical way.

15.9.3 Exhaustive Grid Search

The exhaustive, methodical grid-based search is provided by an object called `GridSearchCV` (the `cv` at the end reminds us that the algorithm will use cross-validation to evaluate the performance of each model it tries out).

`GridSearchCV` produces each combination of parameters one by one, builds and trains the model, and then measures how well the corresponding model performs on our data, returning a final score for that model.

Note that the estimator we hand to this routine shouldn't perform cross-validation itself (that is, its name shouldn't end in `cv`), because the grid searcher is handling that step.

Figure 15.22 shows the pieces that go into making this grid-search object.

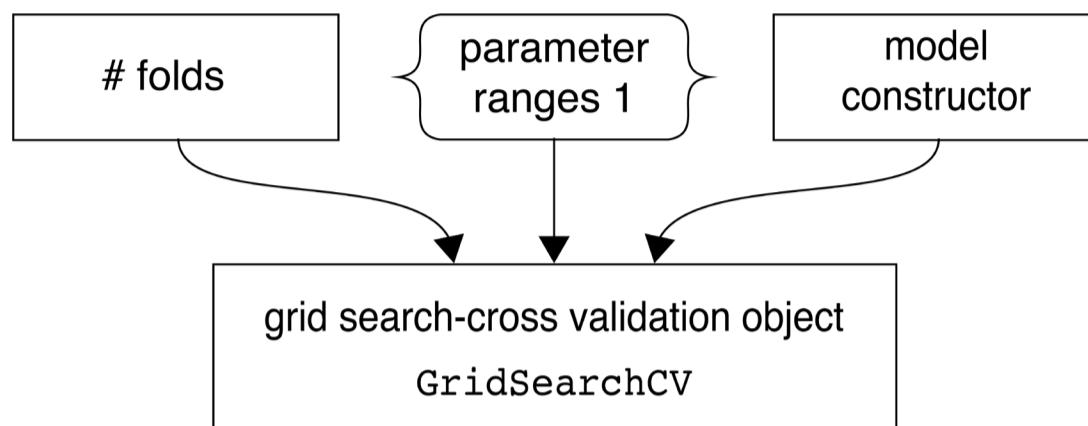


Figure 15.22: The pieces that go into making a basic grid-search object. We supply the number of folds to use during each cross-validation, the parameters we want searched, and the values we want them to take on, and a constructor for the model that we're investigating.

The object takes a number of folds for the cross-validation step (the default is 3), a set of parameters and their values that should be searched, and the routine that builds our learner. There are many more optional parameters that we won't get into.

Conceptually, we can think of the grid search process in two steps. The first step builds a list of every combination of values of the parameters. So if there are just two parameters, we could save this list as a 2D grid. If there are three parameters, we could think of it as a 3D volume, and so on.

The second step runs through those combinations one at a time, building the model, evaluating it with cross-validation, and saving the score. We could save that score in another list (or grid, volume, or bigger structure) of the same shape and size as the list of parameter combinations. Then we can quickly see what score got assigned to each combination.

Figure 15.23 shows a visual summary of the whole process. At the left we see the routine that creates our learning model (perhaps a decision tree algorithm, or a regression algorithm like Ridge). There's also a collection of the parameters we want to vary, and the values we want to

try for each one. Let's assume we're interested in exploring two hyperparameters, and we want to try 7 possible values for the first, and 5 possible values for the other.

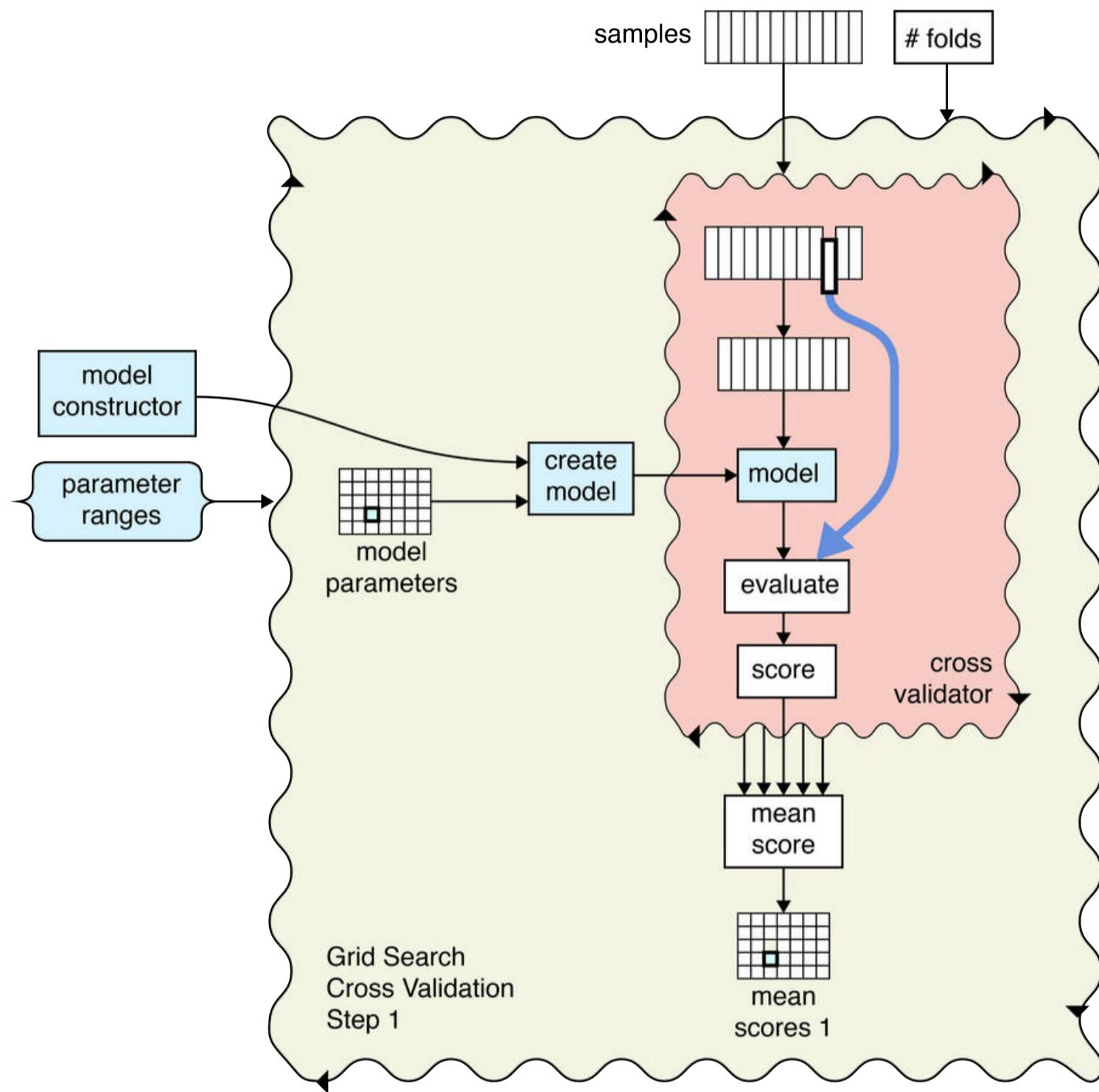


Figure 15.23: Grid searching with cross-validation.

In the first step of processing, we find all combinations of those parameters and values and save them in the grid marked “model parameters.” This grid is 7 by 5, with one entry for each combination of values for the two hyperparameters.

Now we start running the loop. The outer wavy box represents the main loop of the grid searcher. Each time through the loop it uses the model constructor, and one set of model parameters from the grid, to create our model. At the top we can see the samples that make up our training set. If we're using a supervised learner, these samples will have labels. We also provide the number of folds to use when cross-validating.

All of this goes into the inner wavy box, which contains the cross-validation step we saw in Figure 15.18 (though turned on its side). The scores from the cross-validation steps are saved and then averaged, with the result saved in a grid the same shape and size as the model parameters, so it's easy to find the score for each combination of parameters.

Once the searching loop is completed, we look through all the scores saved during the search process and find the parameters that produced the best results. We make a new model using those parameters, train it, and return it, as in Figure 15.24.

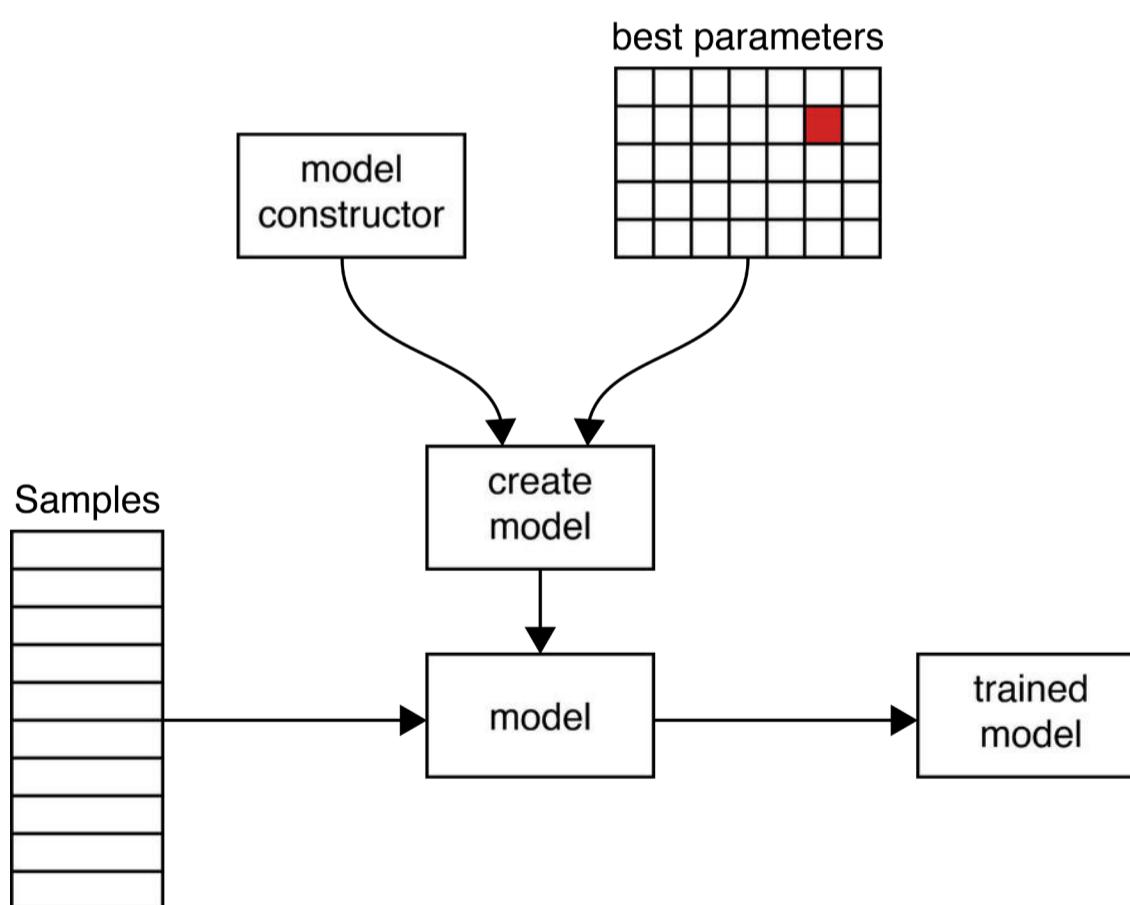


Figure 15.24: We finish the grid-search process by identifying the parameters that gave us the best score. We build a new model from those parameters, train it on all the data, and return that model.

Let's see some code.

We'll use a Ridge classifier from before to classify a bunch of data. The `RidgeClassifier` object takes a lot of arguments, which we've so far been ignoring. Let's pick two of those arguments and search for their best values for a particular set of data.

We'll start with the parameter named `alpha`, which is the regularization strength (this is usually called λ), but it's not unusual to see different Greek letters used for this). Recall from Chapter 9 that regularization helps us prevent over-fitting. In the `RidgeClassifier`, `alpha` is a floating-point value that's 0 or greater. Larger values mean more, or stronger, regularization. As a first guess, we'll try these six values of `alpha`: 1, 2, 3, 5, 10, 20.

The second parameter we'll search for is called `solver`. This is the algorithm used internally by `RidgeClassifier` to do its job. Without going into the details of each one, let's say we'd like to try out a few and see if any one performs particularly well. The documentation for `RidgeClassifier` tells us that we refer to these algorithms by name (that is, with a string) [scikit-learn17g]. Again, mostly arbitrarily, let's pick three: '`svd`', '`lsqr`', and '`sag`'.

Now we need to tell the grid searcher that we want to train and score multiple versions of the `RidgeClassifier`, using these values for the parameters named `alpha` and `solver`.

To communicate which values we want to assign to each parameter, we use a Python **dictionary**. In a nutshell, a dictionary is a list of key/value pairs enclosed in curly braces. Our keys will be the strings that name our parameters, and our values will be lists that the parameters can take on.

Thanks to how Python is structured, we can just name the parameters we want to search on as strings, and use them as the keys in our dictionary. The values we want each argument to take on are named in a list, and assigned to the value for that key. Our dictionary is given in Listing 15.26.

```
parameter_dictionary = {
    'alpha': (1, 2, 3, 5, 10, 20),
    'solver': ('svd', 'lsqr', 'sag')
}
```

Listing 15.26: Building a dictionary to hold the values we want to search over.

When the grid searcher builds a new `RidgeClassifier`, it will assign one value in the `alpha` entry to the `alpha` argument, and one value in the `solver` entry to the `solver` argument. It does this by matching up the names, so the names in our dictionary must exactly match the parameter names used by `RidgeClassifier()`.

The grid searcher will build and score $6 \times 3 = 18$ different models from this dictionary. The models will be made with calls like those in Listing 15.27, though this all happens invisibly to us. Because of how Python's dictionaries are defined, the algorithm might not go through the choices in this order, but that detail will usually also be invisible to us.

```
ridge_model = RidgeClassifier(alpha=1, solver='svd')
ridge_model = RidgeClassifier(alpha=1, solver='lsqr')
ridge_model = RidgeClassifier(alpha=1, solver='sag')

ridge_model = RidgeClassifier(alpha=2, solver='svd')
ridge_model = RidgeClassifier(alpha=2, solver='lsqr')
ridge_model = RidgeClassifier(alpha=2, solver='sag')

ridge_model = RidgeClassifier(alpha=3, solver='svd')
.....
ridge_model = RidgeClassifier(alpha=20, solver='lsqr')
ridge_model = RidgeClassifier(alpha=20, solver='sag')
```

Listing 15.27: The start and end of the 18 models our grid searcher will build and evaluate, based on the dictionary of Listing 15.26. They may not be generated in this order.

To run the search, we make a `GridSearchCV` object with the three items shown in Figure 15.22. These are the number of folds to use, the dictionary of parameters and ranges, and the model constructor. Listing 15.28 shows the code.

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import RidgeClassifier

ridge_model = RidgeClassifier()
parameter_dictionary = {
    'alpha': (1, 2, 3, 5, 10, 20),
    'solver': ('svd', 'lsqr', 'sag')
}
num_folds = 3

grid_searcher = GridSearchCV(estimator=ridge_model,
                             param_grid=parameter_dictionary,
                             cv=num_folds)
```

Listing 15.28: Building a grid searcher. We provide it with the estimator we want it to use (here a `RidgeClassifier`), the parameter dictionary with parameter names and values, and the number of folds to use. We could put the number of folds in the dictionary if we wanted to search different values of that as well. To show that we don't need to, here we're passing it as a constant value.

Hold on a second. Something's fishy here. We saw in Listing 15.27 that the grid searcher is going to make 18 different `RidgeClassifier` objects, each with different parameters. But in Listing 15.28 we made one `RidgeClassifier` and stored it in the variable `ridge_model`, and we gave it no parameters at all. How does the grid searcher take this one object and make 18 new versions of it, each with different parameters?

The answer to this question is also based on how the Python language is set up. Let's look at what's happening at a very high level.

We're providing the searcher the routine that makes a `RidgeClassifier` object as an argument to its `estimator` parameter. As a result, the grid searcher is able to call that routine with new

arguments to make new models. In other words, it can do exactly what Listing 15.27 shows, but using the argument to `estimator`. Since the value we’re assigning to this argument is itself a procedure, invoking that procedure is the same as explicitly invoking `RidgeClassifier()`. This mechanism makes it easy for us to later call `GridSearchCV()` with a completely different estimator if we want, just by changing the routine we provide to `estimator`. If the new model takes the same `alpha` and `solver` parameters, then we can leave our dictionary just as it is. If it takes other parameters, we’d want use a different dictionary.

In short, the grid searcher generates all combinations of the parameters, passing each combination to whatever routine we provided as a value of `estimator` in order to create a new instance of the estimator object. In this case, it will make a new `RidgeClassifier` object. It then runs that object through cross-validation using our data and evaluates its performance.

Creating the `GridSearchCV` object prepares it to run the search, but doesn’t actually start the process. To run the search, we call the `GridSearchCV` object’s `fit()` method. Since in this example we’re training a classifier, we’ll give it our samples and our labels. Listing 15.29 shows the code.

```
grid_searcher.fit(training_samples, training_labels)
```

Listing 15.29: To run the search, we call the searcher’s `fit()` method with our training data and samples.

That’s all it takes. We make that call and then everything happens automatically. When this line returns, the system has exhaustively searched all combinations of our parameters.

We should always pause for a moment before starting a grid search, because we can be in for a long wait. A *long* wait. As we saw before, if we’re searching a lot of parameters, and each run takes a while, we might wait for hours or even weeks `fit()` to finish its work.

When `fit()` is done, we can query our `grid_searcher` object to discover what it found. The object saves its results in internal variables. Each of its variables ends with an underscore to help us avoid confusing those names with our own variables.

The documentation provides a complete list of all the internal variables that contain our results. Let's look at three of the most useful variables. `best_estimator_` (note the trailing underscore) tells us the explicit construction call that the searcher made to make the estimator that resulted in the best score, with all the arguments (including all the defaulted arguments we didn't specify). That best score itself is given by `best_score_`. If we just want the best set of parameters, then `best_parameters_` contains a dictionary that has just the best value for each parameter from our original dictionary.

Let's put this into action. In Figure 15.25 we show some data in a pair of half-moons (generated with the scikit-learn data-making utility `make_moons()`, which we'll see below). On the right we show the results of running the grid search in Listing 15.28 followed by the fitting step in Listing 15.29. Since we're using a `RidgeClassifier`, we'd expect the straight line that does the best job of separating these two classes, though no straight line is doing to do a perfect job.

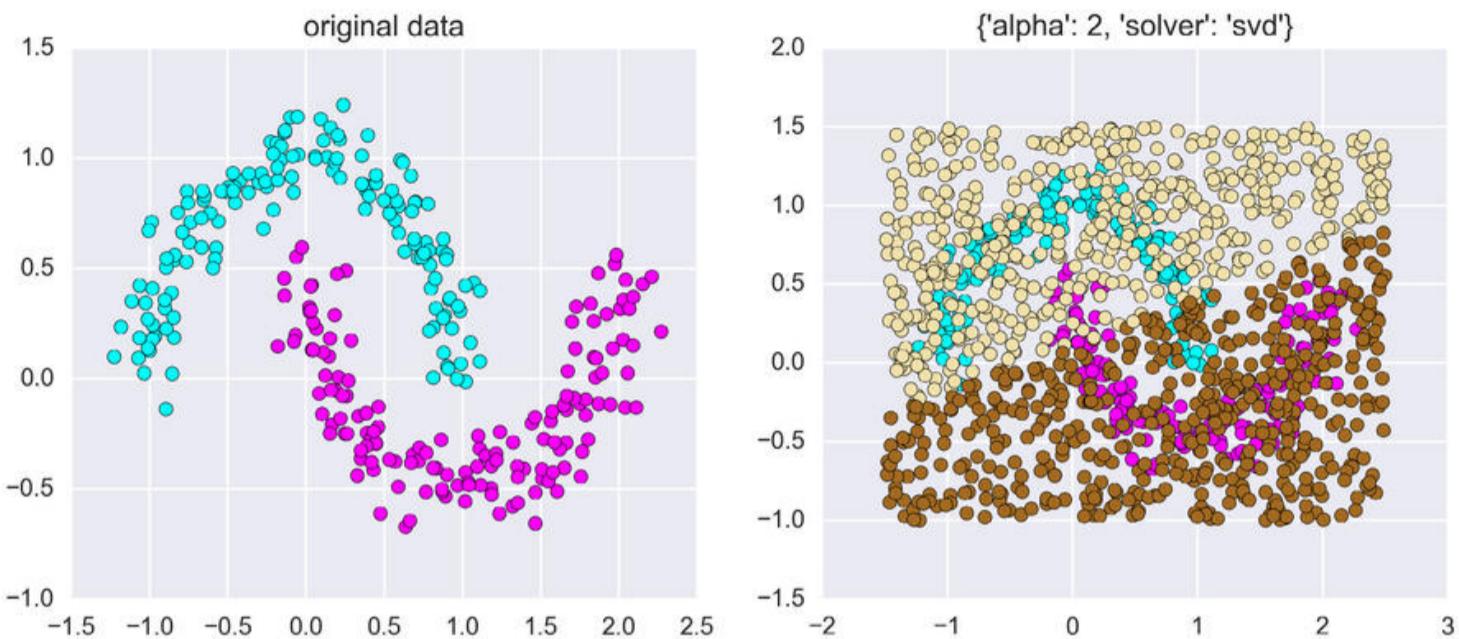


Figure 15.25: On the left, our original training data with 2 categories. On the right, the result of our grid search, with 500 predicted points drawn on top of the original data. The classifier has found a good straight-line approximation to split the data. A more complicated classifier could have found a more complex and accurate boundary curve.

For clarity, Figure 15.26 shows just the test data by itself. Each point is colored by the class it was assigned by the classifier.

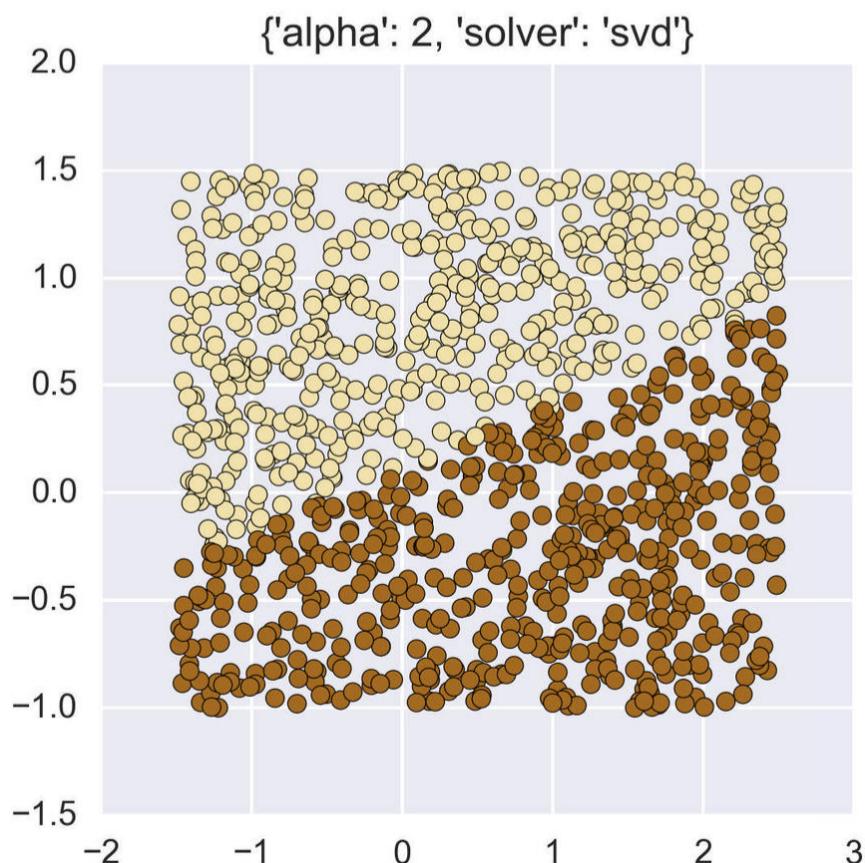


Figure 15.26: Just the test data from Figure 15.25.

In Listing 15.30 we show the resulting values of three variables we just discussed, and their values.

```
grid_searcher.best_estimator_
    RidgeClassifier(alpha=2,
                    class_weight=None, copy_X=True,
                    fit_intercept=True, max_iter=None,
                    normalize=False, random_state=None,
                    solver='svd', tol=0.001)
grid_searcher.best_score_
    0.87
grid_searcher.best_parameters_
    {'solver': 'svd', 'alpha': 2}
```

Listing 15.30: The variables describing the best combination found for our data in Figure 15.25. Output lines produced by Python are colored in light blue. Note that the best estimator provides us with the full call to `RidgeClassifier` with all of its parameters, including the optional ones. The parameter `best_parameters_` is a dictionary, here with 2 key/value pairs.

The output shows that `best_estimator_` contains everything that's in `best_parameters_`, but the latter restricts itself to just the parameters we were searching on.

If we dive deeper into the variables in our `grid_searcher` object we can look at `cv_results_`, which is a huge dictionary with detailed results from the search. In Figure 15.27 we've plotted data from two entries from that variable, `mean_train_score` and `mean_test_score`, which give us the scores for the training and testing data for each of the 18 parameter combinations (these names don't end with an underscore, because they are members of the `cv_results_` dictionary which does have an underscore. It's a quirk of scikit-learn's naming policy). We can see that in this run the three combinations of the parameters gave us the same, best testing results. That is, the choice of `solver` didn't make any difference in this situation.

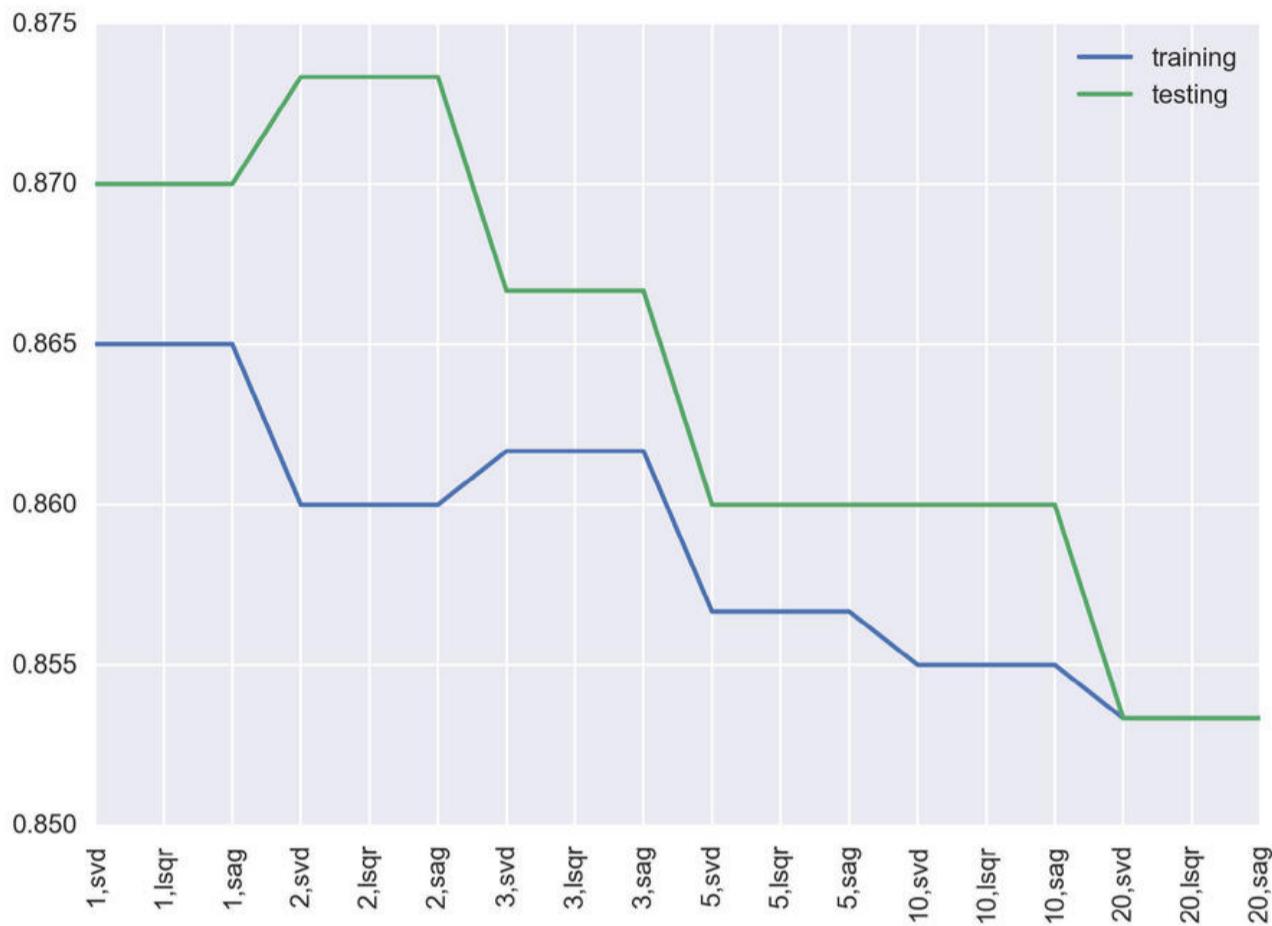


Figure 15.27: Training and testing scores for each of our 18 searches. The best training results came from using `alpha=2` and any of the three solver choices.

The results show that using an `alpha` value of `2` produced the best results on the testing sets, regardless of the algorithm used internally by `RidgeClassifier`. Since the testing data is a proxy for real data, we'd want to use `alpha=2` if we were going to deploy this algorithm. The system picked the `svd` algorithm in the “best” variables from the 3 equally-performing combinations because it was the first one encountered.

15.9.4 Random Grid Search

Now that we know how to do the exhaustive version of grid searching, making the switch to the random version is almost no work at all.

Instead of using a `GridSearchCV` object, we use a `RandomizedSearchCV` object (it also comes from the `model_selection` module).

This object takes the same arguments as `GridSearchCV`, but the randomized version also takes an argument called `n_iter` (for “number of iterations”), telling it how many unique, randomly-chosen parameter combinations it should try (it defaults to 10).

15.9.5 Pipelines

Suppose that as part of our grid search, we’d like to include some form of data pre-processing. We might want to normalize or standardize the training set, or do something more complex like running PCA on it.

It would be natural to search for the best parameters for that transformation while we’re searching for other parameters, like those we saw above. For example, we might want to try using PCA to reduce an 8-dimensional data set down to 5, 4 and 3 dimensions, and see which (if any) of those choices give us the best performance.

But now we have two objects inside the loop: the pre-processing object and the classifying object. How do we tell the searcher, whether systematic or randomized, to use both of these, and how do we tell it which parameters should be delivered to which object?

The answer is to package up the whole sequence of actions we want performed into a **pipeline**. Then the searcher will proceed as usual, only instead of just calling an estimator, it will call the pipeline, and execute all the steps inside of it.

Let’s demonstrate this using the same half-moons data we used in Figure 15.25.

To illustrate the pre-processing step, let’s transform our data every time through the loop in such a way that the `RidgeClassifier` object will be able to fit a curve to the data, rather than just a straight line.

To do this, we’ll use a pre-processing step to produce *more data* for the classifier. This is a technique we haven’t seen before, so let’s see what it does before we go into the code to create and use it.

When a `RidgeClassifier` object finds a boundary curve, it builds it from combinations of the features in the samples. If all we give our classifier are the x and y values of the two features in our samples, as we did earlier, then looking at the math we'd see that the most complex shape the classifier can build from combining them is a straight line. In other words, the reason we got a straight line from `RidgeClassifier` in Figure 15.25 isn't because of the classifier, but because the data we gave it had only the two features of x and y .

If we create some additional features out of the original data, then the classifier will have more to work with. We'll make those features by multiplying together the x and y values in different ways.

For instance, we can multiply the y value with itself a bunch of times. This would make $y \times y$, or $y \times y \times y$, or $y \times y \times y \times y$, and so on. Mathematicians call these **polynomials**. The number of features that are used in these little expressions is called the **degree** of the polynomial. Figure 15.28 shows the shapes of these polynomials made up of repeatedly multiplying x by itself.

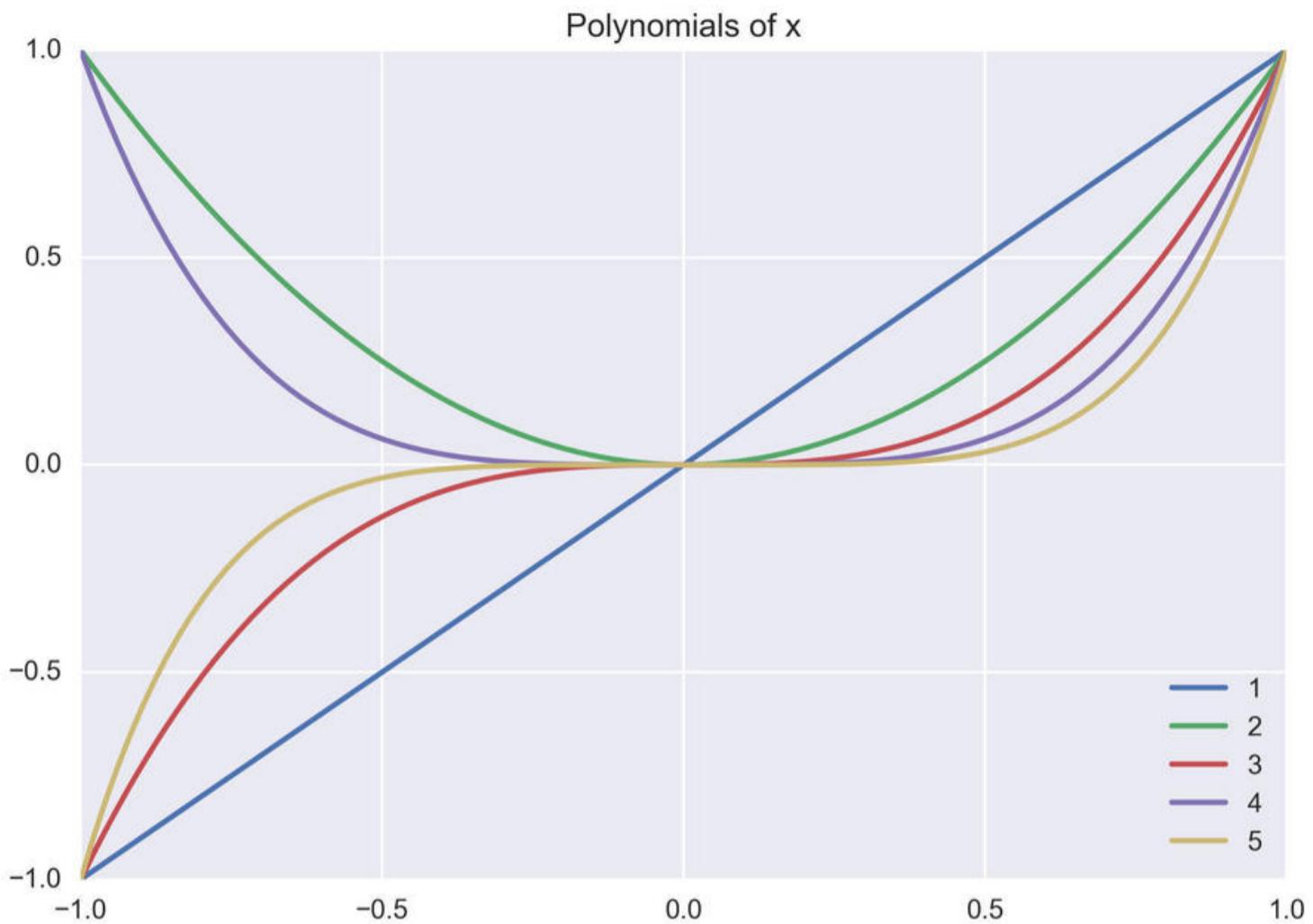


Figure 15.28: The polynomials of x plotted from -1 to 1. The number in the legend is the degree, and tells us how many variables are used to make that curve. So x by itself is a polynomial of degree 1, $x \times x$ is of degree 2, $x \times x \times x$ is of degree 3, and so on. Note that the odd-numbered polynomials are both negative and positive, while the even-numbered polynomials are strictly 0 or larger.

We've shown just polynomials built from x , but we can also build versions of y multiplied by itself any number of times. And we can also mix x and y together.

For example, the three possible second-degree polynomials are $x \times x$, and $y \times y$, and $x \times y$. When we get up to the third degree we have four types of expressions: $x \times x \times x$, and $x \times x \times y$, and $x \times y \times y$, and $y \times y \times y$. There are only four combinations because the order in which we multiply the values doesn't matter to the end result.

As we mentioned, when the Ridge classifier only gets the polynomial of degree 1 for both x and y (that is, we give it just the values of x and y themselves), the most complex shape it can make is a straight line. If

we also give it the polynomials of degree 2 (as we saw, these are $x \times x$, and $y \times y$, and $x \times y$), then it can combine these to make a more interesting curve. The higher the degree of the polynomials we give to the Ridge algorithm, the more complex a curve it can create.

Figure 15.29 shows just a few combinations of the curves in Figure 15.28 (these curves contain only polynomials of x). This shows that by adding up several of these curves, each with its own strength, we can make some pretty complicated shapes.

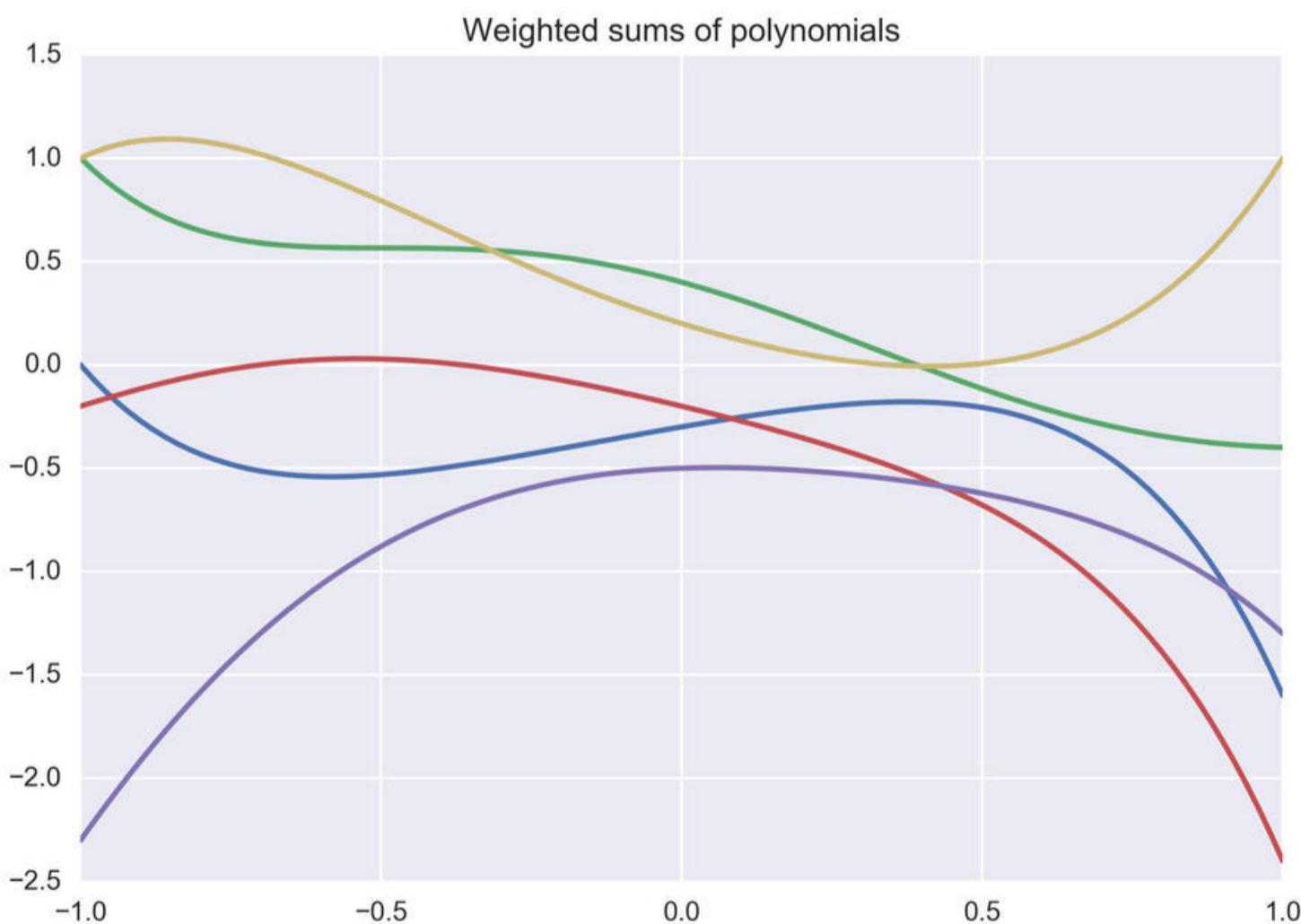


Figure 15.29: Some weighted combinations of the polynomials of x in Figure 15.28. At each point we find the values of the 5 curves, multiply each value by a scaling factor for that curve, and add up the results.

Figure 15.29 shows curves using just the x value. Things get really interesting when we use polynomials in both x and y simultaneously. Without going into the details, we can follow a recipe like that in Figure 15.29 to create a wide variety of 2D curves. Figure 15.30 shows a few examples.

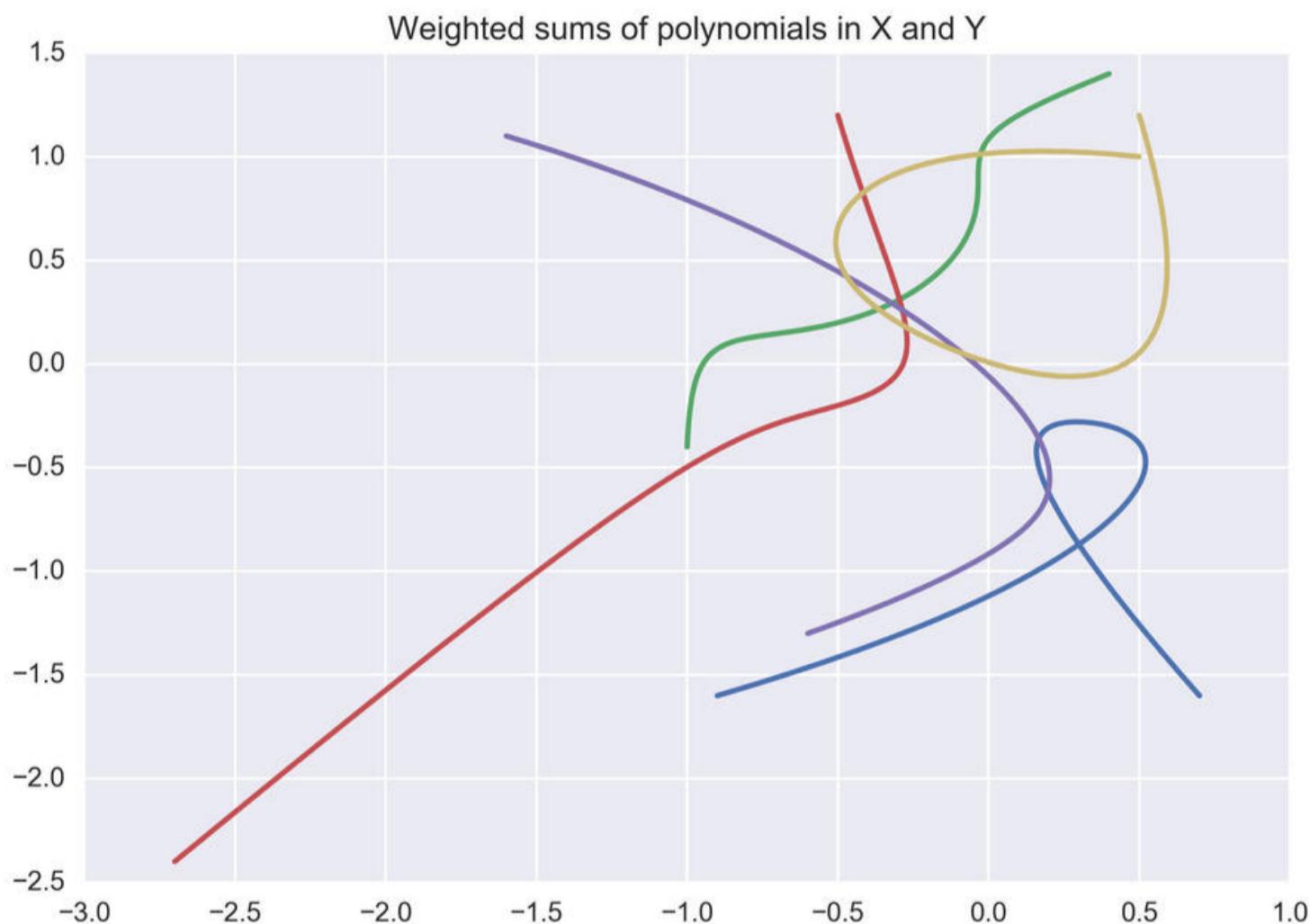


Figure 15.30: Weighted combinations of polynomials in both X and Y.

If we give `RidgeClassifier` not just the x and y values, but these polynomials that come from multiplying those values together in different ways, it can create these kinds of curves. This is much better than a straight line!

We say that by creating these additional features, we're **extending** or **augmenting** our original list of features with new **polynomial features** made from the original values of x and y . In our Numpy array of data, it just means that each row gets longer, from having only 2 features to having more of them, computed by multiplying various combinations of x and y .

The first step of our pipeline will be to build these new polynomial features for every sample, so the Ridge classifier will have them to work with. Now we're ready to identify the new object that will do the job, named `PolynomialFeatures`. We tell it the degree of the features we want it to make, and it cranks them out and appends them to each sample. The more polynomials we include with each sample (that is, the higher their degree), the more complicated Ridge's boundary curve can become.

Wait a second. When we create these polynomial features, we're not including any new information to our data set. We're just using what we have and multiplying the values together. How does that give us curves rather than straight lines?

The answer comes down to the details of how these algorithms are implemented. It's true that we're not providing any new information. But `RidgeClassifier` is designed to work with the data it gets, not all possible variations on that data that might help it produce a better answer. So if we give `RidgeClassifier` our simple 2-feature data, it finds a line. By providing it with the polynomials, it uses those as part of its calculations to find a curve. Conceptually, we could have an argument to `RidgeClassifier` that we could use to tell it to create this extra data all by itself, internally. But scikit-learn is instead set up to make it our responsibility to create that extra data. That way we have complete control over just what we give to `RidgeClassifier`, and thereby control the complexity and shape of the boundaries it can find.

But how much more data is best? As we saw in Chapter 8, at a certain point these increasingly complicated curves can start to overfit the data. So we'll want to stop including new features before that happens. On the other hand, we have the regularization parameter `alpha` in the Ridge classifier which helps us control overfitting. Maybe by increasing the value of `alpha` we can use more of these combined features, and thus use a more complicated (and perhaps better-fitting) curve.

This is getting very complicated! What's the best balance between curve complexity (given by the parameter `degree` to `PolynomialFeatures`), and regularization strength (given by the parameter `alpha` to `RidgeClassifier`)?

There's no need to try to work this out ourselves. Let's just build a two-step pipeline from these two objects and give the searcher a bunch of values to search. Then it can do the work to find which combination works best. If we use the grid searcher, it will try every combination of every parameter.

Figure 15.31 shows a grid searcher with a two-step pipeline replacing the single model we had in Figure 15.23. The first step creates a `PolynomialFeatures` object, and the second creates a `RidgeClassifier` object that uses the augmented data that comes out of the first step. This is a simplified diagram, because the fold we're using for validation is going through a transformation (marked T) that doesn't seem to be coming from anywhere. We'll return to this diagram in the next section and fill in that gap.

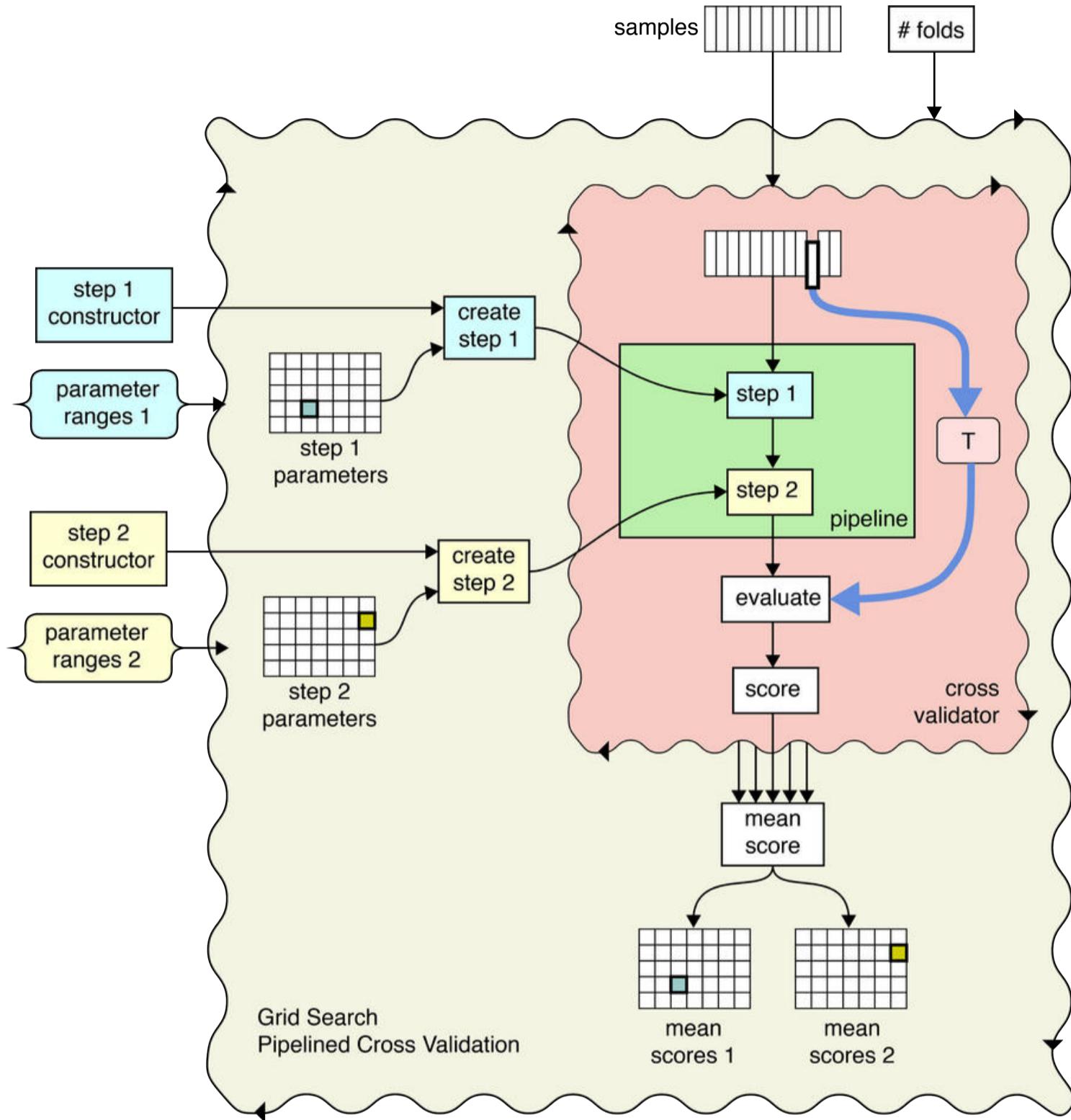


Figure 15.31: A simplified version of our grid search object with a pipeline replacing the single model of Figure 15.23. Note that we're not showing where the fold's transformation comes from, which we'll come back to later. Each step in the pipeline can have its own set of parameters to be searched. As before, each wavy box represents a loop.

Our first step in building our pipeline will be to create our `PolynomialFeatures` object, which produces those extra combinations of x and y . The only argument we care about now for this object is `degree`, which tells it how many polynomials to build from the original

features. Larger values of `degree` will generate and save more of these multiplied-together features, for every sample, which will let the Ridge classifier find more complicated boundary curves.

For the Ridge classifier, let's also search for just one parameter, the regularization strength `alpha`. We'll leave the internal algorithm at its default value (that's the string `auto`, which automatically picks the best algorithm).

Now that we know the two steps that make up our pipeline, let's write the code to create it.

Scikit-learn offers more than one way to build a pipeline. We'll take the approach that's easiest to program and use.

We begin by making the objects that will go into our pipeline. In this case, it's the `PolynomialFeatures` object and the `RidgeClassifier` object. We can make them as in Listing 15.31.

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import RidgeClassifier

pipe_polynomial_features = PolynomialFeatures()
pipe_ridge_classifier = RidgeClassifier()
```

Listing 15.31: We start making our pipeline by creating the objects that will go into it.

As before, neither of these objects has any arguments, because the searcher will fill those in for us as it searches.

Now that we have our objects, we can build the pipeline. To do this, we create an object named `Pipeline`, and we hand it a list that contains one entry for each step. Each entry is itself a list with two elements: a name we pick for that step, and the object that implements it.

The pipeline is built from this list of objects, *in the order in which we name them*. The list can be as long as we like, as long as we have a name and object for each step.

Listing 15.32 shows this construction step for our two-step pipeline. Notice the order of the objects. We name the `PolynomialFeatures` object first because it comes first in our intended sequence of steps.

```
from sklearn.pipeline import Pipeline
pipeline = Pipeline([('poly', pipe_polynomial_features),
                     ('ridge', pipe_ridge_classifier)])
```

Listing 15.32: To create our pipeline object, we give `Pipeline()` a list of steps. Each step is itself a list, containing a name we give to that step, and the object that performs it. Our objects are those we just created in Listing 15.31.

The names we pick when we make the pipeline are like the names we pick for variables: they can be anything we like, but it's best to pick something descriptive. We're using very short names here to conserve space.

We've completed our pipeline, and we will soon hand this to a `GridSearchCV` object as the value of its `estimator` argument.

The last thing to do is make the dictionary of parameters for the searcher to build its combinations from.

If we make our dictionary of parameters for the pipeline in just the same way as we made our dictionary before, sooner or later we'll hit a problem. For example, we know that `RidgeClassifier` takes an argument called `alpha`. What if the `PolynomialFeatures` object also took an argument called `alpha` (it doesn't, but it could)? And what if we wanted to use values `(1,2,3)` for the first `alpha`, and `('dog', 'cat', 'frog')` for the second `alpha`? We need some way to tell the grid searcher which set of values should go to which parameter in which object.

The way scikit-learn answers this question is, frankly, pretty weird. We name each parameter using the name of the pipeline object for that parameter (that's why we gave our objects names when we made the pipeline), followed by the name of the parameter. Note that we don't

use the name of the object (in our example, `pipe_polynomial_features` or `pipe_ridge_classifier`), but the name of the *pipeline step* (in our example, `poly` or `ridge`).

The weird bit is that we join the pipeline object's name and the parameter's name with **two underscore characters**. That is, two `_` in a row, or `__`. This is easy to confuse with just one underscore, which is unfortunate. But two underscores must be used.

So to refer to the `alpha` parameter for our pipeline step named `ridge`, we'd name it in the dictionary as `ridge__alpha`, with two underscores. Similarly, the `degree` parameter for our `poly` pipeline step is named `poly__degree`, again using two underscores.

We always create our dictionary names by assembling the pipeline step name, two underscores, and the parameter name, even when there's no chance of confusion.

Listing 15.33 shows a dictionary for those two parameters, along with some values we'll try for them.

```
pipe_parameter_dictionary = {
    'poly__degree': (0, 1, 2, 3, 4, 5, 6),
    'ridge__alpha': (0.25, 0.5, 1, 2)
}
```

Listing 15.33: A dictionary for our pipeline. Note that each parameter is given by the name of the pipeline step and the name of the parameter, joined with two underscore characters.

This will cause the loop to run $7 \times 4 = 28$ times, but for this small dataset that takes only a few seconds on 2014-era iMac, without even touching the GPU.

Now we're set. We just build our search object as before, and then call its `fit()` routine. Listing 15.34 shows the code.

```
pipe_searcher = GridSearchCV(estimator=pipeline,
    param_grid=pipe_parameter_dictionary,
    cv=num_folds)

pipe_searcher.fit(training_samples, training_labels)
```

Listing 15.34: Building our grid search object for our pipeline, and then executing the search.

The results for our half-moon data are shown in Figure 15.32. The best combination the system found was `alpha=0.25` and `degree=5`. Thanks to the additional features we provided to `RidgeClassifier`, it was able to find a curve to split the data.

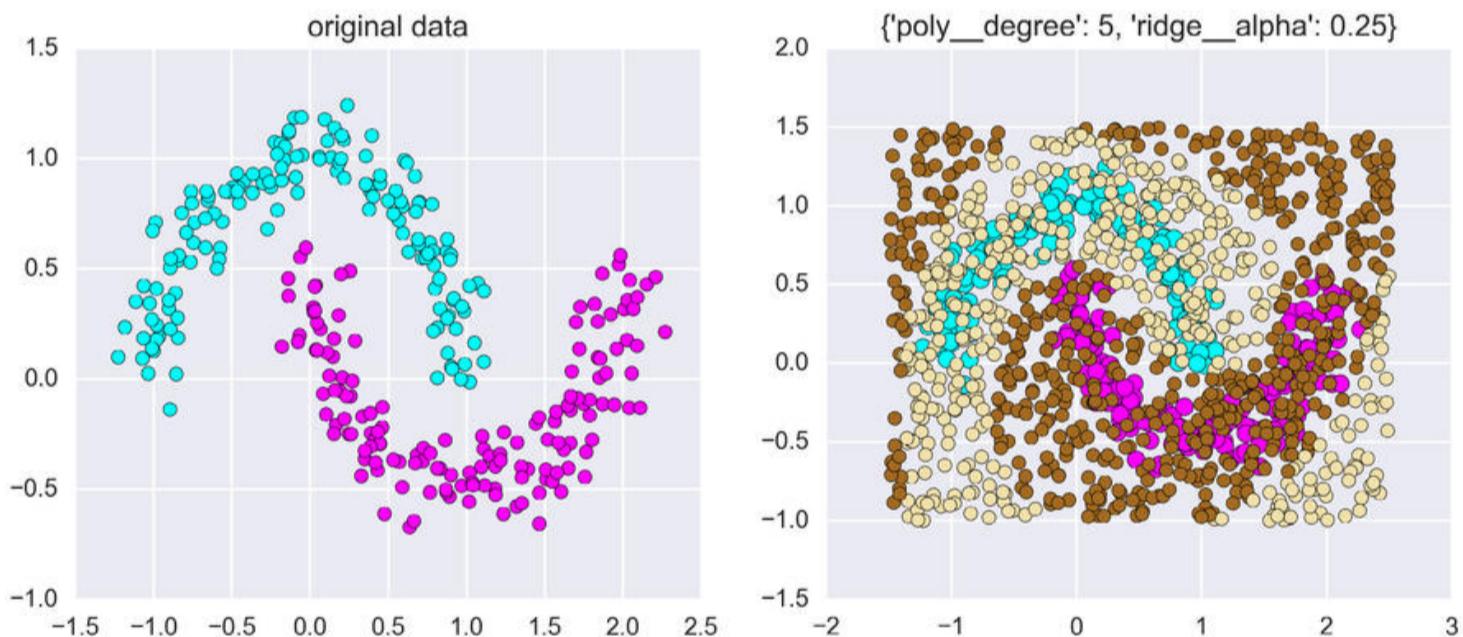


Figure 15.32: Results for searching over our more flexible pipeline object.

For clarity, Figure 15.33 shows just the test data by itself.

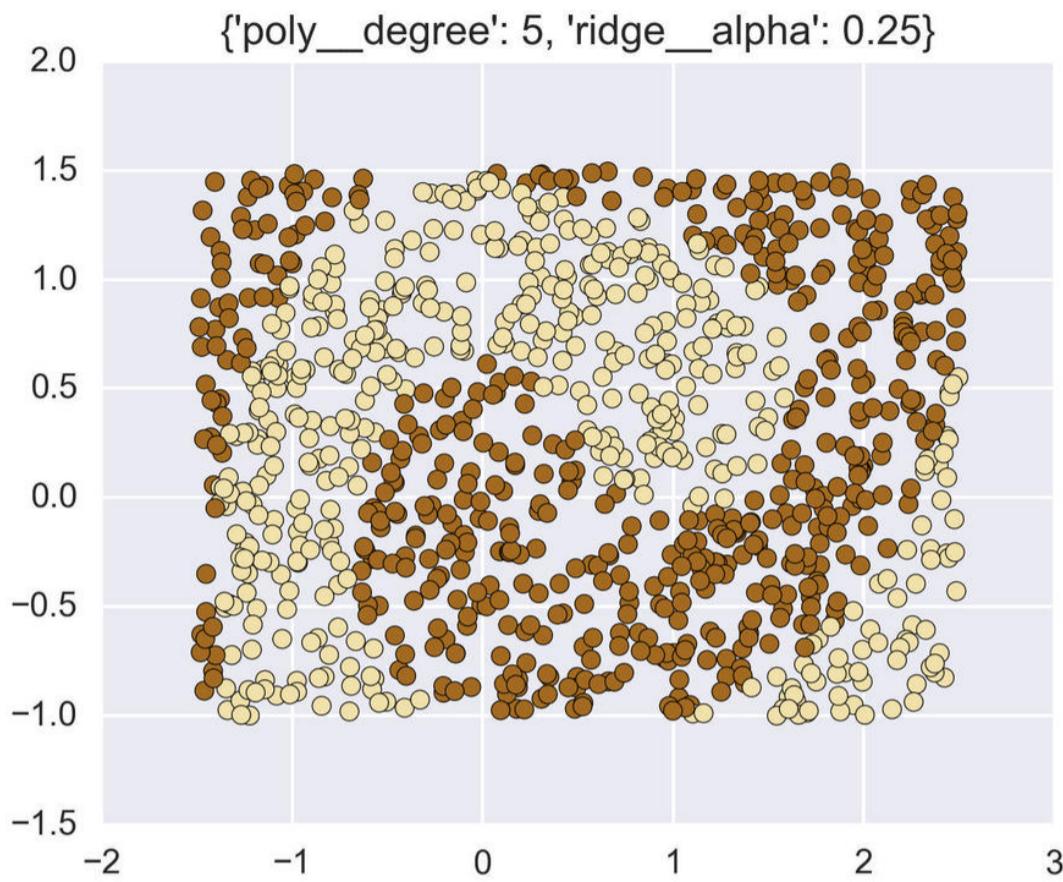


Figure 15.33: Just the test data from Figure 15.25.

It's interesting to note that the boundaries are rather symmetrical, which makes sense given the symmetry in the input. We also see them seeming to curve around and re-appear in the corners. This is kind of wild, but entirely permitted by our data. After all, as long as we're classifying all the data correctly (and we are), it doesn't really matter what happens elsewhere (though we probably prefer simplicity when we can get it, just on general principles [Domingos12]).

Since the dots only give an impression of the boundary, let's look at it in high resolution in Figure 15.34. Just to see what things look like away from the data, we'll also zoom out and look at the boundary curve with a larger range of values.

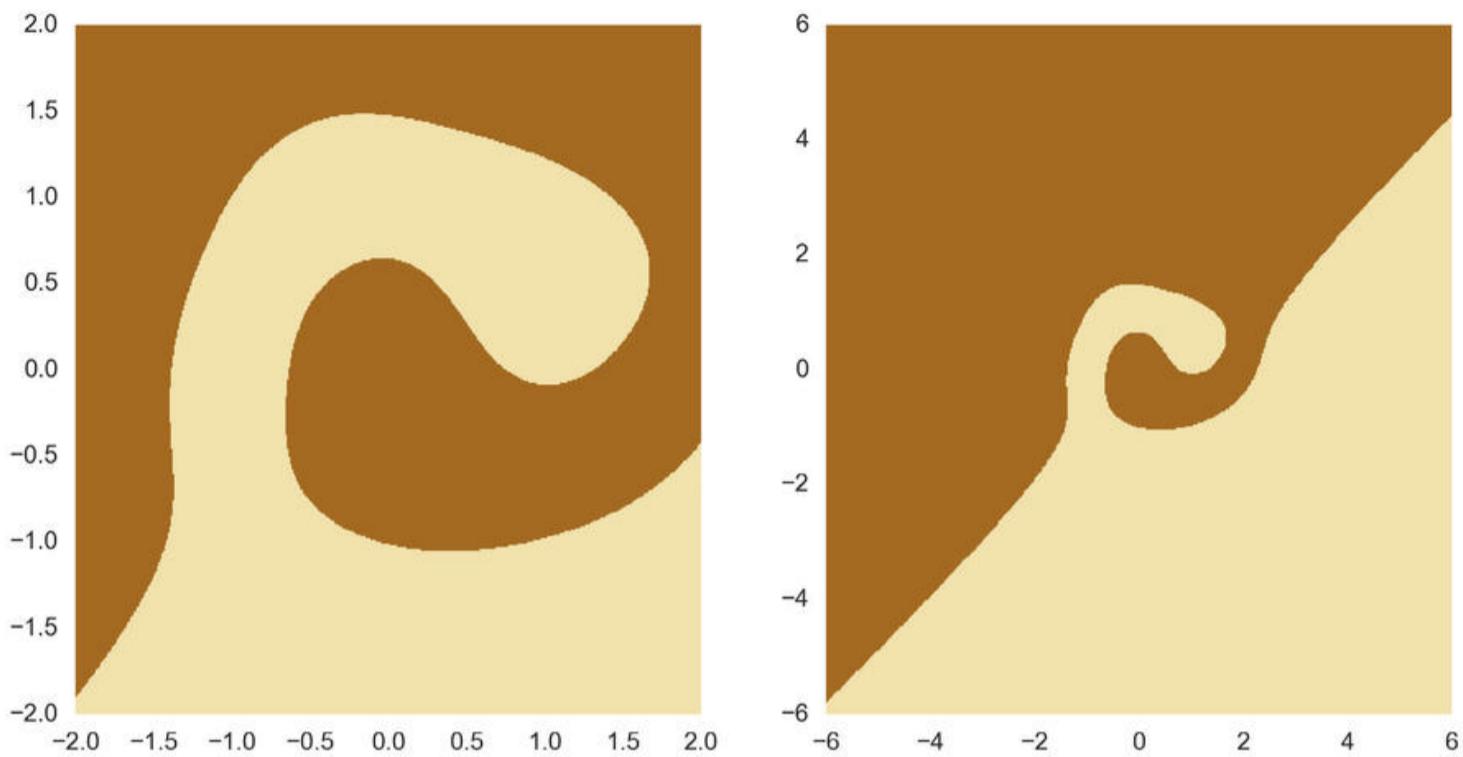


Figure 15.34: Looking at the boundary curves from Figure 15.32 and Figure 15.33. Left: Close up to the data of the two moons, but with a slightly larger range than Figure 15.33. Right: A larger plotting area.

A graph of the scores for the different combinations is shown in Figure 15.35.

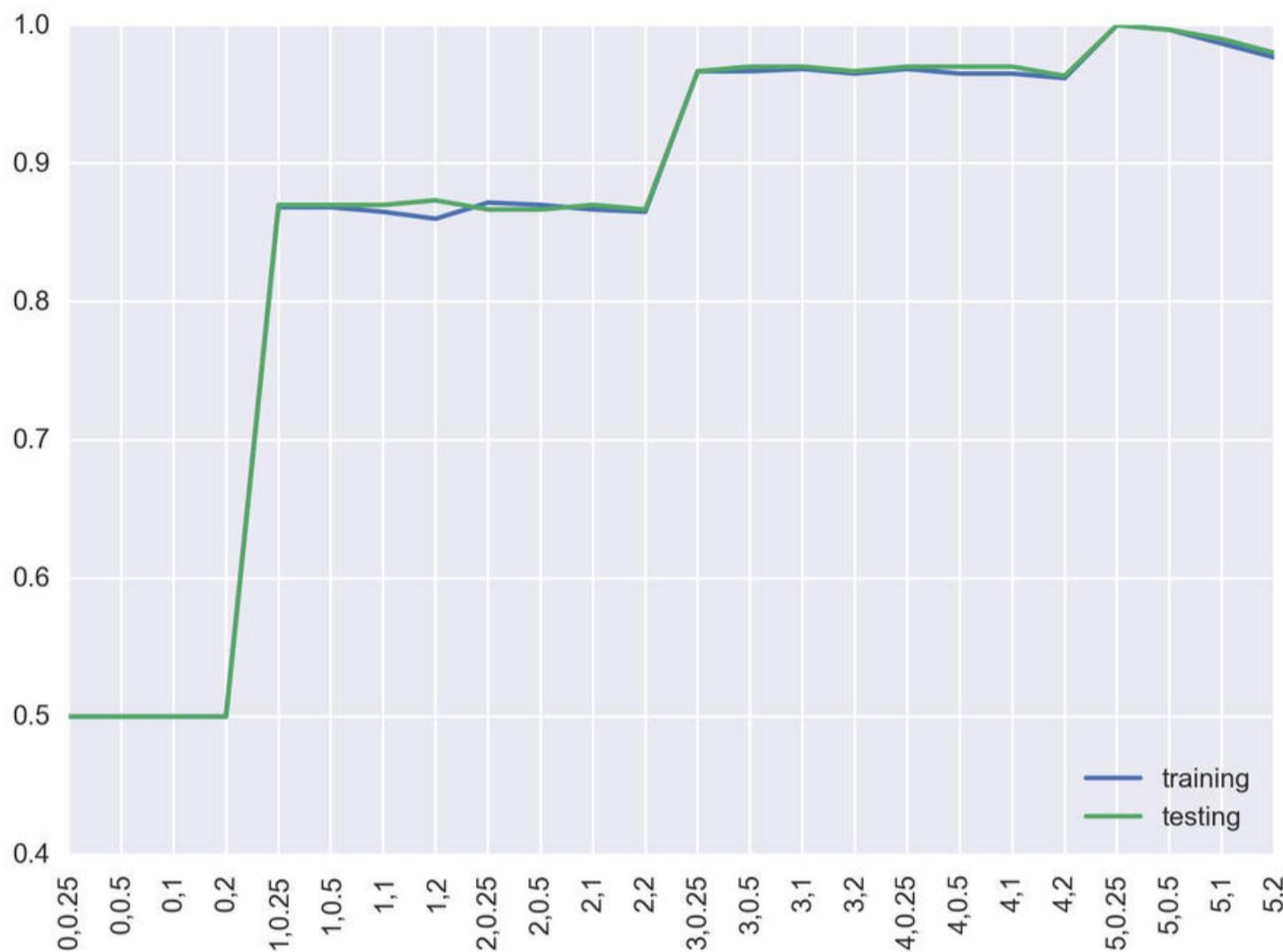


Figure 15.35: Scores from our pipeline search. The straight-line version is shown at the far left, for degree values of 0. The best results came from setting degree to 5, and alpha to 0.25.

The best combination of `alpha` at 0.25 and `degree` at 5, located near the right end of the graph, produced a perfect training and testing score of 1.0. As we might expect, the regularization parameter didn't have much effect until the curve got complicated enough to start overfitting, and even then on this simple example it didn't make much of a difference.

15.9.6 The Decision Boundary

Let's take another look at the decision boundaries we've found for our half-moon data.

In Figure 15.25 and Figure 15.26 we found a linear boundary. In those figures, we drew each dot with the color of the class returned by `predict()`. But as we mentioned before, we can get the relative confidences of the classifier from `decision_function()` and `predict_proba()`. So instead of getting just a single integer telling us which class is assigned to each input, we get back a confidence value for each category. Since we have only 2 classes, this means we'll get two floating-point values, one for each category.

Let's plot the values of `decision_function()` for our straight-line fit in 3D. Figure 15.36(a) shows the result.

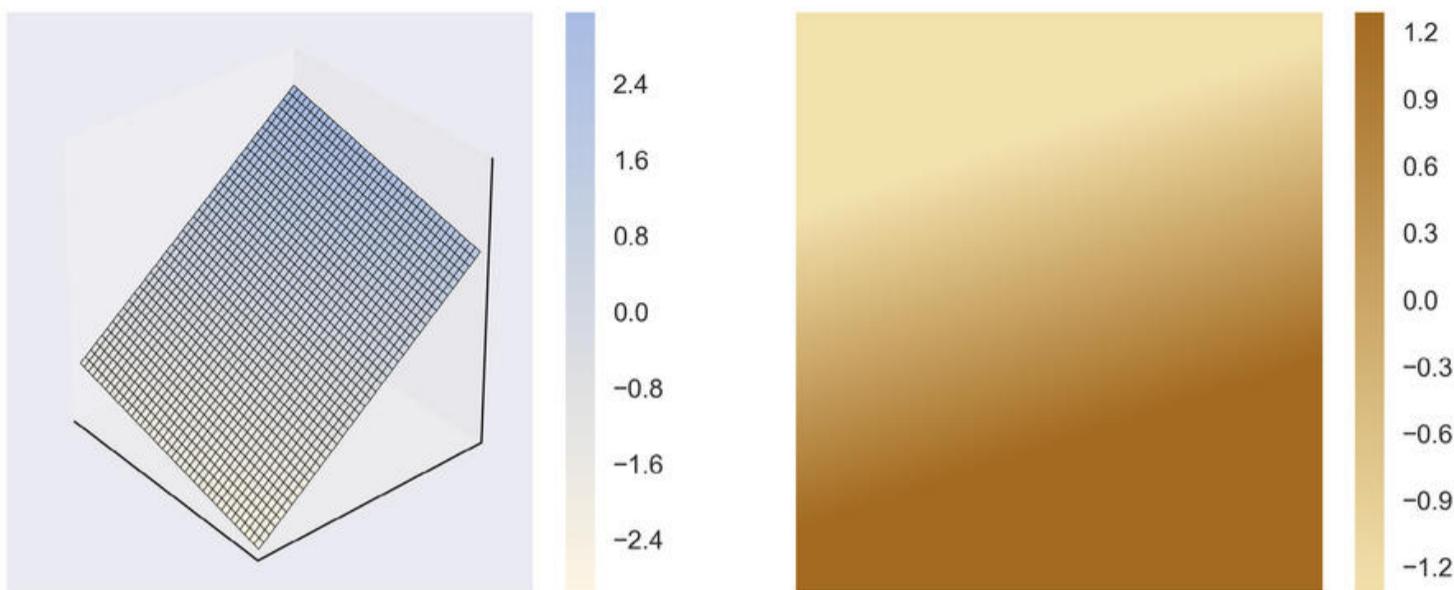


Figure 15.36: Looking at the confidence values for the linear boundary between our two half-moons, as shown in Figure 15.25 and Figure 15.26. When we look at the confidence of each category, rather than asking for just the most likely one, we can see that there's a transition from one to the other. Left: The output of `decision_function()`. Right: A top-down view of the 3D plot.

As we might expect, the boundary from one class to the other isn't instantaneous. This surface is really a single flat plane with no creases.

Let's look at the values from `decision_function()` for our curved boundary in Figure 15.33 in the same way. We'll plot these in 3D, as in Figure 15.37, and then look down on the plot, as in Figure 15.37.

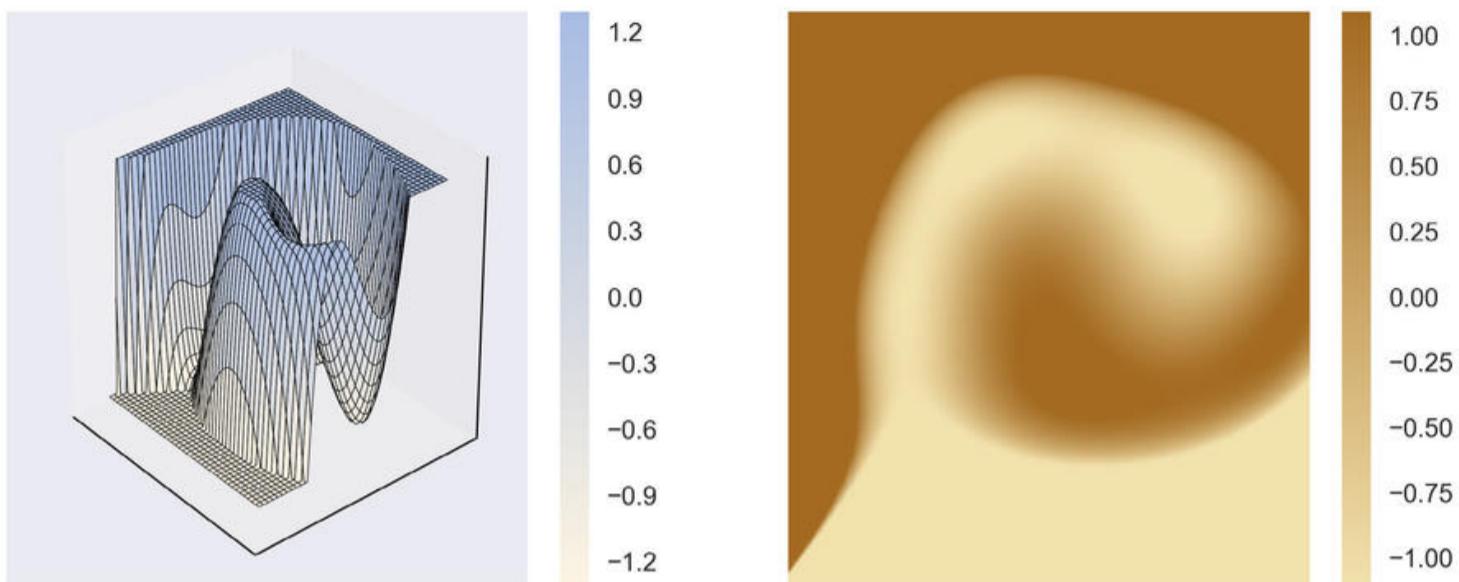


Figure 15.37: The boundary in 3D. The flat regions are because we've clamped the values to $[-1, 1]$ to better see the structure near the middle. Left: A 3D view. Right: A top-down view of the 3D plot.

We can see that there's a smooth transition zone between the two categories (we manually clamped the output of `decision_function()` to $[-1, 1]$ so we could focus on what's happening in that range). In the crossover regions, the probability of a point being in one class or the other varies smoothly.

In some circumstances, getting the most likely class from `predict()` is just what we want. In others, getting the more refined confidences from `decision_function()` (or `predict_proba()`) can be more useful.

15.9.7 Pipelined Transformations

We promised to return to a gap in Figure 15.31, where we showed the fold going through a transformation, but not where the transformation came from. Let's address that now. This will also pay off our earlier promise to show how to use pipelines to correctly apply a transformation while doing cross-validation.

Let's continue with our previous example of using `PolynomialFeatures` followed by `RidgeClassifier`. The first step in our pipeline adds new polynomials to each sample, so it counts as a *transformation* of our training data, since it changes the samples that go into the classifier. In

this example, `PolynomialFeatures` is including new features, rather than changing the features themselves like a scaling transform would. But the samples are changing, so we call it a transformation.

Remember our cardinal rule about transformations: anything we do to the samples in the training data must also be done to all other data.

Since the transformation of our training data (adding new features) is happening inside the pipeline, we have to pull that transformation out so we can apply it to the validation fold. Figure 15.38 shows a close-up of the pipeline in the cross-validation step, where step 1 (our `PolynomialFeatures` object) computes a transformation, and then that transformed data is applied to both the training data used by step 2, and the validation fold used to evaluate the result.

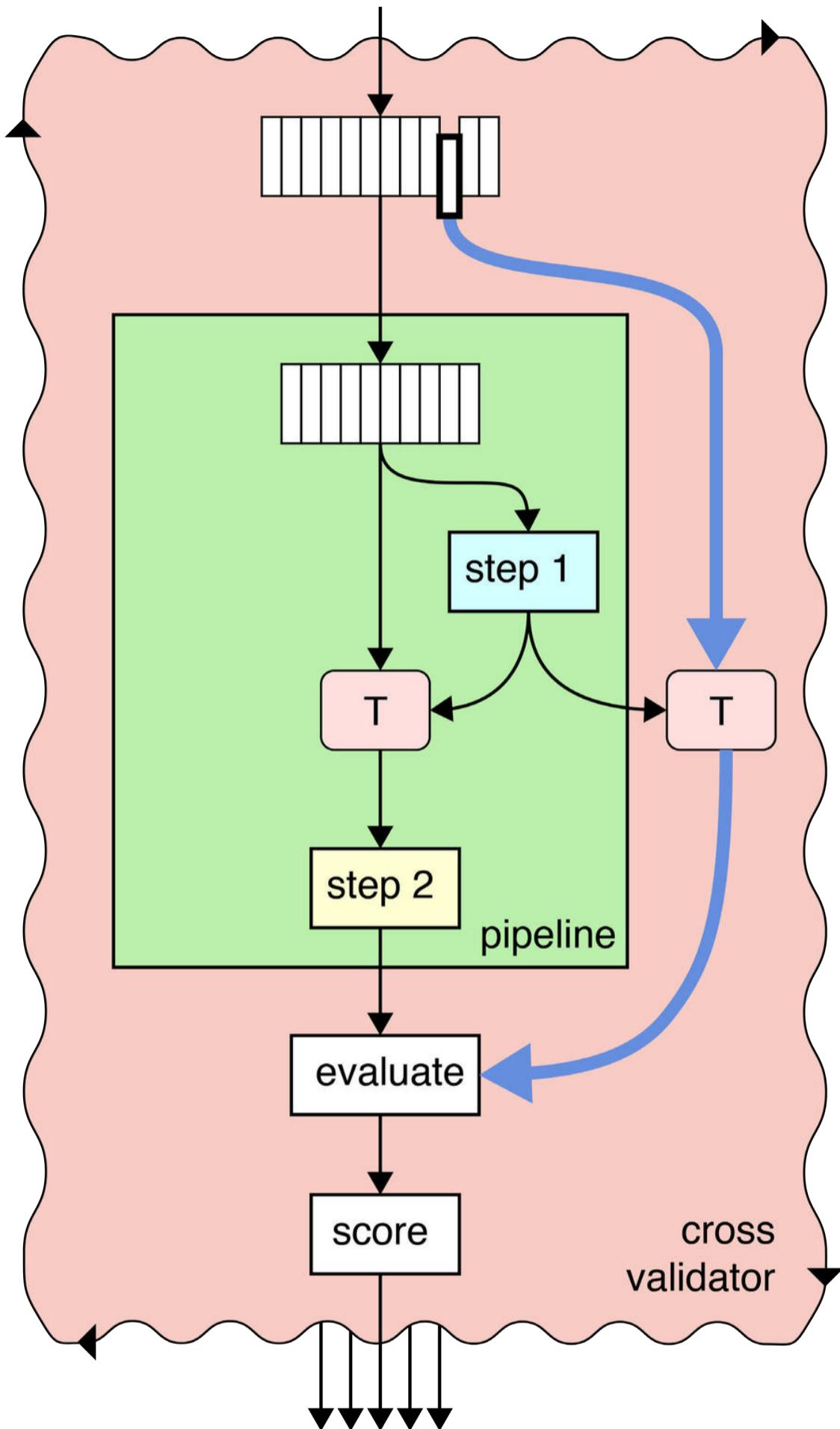


Figure 15.38: A close-up of the cross-validation step in Figure 15.31, filling in the missing source of the transformation to the fold. It's coming from the transformation step inside the pipeline, where the transform is applied to both the training data and the fold.

There's no leaked information here, because we're doing everything by the book. For each model, we extract a fold, compute the transformation on the remaining training data, and then apply that transformation to both the training data and the data in the fold.

This application of the transformation is carried out for us automatically by both the regular and random grid search objects, so we don't have to lift a finger, or change our code from the previous section in any way. In other words, scikit-learn applies the steps in the pipeline in just the right way, automatically.

We discussed this detail because it's a great illustration of how we have to always think about information leakage any time we touch our sample data. It also shows us a nice way to handle this issue. And finally, this example shows how our libraries make our lives easier by handling these issues for us.

We can use multiple transformations in our pipeline. For instance, we might add a `MinMaxScaler` after the `PolynomialFeatures` object. We might also then include a `PCA` object to reduce the dimensionality of the data. We can have as many of these data-transforming steps as we like, and they will all get applied in sequence to both the training and fold data.

15.10 Datasets

The code in this section is in the Jupyter notebook
`Scikit-Learn-Notebook-8-Datasets.ipynb`

Scikit-learn provides a variety of datasets that are clean and ready for immediate use [scikit-learn17h]. Some of these are **real-world** data representing actual field studies, and some are **synthetic** data that is generated procedurally when we ask for it.

For example, we can easily import the famous *Iris* dataset, which describes the lengths of different petals of several species of iris flower. This is a classic database that's used in many discussions of categorization. Scikit-learn also includes the *Boston housing* dataset that records the prices of homes in the Boston area for a period of years, along with other geographical information. This is often used in discussions of regression.

There are other famous datasets as well. For example, the *20newsgroups* dataset provides text data from online discussions, which is frequently used for text-based learners. The *digits* dataset contains small grayscale images of handwritten digits from 0 to 9. And the *Labeled Faces in the Wild*, or *LFW*, dataset provides labeled photographs of people that are useful for face detection and classifying images.

Most of these datasets are returned from NumPy in arrays that are ready for immediate use, but always check the documentation to learn of any idiosyncrasies or exceptions.

To load a dataset, we usually need only to call its loader and save that routine's output in a variable. For example, we can load the Boston data as in Listing 15.35.

```
from sklearn import datasets
house_data = datasets.load_boston()
```

Listing 15.35: Loading the Boston dataset.

The arguments for synthetic datasets are important because they help us control the size and shape of the data that's made.

Some of the most popular synthetic dataset creators are `make_moons()` which generates the two interlocking arcs we used earlier, `make_circles()` which creates a pair of nested circles, and `make_blobs()` which makes sets of points drawn from Gaussian distributions.

As an example, Listing 15.36 shows how we call `make_moons()`. We tell it how many points to make, and we can optionally add a `noise` parameter to break up the uniformity.

```
from sklearn.datasets import make_moons
(moons_xy, moons_labels) = make_moons(
    n_samples=800, noise=.08)
```

Listing 15.36: Using the `make_moons()` routine to generate 800 points of synthetic data.

Figure 15.39 shows these three types of synthetic data, using their defaults and with noise added. Note that each routine provides labels along with the point locations, so the data is ready for classification.

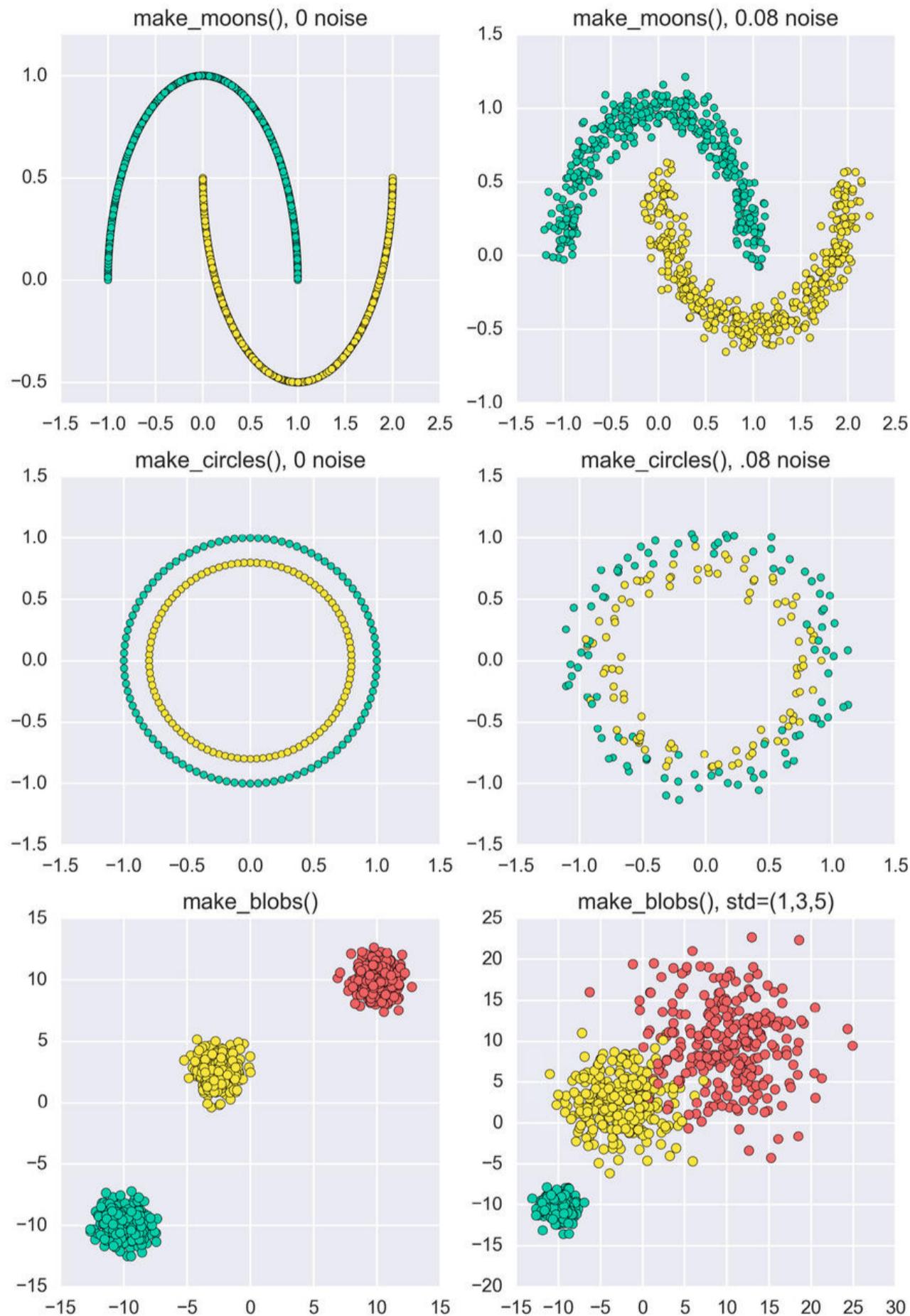


Figure 15.39: Synthetic datasets provided by scikit-learn. Top row: `make_moons` with 800 points, with no noise and with the `noise` parameter set to 0.08. Middle row: `make_circles` with 200 points, with no noise and with the `noise` parameter set to 0.08. Bottom row: `make_blobs()` with 800 points and 3 blobs, with the default standard deviation (all 1's) and with larger standard deviations to spread out the points more.

15.11 Utilities

The code in this section is in the Jupyter notebook
Scikit-Learn-Notebook-9-Utilities.ipynb

Like any library, scikit-learn has its share of utility functions that we can collect together into a kind of grab-bag, or miscellaneous, category.

Perhaps one of the most popular of these is used to split a database into different pieces. The routine `train_test_split()` does just as it says: given a database, it splits it into two pieces that are typically used as a training set and a test (or validation) set. Among other useful arguments, `test_size` is a value from 0 to 1 that specifies the percentage of the database to place into the test set. By default, this has a value of 0.25. Listing 15.37 shows how this works with the nested-circles dataset.

```
from sklearn.model_selection import train_test_split

(circle_xy, circle_labels) = make_circles(
    n_samples=200, noise=.08)

samples_train, samples_test, labels_train, labels_test = \
    train_test_split(circle_xy, circle_labels,
                    test_size=0.25)
```

Listing 15.37: Using `train_test_split()` to break up a dataset of nested circles into training and testing sets.

This produces the datasets shown in in Figure 15.40.

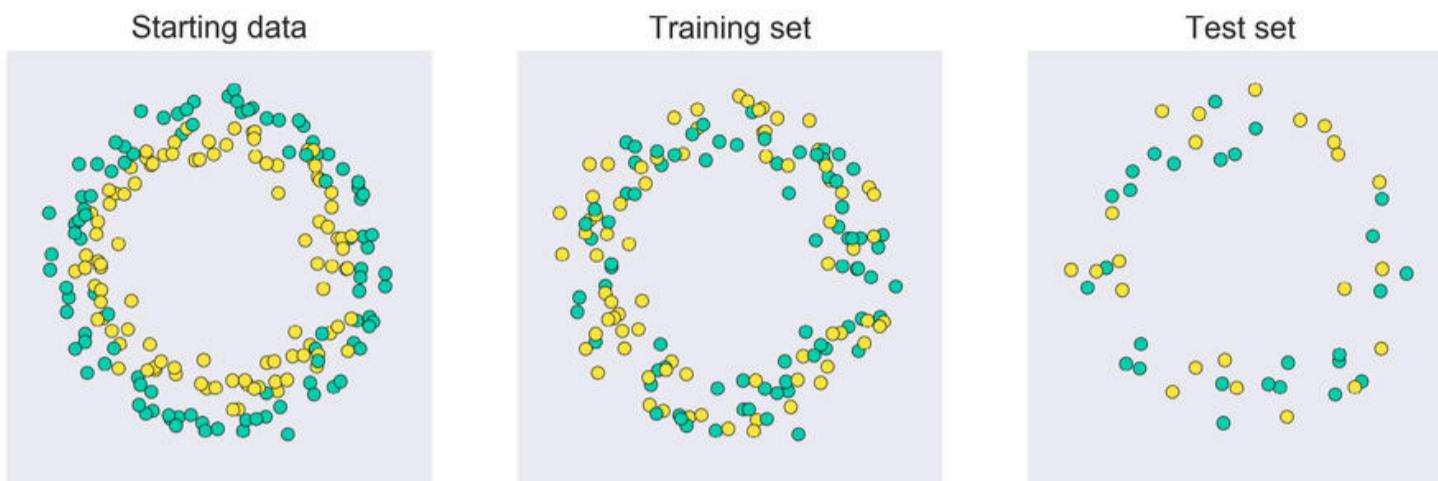


Figure 15.40: Splitting up our original data into a training and testing set with `train_test_split()`. The two sets don't have any samples in common. Left: The starting data of 200 points. Middle: The training set of 150 points. Right: The test set of 50 points.

We've split up our starting data into two new sets. One thing we can't see in the Figure is that the order of the samples has been shuffled. That is, their order in the training and testing sets are not the same order as in the starting data.

To see why shuffling is useful, suppose our circle samples had been generated by starting at 3 o'clock and working clockwise. Then the first 75% of the data would be the samples from 3 o'clock to 12 o'clock, and last 25% would be from 12 to 3, as in Figure 15.41. The test data is nothing like the training data. This would make a terrible training-test split!



Figure 15.41: Data generated by starting at 3 o'clock and working clockwise. Left: The original data of 200 points. Middle: The first 150 points for the training set. Right: The final 50 points for the test set.

We wouldn't have this problem with `make_circles()`, because it generates its samples more randomly, but other programs might not be so careful, and data we get from other sources might have been put into some kind of order before it came to us. To reduce the chance of such datasets producing bad training-test splits like Figure 15.41, `train_test_split()` shuffles the samples before assigning them to the train and test sets. The hope is that this will cause each set to be representative of the totality of the starting data, as it is in Figure 15.40.

15.12 Wrapping Up

As we promised at the start, this chapter has barely hinted at the huge variety of objects and functions offered by scikit-learn.

The scikit-learn online documentation is free and always available, but it's typically aimed at the working programmer who already understands the concepts and just needs to be reminded of syntax or argument names. There are some explanatory and tutorial articles, and some examples of use, but they, too, are often terse. For these reasons, the library's documentation is probably best used for reference, and not for explanations, though the FAQ can sometimes help clear up a question [scikit-learn17d].

An easier way to dig deeper into the mechanics of this versatile library is the book by Müller and Guido [Müller-Guido16]. The book assumes familiarity with machine-learning algorithms, though there is some introductory and review material.

A much more in-depth approach, with plenty of mathematical details, is offered in the book by Raschka [Raschka15].

References

- [Domingos12] Pedro Domingos, “A Few Useful Things to Know About Machine Learning”, Communications of the ACM, Volume 55 Issue 10, October 2012. <https://homes.cs.washington.edu/~pedrod/papers/cacm12.pdf>
- [Jupyter17] The Jupyter authors, “Jupyter”, 2017. <http://jupyter.org/>
- [Kassambara17] Alboukadel Kassambara, “Determining The Optimal Number Of Clusters: 3 Must Know Methods”, STHDA blog, Statistical tools for high-throughput data analysis, 2017. <http://www.sthda.com/english/articles/29-cluster-validation-essentials/96-determining-the-optimal-number-of-clusters-3-must-know-methods/>
- [Müller-Guido16] Andreas C. Müller and Sarah Guido, “Introduction to Machine Learning with Python”, O’Reilly Press, 2016.
- [Raschka15] Sebastian Raschka, “Python Machine Learning,” Packt Publishing, 2015.
- [scikit-learn17a] Scikit-learn authors, “API Reference”, 2017. <http://scikit-learn.org/stable/modules/classes.html>
- [scikit-learn17b] Scikit-learn authors, “Documentation of scikit-learn 0.19”, 2017. <http://scikit-learn.org/stable/documentation.html>
- [scikit-learn17c] Scikit-learn authors, “3.3. Model evaluation: quantifying the quality of predictions”, 2017. http://scikit-learn.org/stable/modules/model_evaluation.html
- [scikit-learn17d] Scikit-learn authors, “scikit-learn FAQ”, 2017. <http://scikit-learn.org/stable/faq.html>
- [scikit-learn17e] Scikit-learn authors, “Home page of scikit-learn 0.19, 2017. <http://scikit-learn.org/stable/>

[scikit-learn17f] Scikit-learn authors, “Installing scikit-learn”, 2017.

<http://scikit-learn.org/stable/install.html>

[scikit-learn17g] Scikit-learn authors, “RidgeClassifier”, 2017. http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.RidgeClassifier.html

[scikit-learn17h] Scikit-learn authors, “sklearn.datasets Datasets”, 2017. <http://scikit-learn.org/stable/modules/classes.html#module-sklearn.datasets>

[VanderPlas16] Jake VanderPlas, “Python Data Science Handbook”, O’Reilly, 2016.

[Waskom17] Michael Waskom, “seaborn: statistical data visualization”, Seaborn website, 2017. <https://seaborn.pydata.org/>

Chapter 16

Feed-Forward Networks

Deep learning systems are structured as networks of neurons. Much of their work is performed as data flows through them in a forward direction, starting at the inputs until it reaches the outputs.

Contents

16.1 Why This Chapter Is Here	657
16.2 Neural Network Graphs	658
16.3 Synchronous and Asynchronous Flow	661
16.3.1 The Graph in Practice	664
16.4 Weight Initialization	664
16.4.1 Initialization	667
References	670

16.1 Why This Chapter Is Here

In this chapter we'll start our transition from general machine-learning concepts and algorithms to those that apply in particular to the relatively newer field of deep learning.

As we mentioned in the introduction, “deep learning” is a catch-all phrase that usually refers to a network of artificial neurons that’s structured in a sequence of layers. Typically, the neurons on each layer get their inputs from a previous layer, and send their outputs to a following layer. In most types of deep networks, neurons do not communicate with other neurons on the same layer.

This naturally allows us to process data **sequentially**, with each stage of neurons building on the work done by the previous stage. 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 [Meunier09] [NVRI17].

It's amazing that hooking up neurons like this produces anything useful. As we saw in Chapter 10, a single artificial neuron can hardly manage to do anything. It takes a bunch of inputs, weights them, adds the results together, and then passes that through a little function. It is remarkable that this process could manage to draw a straight line between a couple of clumps of 2D data, but that's all it can do. But if we assemble many thousands of these little units together, wire them up the right way, 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 skill.

All thanks to these little neurons.

Over time people have developed a number of distinct ways to organize layers of neurons, resulting in a collection of common layer structures. Many deep-learning algorithms are based on stacks of carefully-chosen layers with finely-tuned hyperparameters.

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. Sometimes we get lucky on the first try, but often we need to experiment with our system and try out variations in order to achieve our goals.

In this chapter we'll look at how collections of neurons communicate, and how to set up the initial weights before learning begins.

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.

In this brief chapter we'll look at some of the common principles of these feed-forward networks. We'll draw on this foundation of vocabulary and ideas in later chapters when we look at network-based learning algorithms.

16.2 Neural Network Graphs

Most drawings of **neural networks** (or **neural nets**) look something like one of the drawings in Figure 16.1. Each of these drawings is called a **graph**. The graph is made up of **nodes** (also called **vertices** or **elements**), shown as circles. Typically, these will be neurons, and throughout this book we'll 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** or simply **lines**). Information flows along the edges, carrying the output of one node to the inputs of others.

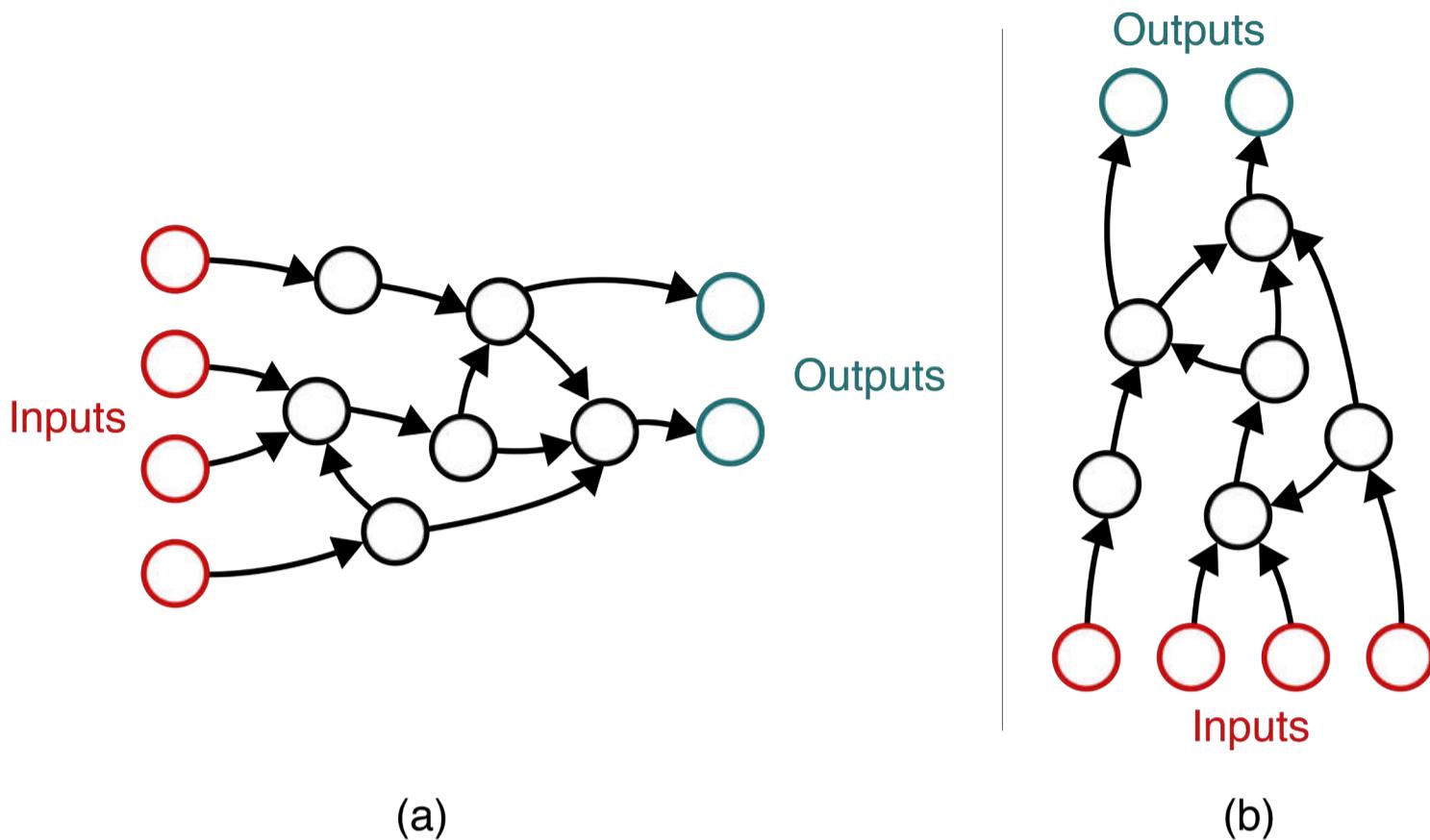


Figure 16.1: Two neural networks drawn as graphs. Data flows from node to node along the edges, following the arrows. When the edges are not labeled with an arrow, data usually flows left-to-right or bottom-to-top. No data ever returns to a node once it has left. In other words, information only flows forwards, and there are no loops. (a) Left-to-right flow. (b) Bottom-to-top flow.

Each edge has an arrow that indicates a **direction**, showing which way data flows along that edge. Usually the inputs are at the left and data flows to the right, or the inputs are at the bottom and the data flows upwards. When there's no ambiguity in the direction of information flow, the arrowheads are frequently left off.

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.

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.

We say that a node near the inputs in Figure 16.1(a) is **before** a node near the outputs, which comes **after** it. In Figure 16.1(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.

The study of graphs is so large that it is considered a field of mathematics in its own right, called **graph theory** [Trudeau94]. Here, we're going to stick to the basic ideas of graphs, mostly as conceptual tools to help us organize our neural networks.

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**, which is pronounced as the word “dag,” rhyming with “drag”). The word “directed” here means that the edges have arrows (which may be implied, as we mentioned above). The word “acyclic” means “no cycles.” An important exception to this rule is a type of network called a **recurrent neural network**, or **RNN**, which we'll see in Chapter 22. But even in those networks, we can usually re-draw things so that they form a DAG.

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 heard audio feedback resulting from moving a microphone too close to a speaker will be 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.

Because the data only flows “forward” from inputs to outputs, this is called a **feed-forward** graph or network. The idea is that each node is “feeding” data to the nodes that come after it.

We’ll see in Chapter 18 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 backwards we call that generally a **feed-backward**, **backward-flow**, or **reverse-feed** algorithm. We reserve the word “feedback” for the types of situations described above, where a loop in the graph could enable a node to receive its own output as input.

16.3 Synchronous and Asynchronous Flow

Interpreting graphs like those in Figure 16.1 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. That is a **continuous** process: there are new molecules of water flowing 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, or physical deliveries by the postal service.

This type of flow is also called a **sample-and-hold** system. When a piece of data arrives at a node, it stays there on the input (where it has been “sampled,” or retained), its value remaining fixed (or “held”), like a text message sitting on a screen. It sits there until a new piece of data comes along, replacing it. Then that data sits there until it’s replaced, and so on.

Some networks are built around the idea of a master clock somewhere, ticking along forever like a grandfather clock. On each tick of the clock, every node that has computed some data sends that data out along edges to its children. When the information reaches the child node, it sits there without changing, and in the interval between ticks all the nodes process the data sitting at their inputs. Then the clock ticks again, and every node that has created an output sends its data down all of its output connections to the inputs of its children. Not all nodes compute their outputs with the same speed. Nodes with more complex work to do will often take more time to produce their output. To make sure that new data flows on every tick of the clock, we usually pick the timing of the clock so that the slowest node in the network has time to complete its work before the next tick arrives. We call this a **synchronous** system.

In other words, in a synchronous system, data moves only when the clock ticks. Between ticks, the data on the wires remains constant, and the nodes can consume that information and work with it to prepare their output, which is passed on to their descendants on the next tick.

An example of this type of system is a multi-stage conveyor belt, as used in factories. These types of belts move in steps. They carry a product from one station to the next, and then pause so that some step of work can be carried out. So every station does some work on the assembly it's received, the belt moves one step forward, and the process repeats.

By contrast, an **asynchronous** (meaning “not-synchronous”) system has no master clock. Instead, information moves as soon as it’s computed.

For example, a group of people making plans by sending text messages back and forth is asynchronous, since each person responds when they have the chance. There’s no master clock, or “heartbeat,” driving the system. Some people might respond to messages instantly, and others might be busy and not reply for a long time. At all times everyone

works with the most up-to-date information they have from everyone else. If someone is so slow that their most recent message is a week old, and the decision has already been made, then so be it.

The asynchronous case is the more general one, since each individual piece of data holds its value until it changes. The synchronous network is a special case of the asynchronous one, where the updates happen in lock-step.

Because asynchronous networks are the more general version, understanding their workings gives us insight into how both kinds of systems operate.

A problem in asynchronous networks arises when a node only wants to proceed when it has new information from all of its predecessors. Since data just sits there holding its value, how can that node distinguish a new value from an existing one? For instance, suppose a node adds up the numbers given to it by three other nodes. At one point those values are 3, 5, and 2, so the node makes the value 10 and sends it out. Soon after that, the 3 turns into a 1, and the 5 turns into a 4. But the 2 doesn't change. Or did it? Maybe that node's output was just re-computed, and the result happened to just be 2 again. Should our node add up these values and produce a new output, or not?

The node needs a way to make sure it only adds up the inputs and then produce a new output when all of its inputs are new, compared to the last time it calculated a sum.

A useful way to know when new data has arrived is to think of data as having an associated **time stamp** that records when it was sent. Such a stamp records the time the data was computed and sent onward, and not the time when it was used. In other words, the node that computes the data assigns the time stamp, not the nodes that receive it.

Each node can then also remember the moment when it computed its most recent result.

When all of the inputs are time-stamped later than the most recent time the node produced its own output, then they're all new, even if they happen to have the same values. Now the node can compute a new result, and remember the moment of that computation.

Then it waits until a completely new set of data has arrived, so it can produce a new value, record the moment of doing so, and restart the cycle.

16.3.1 The Graph in Practice

Attaching a time-stamp to every piece of data was a useful concept for discussion, but we often don't need it in a computer program. Traditionally, we enforce this rule simply by how we structure the code.

Modern computers often use a wide variety of hardware and software to accelerate computations. The basic scheme described above gets more complex in such situations, but the principles still apply.

16.4 Weight Initialization

We mentioned before that our graphs will represent networks of neurons and their connections.

When we draw a neural network in graph form, the nodes represent neurons, and the edges are the connections between them. The output of one neuron travels along an edge to the input of the next, as in Figure 16.2.

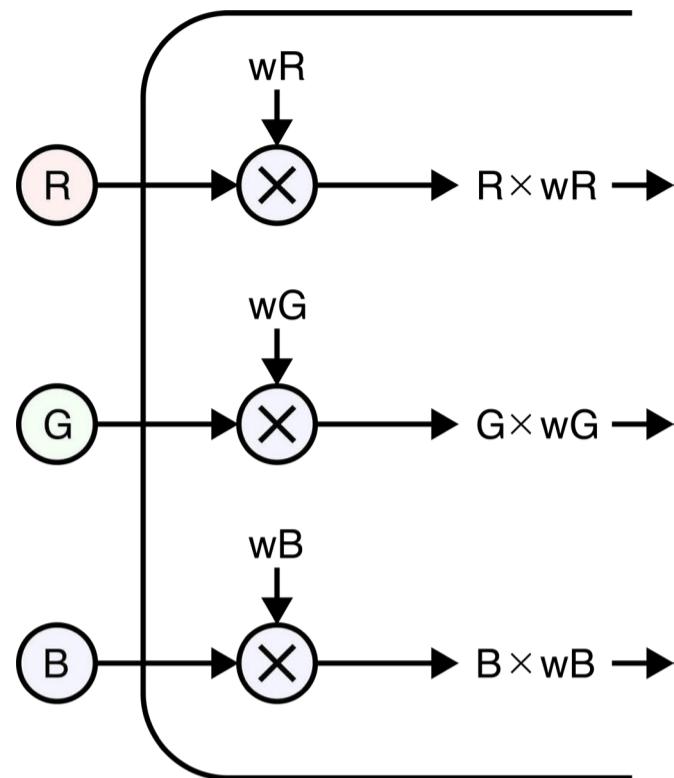


Figure 16.2: Here three values, named R, G, and B, are heading into a neuron. As each value enters the neuron, it's weighted, or multiplied by a number associated with that input. The resulting scaled value is then used by the rest of the neuron to compute its output value.

Recall from Chapter 10 that the first thing an artificial neuron does with an input is to multiply it by a number called the *weight*. We sometimes call this **weighting** the input.

As we saw in Chapter 10, we can conceptually pull the weights out of the neuron and put them onto the wires carrying the data. In this version, after a neuron outputs a value, it's multiplied by a weight as it travels down the connecting line, and then arrives at the next neuron already weighted, as shown in Figure 16.3.

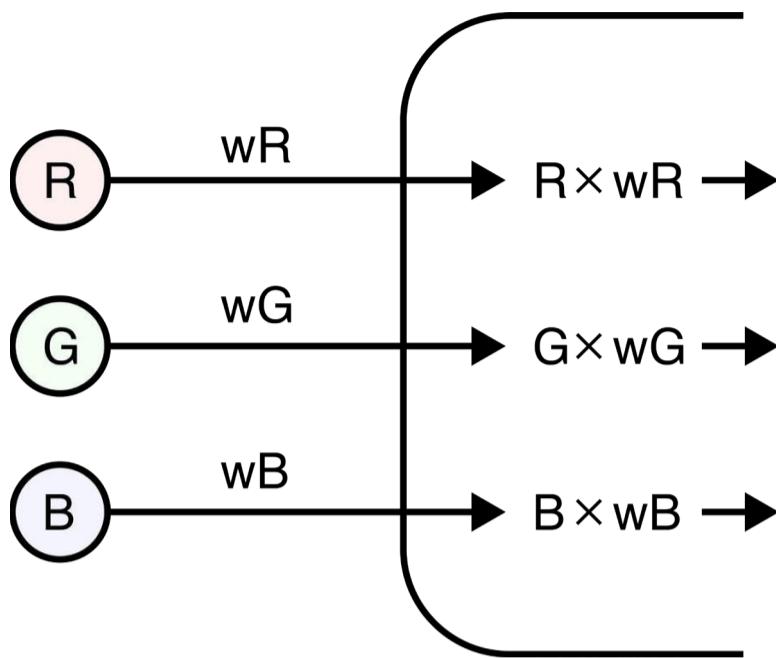


Figure 16.3: Here we've pulled the weights out of the neuron in Figure 16.2, and written them on the wires carrying the values into the neuron. In this style of diagram, the multiplication of each value by the named weight is implied. The values going into the rest of the neuron are the same as before, since we've only made cosmetic changes.

Each line has its own weight, so if one neuron sends its output to three children, there will be three connecting lines, each with its own weight. The idea is in Figure 16.4.

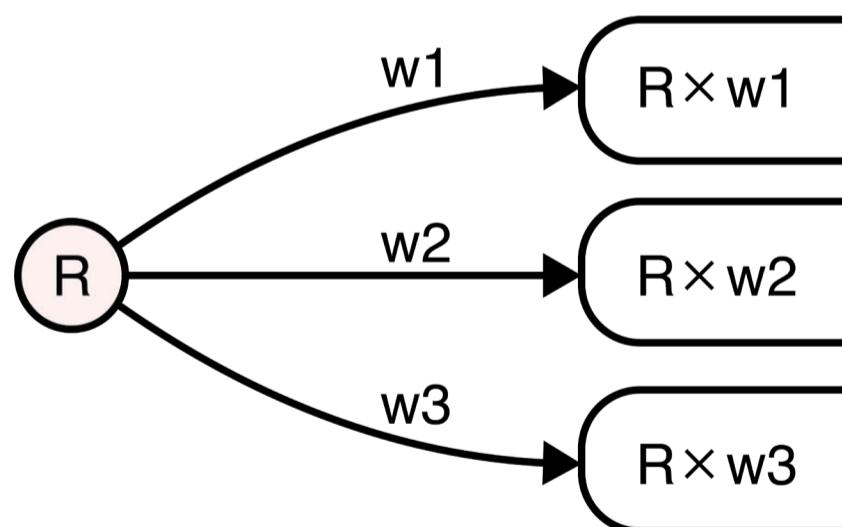


Figure 16.4: If a single neuron, here labeled R, sends its output to three different neurons, then R's output value will be weighted by three potentially different values, one on each wire.

This style of drawing, where the weights are implicitly attached to the edges, is by far the most common, and the one we'll adopt here. As we mentioned in Chapter 10, we often don't even write the weight explicitly, but they should be understood as implied.

This is important enough to repeat. In any neural network graph, if no weights are explicitly shown, it is implied that *there is a unique weight on each wire*, and as a value moves from one neuron to another along that wire, that value is modified by the weight.

16.4.1 Initialization

So far, we've seen that data is applied to the inputs, flows through wires, and gets weighted before arriving at neurons as inputs. Those neurons calculate new values based on the inputs, and those values go out on their own wires, where they get weighted before arriving as inputs at other neurons.

The learning process is all about modifying the weights on the wires (or in the neurons) so that we get the best results at the end.

But how do we initialize those weights in the first place, before there's been any data or learning?

People have explored many different ways to initialize the weights, and compared the results. A few options have proven to consistently provide the best results for learning, and these are supported by most machine-learning libraries. Other initialization schemes are often available for special cases, or just in case we want to try something new and see how it works with our data.

Perhaps the simplest scheme is just to assign the same value to every weight. This turns out to be a bad idea, since the algorithms we'll later use to adjust the weights will tend to change all those weights of the same value by the same amount, causing them to change in lockstep. We'd really like our weights to start out with different values, so that they will be more amenable to being individually adjusted.

Perhaps the easiest way to assign a different value to each weight is called **uniform initialization**. Here the word “uniform” refers to a **uniform random distribution**, like we saw in Chapter 2. This just means that all of the values between one extreme and another are equally likely. In machine learning, we often use a small range around zero, such as $[-0.05, 0.05]$. Assigning random values to our weights may sound crazy, but the values are quickly improved during learning and their initial influence rapidly fades. Figure 16.5 shows the idea.

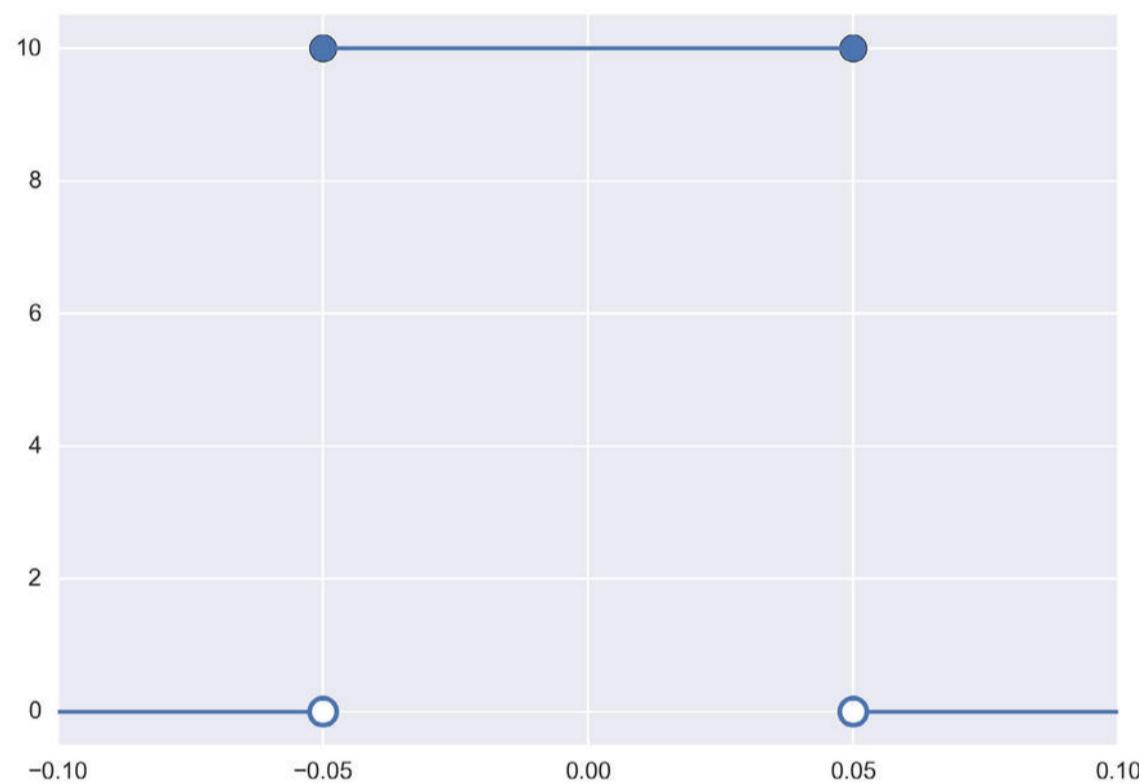


Figure 16.5: When we sample values from a uniform distribution, there’s an equal chance of getting back any value in the non-zero range. Here the specific height of the upper region isn’t important for understanding what’s happening. What matters is that every value between -0.05 and 0.05 is equally likely to be selected.

When we ask for random values using the distribution of Figure 16.5, we’re telling the computer that we want no returned values larger than 0.05 , and no returned values less than -0.05 . Furthermore, every value between those extremes should have an equal probability of being chosen and returned.

The **normal initialization** scheme uses a **normal distribution** or **Gaussian distribution** of random numbers, as discussed in Chapter 2. This distribution is the famous “bell curve.” When we use this approach for choosing the initial weights, it means that values near 0 are most likely to be used, and values to the sides are less likely. Figure 16.6 shows this visually.

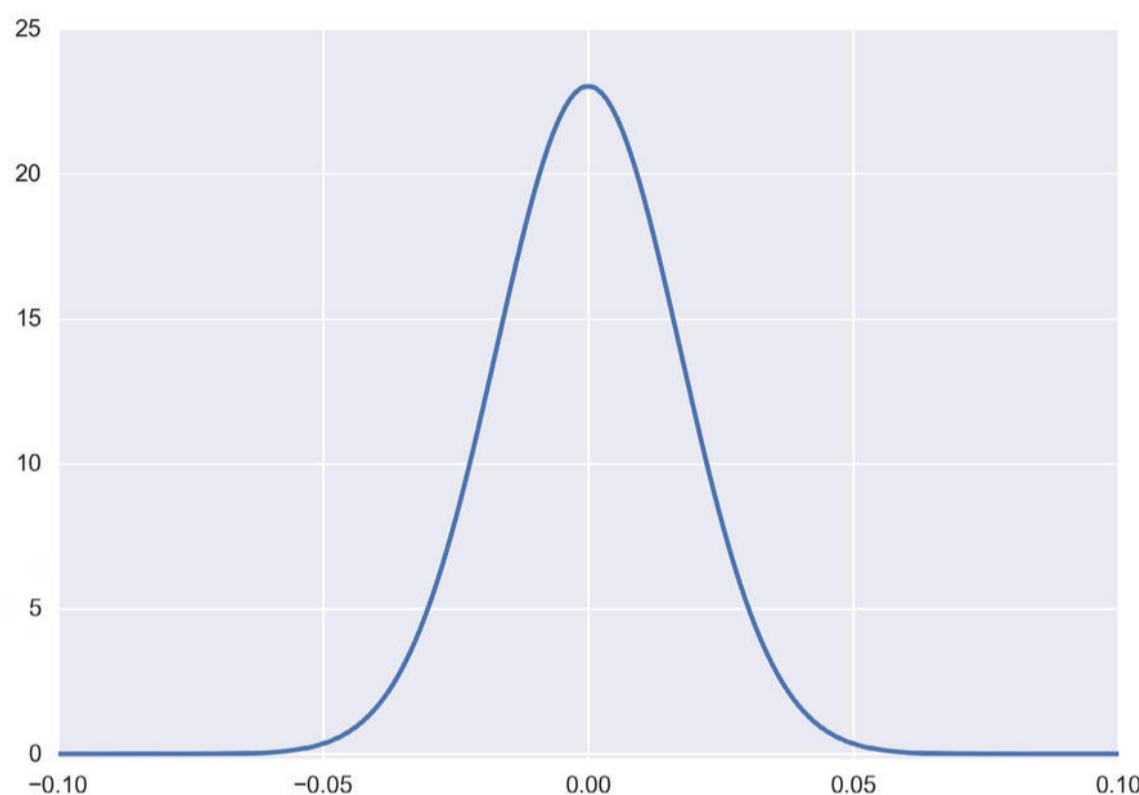


Figure 16.6: When we sample values from a normal distribution, we’re more likely to get back values near the center of the distribution, with decreasing probability as we move away from the center. Again, the absolute values are less important than noting that 0 is most likely, and the probabilities drop off smoothly to the sides, reaching nearly zero at the limits that we’ve chosen here at -0.05 and 0.05.

As with uniform initialization, the random values we start with are quickly replaced by better values as we start learning.

We rarely use uniform or normal initialization directly, because we need to pick their parameters (for instance, their minimum and maximum values). Researchers have found that there are good rules of thumb for picking these parameters, and the various algorithms that have emerged 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. [Lecun98] [Glorot10] [He15].

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.

These techniques have all proven to work well in practice. If a library offers more than one of them, it often pays to try a few and see which performs the best for a given combination of a specific network and the specific data we're teaching it with. Most libraries offer one or more of these schemes, plus variations on them.

Some particular types of algorithms have been found to work best with particular initialization schemes, and those details are usually noted in the documentation of those algorithms in our library of choice.

References

[Glorot10] Xavier Glorot and Yoshua Bengio, “Understanding the Difficulty of Training Deep Feedforward Neural Networks,” Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS), 2010. <http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf>

[He15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, arXiv 1502.01852, 2015. <https://arxiv.org/abs/1502.01852>

[Lecun98] Yann LeCun, Leon Bottou, Genevieve B. Orr, Klaus-Rober Müller, “Efficient BackProp”, in “Neural Networks: Tricks of the Trade”, editors Genevieve B. Orr and Klaus-Rober Müller, Lecture Notes in Computer Science volume 1524, Springer-Verlag, 1998. <http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf>

[Meunier09] David Meunier, Renaud Lambiotte, Alex Fornito, Karen D. Ersche and Edward T. Bullmore, “Hierarchical Modularity in Human Brain Functional Networks”, Frontiers in Neuroinformatics, 2009. <https://www.frontiersin.org/articles/10.3389/neuro.11.037.2009/full>

[NVRI17] National Vision Research Institute of Australia, “Hierarchical Visual Processing”, NVRI Research Blog, 2017. <http://www.nvri.org.au/pages/hierarchical-visual-processing.html>

[Trudeau94] Richard J. Trudeau, “Introduction to Graph Theory”, 2nd Edition. Dover Books on Mathematics, 1994.

Chapter 17

Activation Functions

The last step in an artificial neuron is to apply an activation function to the value it computes. Here we'll see a variety of popular activation functions in use today.

Contents

17.1 Why This Chapter Is Here.....	674
17.2 What Activation Functions Do	674
17.2.1 The Form of Activation Functions	679
17.3 Basic Activation Functions.....	679
17.3.1 Linear Functions.....	680
17.3.2 The Stair-Step Function.....	681
17.4 Step Functions	682
17.5 Piecewise Linear Functions.....	685
17.6 Smooth Functions.....	690
17.7 Activation Function Gallery	698
17.8 Softmax.....	699
References	702

17.1 Why This Chapter Is Here

Artificial neurons are at the core of neural networks. We've looked at the structure of artificial neurons in Chapter 10, but we didn't go into detail on the final step in its processing. This chapter is all about that final step.

As we saw in Chapter 10, an artificial neuron takes three steps to produce its output. First, it weights every input by multiplying it with a corresponding weight value. Second, it sums up those weighted values. And finally, it passes that summed value through a mathematical function that changes it.

There are many choices for this function, but collectively they're called **activation functions**. The output of the activation function is the output of the neuron.

In this chapter we'll see why activation functions are useful, and then we'll survey the most popular ones. When we build neural networks in later chapters, they will all use one or more of these functions.

17.2 What Activation Functions Do

The activation function, while a small piece of the overall structure, is critical to a successful neural network. Without it, the whole collection of neurons can be mathematically crunched together into just one neuron. As we saw in Chapter 10, a single neuron has very little computational power.

As an analogy, imagine a series of train cars making up a long freight train. Each car is connected to the one after it with a pair of interlocking hooks, collectively called a **coupling**. The coupling lets the cars move and rotate a little with respect to one another, so they can follow the track, but it prevents them from disconnecting from each other. A coupling is shown in Figure 17.1.

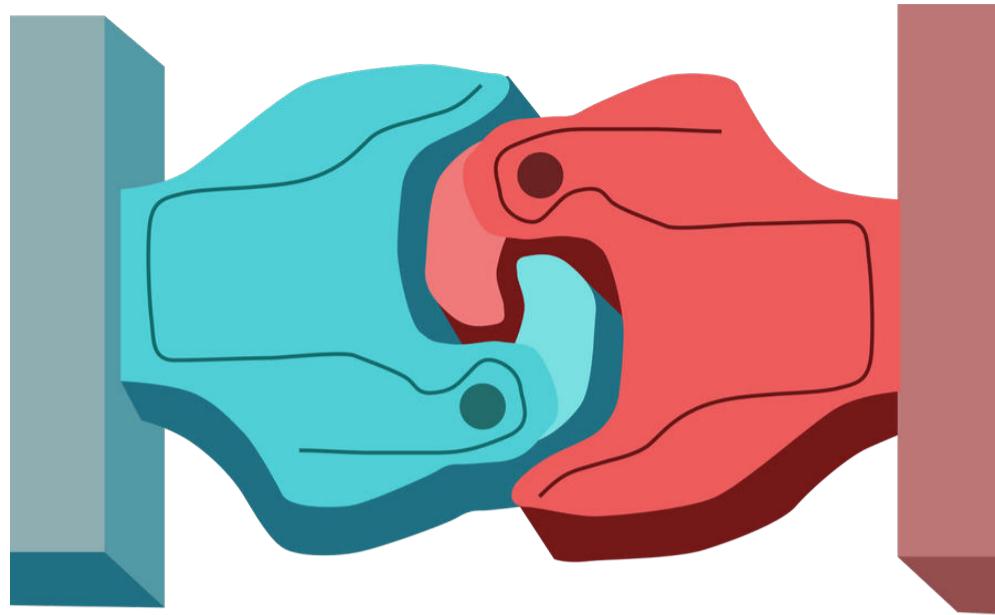


Figure 17.1: A train coupling between two railway cars. The two pieces meet in a pair of hinged “fingers” that interlock. This allows the cars to separate and close up a little as they travel, and for the two cars to rotate with respect to each other in either direction and still stay connected. (Image after photo on [Stoltz16]).

Without these couplings, the cars couldn’t rotate with respect to one another, giving us the equivalent of one extremely long train car. This would be impractical for many reasons, including danger to anyone or anything located off the tracks but inside a curve, as in Figure 17.2.

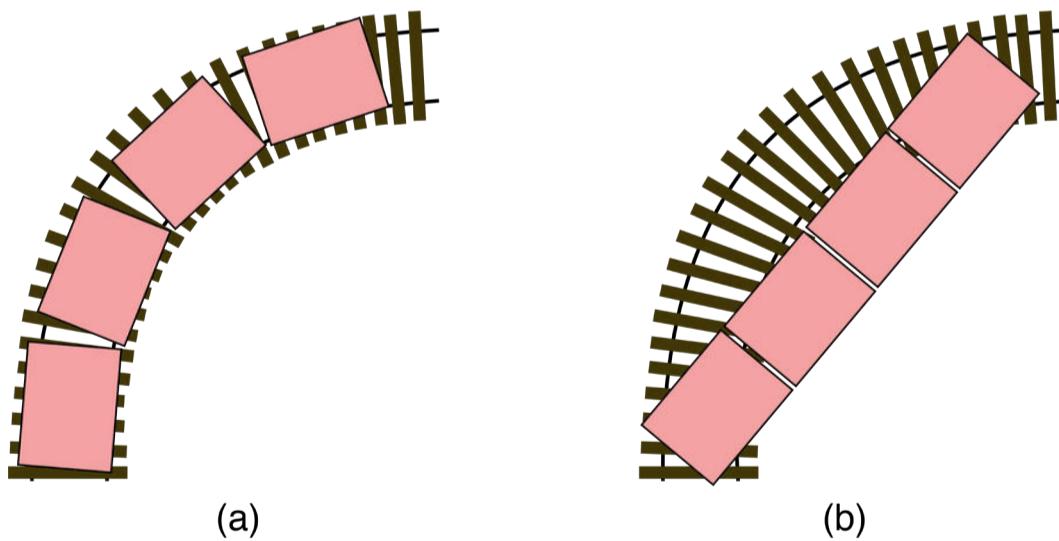


Figure 17.2: One of the problems that are solved by couplings. (a) With couplings, many individual train cars can each follow the curved track, yet stay in contact. (b) Without couplings, it’s as if the cars were all one big rigid car.

The analogy between couplings and activation functions is more suggestive than literal, because the mathematical purpose of activation functions is a bit more abstract than the mechanical purposes of couplings. But what is common is both ideas serve a similar purpose in allowing many units to remain connected, yet distinct from one another. Without activation functions, our neural network would mathematically turn into one big neuron, just as the train cars turn into one big rigid car.

Let's see an example of how this kind of **network collapse** could happen. Figure 17.3 shows a network with three inputs named A, B, and C. There are six internal nodes with names D, E, F, and G, H, I, and one output neuron labeled J. This is a **fully-connected** network, where every neuron receives input from every neuron on the previous layer.

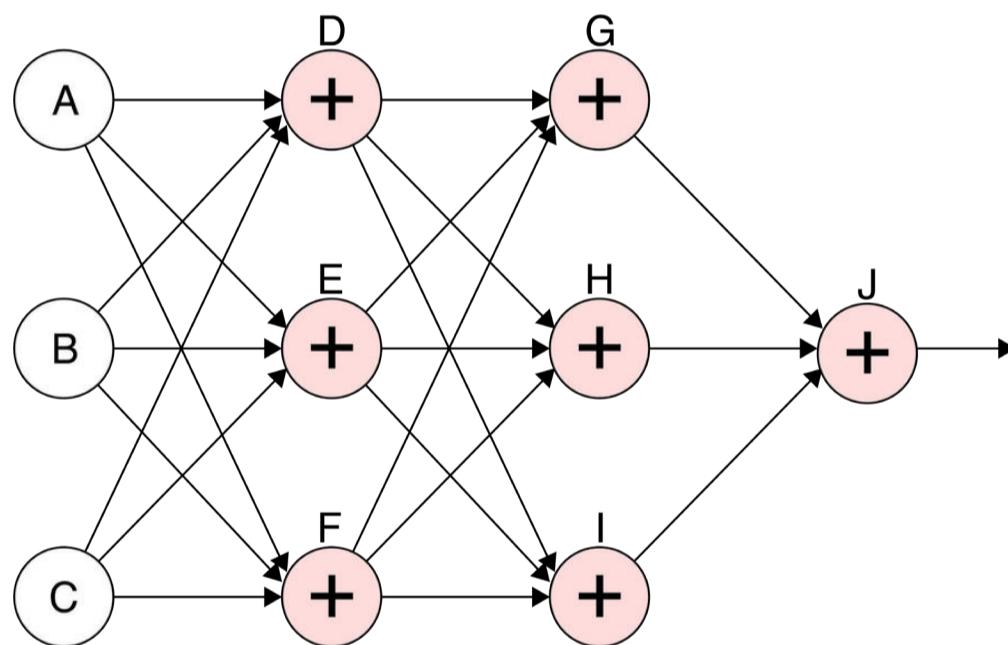


Figure 17.3: A fully-connected network of 3 inputs, 1 output, and two internal layers of 3 neurons each. Each arrow represents a connection and implicitly has an attached weight.

Figure 17.3 is the kind of complicated diagram one often sees for fully-connected networks. Nobody likes these messy and complicated things, so let's redraw our diagram. We won't change anything about the network or its connections, but we'll make it easier to read by pulling the pieces apart into their own little diagrams. We'll also label each edge with its weight. Figure 17.4 shows the result.

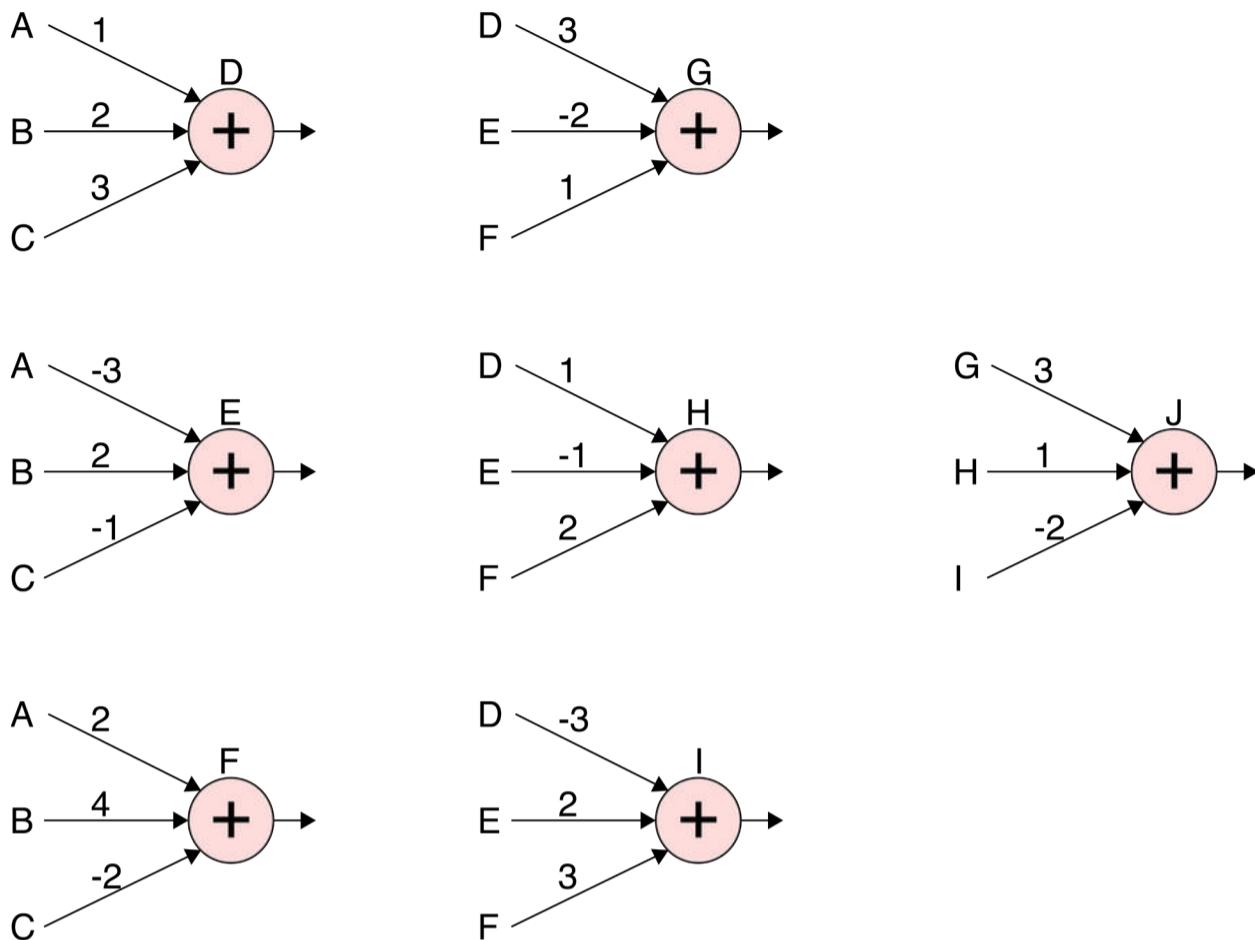


Figure 17.4: Re-drawing Figure 17.3 so that we can make sense of it. We've labeled the weights on the edges explicitly. Notice that every neuron from D through J presents at its output just some combination of the inputs A, B, and C.

The key thing to notice about Figure 17.4 is that the outputs of D, E, and F are just the result of additions and multiplications with A, B, and C. For instance, the output of D is $A + 2B + 3C$, and the output of F is $2A + 4B - 2C$ (here we're implying multiplication, so $2B$ is the same as $2 \times B$). Since nodes G, H, and I just add scaled versions of these combinations, their outputs too are just some combination of A, B, and C. Finally, the output of J is a combination of the outputs of G, H, and I, and thus it, too, is just some combination of A, B, and C.

If we grind through all the numbers, then we'll find that Figure 17.4 can be drawn, or “collapsed,” into just one node with new weights on the three inputs. Figure 17.5 shows that result.

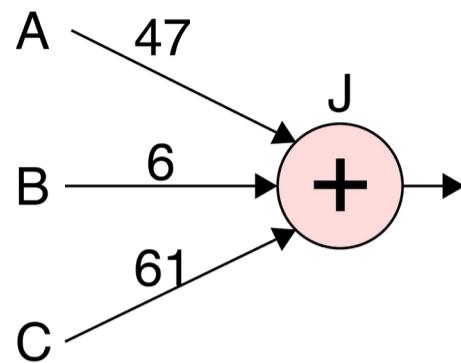


Figure 17.5: If we grind through the numbers in Figure 17.4, we find that the output of J is just this combination of A, B, and C.

The value that comes out of the single neuron of Figure 17.5 will be identical to the value that comes out of node J in the big network of Figure 17.3 or Figure 17.4. It'll even come out faster and use up less computer memory.

But this is bad news if we want our network to be able to do anything more than what one neuron can do. If we let this collapse happen, then our neural network will never be any better than a single neuron. That's not going to be a versatile system for learning about complex problems.

An activation function is just a tiny little bit of processing that goes at the end of each neuron, but it prevents this collapse. That's because the math tells us that a network can only collapse in this way if the network does nothing but multiplication and addition, which are called **linear functions**. By contrast, the activation function is called a **non-linear function**, or sometimes just a **non-linearity**. The presence of this non-linear step after each neuron (that is, something beyond addition and multiplication) prevents the collapse of the network.

The result of using these functions is that we keep our network of many neurons, rather than having it turn into just one neuron, and that makes all the difference.

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 ceasing altogether.

If that happens, we can substitute an alternative activation function that avoids the problem (though of course it will have its own weak points).

Here we'll look at various activation functions and consider their pros and cons. This will help us make informed choices when building our own neural networks.

17.2.1 The Form of Activation Functions

An activation function (sometimes also called a **transfer function**) is an operation that takes a floating-point number as input, and returns a new floating-point number as output. The word “function” comes from how the operation is structured mathematically, but we can also think of it as a programming-language function, or subroutine. It takes one floating-point value as a parameter, and returns a new floating-point value as output.

We can characterize these functions in entirely visual terms, without any equations or code.

In theory we could apply a different activation function to every neuron in our network. But in practice, there's usually one best choice for each layer in a neural network, so we use that one function for every neuron in the layer.

17.3 Basic Activation Functions

To picture an activation function, we can draw it as a 2D curve. The horizontal, or X, axis is the input value, and the vertical, or Y, axis is the output value. So to find the output for any input, we locate the input along the X axis, and move directly upwards until we hit the curve. That's the output value.

17.3.1 Linear Functions

Figure 17.6 shows a few “curves” that are just straight lines. Let’s look at the leftmost example. If we pick any point on the X axis, and go vertically up until we hit the “curve,” the value of that intersection on the Y axis is the same. So the output, or Y value, of this curve is always the same as the input, or X value. We call this the **identity function**.

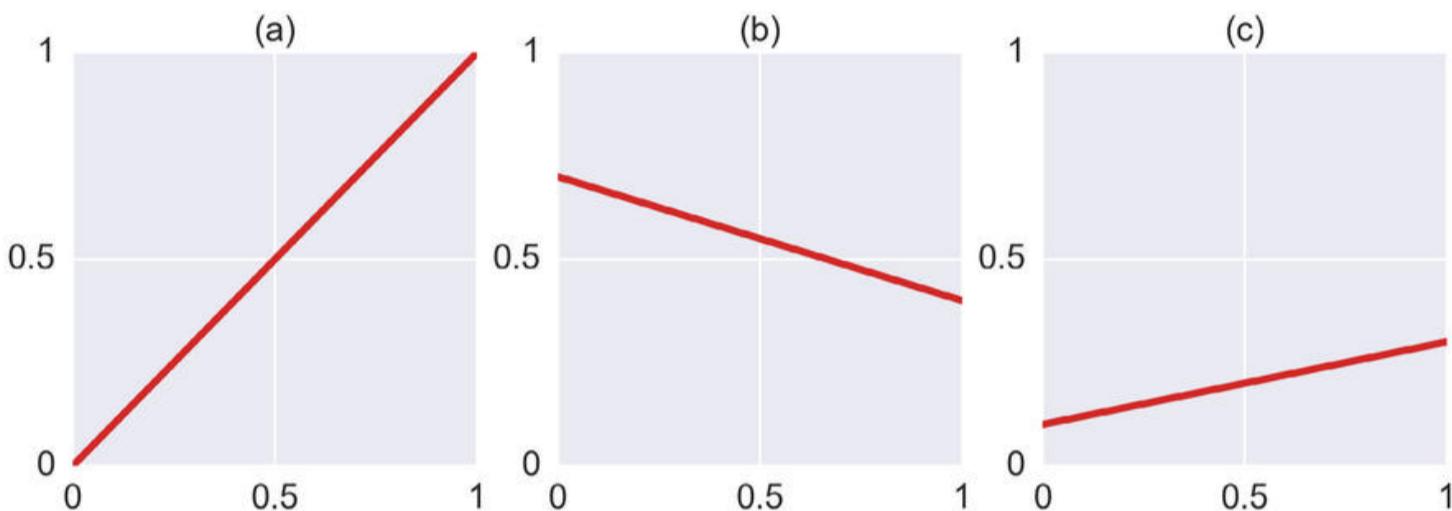


Figure 17.6: A few “curves” that are really just straight lines. At the left the Y value is always the same as the X value. We call this the identity function.

The other “curves” in Figure 17.6 are also straight lines, but they’re tilted at different slopes. We call any “curve” that’s just a single straight line a **linear function**, or even (slightly confusingly) a **linear curve**.

These linear “curves” are exactly the kind of shapes that prevent the activation function from doing its job. When the activation function is a straight line, then mathematically it’s only doing multiplication and addition, and that means that the network can collapse. To avoid that we can change our function in lots of ways, for example by adding a kink or bend, or curving the line, or breaking it up into pieces. But we need to get away from a single straight line.

One thing that we have to keep in mind, though, is that the curve needs to be **single-valued**. As we discussed in Chapter 5, this means that if we look upward from any value along the X axis, there’s only one value of Y above us.

An easy variation is to start with a straight line and break it up into several pieces. They don't even have to join up. In the language of Chapter 5, they don't have to be **continuous**.

17.3.2 The Stair-Step Function

For example, Figure 17.7 shows a **stair-step function** that outputs the value 0 if the input is from 0 to just less than 0.2, but then 0.2 if the 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 value at each point.

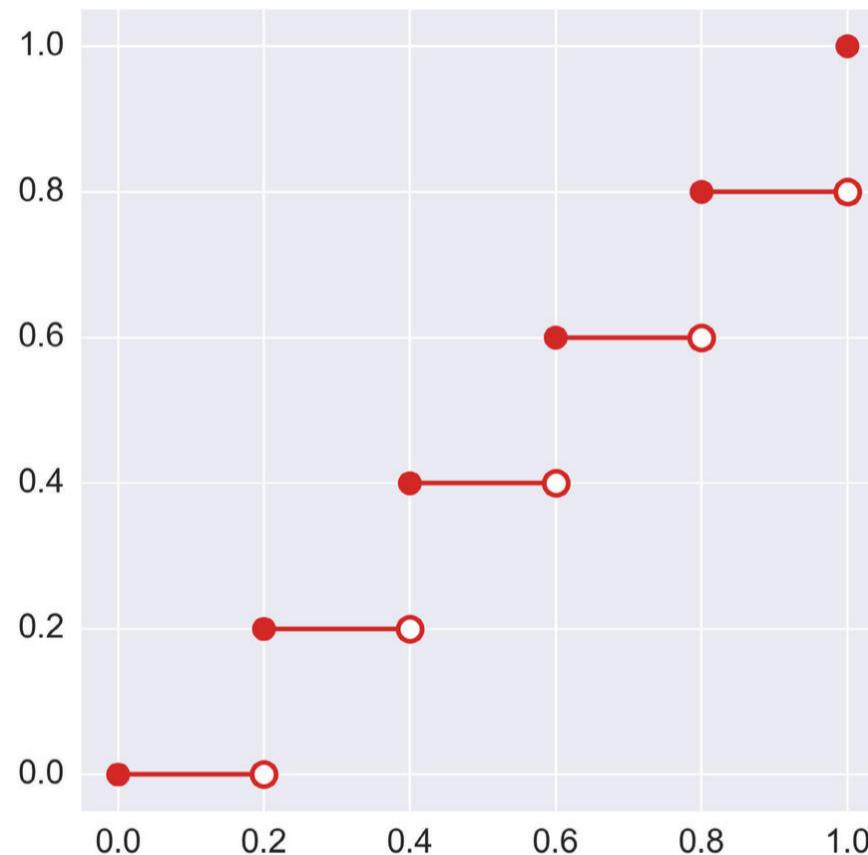


Figure 17.7: This “curve” is made up of multiple straight lines. It is discontinuous, meaning that we'd have to lift our pencil from the paper to draw it. A filled circle tells us that the Y value there is valid, while an open circle tells us that there is “no curve” at that point.

Earlier we said that any activation function that is a single straight line is called *linear*, and everything else is *non-linear*. This holds even when we're using multiple straight lines: unless it's a single straight line, the function is *non-linear*. Except in rare special circumstances,

we always use non-linear activation functions on the neurons in our network. The most common exception is in the final output neuron, or neurons. Using a linear function there isn't risky, because there are no following neurons to collapse with.

The term “linear” has a broader mathematical meaning. In many situations, mathematicians actively seek linear problems because they are usually easier to solve than their non-linear cousins. But when building a neural network, a non-linear activation function is our friend.

Because the activation function is often the only non-linear part of an artificial neuron's processing, the activation function is often referred to as **the non-linearity**.

17.4 Step Functions

The simplest activation function is the identity function, which is the same as having no activation function at all. The output of the neuron's summing stage becomes its final output with no further change. As we've mentioned, linear activation functions are rarely used anywhere but the output neurons.

Activation functions are able to take in any real number as an input. We usually only show a neighborhood around 0, because that's where most of the variation is located. Well beyond 0, most activation functions have a predictable structure that we don't need to explicitly draw.

A simple non-linear activation function is the **step function**. The perceptron, discussed in Chapter 10, uses a step function as its activation function.

A step function is usually drawn as in Figure 17.8(a). It has one value until some **threshold**, and then it has some other value. What value does the function have when the input is precisely at the threshold?

Different people have different preferences for this. In Figure 17.8(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.

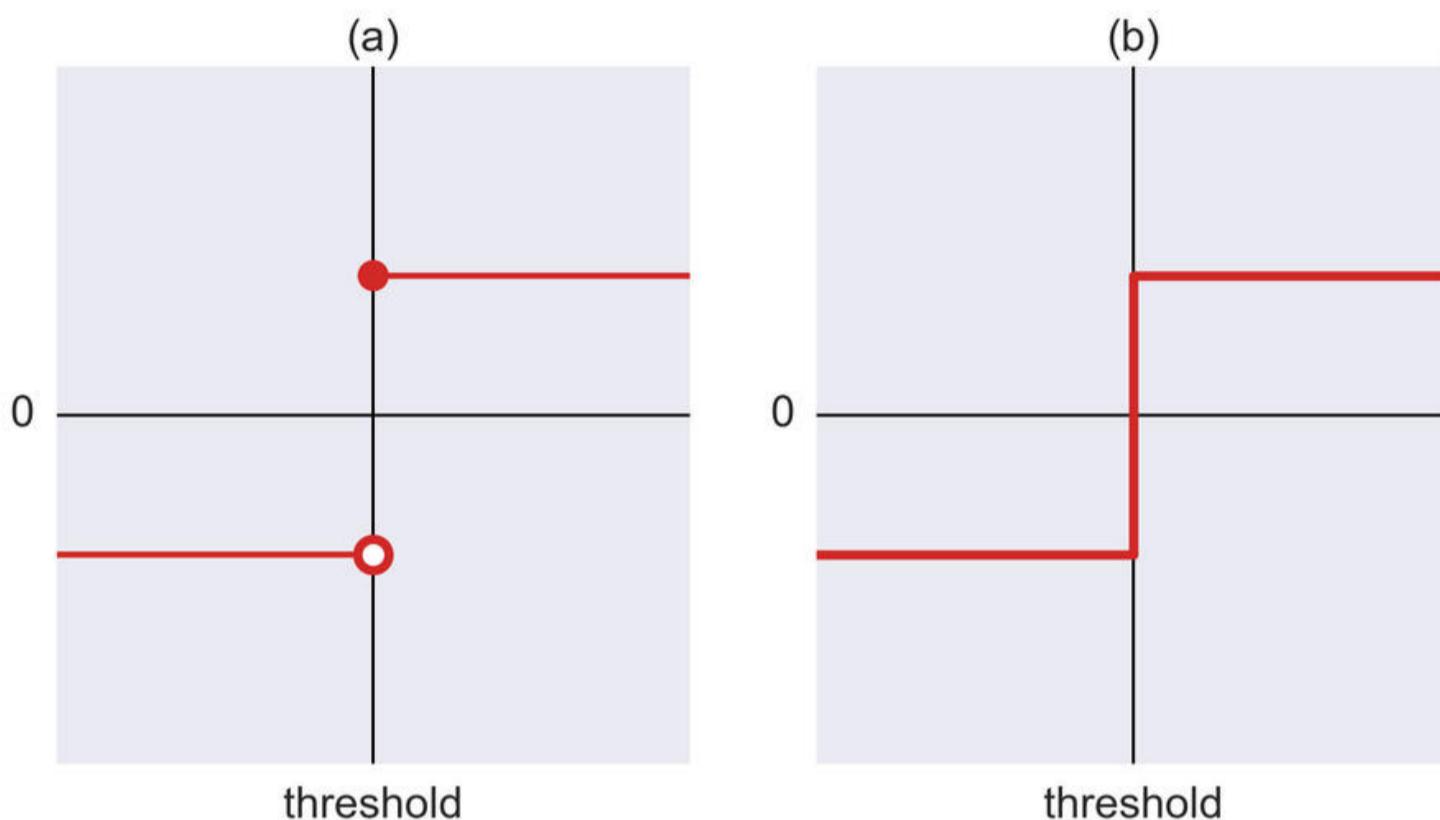


Figure 17.8: A step function has two fixed values, one each to the left and right of a threshold value of X . (a) Using our solid and empty circles to show that when the X value is exactly the threshold, we use the larger value. (b) Often authors are casual about what happens exactly at the transition and draw the picture this way, in order to stress the “step” of the function. This is an ambiguous way to draw the curve, but it’s common (often we don’t care which value is used at the threshold, so we can imagine whichever one we prefer).

There are a couple of popular versions of the step that have their own names. The **unit step** is 0 to the left of the threshold, and 1 to the right. Figure 17.9 shows this function.

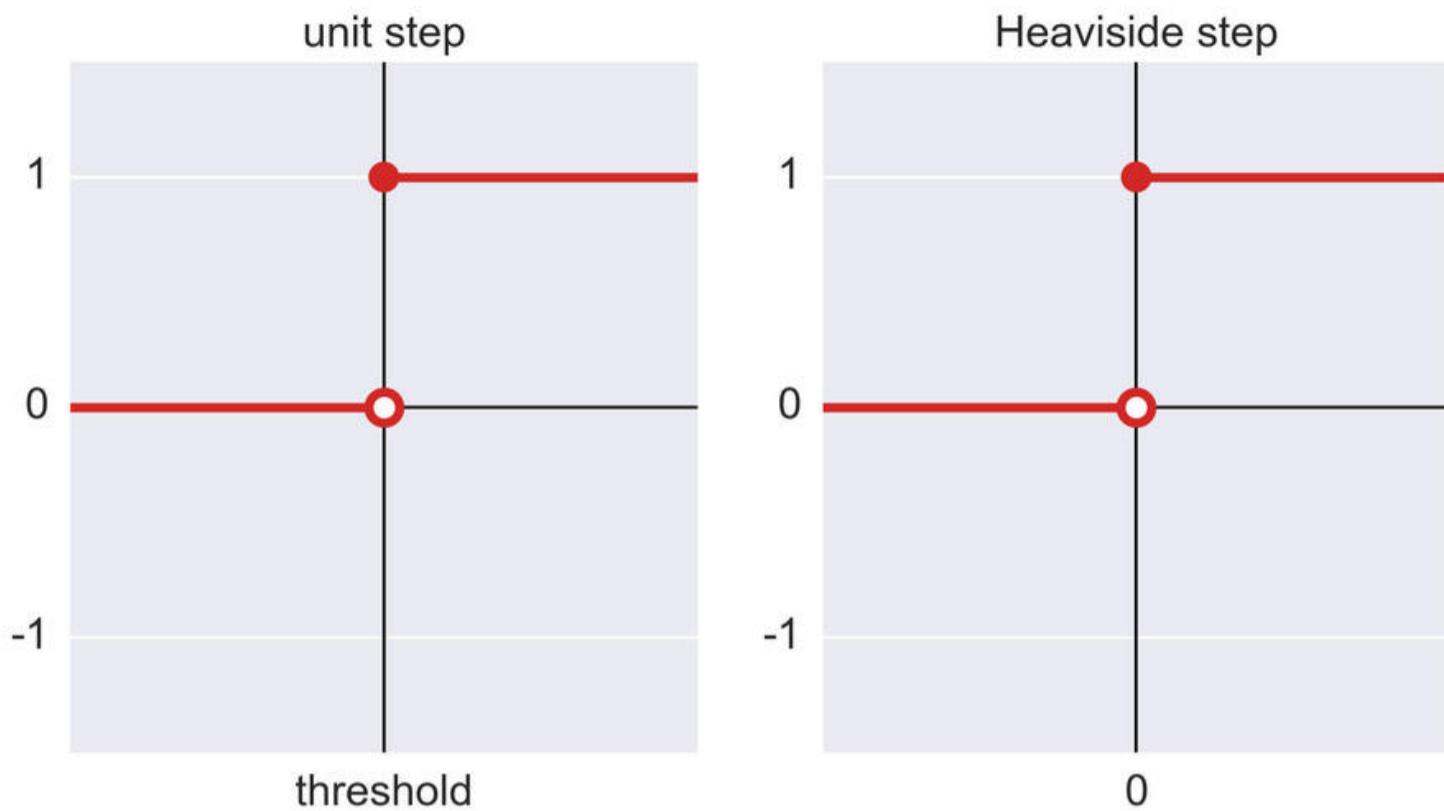


Figure 17.9: A couple of popular step functions. Left: The unit step has a value of 0 to the left of the threshold, and 1 to the right. Right: The Heaviside step is a unit step where the threshold is 0.

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 17.9.

Finally, if we have a Heaviside step (so the threshold is at zero) but the value to the left is -1 rather than 0, we call this the **sign function**, shown in Figure 17.10. 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 one has to pay closer attention to context to figure out which one is being referred to.

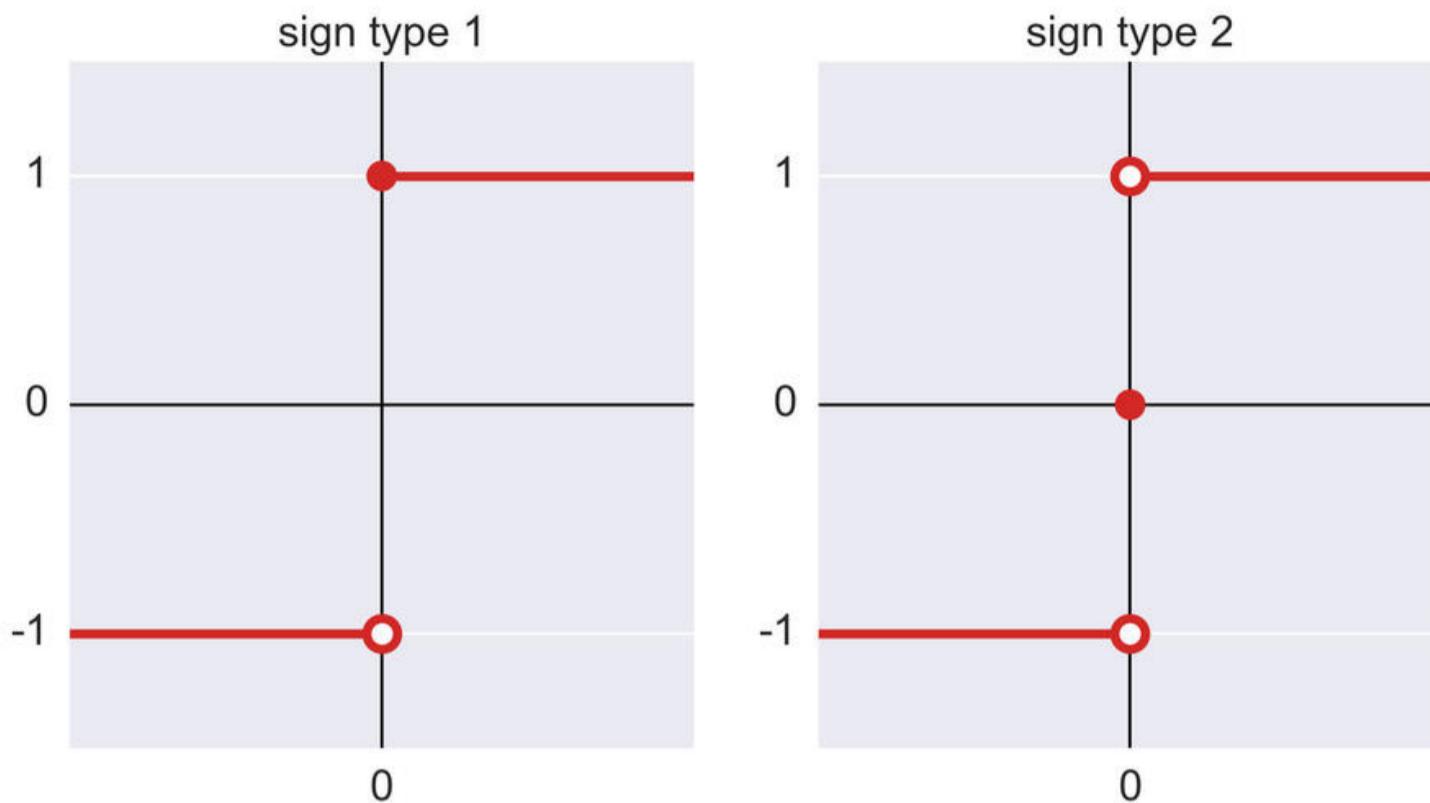


Figure 17.10: Two popular types of sign functions. Left: Values less than 0 are assigned an output of -1 , all others are 1 . Right: Values less than 0 are given the value -1 , values greater than 0 are given the value 1 , and an input of exactly 0 gets the value 0 .

As we saw in Chapter 10, the perceptron uses the unit step in Figure 17.9 as its activation function.

17.5 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 non-linear function as long as the pieces don't together 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 in lower case). The name comes from an electronics part called a “rectifier,” which can be used to prevent negative voltages from passing through to another part of a circuit

[Kuphaldt17]. When the voltage goes negative the rectifier “clamps” it to 0, and the rectified linear unit does the same thing with the numbers that are fed into it.

The ReLU’s graph is shown in Figure 17.11. It’s made up of only two pieces, each straight lines, but thanks to the kink, 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.

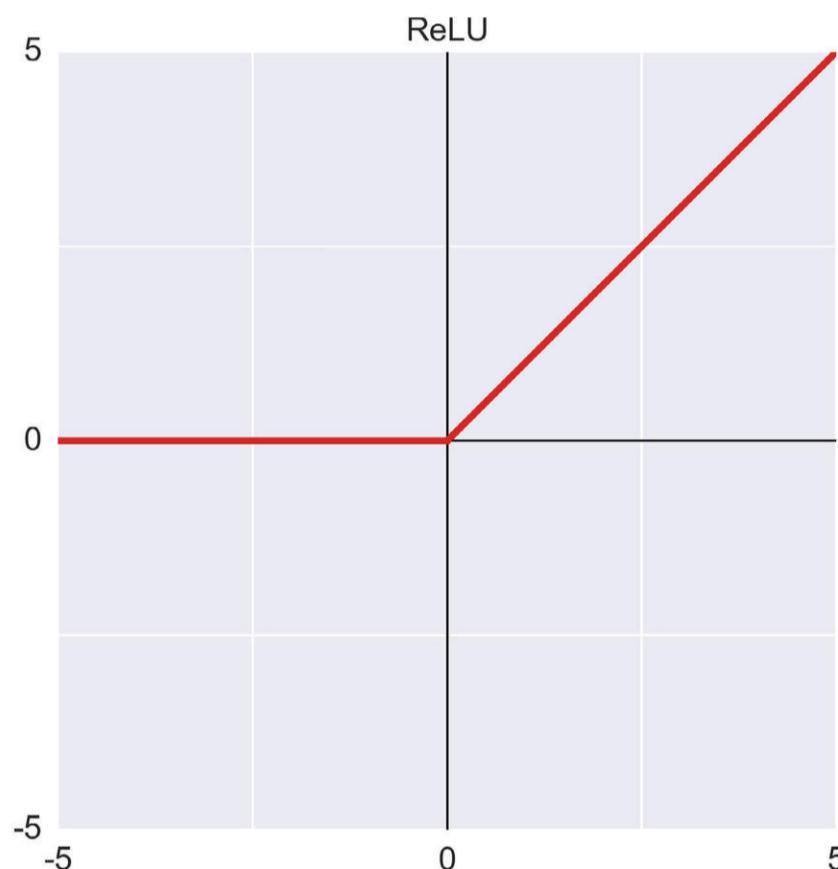


Figure 17.11: The ReLU, or rectified linear unit. It outputs 0 for all negative inputs, otherwise the output is the input.

The ReLU activation function is popular because it’s a simple and fast way to include a non-linearity step in our artificial neurons. We’ll see in Chapter 18 that it can run into some problems, which has led to the development of the ReLU variations below. But on the whole, ReLU (or leaky ReLU, which we’ll see next) performs well and is often the first choice of activation function when building a new network. Beyond the fact that these choices work well in practice, there are good mathematical reasons to like using ReLU [Limmer17].

The **leaky ReLU** changes the response for negative values. Rather than output a 0 for any negative value, this function outputs the input scaled down by a factor of 10. Figure 17.12 shows this function.

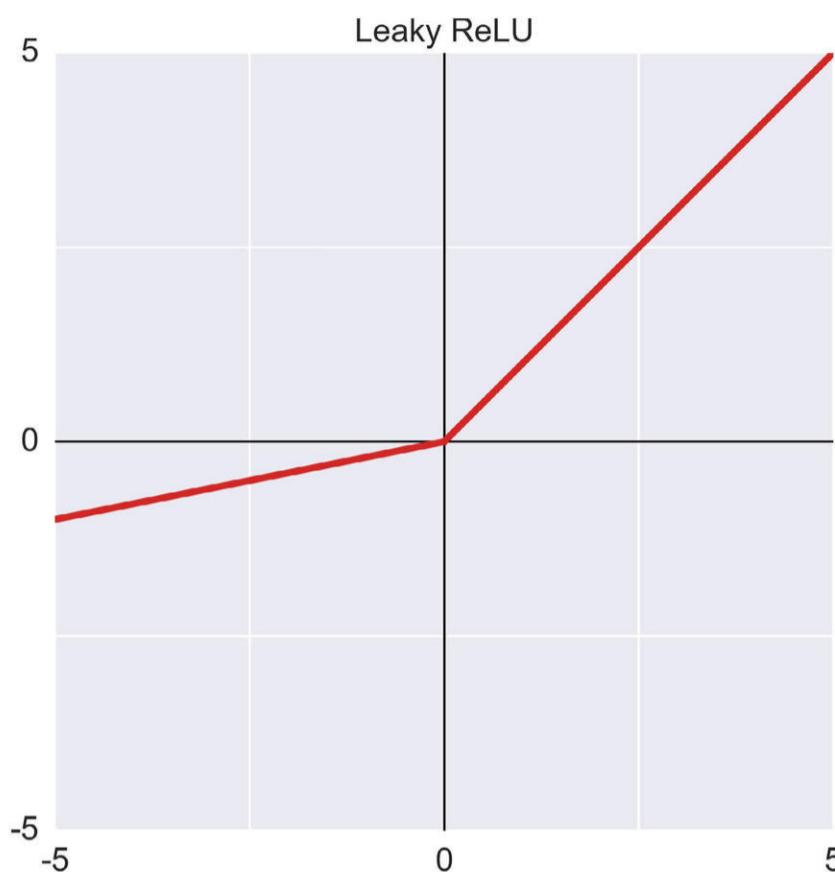


Figure 17.12: The leaky ReLU is like the ReLU, but returns a scaled-down value of X when X is negative.

Of course, there's no need to always scale down the negative values by a factor of 10. A **parametric ReLU** lets us choose how much negative amounts are scaled, as shown in Figure 17.13.

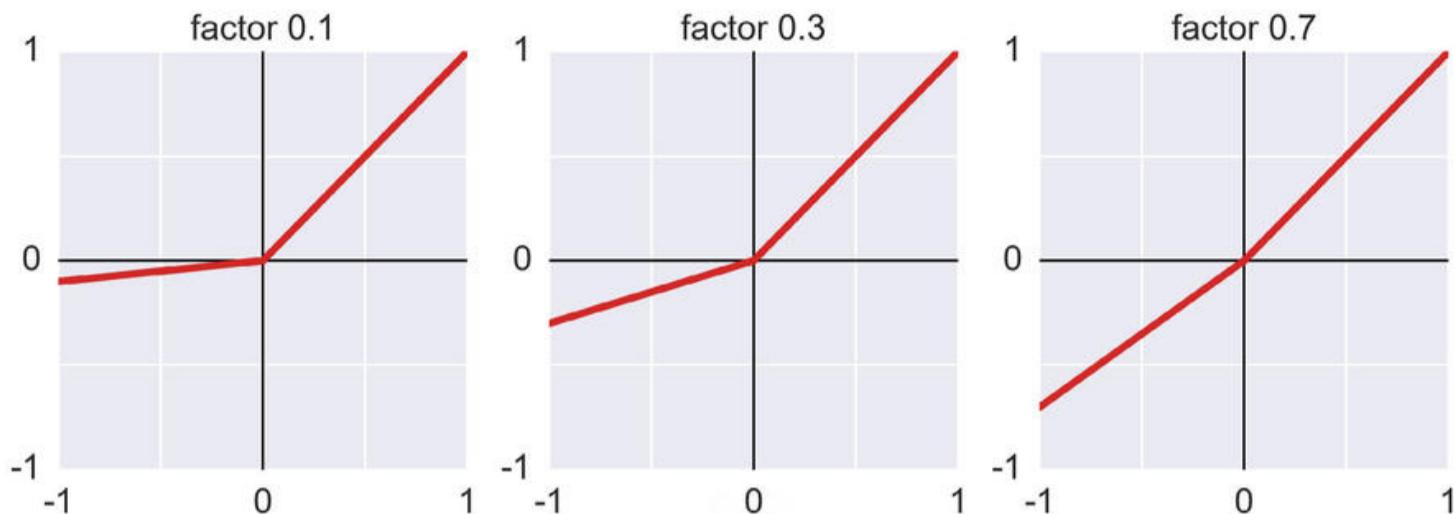


Figure 17.13: A parametric ReLU is like a leaky ReLU, but the slope for values of X less than 0 can be adjusted. Left: A scaling factor of 0.1, resulting in a standard leaky ReLU. Center: A scaling factor of 0.3. Right: A scaling factor of 0.7.

The essential thing when using a parametric ReLU is to never select a factor of exactly 1.0, because then we'd lose the kink, the function would be a straight line, and we risk that this neuron will collapse, or combine, with one that follows it.

Another variation on the basic ReLU is the **shifted ReLU**, which just moves the bend down and left. Figure 17.14 shows an example.

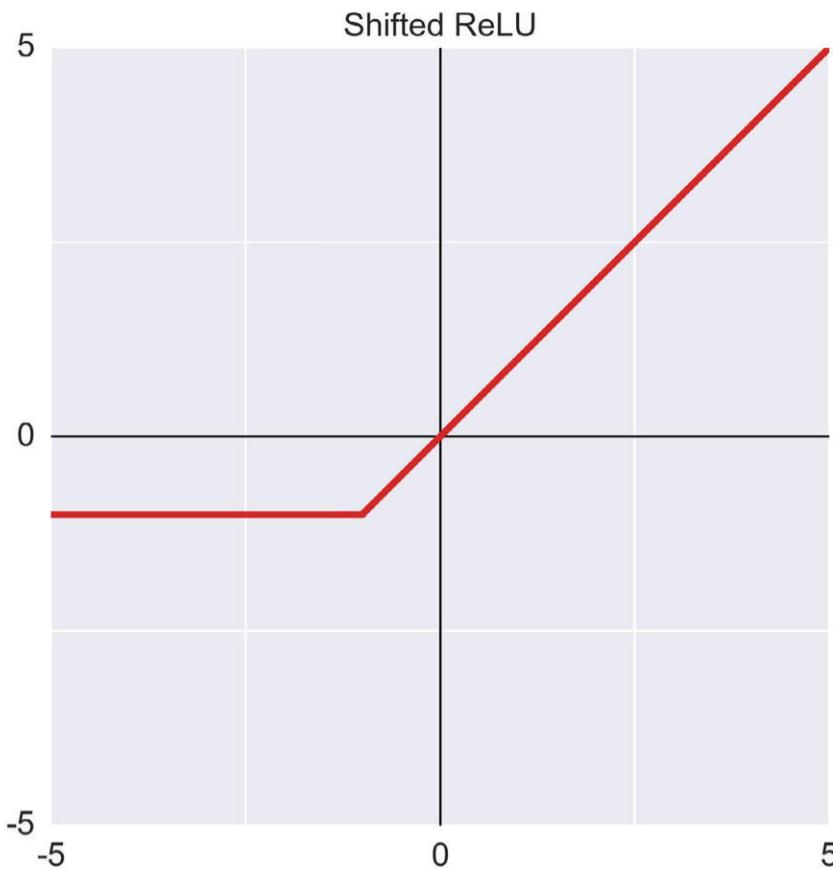


Figure 17.14: The shifted ReLU moves the bend in the ReLU function down and left.

We can combine the basic ReLU and the leaky (or parametric) ReLU, creating an activation function called **maxout** [Goodfellow13]. 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. In other words, we find our input on the X axis, and then select the output with the largest Y value. One way to think about this is to imagine pouring paint downwards, into the graph with all the straight lines plotted. The surface that gets covered with paint is the output of maxout. Figure 17.15 shows maxout with just two lines, approximating a ReLU, as well as with a few more lines to create more complex shapes.

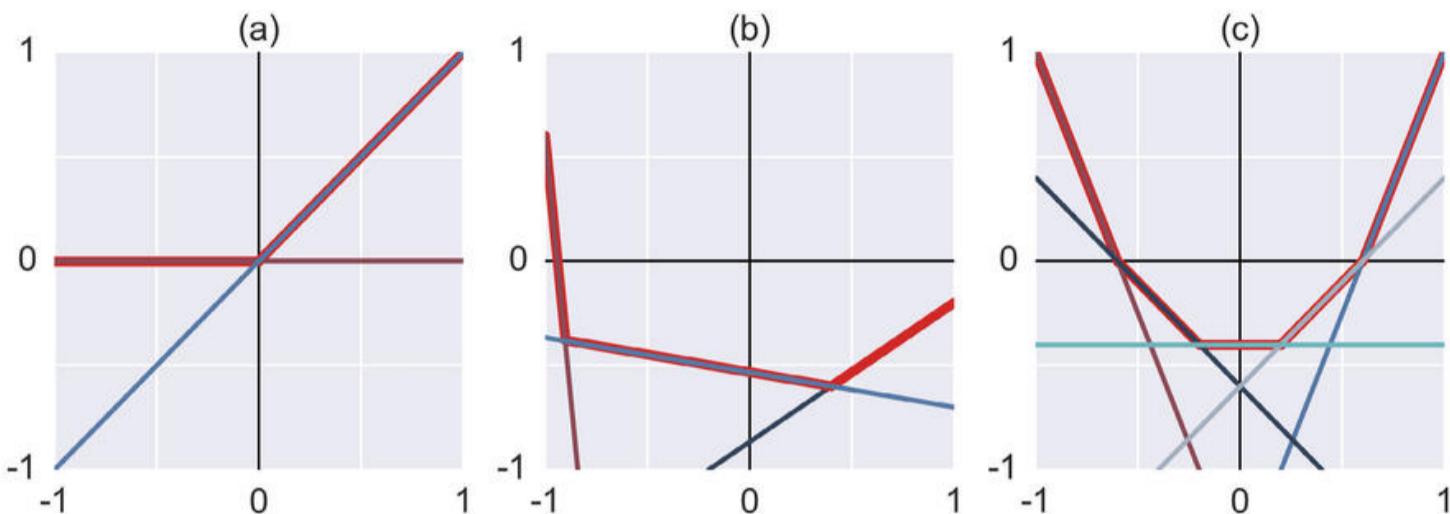


Figure 17.15: The maxout function lets us build up a function from multiple straight lines. The output of the function for any value of X is that of the line with the largest value of Y for that X . The heavy red line is the output of maxout for each set of lines. (a) Maxout forming a ReLU. (b) Maxout forming a lopsided bowl. (c) Maxout forming a symmetrical bowl.

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**, but is not frequently used in basic neural networks.

17.6 Smooth Functions

As we'll see in Chapter 18, a key step in teaching neural networks involves computing derivatives for the outputs of neurons, including their activation functions.

The various forms of ReLU that we saw in the last section create their non-linearities by using multiple straight lines with at least one kink, or bend, in the collection. We said in Chapter 5 that mathematically, there is no derivative in a kink between a pair of straight lines.

So if these kinks prevent the computation of derivatives, which are necessary for teaching a network, why are ReLU functions so popular? It turns out that there are standard mathematical tools that can finesse the sharp corners like those in ReLU functions and still get a

derivative [Oppenheim96]. These tricks don't work on all functions, but one of the principles that guided the development of the functions we saw above 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. So let's look at a few popular smooth activation functions.

The **softplus** function simply smooths out the ReLU, as shown in Figure 17.16.

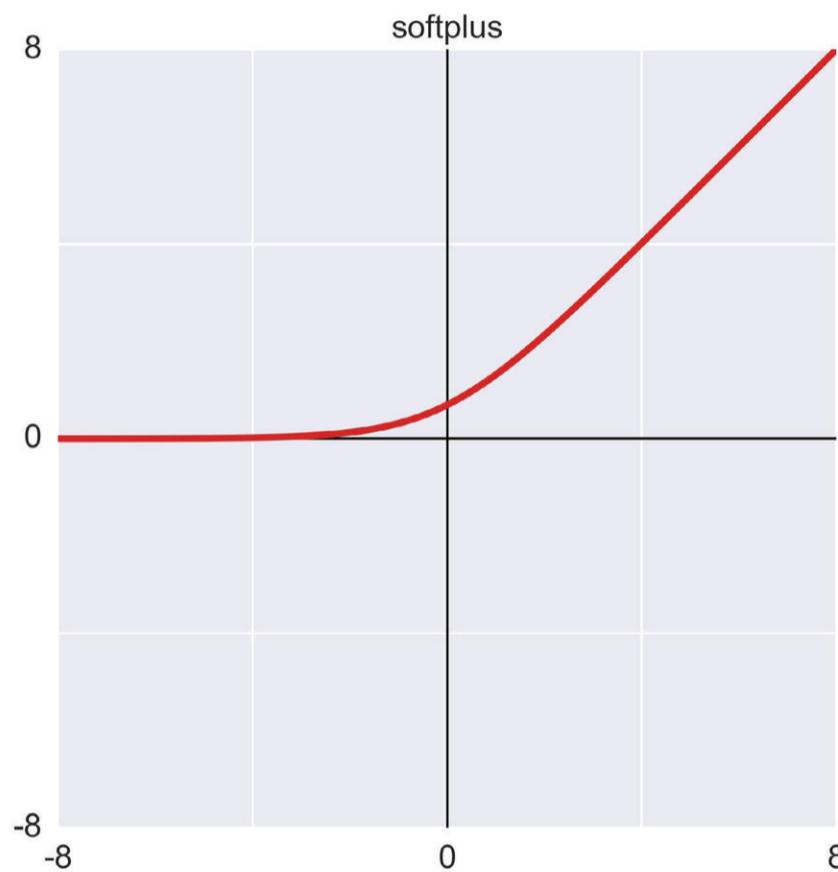


Figure 17.16: The softplus function is smoothed version of the ReLU.

We can smooth out the shifted ReLU as well. This is called the **exponential ReLU**, or **ELU** [Clevert16]. It's shown in Figure 17.17.

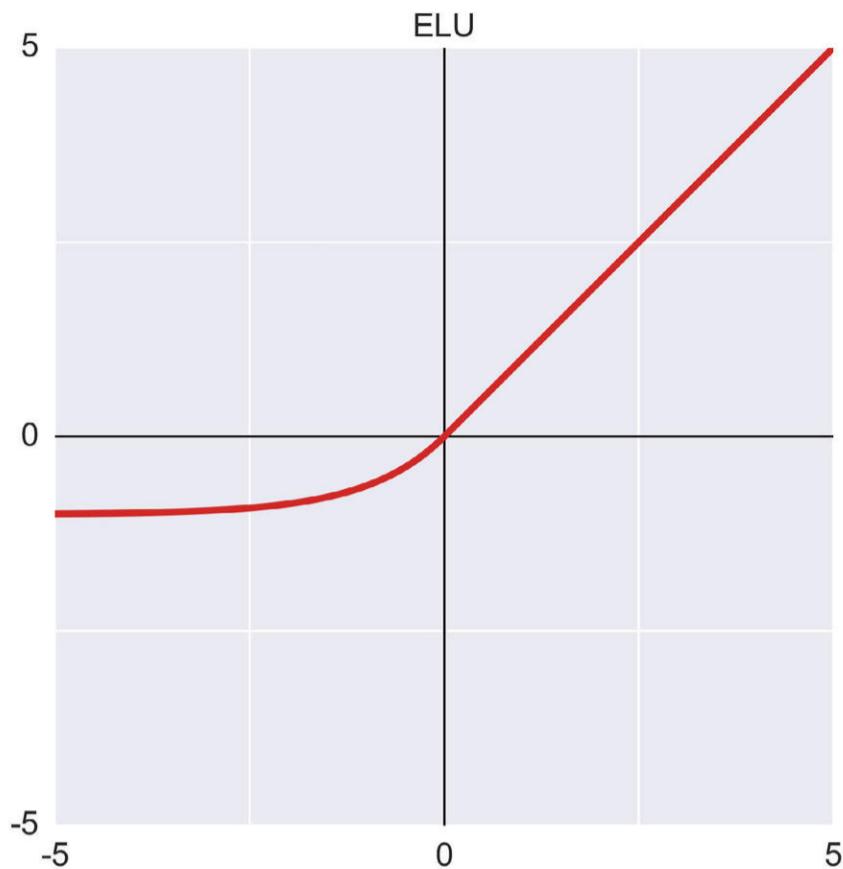


Figure 17.17: The exponential ReLU, or ELU.

Another popular smooth activation function is called the **sigmoid**, also called the **logistic function** or **logistic curve**. The name “sigmoid” comes from the resemblance of the curve to an S shape, while the other names refer to its mathematical structure. Figure 17.18 shows this function.

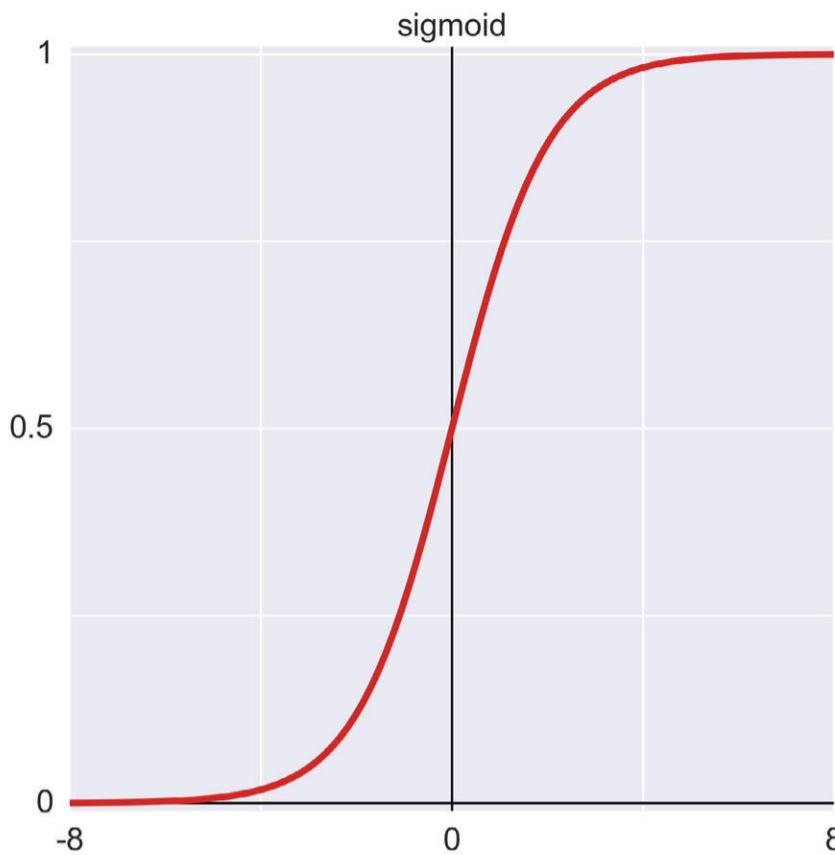


Figure 17.18: The S-shaped sigmoid function is also called the logistic function or logistic curve. It has a value of 0 for very negative inputs, and a value of 1 for very positive inputs. In the range of about -6 to 6, it smoothly transitions between the two.

Closely related to the sigmoid is another mathematical function called the **hyperbolic tangent**. 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 17.19.

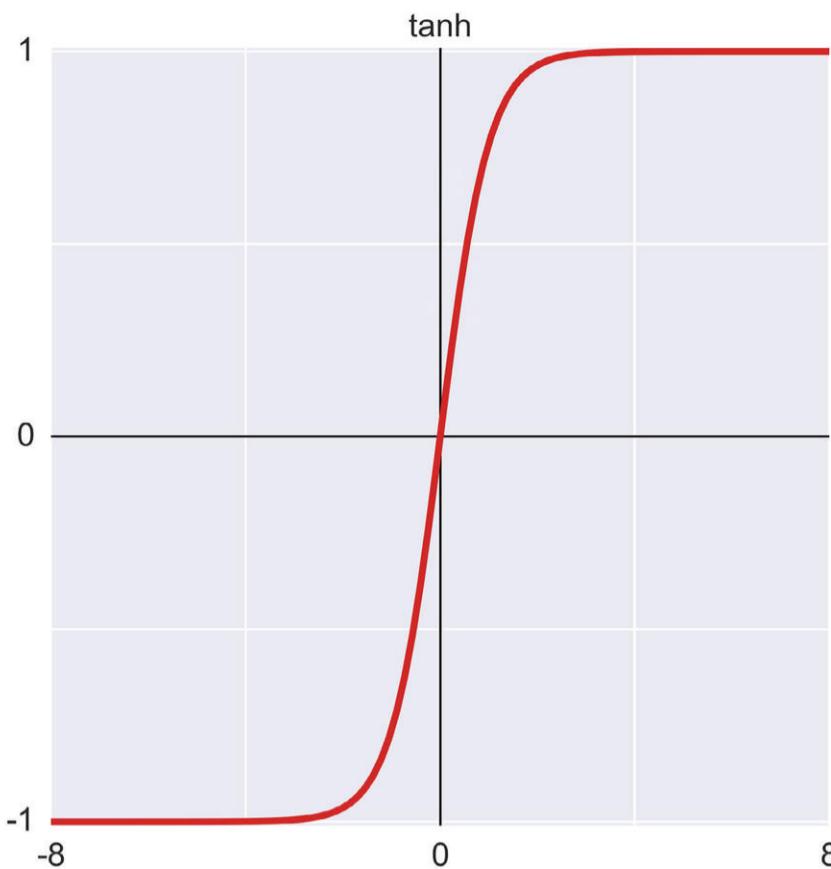


Figure 17.19: The hyperbolic tangent function, written \tanh , is S-shaped like the sigmoid of Figure 17.18. The key differences are that it returns a value of -1 for very negative inputs, and the transition zone is a bit narrower.

We say that the sigmoid and tanh functions both **squash** the entire input range (from negative to positive infinity) into a small range of values. The sigmoid squashes all inputs to the range $[0,1]$, while tanh squashes them to $[-1,1]$.

The two are shown on top of one another in Figure 17.20.

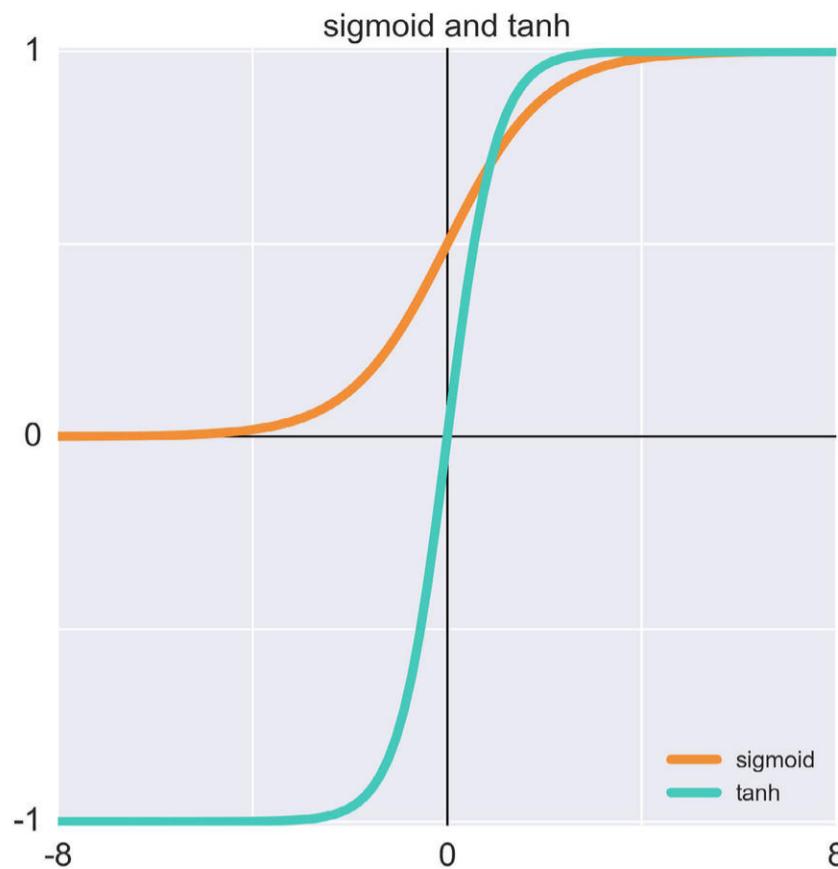


Figure 17.20: The sigmoid function (red) and tanh function (blue), both plotted for the range -8 to 8 .

A combination of the basic ReLU and sigmoid shapes is called **swish** [Ramachandran17]. Figure 17.21 shows what this looks like. In essence it's a ReLU, but with a small, smooth bump just to left of 0, which then flattens out.

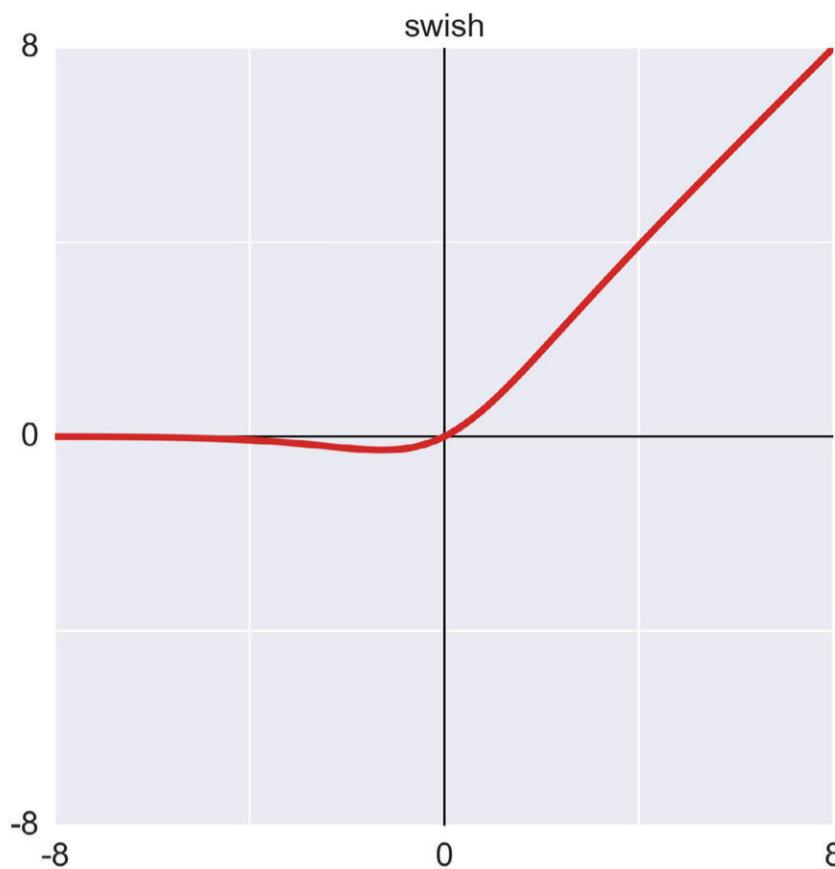


Figure 17.21: The swish activation function, combining the ReLU and sigmoid.

We've mentioned that the ReLU is probably the most popular activation in use today. But when we're using an algorithm that requires a real derivative from the activation function (rather than the tricks that let us use the piecewise-linear ReLU and maxout functions), then sigmoid and tanh are usually our first choices.

A downside of the smooth curves is that they only produce useful results in a narrow range of inputs. For example, when input values get to be larger than about 6, or smaller than about -6 , sigmoid and tanh will always give us back about the same value. This phenomenon is called **saturation**. When the output is always the same value, the derivative of the function becomes 0. We'll see in Chapter 18 that the value of the derivative is essential information when we train a neural network. When the derivative goes to 0, training can come to a standstill.

The ReLU suffers the same problem for negative values, since it always returns 0 for negative values, just a sigmoid does when they're negative enough. But the ReLU doesn't saturate for positive values, since it always returns the input.

These variations from one activation function to the next are some of the reasons why there are multiple activation functions in popular use. There is no firm theory to tell us which activation function will work best in a specific layer of a specific network, so we often need to try out a few of them to see which works best.

17.7 Activation Function Gallery

Figure 17.22 summarizes the activation functions we've discussed.

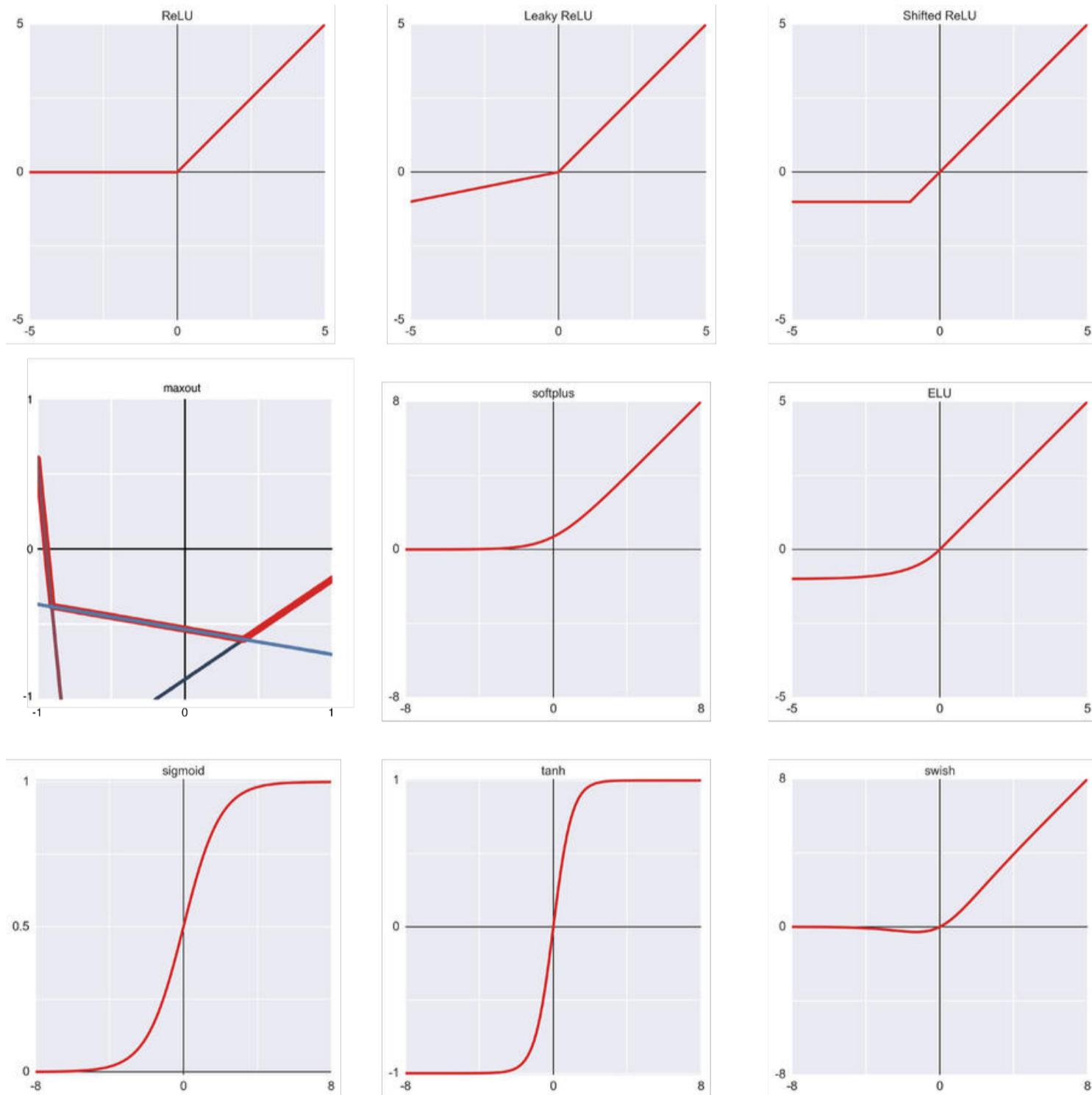


Figure 17.22: A gallery of popular activation functions. Top row, left to right: ReLU, leaky ReLU, shifted ReLU. Middle row: maxout, softplus, ELU. Bottom row: sigmoid, tanh, swish.

17.8 Softmax

There's another kind of operation that we typically apply only at the output neurons of a classifier neural network, and even then only if there are 2 or more such neurons. It's not an activation function in the sense we've been using the term, because it applies simultaneously to all the output neurons, not just one. But we'll cover it here because this is a natural place to discuss this important tool.

The technique is called **softmax**, and we often use it as a final step for classification networks with multiple outputs.

The name is the same as the softmax activation function we saw above, because they're based on similar mathematics. But this technique is not an activation function.

This version of softmax turns the raw numbers that come out of the network into class probabilities. Figure 17.23 shows the idea.

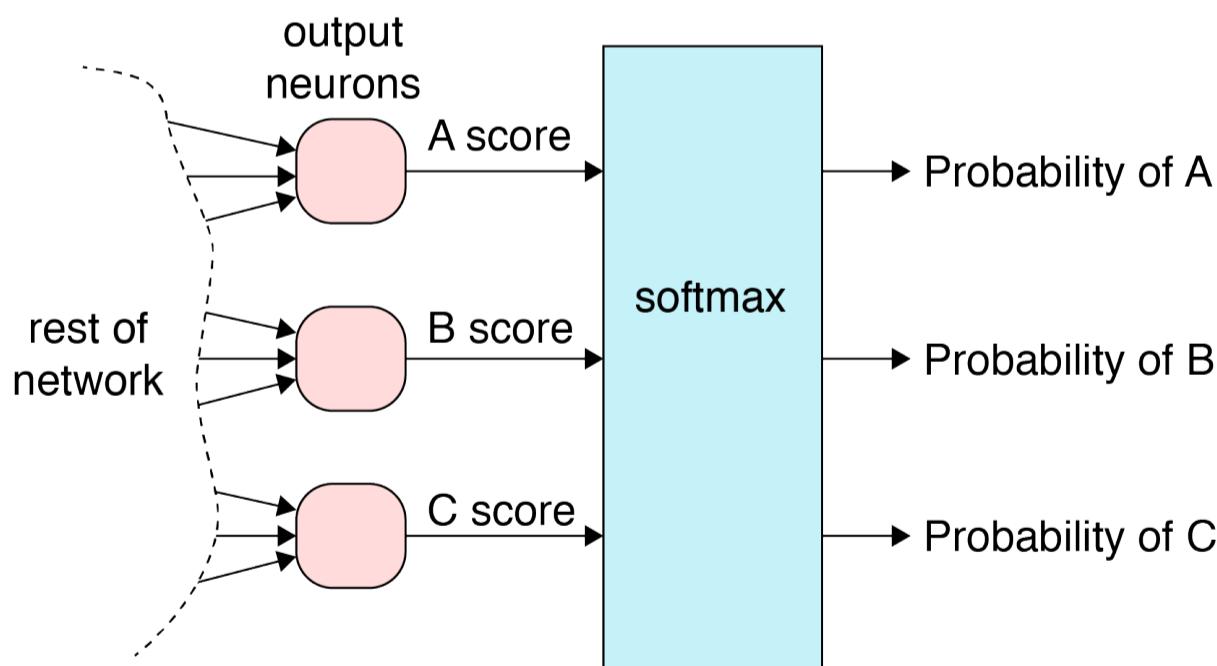


Figure 17.23: The softmax function takes all the network's outputs and modifies them simultaneously. The result is that the scores are turned into probabilities.

Each output neuron presents a value, or score, that corresponds to how much the network thinks that particular input belongs to the corresponding category. In Figure 17.23 we're assuming that we have three categories in our data, named A, B, and C, so each of the three output neurons gives us a score for its category. The larger the score, the more certain the system is that the input belongs to that category.

From this, we can sort the scores in order of how much the network considers the corresponding category to be the correct one. The category with the highest score is the one that the network considers its top choice. That's useful.

But even more useful would be if we could get **probabilities** from the network. For example, suppose that the three values coming out of the three neurons in Figure 17.23 were, say, 0.6 for A, 0.3 for B, and 0.1 for C. Then we could say that A is twice as probable as B, and 6 times more probable than C. We could also say B is 3 times more probable than C.

We can't make these kinds of statements working just from the scores, because they're not designed to be interpreted this way. If we want to make statements about likelihoods and probabilities, we have to turn them into probabilities. Softmax is the most common way to do this.

A mathematically important characteristic of any set of probabilities is that they add up to 1. If we were just to modify each output of the network independently, we wouldn't know the other values, so we couldn't make sure they added up to anything in particular. By handing all the outputs to softmax, it can simultaneously adjust all the values so that they represent probabilities and sum to 1.

Let's look at softmax in action.

Consider the top-left graph of Figure 17.24. This shows the outputs for a classifier with six output neurons, which we've labeled A through F. From this graph, we can see that the value for category B is 0.1 and the value for category C is 0.8. As we've discussed, it would be a mistake to conclude from this that the input is 8 times more likely to be

in category C than category B, because these are scores and not probabilities. To compare them to each other, we need to change them into probabilities.



Figure 17.24: Applying softmax to different sets of output. Top row: Scores from a classifier. Bottom row: Results of running the scores in the top row through softmax.

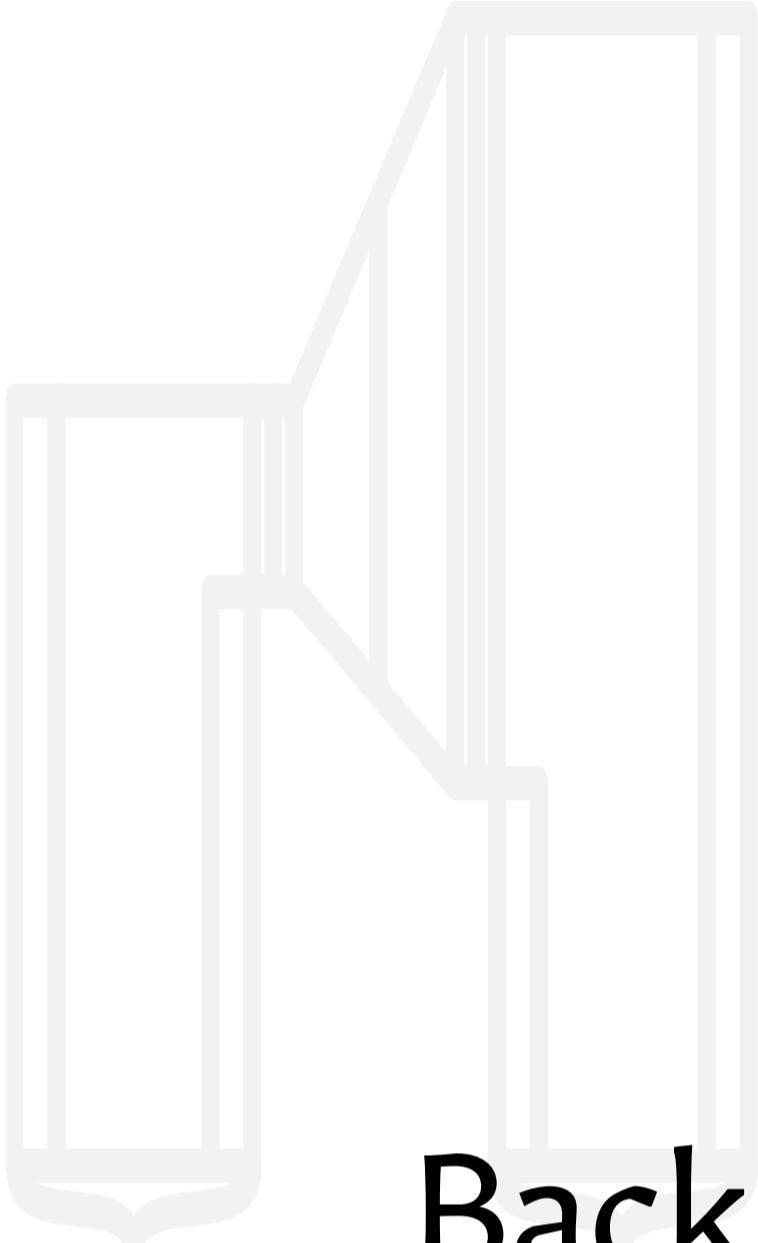
We show the results of applying softmax in the graph in the lower left. These are the probabilities of the input belonging to each of the six classes. It's interesting to note that the big values, like C and F, get scaled down by a lot, but the small values, like B, are hardly scaled at all. But the 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 this we can conclude that the input is a little less than 2 times more probable to be in category C than category B.

The middle and right columns of Figure 17.24 show the outputs for two other hypothetical networks and inputs, before and after softmax.

Because softmax turns our scores into probabilities, it's widely used at the end of neural networks that are used for classification.

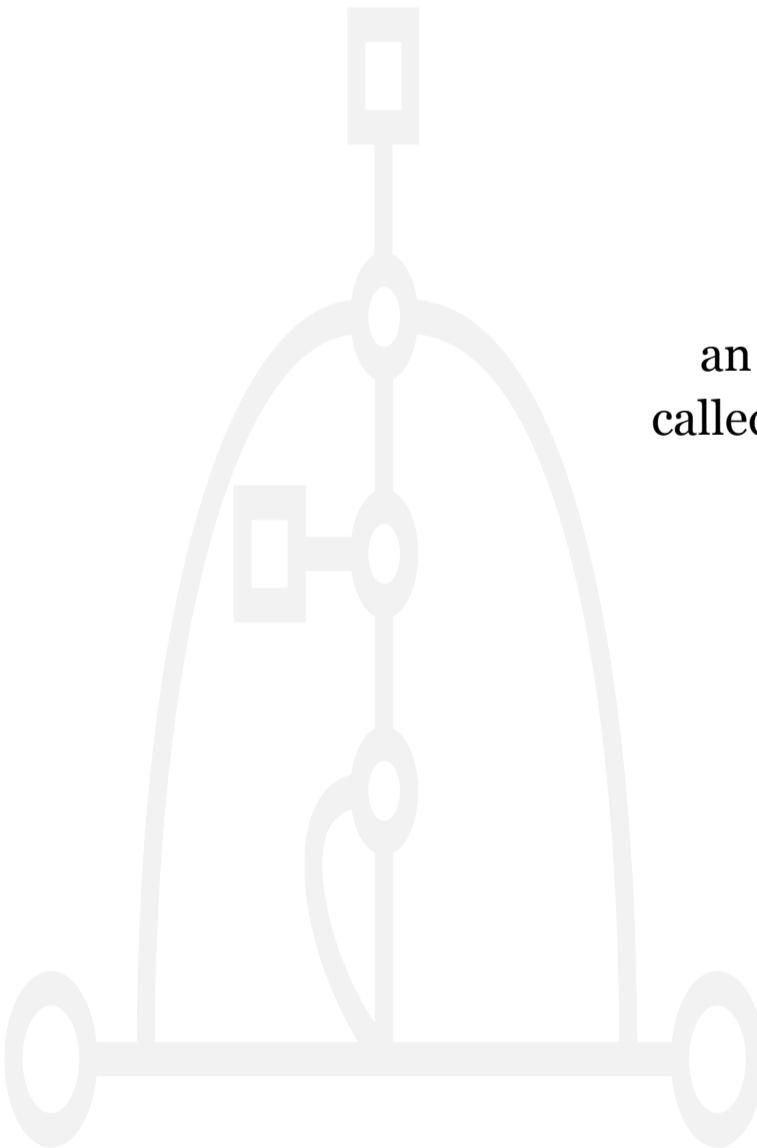
References

- [Clevert16] Djork-Arné Clevert, Thomas Unterthiner, Sepp Hochreiter, “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)”, ICLR 2016. <https://arxiv.org/abs/1511.07289>
- [Kuphaldt17] Tony R. Kuphaldt, “Introduction to Diodes And Rectifiers, Chapter 3 - Diodes and Rectifiers”, in “Lessons in Electric Circuits”, 2017. <https://www.allaboutcircuits.com/textbook/semiconductors/chpt-3/introduction-to-diodes-and-rectifiers/>
- [Goodfellow13] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, Yoshua Bengio, “Maxout Networks”, ICML 28(3), pp. 1319-1327, 2013. <http://jmlr.org/proceedings/papers/v28/goodfellow13.pdf>
- [Limmer17] Steffen Limmer and Slawomir Stanczak, “Optimal deep neural networks for sparse recovery via Laplace techniques”, arXiv 1709.01112, 2017. <https://arxiv.org/abs/1709.01112>
- [Oppenheim96] Alan V. Oppenheim and S. Hamid Nawab, “Signals and Systems, Second edition”, Prentice Hall, 1996.
- [Ramachandran17] Prajit Ramachandran, Barret Zoph, Quoc V. Le, “Swish: A Self-Gated Activation Function”, arXiv:1710.05941, 2017. <https://arxiv.org/abs/1710.05941>
- [Stoltz16] Stefan Stoltz, Trains4Africa blog, 2016. <http://trains4africa.co.za/?p=1172>



Chapter 18

Backpropagation



A neural network learns from its mistakes. Each time the system makes an incorrect prediction, we use an algorithm called backpropagation to improve its weights.

Contents

18.1 Why This Chapter Is Here	706
18.1.1 A Word On Subtlety.....	708
18.2 A Very Slow Way to Learn.....	709
18.2.1 A Slow Way to Learn	712
18.2.2 A Faster Way to Learn	716
18.3 No Activation Functions for Now.....	718
18.4 Neuron Outputs and Network Error.....	719
18.4.1 Errors Change Proportionally.....	720
18.5 A Tiny Neural Network.....	726
18.6 Step 1: Deltas for the Output Neurons	732
18.7 Step 2: Using Deltas to Change Weights....	745
18.8 Step 3: Other Neuron Deltas.....	750
18.9 Backprop in Action	758
18.10 Using Activation Functions	765
18.11 The Learning Rate	774
18.11.1 Exploring the Learning Rate.....	777

18.12 Discussion	787
18.12.1 Backprop In One Place	787
18.12.2 What Backprop Doesn't Do	789
18.12.3 What Backprop Does Do	789
18.12.4 Keeping Neurons Happy	790
18.12.5 Mini-Batches.....	795
18.12.6 Parallel Updates	796
18.12.7 Why Backprop Is Attractive	797
18.12.8 Backprop Is Not Guaranteed	797
18.12.9 A Little History	798
18.12.10 Digging into the Math.....	800
References	802

18.1 Why This Chapter Is Here

This chapter is about training a neural network. The very basic idea is appealingly simple. Suppose we’re training a categorizer, which will tell us which of several given labels should be assigned to a given input. It might tell us what animal is featured in a photo, or whether a bone in an image is broken or not, or what song a particular bit of audio belongs to.

Training this neural network involves handing it a sample, and asking it to **predict** that sample’s label. If the prediction matches the label that we previously determined for it, we move on to the next sample. If the prediction is wrong, we change the network to help it do better next time.

Easily said, but not so easily done. This chapter is about how we “change the network” so that it **learns**, or improves its ability to make correct predictions. This approach works beautifully not just for classifiers, but for almost any kind of neural network.

Contrast a feed-forward network of neurons to the dedicated classifiers we saw in Chapter 13. Each of those dedicated algorithms had a customized, built-in learning method that measured the incoming data to provide the information that classifier needed to know.

But a neural network is just a giant collection of neurons, each doing its own little calculation and then passing on its results to other neurons. Even when we organize them into layers, there’s no inherent learning algorithm.

How can we train such a thing to produce the results we want? And how can we do it efficiently?

The answer is called **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 our models in reasonable amounts of time. With backprop, deep learning algorithms are practical and plentiful.

Backprop is a low-level algorithm. When we use libraries to build and train deep learning systems, their finely-tuned routines give us both speed and accuracy. Except as an educational exercise, or to implement some new idea, we're likely to never write our own code to perform backprop.

So why is this chapter here? Why should we bother knowing about this low-level algorithm at all? There are at least four good reasons to have a general knowledge of backpropagation.

First, it's important to understand backprop because knowledge of one's tools is part of becoming a master in any field. Sailors at sea, and pilots in the air, need to understand how their autopilots work in order to use them properly. A photographer with an auto-focus camera needs to know how that feature works, what its limits are, and how to control it, so that she can work with the automated system to capture the images she wants. A basic knowledge of the core techniques of any field is part of the process of gaining proficiency and developing mastery. In this case, knowing something about backprop lets us read the literature, talk to other people about deep learning ideas, and better understand the algorithms and libraries we use.

Second, and more practically, knowing about backprop can help us design networks that learn. When a network learns slowly, or not at all, it can be because something is preventing backprop from running properly. Backprop is a versatile and robust algorithm, but it's not bulletproof. We can easily build networks where backprop won't produce useful changes, resulting in a network that stubbornly refuses to learn. For those times when something's going wrong with backprop, understanding the algorithm helps us fix things [Karpathy16].

Third, many important advances in neural networks rely on backprop intimately. To learn these new ideas, and understand why they work the way they do, it's important to know the algorithms they're building on.

Finally, backprop is an elegant algorithm. It efficiently solves a problem that would otherwise require a prohibitive amount of time and computer resources. It's one of the conceptual treasures of the field. As curious, thoughtful people it's well worth our time to understand this beautiful algorithm.

For these reasons and others, this chapter provides an introduction to backprop. Generally speaking, introductions to backprop are presented mathematically, as a collection of equations with associated discussion [Fullér10]. As usual, we'll skip the mathematics and focus instead on the concepts. The mechanics are common-sense at their core, and don't require any tools beyond basic arithmetic and the ideas of a derivative and gradient, which we discussed in Chapter 5.

18.1.1 A Word On Subtlety

The backpropagation algorithm is not complicated. In fact, it's remarkably simple, which is why it can be implemented so efficiently.

But simple does not always mean easy.

The backprop algorithm is subtle. In the discussion below, the algorithm will take shape through a process of observations and reasoning, and these steps may take some thought. We'll try to be clear about every step, but making the leap from reading to understanding may require some work.

It's worth the effort.

18.2 A Very Slow Way to Learn

Let's begin with a very slow way to train a neural network. This will give us a good starting point, which we'll then improve.

Suppose we've been given a brand-new neural network consisting of hundreds or even tens of thousands of interconnected neurons. The network was designed to classify each input into one of 5 categories. So it has 5 outputs, which we'll number 1 to 5, and whichever one has the largest output is the network's prediction for an input's category. Figure 18.1 shows the idea.

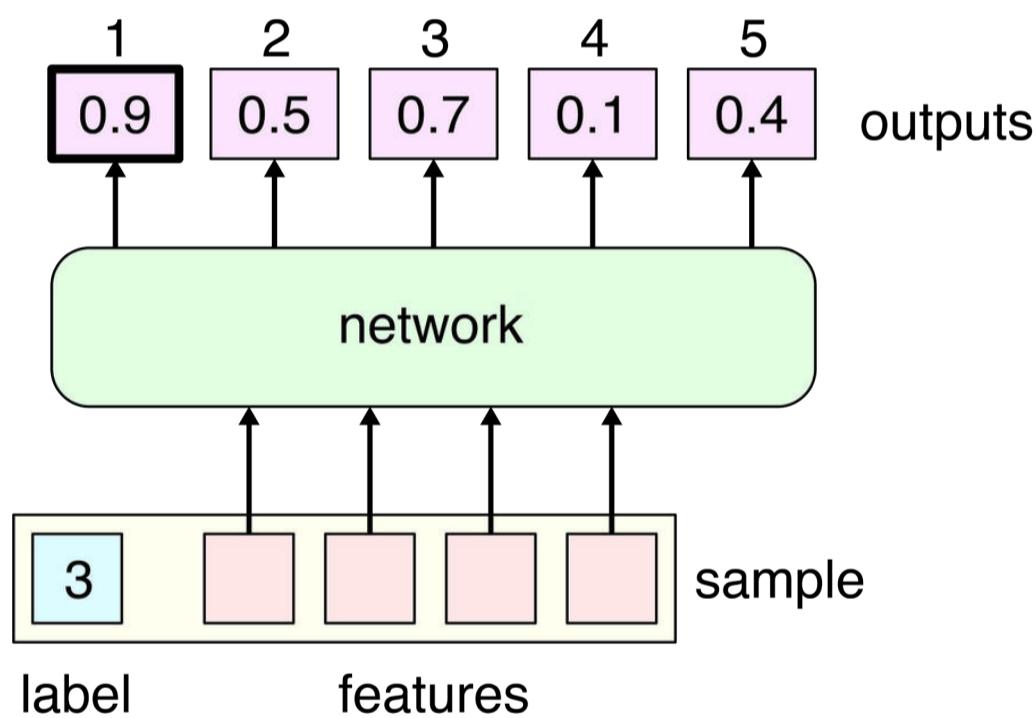


Figure 18.1: A neural network predicting the class of an input sample.

Starting at the bottom of Figure 18.1, we have a sample with four features and a label. The label tells us that the sample belongs to category 3. The features go into a neural network which has been designed to provide 5 outputs, one for each class. In this example, the network has incorrectly decided that the input belongs to class 1, because the largest output, 0.9, is from output number 1.

Consider the state of our brand-new network, before it has seen any inputs. As we know from Chapter 16, each input to each neuron has an associated weight. There could easily be hundreds of thousands, or many millions, of weights in our network. Typically, all of these weights will have been initialized with small random numbers.

Let's now run one piece of labeled training data through the net, as in Figure 18.1. The sample's features go into the first layer of neurons, and the outputs of those neurons go into more neurons, and so on, until they finally arrive at the output neurons, when they become the output of the network. The index of the output neuron with the largest value is the predicted class for this sample.

Since we're starting with random numbers for our weights, we're likely to get essentially random outputs. So there's a 1 in 5 chance the network will happen to predict the right label for this sample. But there's a 4 in 5 chance it'll get it wrong, so let's assume that the network predicts the wrong category.

When the prediction doesn't match the label, we can measure the error numerically, coming up with a single number to tell us just how wrong this answer is. We call this number the **error score**, or **error**, or sometimes the **loss** (if the word "loss" seems like a strange synonym for "error," it may help to think to think of it as describing how much information is "lost" if we categorize a sample using the output of the classifier, rather than the label.).

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 0 means that the network predicted this sample's label correctly. In a perfect world, we'd get the error down to 0 for every sample in the training set. In practice, we usually settle for getting as close as we can.

Let's briefly recap some terminology from previous chapters. When we speak of "the network's error" with respect to a training set, we usually mean some kind of overall average that tells us how the network is doing when taking all the training samples into consideration. We call this the **training error**, since it's the overall error we get from predicting results from the training set. Similarly, the error from the test or validation data is called the **test error** or **validation error**. When the system is deployed, a measure of the mistakes it makes on new data is called the **generalization error**, because it represents how well (or poorly) the system manages to "generalize" from its training data to new, real-world data.

A nice way to think about the whole training process is to anthropomorphize the network. We can say that it "wants" to get its error down to zero, and the whole point of the learning process is to help it achieve that goal.

One advantage of this way of thinking is that we can make the network do anything we want, just by setting up the error to "punish" any quality or behavior that we don't want. Since the algorithms we'll see in this chapter are designed to minimize the error, we know that anything about the network's behavior that contributes to the error will get minimized.

The most natural thing to punish is getting the wrong answer, so the error almost always includes a term that measures how far the output is from the correct label. The worse the match between the prediction and the label, the bigger this term will be. Since the network wants to minimize the error, it will naturally minimize such mistakes.

This approach of "punishing" the network through the error score means we can choose to include terms in the error for anything we can measure and want to suppress. For example, another popular measure to add into the error is a **regularization term**, where we look at the magnitude of all the weights in the network. As we'll see later in this chapter, we usually want those weights to be "small," which often means between -1 and 1 . As the weights move beyond this range, we

add a larger number to the error. Since the network “wants” the smallest error possible, it will try to keep the weights small so that this term remains small.

All of this raises the natural question of how on earth the network is able to accomplish this goal of minimizing the error. That’s the point of this chapter.

Let’s start with a basic error measure that only punishes a mismatch between the network’s prediction and the label.

Our first algorithm for teaching the network will be just a thought experiment, since it would be absurdly slow on today’s computers. But the motivation is right, and this slow algorithm will form the conceptual basis for the more efficient techniques we’ll see later in this chapter.

18.2.1 A Slow Way to Learn

Let’s stick with our running example of a classifier. We’ll give the network a sample and compare the system’s prediction with the sample’s label.

If the network got it right and predicted the correct label, we won’t change anything and we’ll move on to the next sample. As the wise man said, “If it ain’t broke, don’t fix it” [Seung05].

But if the result for a particular sample is incorrect (that is, the category with the highest value does not match our label), we will try to improve things. That is, we’ll learn from our mistakes.

How do we learn from this mistake? Let’s stick with this sample for a while and try to help the network do a better job with it. First, we’ll pick a small random number (which might be positive or negative). Now we’ll pick one weight at random from the thousands or millions of weights in the network, and we’ll add our small random value to that weight.

Now we'll evaluate our sample again. Everything up to that change will be the same as before. But there will be a chain reaction of changes in the outputs of the neurons starting at the weight we modified. The new weight will produce a new input for the neuron that uses that input, which will change that neuron's output value, which will change the output of every neuron that uses that output, which will change the output of every neuron that uses any of *those* outputs, and so on. Figure 18.2 shows this idea graphically.

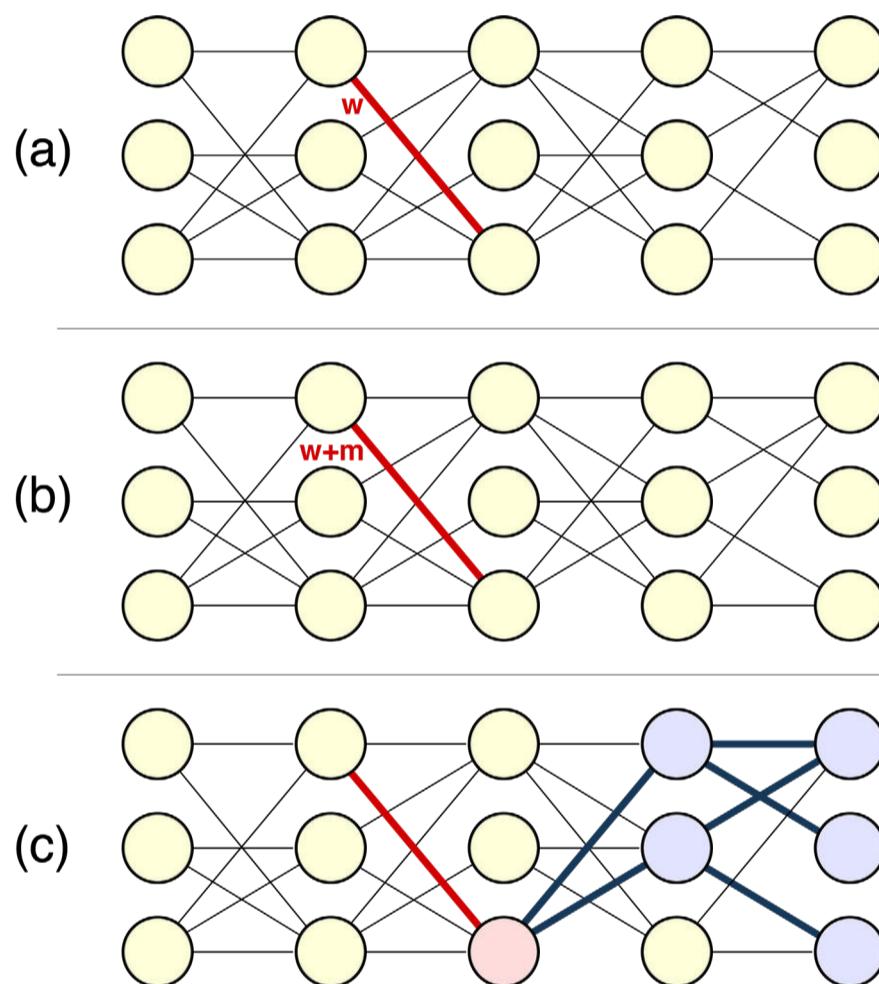


Figure 18.2: Updating a single weight causes a chain reaction that ultimately can change the network's outputs.

Figure 18.2 shows a network of 5 layers with 3 neurons each. Data flows from the inputs at the left to the outputs at the right. For simplicity, not every neuron uses the output of every neuron on the previous layer. In part (a) we select one weight at random, here shown in red and marked w . In part (b) we modify the weight by adding a value m to it, so the weight is now $w+m$. When we run the sample through the network again, as shown in part (c), the new weight causes a change

in the output of the neuron it feeds into (in red). The output of that neuron changes as a result, which causes the neurons it feeds into to change their outputs, and the changes cascade all the way to the output layer.

Now that we have a new output, we can compare it to the label and measure the new error. If the new error is less than the previous error, then we've made things better! We'll keep this change, and move on to the next sample.

But if the results didn't get better then we'll undo this change, restoring the weight back to its previous value. We'll then pick a new random weight, change it by a newly-selected small random amount, and evaluate the network again.

We can continue this process of picking and nudging weights until the results improve, or we decide we've tried enough times, or for any other reason we decide to stop. Then we just move on to the next sample.

When we've used all the samples in our training set, we'll just go through them all again (maybe in a different order), over and over. The idea is that we'll improve a little bit from every mistake.

We can continue this process until the network classifies every input correctly, or we've come close enough, or our patience is exhausted.

With this technique, we would expect the network to slowly improve, though there may be setbacks along the way. For example, adjusting a weight to improve one sample's prediction might ruin the prediction for one or more other samples. If so, when those samples come along they will cause their own changes to improve their performance.

This thought algorithm isn't perfect, because things could get stuck. For example, there might be times when we need to adjust more than one weight simultaneously. To fix that, we can imagine extending our algorithm to assign multiple random changes to multiple random weights. But let's stick with the simpler version for now.

Given enough time and resources, the network would eventually find a value for every weight that either predicts the right answer for every sample, or it comes as close as that network possibly 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” [NASA12].

This technique, while a valid way to teach a network, 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.

But this *is* the core idea. To train our network, we’ll watch its output, and when it makes mistakes, we’ll adjust the weights to make those mistakes less likely. Our goal in this chapter will be to take this rough idea and re-structure it into a vastly more practical algorithm.

Before we move on, it’s worth noting that we’ve been talking about weights, but not the bias term belonging to every neuron. We know that every neuron’s bias gets added in along with the neuron’s weighted inputs, so changing the bias would also change the output. Doesn’t that mean that we want to adjust the bias values as well? We sure do. But thanks to the **bias trick** we saw in Chapter 10, we don’t have to think about the bias explicitly. That little bit of relabeling sets up the bias to look like an input with its own weight, just like all the other inputs. The beauty of this arrangement is that it means that as far as our training algorithm is concerned, the bias is just another weight to adjust. In other words, all we need to think about is adjusting weights, and the bias weights will automatically get adjusted along the way with all the other weights.

Let’s now consider how we might improve our incredibly slow weight-changing algorithm.

18.2.2 A Faster Way to Learn

The algorithm of the last section would improve our network, but at a glacial pace.

One big source of inefficiency is that half of our adjustments to the weights are in the wrong direction: we add a value when we should instead 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, requiring us to evaluate an immense number of samples. Let's solve these problems.

We could avoid making mistakes if we knew beforehand whether we wanted to nudge each weight along the number line to the right (that is, make it more positive) or to the left (and make it more negative).

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. In 1D (where the gradient is also called the **derivative**), the gradient is the slope of a curve above a specific point. Our curve describes the network's error, and our point is the value of a weight. If the slope of the error (the gradient) above the weight is positive (that is, the line goes up as we move to the right), then moving the point to the right will cause the error to go up. More useful to us is that moving the point to the left will cause the error to go down. If the slope of the error is negative, the situations are reversed.

Figure 18.3 shows two examples.

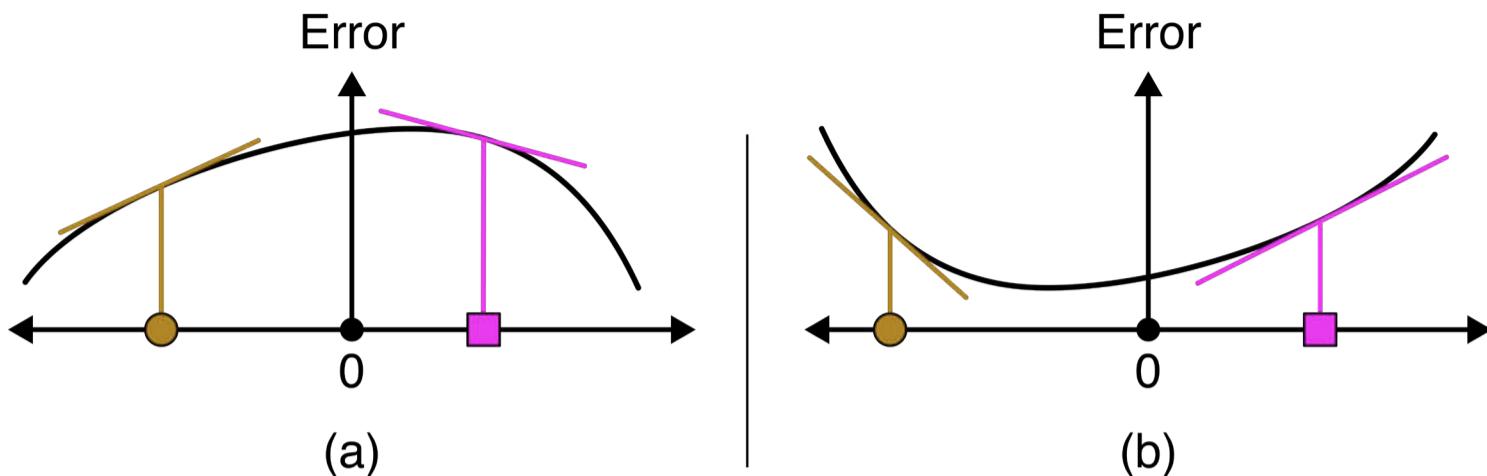


Figure 18.3: The gradient tells us what will happen to the error (the black curves) if we move a weight to the right. The gradient is given by the slope of the curve directly above the point we're interested in. Lines that go up as we move right have a positive slope, otherwise they are negative.

In Figure 18.3(a), we see that if we move the round weight to the right, the error will increase, because the slope of the error is positive. To reduce the error, we need to move the round point left. The square point's gradient is negative, so we reduce the error by moving that point right. Part (b) shows the gradient for the round point is negative, so moving to the right will reduce the error. The square point's gradient is positive, so we reduce the error by moving that point to the left.

If we had the gradient for a weight, we could always adjust it exactly as needed to make the error go down.

Using the gradients wouldn't be much of an advantage if they were time-consuming to compute, so as our second improvement let's suppose that we can calculate the gradients for the weights very efficiently. In fact, let's suppose that we could quickly calculate the gradient for *every* weight in the whole network. Then we could update *all of the weights simultaneously* by adding a small value (positive or negative) to each weight in the direction given by its own individual gradient. That would be an immense time-saver.

Putting these together gives us a plan where we'll run a sample through the network, measure the output, compute the gradient for every weight, and then use the gradient at each weight to move that weight to the right or the left. This is exactly what we're going to do.

This plan makes knowing the gradient an important issue. Finding the gradient efficiently is the main goal of this chapter.

Before we continue, it's worth noticing that this algorithm makes the assumption that tweaking all the weights independently and simultaneously will lead to a reduction in the error. This is a bold assumption, because we've already seen how changing one weight can cause ripple effects through the rest of the network. Those effects could change the values of other neurons, which in turn would change their gradients. We won't get into the details now, but we'll see later that if we make the changes to the weights small enough, that assumption will generally hold true, and the error will indeed go down.

18.3 No Activation Functions for Now

For the next few sections in this chapter, we're going to simplify the discussion by pretending that our neurons don't have activation functions.

As we saw in Chapter 17, activation functions are essential to keep our whole network from becoming nothing more than the equivalent of a single neuron. So we need to use them.

But if we include them in our initial discussion of backprop, things will get complicated, fast. If we leave activation functions out for just a moment, the logic is much easier to follow. We'll put them back in again at the end.

Since we're temporarily pretending that there are no activation functions in our neurons, neurons in the following discussions just sum up their weighted inputs, and present that sum as their output, as in Figure 18.4. As before, each weight is named with a two-letter composite of the neuron it's coming from and the neuron it's going into.

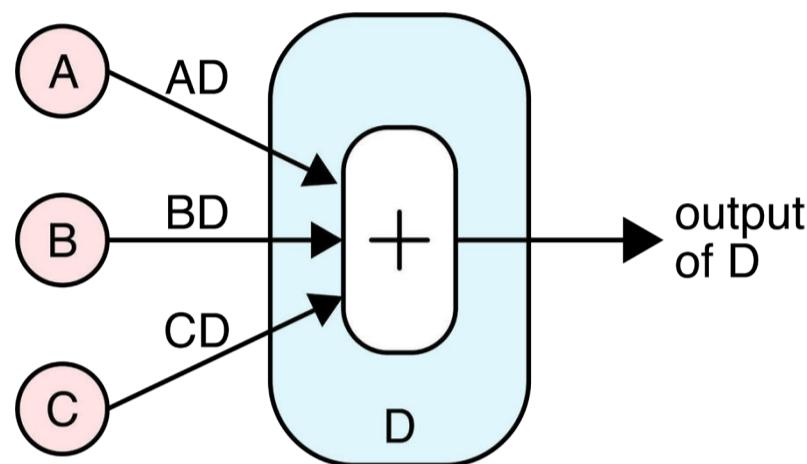


Figure 18.4: Neuron D simply sums up its incoming values, and presents that sum as its output. Here we've explicitly named the weights on each connection into neuron D.

Until we put explicitly put activation functions back in, our neurons will emit nothing more than the sum of their weighted inputs.

18.4 Neuron Outputs and Network Error

Our goal is to reduce the overall error for a sample, by adjusting the network's weights.

We'll do this in two steps. In the first step, we calculate and store a number called the “delta” for every neuron. This number is related to the network's error, as we'll see below. This step is performed by the **backpropagation** algorithm.

The second step uses those delta values at the neurons to update the weights. This step is called the **update** step. It's not typically considered part of backpropagation, but sometimes people casually roll the two steps together and call the whole thing "backpropagation."

The overall plan now is to run a sample through the network, get the prediction, and compare that prediction to the label to get an error. If their error is greater than 0, we use it to compute and store a number we'll call "delta" at every neuron. We use these delta values and the neuron outputs to calculate an update value for each weight. The final step is to apply every weight's individual update so that it takes on a new value.

Then we move on to the next sample, and repeat the process, over and over again until the predictions are all perfect or we decide to stop.

Let's now look at this mysterious "delta" value that we store at each neuron.

18.4.1 Errors Change Proportionally

There are two key observations that will make sense of everything to follow. These are both based on how the network behaves when we ignore the activation functions, which we're doing for the moment. As promised above, we'll put them back in later in this chapter.

The first observation is this: *When any neuron output in our network changes, the output error changes by a proportional amount.*

Let's unpack that statement.

Since we're ignoring activation functions, there are really only two types of values we care about in the system: weights (which we can set and change as we please), and neuron outputs (which are computed automatically, and which 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 connection that output travels on. Each neuron's output is just the sum of all of these weighted inputs. Figure 18.5 recaps this idea graphically.

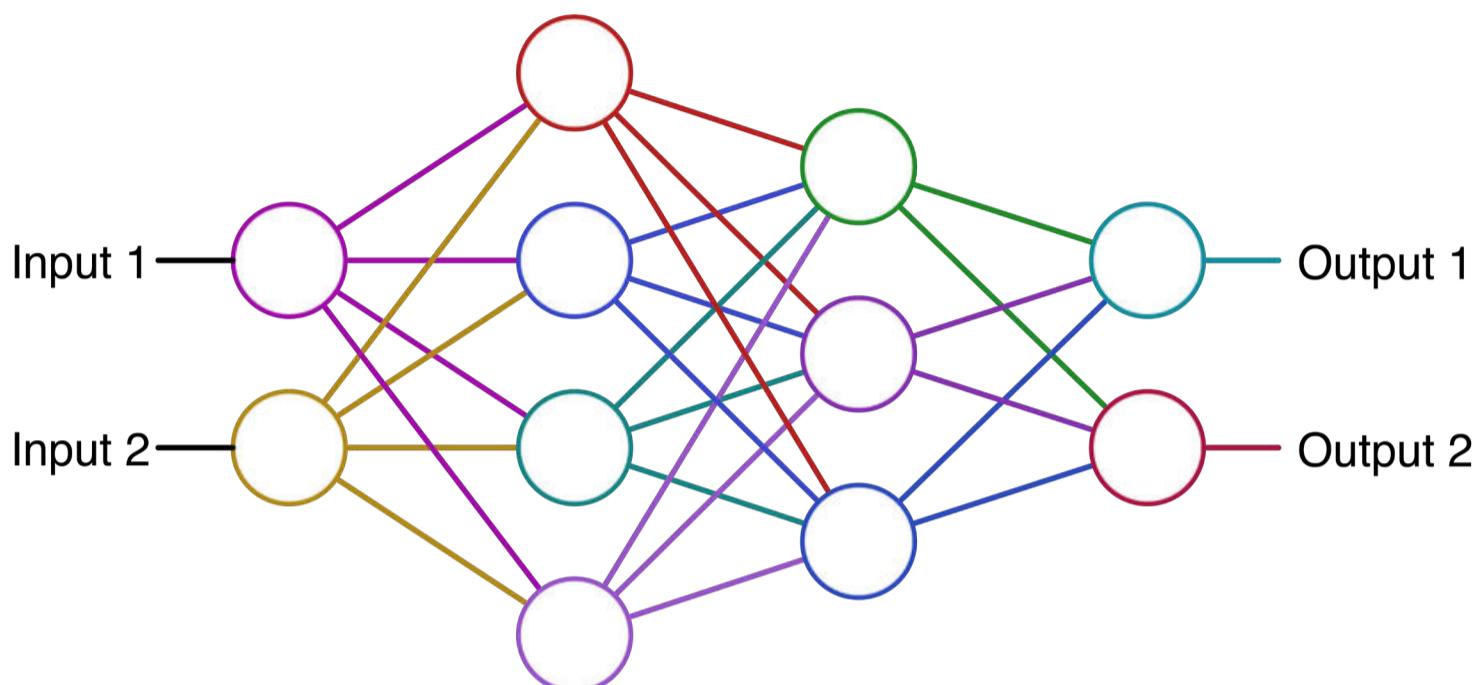


Figure 18.5: A small neural network with 11 neurons organized in 4 layers. Data flows from the inputs at the left to the outputs at the right. Each neuron's inputs come from the outputs of the neurons on the previous layer. This type of diagram, though common, easily becomes dense and confusing, even with color-coding. We will avoid it when possible.

We know that we'll be changing weights to improve our network. But sometimes it's easier to think about looking at the change in a neuron's output. As long as we keep using the same input, the only reason a neuron's output can change is because one of its weights has changed. So in the rest of this chapter, any time we speak of the result of a change in a neuron's output, that came about because we changed one of the weights that neuron depended on.

Let's take this point of view now, and imagine we're looking at a neuron whose output has just changed. What happens to the network's error as a result? Because the only operations that are being carried out in our network are multiplication and addition, if we work through the numbers we'll see that the result of this change is that the change in the error is **proportional** to the change in the neuron's output.

In other words, to find the change in the error, we find the change in the neuron’s output and multiply that by some particular value. If we double the amount of change in the neuron’s output, we’ll double the amount of change in the error. If we cut the neuron’s output change by one-third, we’ll cut the change in the output by one-third.

The connection between any *change* in the neuron’s output and the resulting *change* in the final error is just the neuron’s change times some number. This number goes by various names, but the most popular is probably the lower-case Greek letter δ (delta), though sometimes the upper-case 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.

So every neuron has a “delta,” or δ , associated with it. This is a real number that can be big or small, positive or negative. If the neuron’s output changes by a particular amount (that is, it goes up or down), we multiply that change by that neuron’s delta, and that tells us how the entire network’s output will change.

Let’s draw a couple of pictures to show the “before” and “after” conditions of a neuron whose output changes. We’ll change the output of the neuron using brute force: we’ll add some arbitrary number to the summed inputs just before that value emerges as the neuron’s output. As in Figure 18.2, we’ll use the letter m (for “modification”) for this extra value.

Figure 18.6 shows the idea graphically.

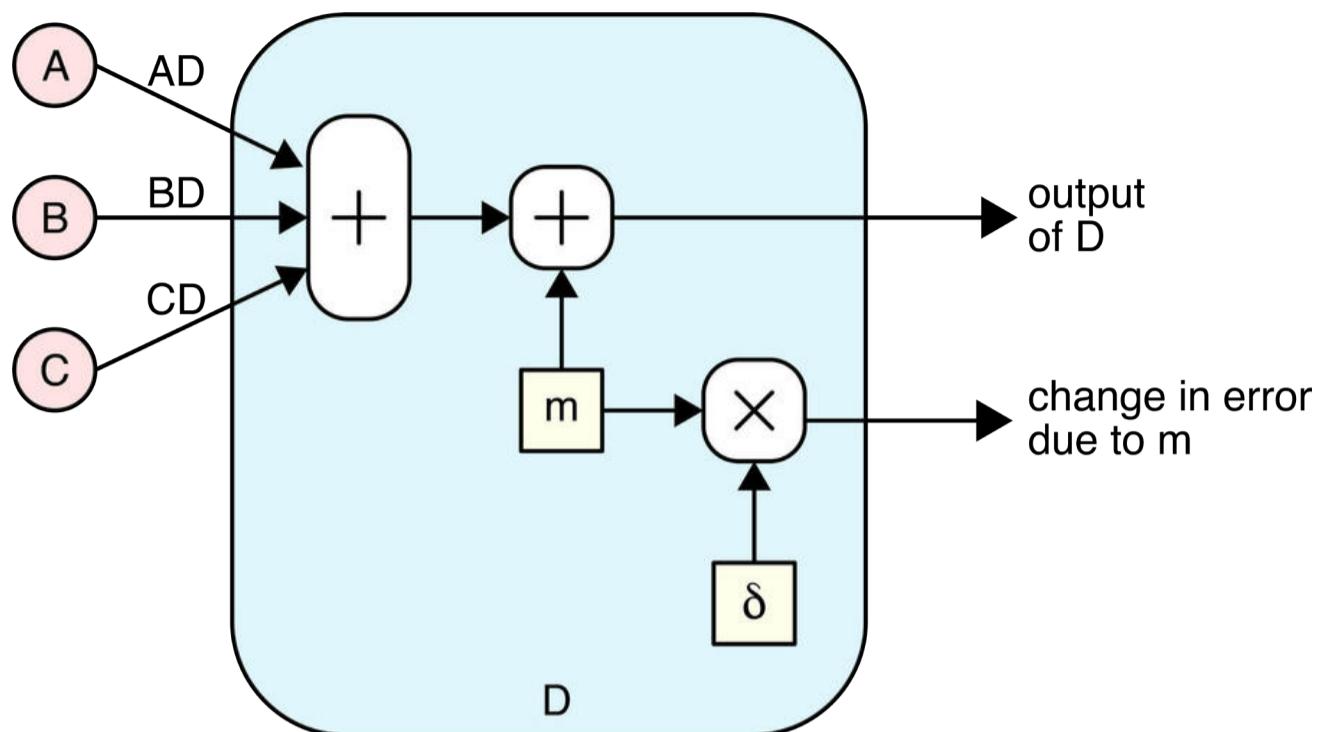


Figure 18.6: Computing the change in the error due to a change in a neuron's output. Here we're forcing a change in the neuron's output by adding an arbitrary amount m to the sum of the inputs. Because the output will change by m , we know the change in the error is this difference m times the value of δ belonging to this neuron.

In Figure 18.6 we placed the value m inside the neuron. But we can also change the output by changing one of the inputs. Let's change the value that's coming in from neuron B. We know that the output of B will get multiplied by the weight BD before it's used by neuron D. So let's add our value m right after that weight has been applied. This will have the same result as before, since we're just adding m to the overall sum that emerges from D. Figure 18.7 shows the idea. We can find the change in the output like before, multiplying this change m in the output by δ .

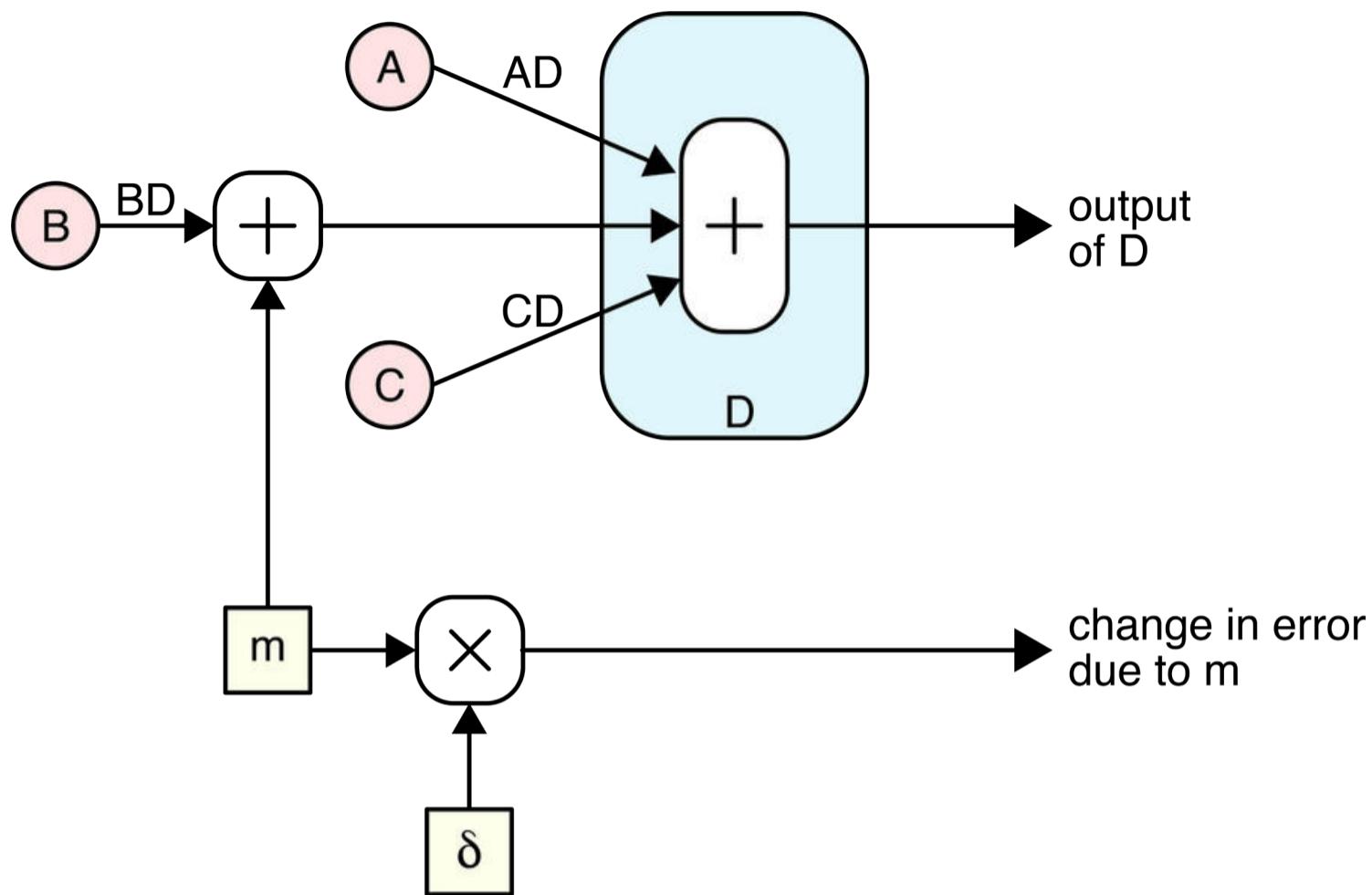


Figure 18.7: A variation of Figure 18.6, where we add m to the output of B (after it has been multiplied by the weight BD). The output of D is again changed by m , and the change in the error is again m times this neuron's value of δ .

To recap, if we know the change in a neuron's output, and we know the value of delta for that neuron, then we can predict the change in the error by multiplying that change in the output by that neuron's delta.

This is a remarkable observation, because it shows us explicitly how the error changes based on the change in output of each neuron. The value of delta acts like an amplifier, making any change in the neuron's output have a bigger or smaller effect on the network's error.

An interesting result of multiplying the neuron's change in output with its delta is that if the change in the output and the value of delta both have the same **sign** (that is, both are positive or negative), then the change in the error will be positive, meaning that the error will increase. If the change in the output and delta have opposite signs (that is, one

is negative and one is positive), then the change in the error will be negative, meaning that the error will decrease. That's the case we want, since our goal is always to make the error as small as possible.

For instance, suppose that neuron A has a delta of 2, and for some reason its output changes by -2 (say, changing from 5 to 3). Since the delta is positive and the change in output is negative, the change in the error will also be negative. In numbers, $2 \times -2 = -4$, so the error will drop by 4.

On the other hand, suppose the delta of A is -2 , and its output changes by $+2$ (say from 3 to 5). Again, the signs are different, so the error will change by $-2 \times 2 = -4$, and again the error will reduce by 4.

But if the change in A's output is -2 , and the delta is also -2 , then the signs are the same. Since $-2 \times -2 = 4$, the error will increase by 4.

At the start of this section we said there were two key observations we wanted to note. The first, as we've been discussing, is that if a neuron's output changes, the error changes by a proportional amount.

The second key observation is: *this whole discussion applies just as well to the weights*. After all, the weights and the outputs are multiplied together. When we multiply two arbitrary numbers, such as a and b , then we make the result bigger by adding something to *either* the value of a or b . In terms of our network, we can say that *when any weight in our network changes, the error changes by a proportional amount*.

If we wanted, we could work out a delta for every weight. And that would be perfect. We would know just how to tweak each weight to make the error go down. We just add in a small number whose sign is opposite that of the weight's delta.

Finding those deltas is what backprop is for. We find them by first finding the delta for every neuron's output. We'll see below that with a neuron's delta, and its output, we can find the weight deltas.

We already know every neuron's outputs, so let's turn our attention to finding those neuron deltas.

The beauty of backpropagation is that finding those values is incredibly efficient.

18.5 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 could do this job with just one perceptron, but we'll use a little network because it lets us see the general principles.

In this section we'll look at the network and give a label to everything we care about. That will make later discussions simpler and easier to follow.

Figure 18.8 shows our network. The inputs are the X and Y coordinates of each point, there are four neurons, and the outputs of the last two serve as the outputs of the network. We call their outputs the predictions P_1 and P_2 . The value of P_1 is the network's prediction of the likelihood that our sample (that is, the X and Y at the input) belongs to class 1, and P_2 is its prediction of the likelihood that the sample belongs to class 2. These aren't actually probabilities because they won't necessarily add up to 1, but whichever is larger is the network's preferred choice of category for this input. We could make them into probabilities (by adding a **softmax** layer, as discussed in Chapter 17), but that would just make the discussion more complicated without adding anything useful.

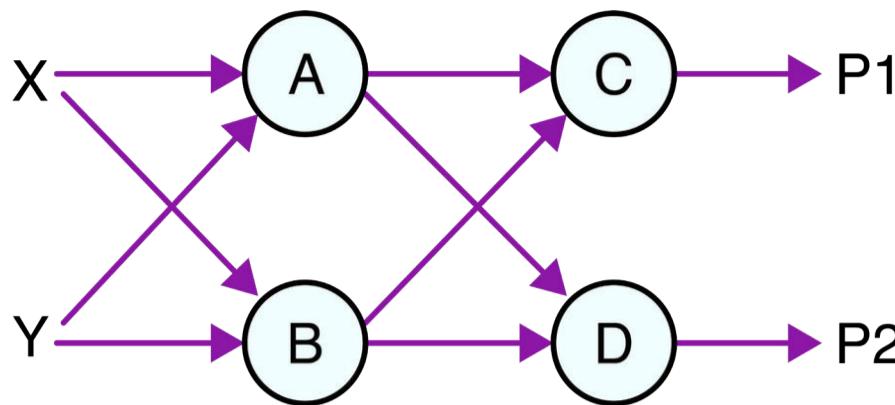


Figure 18.8: A simple network. The input has two features, which we call X and Y. There are four neurons, ending with two predictions, P1 and P2. These predict the likelihoods (not the probabilities) that our sample belongs to class 1 or class 2, respectively.

Let's label the weights. As usual, we'll imagine that the weights are sitting on the wires that connect neurons, rather than stored inside the neurons. The name of each weight will be the name of the neuron providing that value at its output followed by the neuron using that value as input. Figure 18.9 shows the names of all 8 weights in our network.

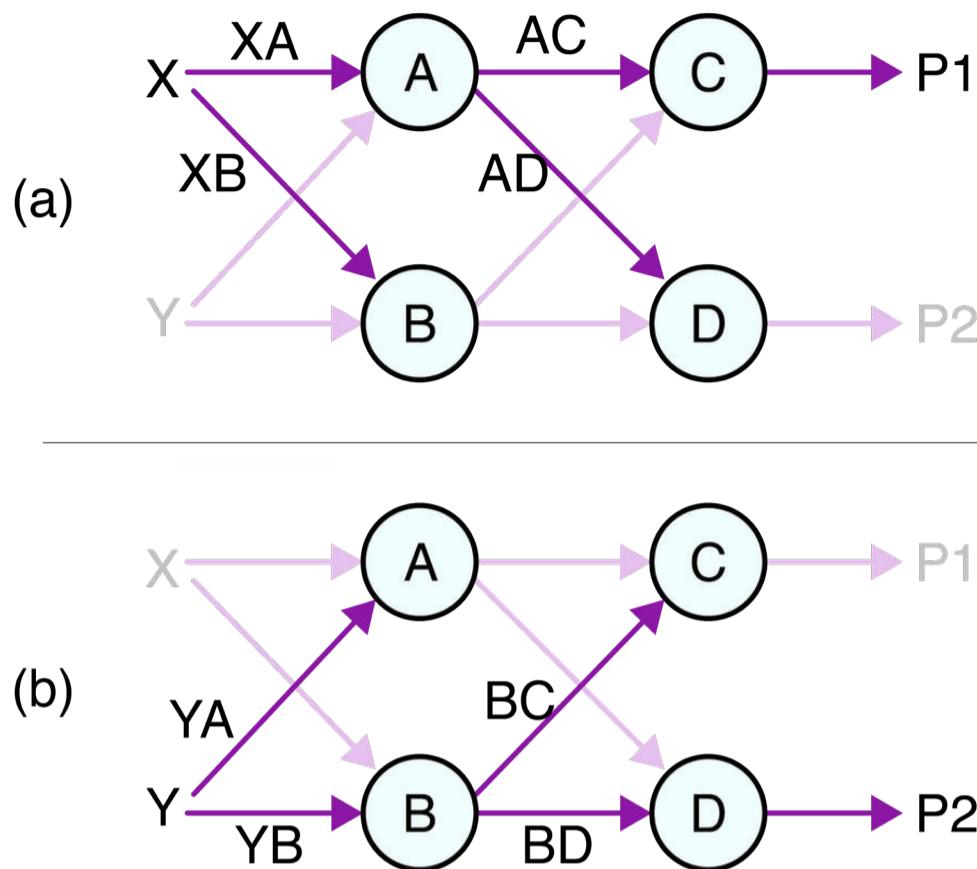


Figure 18.9: Giving names to each of the 8 weights in our tiny network. Each weight is just the name of the two neurons it connects, with the starting neuron (on the left) first, and then the destination neuron (on the right) second. For the sake of consistency, we pretend that X and Y are “neurons” when it comes to naming the weights, so XA is the name of the weight that scales the value of X going into neuron A.

This is a tiny **deep-learning network** with two **layers**. The first layer contains neurons A and B, and the second contains neurons C and D, as shown in Figure 18.10.

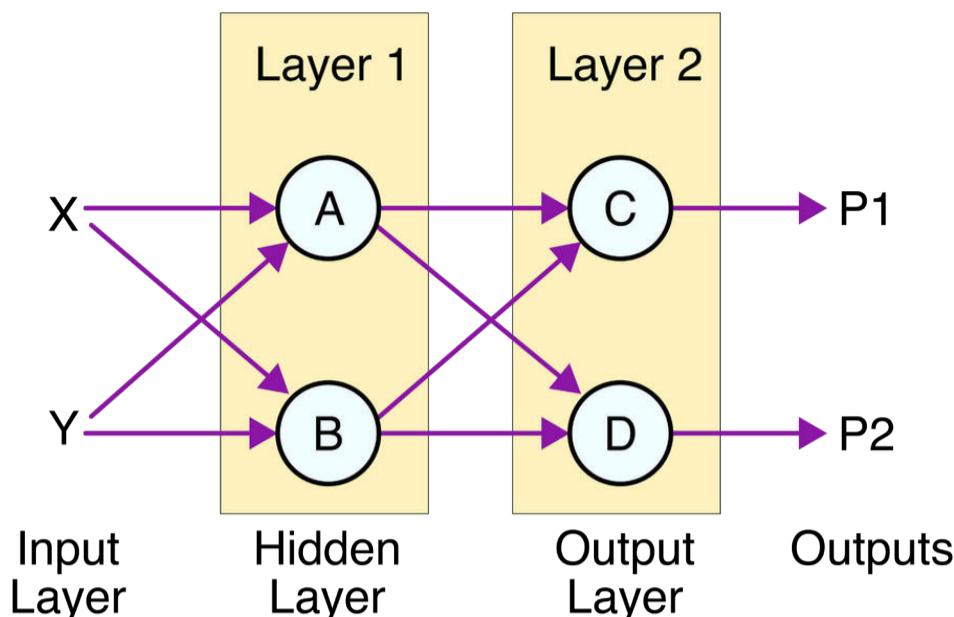


Figure 18.10: Our tiny neural network has 2 layers. The input layer doesn't do any computing, so it's usually not included in the layer count.

Two layers is not a terribly deep network, and two neurons per layers is not a lot of computing power. We usually work with systems with more layers, and more neurons on each layer. Determining how many layers we should use for a given task, and how many neurons should be on each layer, is something of an art and an experimental science. In essence, we usually take a guess at those values, and then vary our choices to try to improve the results.

In Chapter 20 we'll discuss deep learning and its terminology. Let's jump ahead a little bit here and use that language for the various pieces of the network in Figure 18.10. The **input layer** is just a conceptual grouping of the inputs X and Y. These don't correspond to neurons, because these are just pieces of memory for storing the features in the sample that's been given to the network. When we count up the layers in a network, we don't usually count the input layer.

The **hidden layer** is called that because neurons A and B are “inside” the network, and thus “hidden” from a viewer on the outside, who can see only the inputs and outputs. The **output layer** is the set of neurons that provide our **outputs**, here P1 and P2. These layer names are a little asymmetrical because the input layer has no neurons, and the output layer does, but they're how the convention has developed.

Finally, we'll want to refer to the output and delta for every neuron. For this, we'll make little two-letter names by combining the neuron's name with the value we want to refer to. So Ao and Bo will be the names of the outputs of neurons A and B, and $A\delta$ and $B\delta$ will be the delta values for those two neurons.

Figure 18.11 shows these values stored with their neurons.

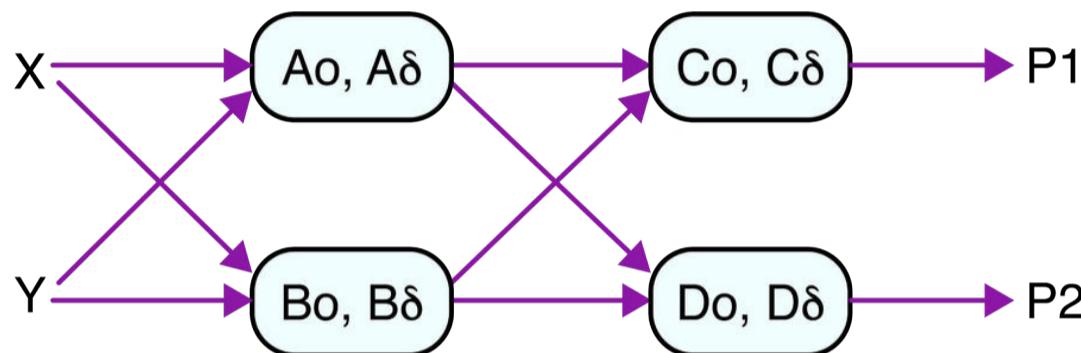


Figure 18.11: Our simple network with the output and delta values for each neuron.

We'll be watching what happens when neuron outputs change, causing changes to the error. We'll label the change in the output of neuron A as Am . We'll label the error simply E , and a change to the error as Em .

As we saw above, 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'll 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 18.12 shows the schematic setup we'll use for visualizing the way changes in a neuron's output are scaled by its delta to produce changes to the error.

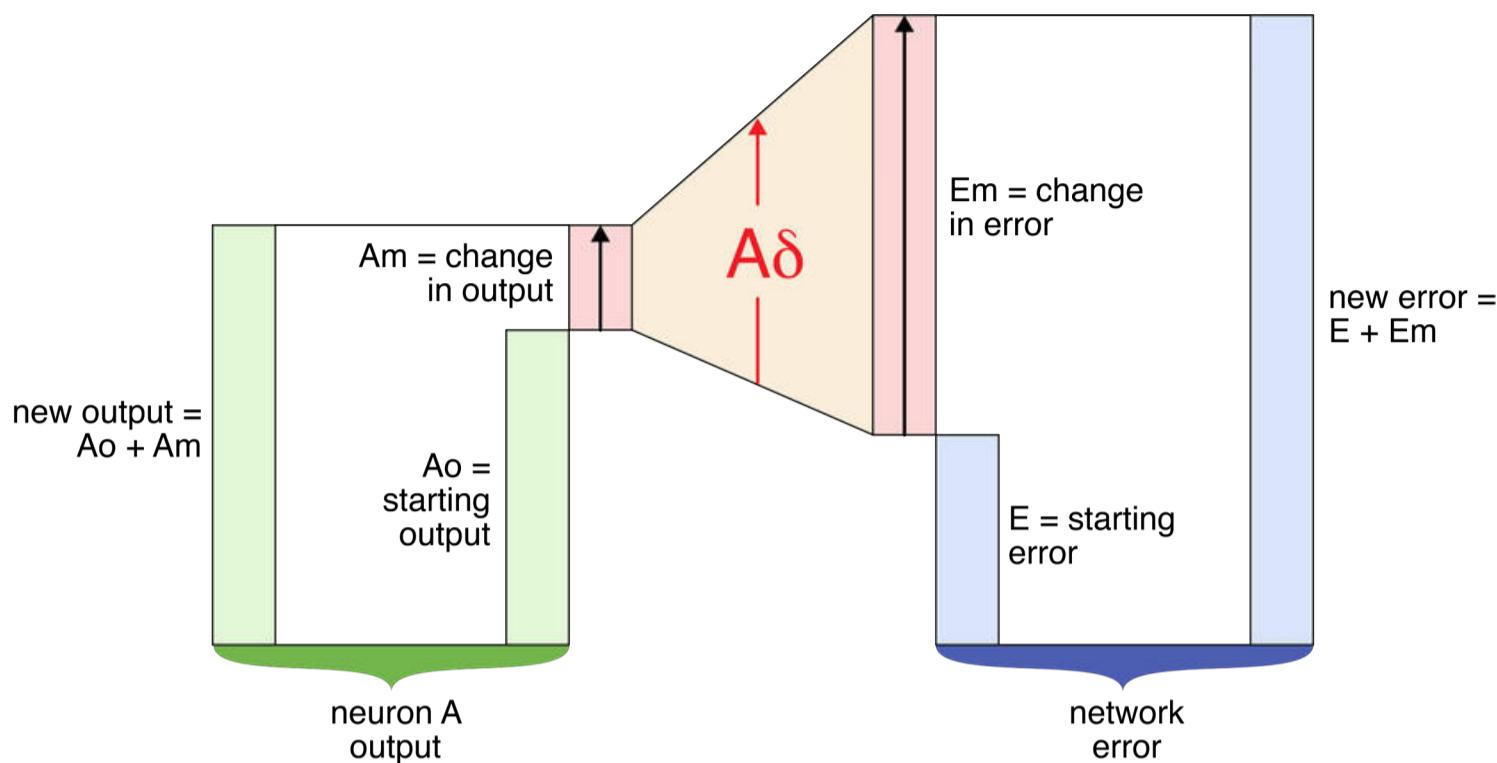


Figure 18.12: Our schematic for visualizing how changes in a neuron's output can change the network's error. Read the diagram roughly left to right.

At the left of Figure 18.12 we start with a neuron A. It starts with value Ao , but we change one of the weights on its inputs so that the output goes up by Am . The arrow inside the box for Am shows that this change is positive. This change is multiplied by $A\delta$ to give us Em , the change in the error. We show $A\delta$ as a wedge, illustrating the amplification of Em . Adding this change to the previous value of the error, E , gives us the new error $E+Em$. In this case, both Am and $A\delta$ are positive, so the change in the error $Am \times A\delta$ is also positive, increasing the error.

Keep in mind that the delta value $A\delta$ relates a *change* in a neuron's output to a *change* in the error. These are not relative or percentage changes, but the actual amounts. So if the output of A goes from 3 to 5, that's a change of 2, so the change in the error would be $A\delta \times 2$. If the output of A goes from 3000 to 3002, that's still a change of 2, and the error would change by the same amount, $A\delta \times 2$.

Now that we've labeled everything, we're finally ready to look at the backpropagation algorithm.

18.6 Step 1: Deltas for the Output Neurons

Backpropagation is all about finding the delta value for each neuron. To do that, we'll find gradients of the error at the end of the network, and then propagate, or move, those gradients back to the start. So we'll begin at the end: the output layer.

The outputs of neuron C and D in our tiny network give us the likelihoods that the input is in class 1 or class 2, respectively. In a perfect world, a sample that belongs to group 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=0.1, telling us that it's much more likely that the sample is in class 1 (remember that these aren't probabilities, so they probably won't sum to 1).

We'd like to come up with a single number to represent the network's error. To do that, we'll 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 12. Recall that one-hot encoding makes a list of numbers as long as the number of classes, and puts a 0 in every entry. Then it puts a 1 in the entry corresponding to the correct class. In our case, we have only two classes, so the encoder would always start with list of two zeros, which we can write as (0, 0). For a sample that belongs to class 1, it would put a 1 in the first slot, giving us (1, 0). A sample from class 2 would get the label (0, 1). Sometimes such a label is also called a **target**.

Let's put the predictions P_1 and P_2 into a list as well: (P_1, P_2) . Now we can just compare the lists. There are lots of ways to do this. For example, a simple way would be to find the difference between corresponding elements from each list and then add up those differences. Figure 18.13 shows the idea.

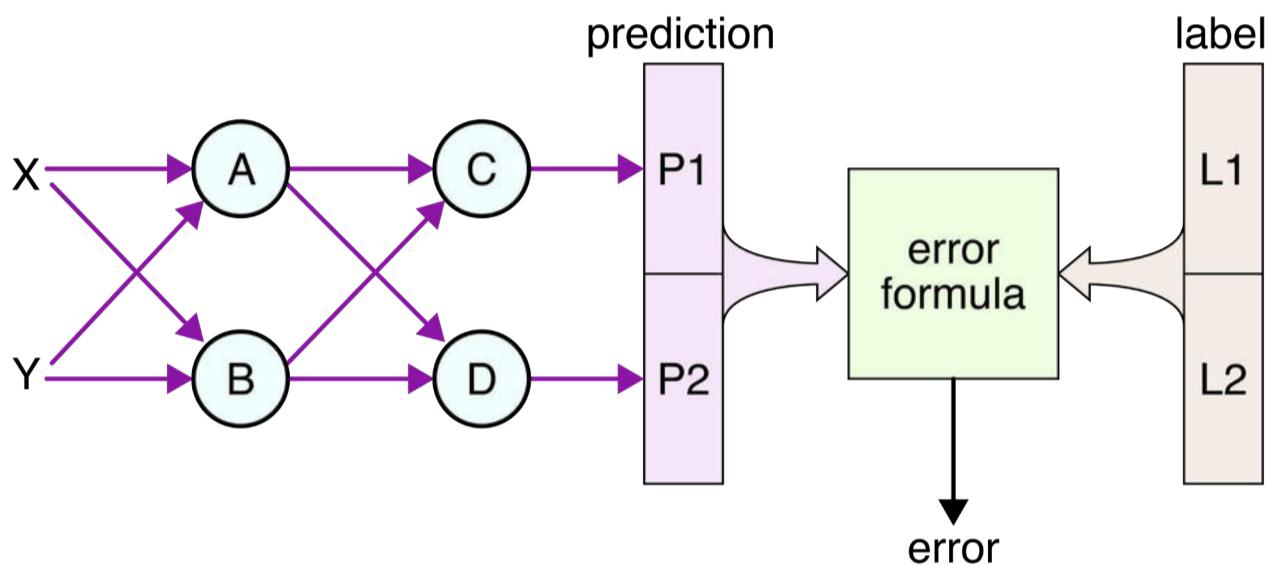


Figure 18.13: To find the error from a specific sample, we start by supplying the sample's features X and Y to the network. The outputs are the predictions P_1 and P_2 , telling us the likelihoods that the sample is in class 1 and class 2, respectively. We compare those predictions with the one-hot encoded label, and from that come up with a number representing the error. If the predictions match the label perfectly, the error is 0. The bigger the mismatch, the bigger the error.

If the prediction list is identical to the label list, then the error is 0. If the two lists are close (say, $(0.9, 0.1)$ and $(1, 0)$), then we'd want to come up with an error number that's bigger than 0, but maybe not enormous. As the lists become more and more different, the error should increase. The maximum error would come if the network is absolutely wrong, for example predicting $(1, 0)$ when the label says $(0, 1)$.

There are many formulas for calculating the network error, and most libraries let us choose among them. We'll see in Chapters 23 and 24 that this error formula is one of the critical choices that defines what our network is for. For instance, we'll choose one type of error formula if we're building a network to classify inputs into categories, another type of formula if our network is for predicting the next value in a

sequence, and yet another type of formula if we're trying to match the output of some other network. These formulas can be mathematically complex, so we won't go into those details here.

As Figure 18.13 shows, that formula for our simple network takes in 4 numbers (2 from the prediction and 2 from the label), and produces a single number as a result.

But all the error formulas share the property that when they compare a classifier's output to the label, a perfect match will give a value of 0, and increasingly incorrect matches will give increasingly large errors.

For each type of error formula, our library function will also provide us with its **gradient**. The gradient tells us how the error will change if we increase any one of the four inputs. This may seem redundant, since we know that we want the outputs to match the label, so we can tell how the outputs should change just by looking at them. But recall that the error can include other terms, like the regularization term we discussed above, so things can get more complicated.

In our simple case, we can use the gradient to tell us whether we'd like the output of C to go up or down, and the same for D. We'll pick the direction for each neuron that causes the error to decrease.

Let's think about drawing our error. We could also draw the gradient, but usually that's harder to interpret. When we draw the error itself, we can usually see the gradient just by looking at the slope of the error.

Unfortunately, we can't draw a nice picture of the error for our little network because it would require five dimensions (four for the inputs and one for the output). But things aren't so bad. We don't care about how the error changes when the label changes, because the label can't change. For a given input, the label is fixed. So we can ignore the two dimensions for the labels. That leaves us with just 3 dimensions.

And we can draw a 3D shape! So let's plot the error. Remember that we can visualize the gradient at any point just by imagining which way a drop of water would flow if we placed it on the surface of the error above that location.

Let's draw a 3D diagram that shows the error for any set of values P_1 and P_2 , *for a given label*. That is, we'll set the value of the label, and explore the error for different values of P_1 and P_2 . Every error formula will give us a somewhat different surface, but most will look roughly like Figure 18.14.

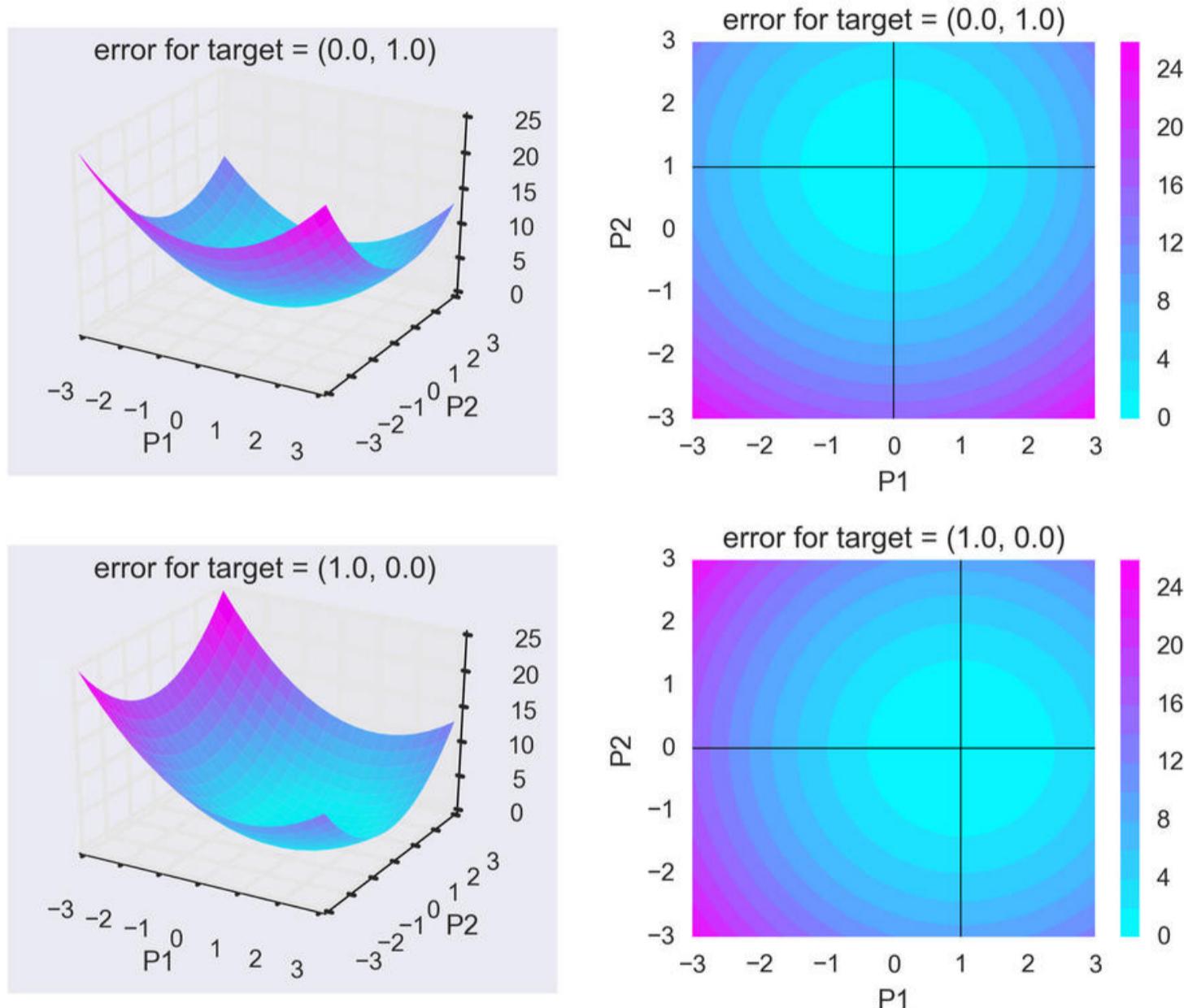


Figure 18.14: Visualizing the error of our network, given a label (or target) and our two predictions, P_1 and P_2 . Top row: The label is $(0, 1)$, so the error is a bowl with its bottom at $P_1 = 0$ and $P_2 = 1$. As P_1 and P_2 diverge from those values, the error goes up. Left: The error for each value of P_1 and P_2 . Right: A top-down view of the surface at the left, showing the height using colored contours. Bottom row: The label is $(1, 0)$, so now the error is a bowl with the bottom at $P_1 = 1$ and $P_2 = 0$.

For both labels, the shape of the error surface is the same: a bowl with a rounded bottom. The only difference is the location of the bottom of the bowl, which is directly over the label. This makes sense, because our whole intention is to get P_1 and P_2 to match the label. When they do, we have zero error. So the bottom of the bowl has a value of 0, and it sits right on top of the label. The more different P_1 and P_2 are from the label, the more the error grows.

These plots let us make a connection between the gradient of this error surface and the delta values for the output layer neurons C and D. It will help to remember that P_1 , the likelihood of the sample belonging to class 1, is just another name for the output of C, which we also call C_o . Similarly, P_2 is another name for D_o . So if we say that we want to see a specific change in the value of P_1 , we're saying that we want the output of neuron C to change in that way, and the same is true for P_2 and D.

Let's look at one of these error surfaces a little more carefully so we can really get a feeling for it. Suppose we have a label of $(1,0)$, like in the bottom row of Figure 18.14. Let's suppose that for a particular sample, output P_1 has the value -1 , and output P_2 has the value 0 . In this example, P_2 matches the label, but we want P_1 to change from -1 to 1 .

Since we want to change P_1 while leaving P_2 alone, let's look at the part of the graph that tells us how the error will change by doing just that. We'll set $P_2=0$ and look at the cross-section of the bowl for different values of P_1 . We can see it follows the overall bowl shape, as in Figure 18.15.

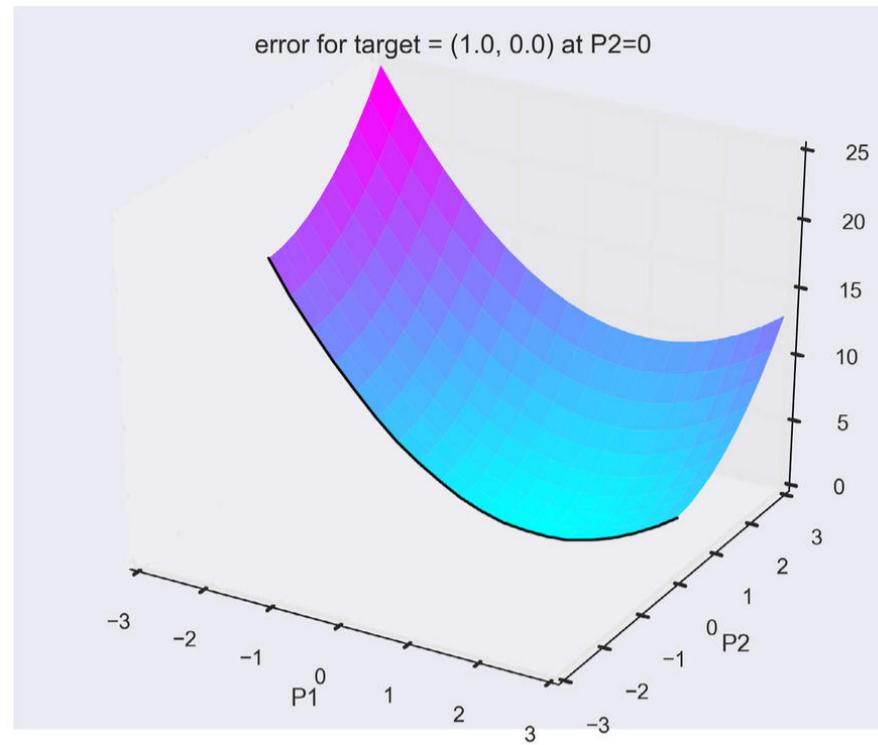


Figure 18.15: Slicing away the error surface from the bottom left of Figure 18.14, where the label is $(1,0)$. The revealed cross-section of the surface shows us the values of the error for different values of $P1$ when $P2 = 0$.

Let's look just at this slice of the error surface, shown in Figure 18.16.

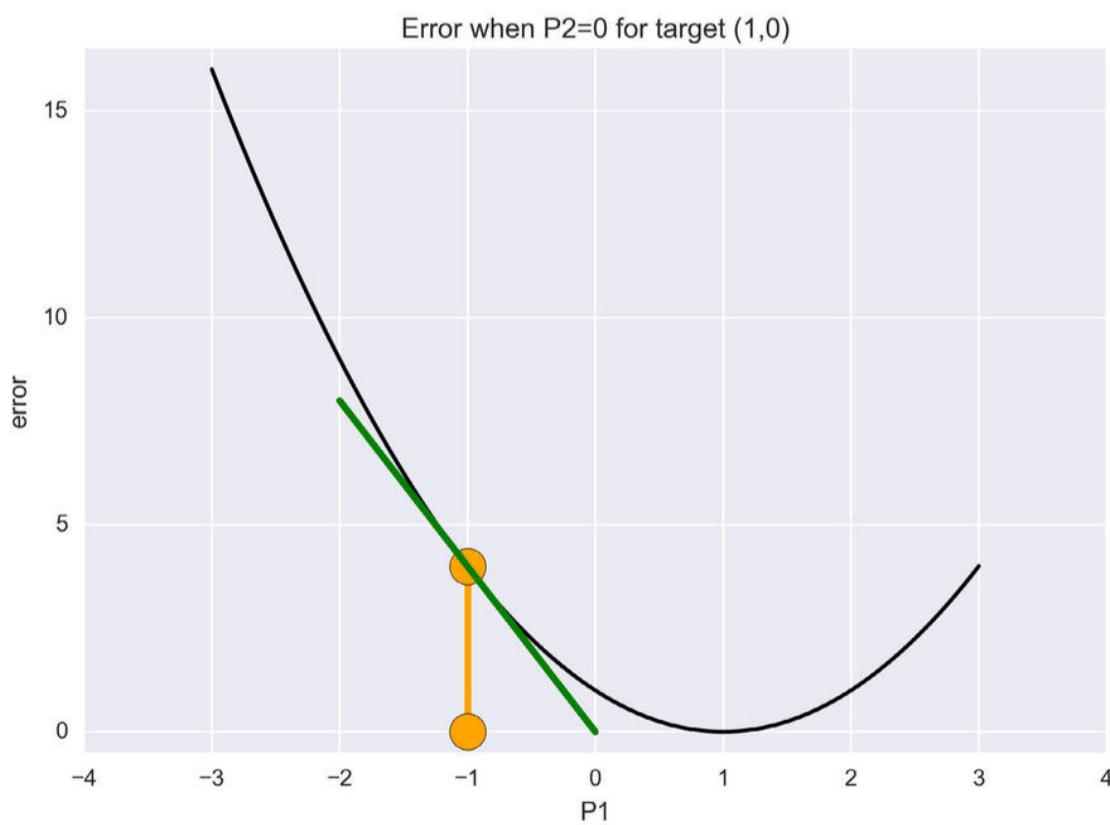


Figure 18.16: Looking at the cross-section of the error function shown in Figure 18.15, we can see how the error depends on different values of $P1$, when $P2$ is fixed at 0.

In Figure 18.16 we've marked the value $P_1 = -1$ with an orange dot, and we've drawn the derivative at the location on the curve directly above this value of P_1 . This tells us that if we make P_1 more positive (that is, we move right from -1), the error in the network will decrease. But if we go too far and increase P_1 beyond the value of 1 , the error will start to increase again. The derivative is just the piece of the gradient that applies to only P_1 , and tells us how the error changes as the value of P_1 changes, *for these values of P_2 and the label*. As we can see from the figure, if we get too far away from -1 the derivative no longer matches the curve, but close to -1 it does a good job.

We'll come back to this idea again later: the derivative of a curve tells us what happens to the error if we move P_1 *by a very small amount* from a given location. The smaller the move, the more accurate the derivative will be at predicting our new error. This is true for any derivative, or any gradient.

We can see this characteristic in Figure 18.16. If we move P_1 by 1 unit to the right from -1 , the derivative (in green) would land us at an error of 0, though it looks like the error for $P_1=0$ (the value of the black curve) is really about 1. We *can* use the derivative to predict the results for large changes in P_1 , but our accuracy will go down the farther we move, as we just saw. In the interests of clear figures that are easy to read, we'll sometimes make large moves when the difference between where the derivative would land us, and where the real error curve tells us we should be, are close enough.

Let's use the derivative to predict the change in the error due to a change in P_1 . What's the slope of the green line in Figure 18.16? The left end is at about $(-2, 8)$, and the right end is at about $(0, 0)$. Thus the line descends about 4 units for every 1 unit we move to the right, for a slope of $-4/1$ or -4 . So if P_1 changed by 0.5 (that is, it changed from -1 to -0.5), we'd predict that the error would go down by $0.5 \times -4 = -2$.

That's it! That's the key observation that tells us the value of $C\delta$.

Remember that P_1 is just another name for C_o , the output of C. We've found that a change of 1 in C_o results 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 P_1 . But for small moves, the proportion is right. For instance, if we were to increase P_1 by 0.01, then we'd expect the error to change by $-4 \times 0.01 = -0.04$, and for such a small change in P_1 the predicted change in the error should be pretty accurate. If we increased P_1 by 0.02, then we'd expect the error to change by $-4 \times 0.02 = -0.08$. If we move P_1 to the left, so it changed from -1 to, say, -1.1 we'd 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 C_o , we can predict the change in the error by multiplying C_o by -4 .

That's exactly what we've been looking for! The value of $C\delta$ is -4 . Note that this only holds for this label, and these values of C_o and D_o (or P_1 and P_2).

We've just found our first delta value, telling 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 P_1 (or C_o).

Figure 18.17 shows what we've just described using our error diagram.

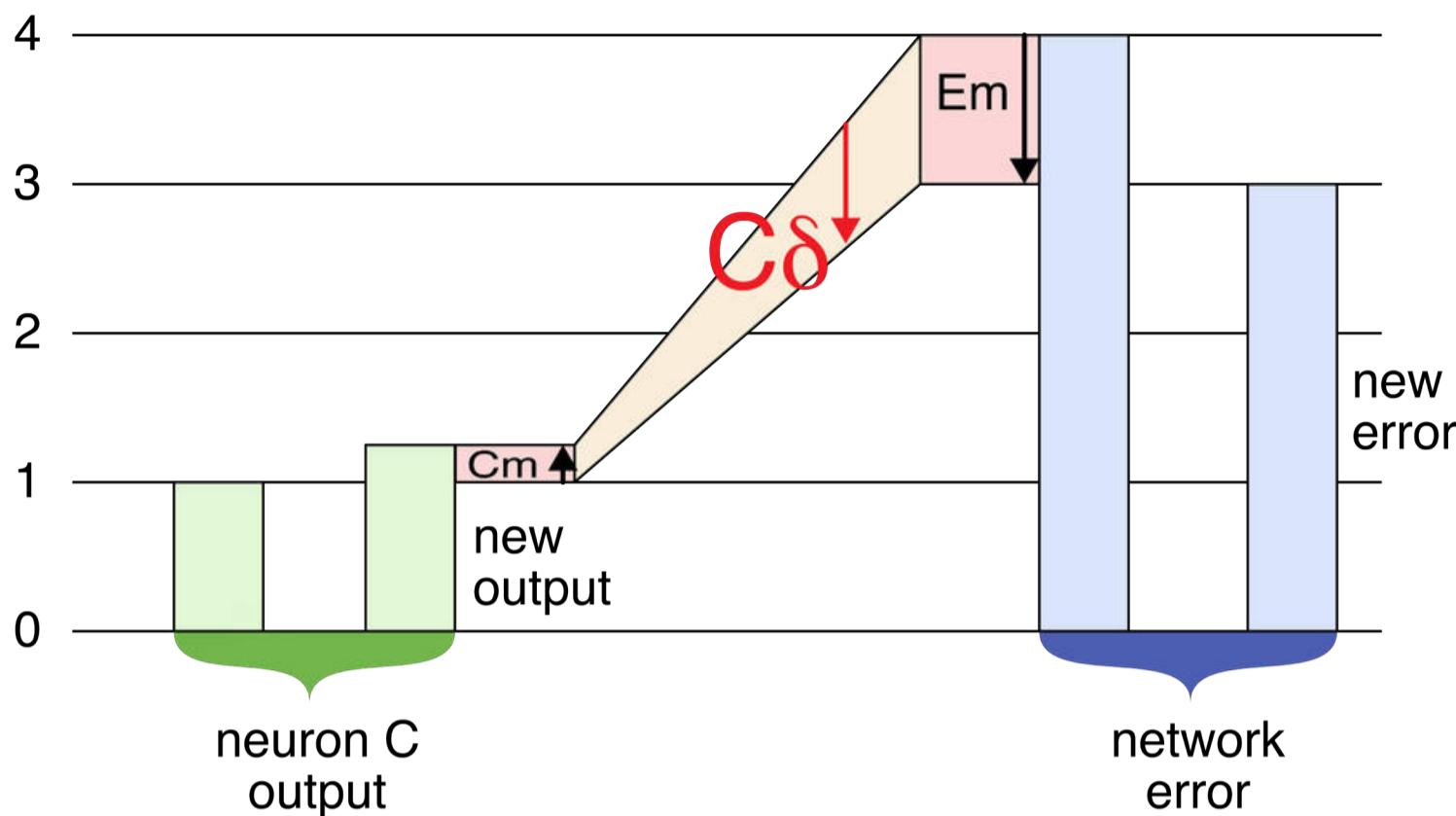


Figure 18.17: Our error diagram illustrating the change in the error from a change in the output of neuron C. The original output is the green bar at the far left. We imagine that due to a change in the inputs, the output of C increases by an amount C_m . This is amplified by multiplying it with $C\delta$, giving us the change in the error, E_m . That is, $E_m = C_m \times C\delta$. Here the value of C_m is about 1/4 (the upward arrow in the box for C_m 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 $E_m = -4 \times 1/4 = -1$. The new error, at the far right, is the previous error plus E_m .

Remember that at this point we're not going to do anything with this delta value. Our goal right now is just to find the deltas for our neurons. We'll use them later.

We assumed above that P_2 already had the right value, and we only needed to adjust P_1 . But what if they were both different than their corresponding label values?

Then we'd repeat this whole process for P_2 , to get the value of $D\delta$, or delta for neuron D. Let's do just that.

In Figure 18.18 we can see the slices of the error for an input with a corresponding label, or target, of $(1,0)$ when $P_1 = -0.5$, and $P_2 = 1.5$.

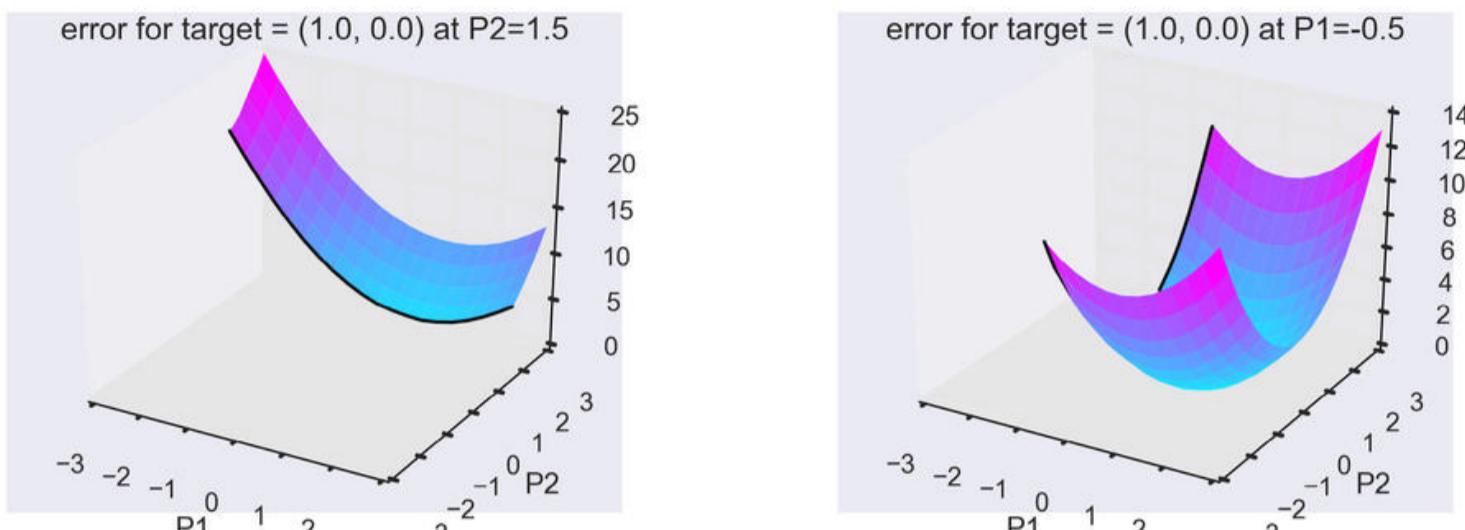


Figure 18.18: Slicing our error function when both P_1 and P_2 are different from the label. Left: The error due to different values of P_1 when $P_2=1.5$. Right: The error due to different values of P_2 when $P_1 = -0.5$.

The cross-section curves are shown in Figure 18.19. Notice that the curve for P_1 has changed from Figure 18.16. That's because the change in P_2 means we're taking the P_1 cross-section at $P_2=1.5$ instead of $P_2=0$.

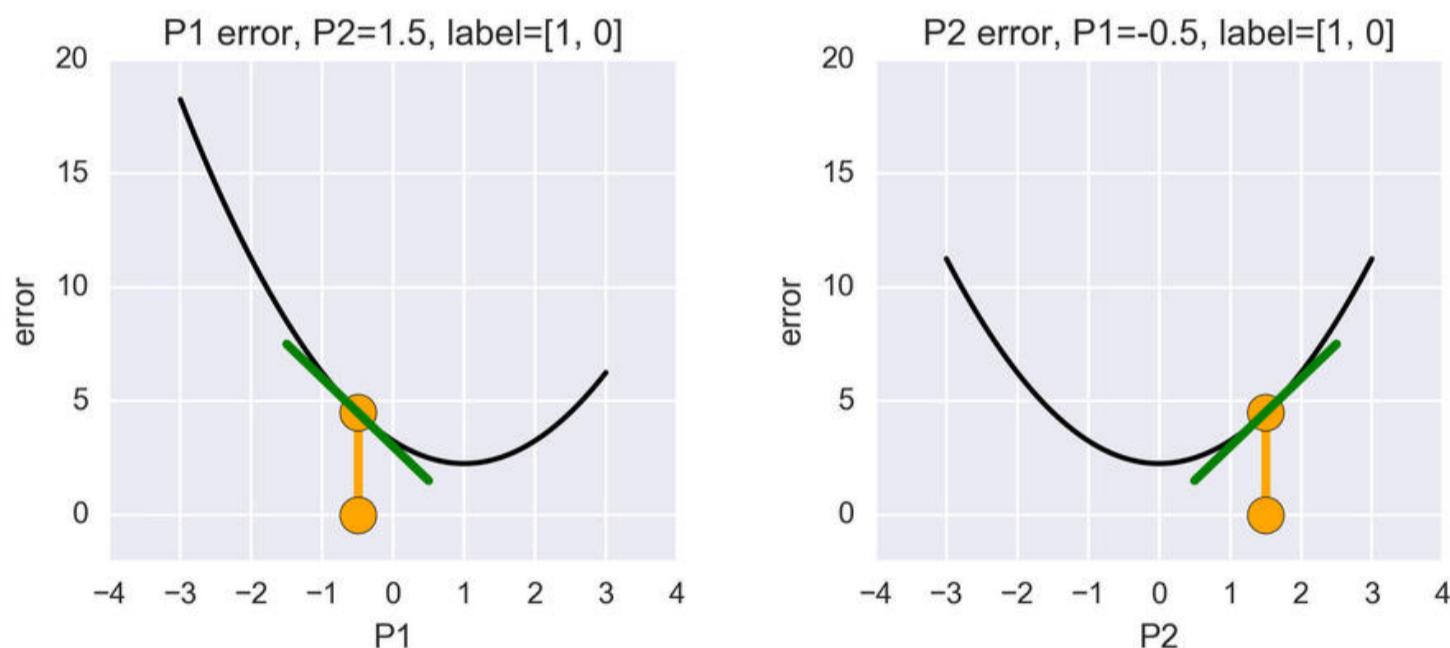


Figure 18.19: When neither P_1 or P_2 match the label, we can try to reduce the error by adjusting each one individually. In this example, the label is $(1, 0)$. Left: The error for different values of P_1 when P_2 is 1.5. The smallest value is a little more than 2. The derivative tells us that we can get to that lower value by making P_1 larger than its current value of -0.5 (that is, moving to the right). Right: The error for different values of P_2 when $P_1 = -0.5$. The current value of P_2 is 1.5. The derivative tells us that we can make the error smaller by making P_2 smaller (that is, moving to the left).

For the specific values in this figure, it looks like a change of about 0.5 in P_1 would result in a change of about -1.5 in the error, so $C\delta$ is about $-1.5/0.5 = -3$. Instead of changing P_1 , what if we changed P_2 ? Looking at the graph on the right, a change of about -0.5 (moving left this time, towards the minimum of the bowl) would result in a change of about -1.25 in the error, so $D\delta$ is about $1.25/-0.5 = 2.5$. The positive sign here tells us that moving P_2 to the right will cause the error to go up, so we want to move P_2 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 their respective label values. Second, because the current values of P_1 and P_2 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 the minimum available error is not 0. In particular, the curves never get lower than a bit more than 2. That's because each curve looks at changing just one of the two values, while the other is left fixed. So even if P_1 got to its ideal value of 1, there would still be error in the result because P_2 is not at its ideal value of 0, and vice-versa.

This means that if we change just one of these two values, we'll never get down to the minimum error of 0. To get the error down to 0, both P_1 and P_2 have to work their way down to the bottom of their respective curves.

Here's the wrinkle: each time either P_1 or P_2 changes, we pick out a different cross-section of the error surface. That means we get different curves for the error, just as Figure 18.16, when $P_2=0$, is different from Figure 18.19 when $P_2 = 1.5$. Because the error curve has changed, the deltas change as well.

So if we change either value, we have to restart this whole process again from scratch before we know how to change the other value.

Well, not exactly. We'll see later that we actually *can* update both values at the same time, as long as we take very small steps. But that's about as far as we can cheat before we risk driving the error up again. Once we've taken our small steps, we have to evaluate the error surface again to find new curves and then new derivatives before we can adjust P1 and P2 again.

We've just described the general way to compute the delta values for the output neurons (we'll look at hidden neurons in a moment). In practice, we often use a particular measure of error that makes things easier, because we can write down a super simple formula for the derivative, which in turns gives us an easy way to find the deltas. This error measure is called the **quadratic cost function**, or the **mean squared error** (or **MSE**) [Neilsen15a]. As usual, we won't get into the mathematics of this equation. What matters for us now is that this choice of function for the error means that the derivative at any neuron's value (that is, the neuron's delta value) is particularly easy to calculate. The delta for an output neuron is the difference between the neuron's value and the corresponding label entry [Seung05]. Figure 18.20 shows the idea graphically.

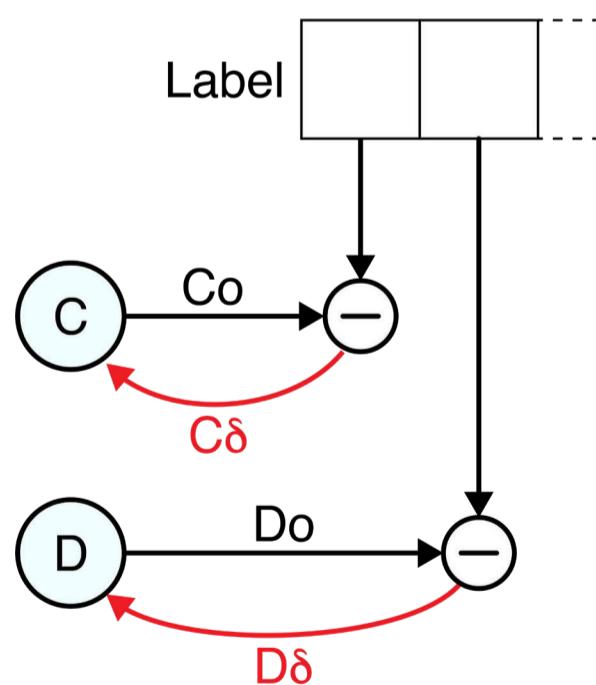


Figure 18.20: When we use the quadratic cost function, the delta for any output neuron is just the value in the label minus the output of that neuron. As shown in red, we save that delta value with its neuron.

This little calculation matches up nicely with our bowl-shaped pictures above. Remember that C_0 and P_1 are two names for the same value, as are D_0 and P_2 .

Let's consider C_0 (or P_1) when the first label is 1. If $C_0=1$, then $1-C_0=0$, so a tiny change to C_0 will have no effect on the output error. That makes sense, because we're at the bottom of the bowl where it's flat.

Suppose that $C_0=2$. Then the difference $1-C_0=-1$, telling us that a change to C_0 will change the error by the same amount, but with opposite sign (for example, a change of 0.2 in C_0 would result in a change of -0.2 to the error). If C_0 is much larger, say $C_0=5$, then $1-C_0=-4$, telling us that any change to C_0 will be amplified by a factor of -4 in the change to the error. That also makes sense, because we're now at a very steep part of the bowl where a small change in the input (C_0) will cause a big change in the output (the change in the error). We've been using large numbers for convenience, but remember that the derivative only tells us what happens if we take a very small step.

The same thought process holds for neuron D, and its output D_0 (or P_2).

We've now completed the first step in backpropagation: we've found the delta values for all the neurons in the output layer.

We've seen that the delta for an output neuron depends on the value in the label and the neuron's output. If the neuron's output changes, the delta will change as well.

So “the delta” is a temporary value that changes with every new label, and every new output of the neuron. Following this observation, any time we update the weights in our network, we'll need to calculate new deltas.

Remember that our goal is to find the deltas 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.

18.7 Step 2: Using Deltas to Change Weights

We've seen how to find a delta value for every neuron in the output layer. If the output of any of those neurons changes by some amount, we multiply that change by the neuron's delta to find the change to the output.

We know that a change to the neuron's output can come from a change in an input, which in turn can come from a change in a previous neuron's output or the weight connecting that output to this neuron. Let's look at these inputs.

We'll focus on output neuron C, and the value it receives from neuron A on the previous layer. Let's use the temporarily name V to refer to the value that arrives into C from A. It has a value given by the output of A, or Ao , and the weight between A and C, or AC , so the value V is $Ao \times AC$. This setup is shown in Figure 18.21.

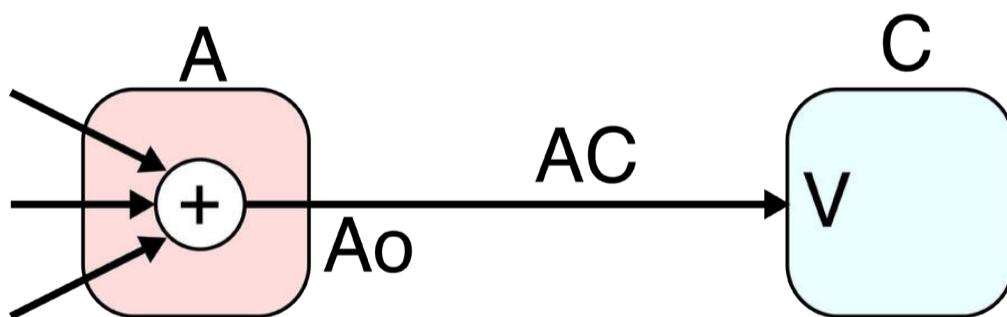


Figure 18.21: The value coming into neuron C, which we're calling V for the moment, is Ao (the output of A) times the weight AC , or $Ao \times AC$.

If V changes for any reason, we know that the output of C will change by V as well (since C is just adding up its inputs and passing on that sum). Since the change in C is V , the network's error will change by $V \times C\delta$, since we built $C\delta$ to do just that.

There are only two ways the value of V can change in this setup: either the output of A changes or the value of the weight changes.

Since the output of A is computed by the neuron automatically, there's not much we can do to adjust that value directly. But we can change the weight, so let's look at that.

Let's modify the weight AC by adding some new value to it. We'll call that new value ACm , so the new value of V is $(AC+ACm)$. Then the value arriving at C is $Ao \times (AC+ACm)$. This is shown in Figure 18.22.

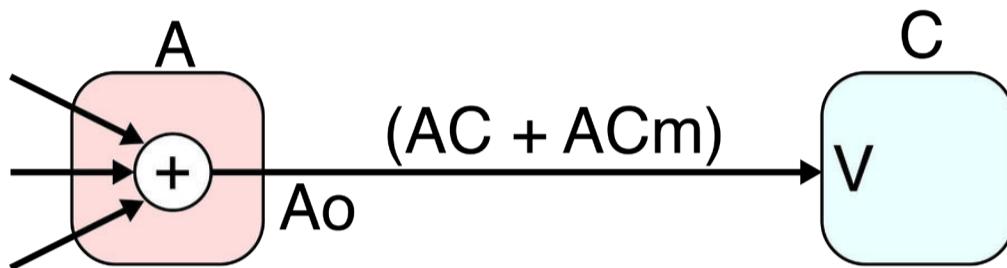


Figure 18.22: If we change the weight AC , then this will change the value of V . Here, the value of V coming into neuron C is $Ao \times (AC+ACm)$.

If we subtract the old value $Ao \times AC$ from the new value $Ao \times (AC+ACm)$, we find that the change in the value coming into C due to our modifying the weight is $Ao \times ACm$.

Since this change goes into C, the change will then get multiplied by $C\delta$, telling us the change in the network's error as a result of modifying the weight.

Hold on a second. That's what we've wanted this whole time, to find out what a change in the weight would do to the error. Have we just found that?

Yup, we have. We've achieved our goal! We discovered how a change to a weight would affect the network's error.

Let's look at that again.

If we change the weight AC by adding ACm to it, then we know the change in the network's error is given by the change in C, $(Ao \times ACm)$, times the delta for C, or $C\delta$. That is, the change in error is $(Ao \times ACm) \times C\delta$.

Let's suppose we increase the weight by 1. Then $ACm=1$, so the error will change by $Ao \times C\delta$.

So every change of $+1$ to the weight AC will cause a change of $Ao \times C\delta$ to the network error. If we add 2 to the weight AC , we'll get double this change, or $2 \times (Ao \times C\delta)$. If we add 0.5 to the weight, we'll get half as much, or $0.5 \times (Ao \times C\delta)$.

We can turn this around, and say that if we want to increase the error by 1 , we should add $Ao \times C\delta$ to the weight. But we want to make the error go down. The same logic says that we can reduce the error by 1 if we instead *subtract* the value $Ao \times C\delta$ from the weight. So now we know how to change this weight in order to reduce the overall error.

We've found what to do to the weight AC to decrease the error in this little network. To subtract 1 from the error, we subtract $Ao \times C\delta$ from AC .

Figure 18.23 shows that every change of $Ao \times C\delta$ in the weight AC leads to a corresponding change of 1 in the network error in our network. And subtracting that value leads to a change of -1 in the error.

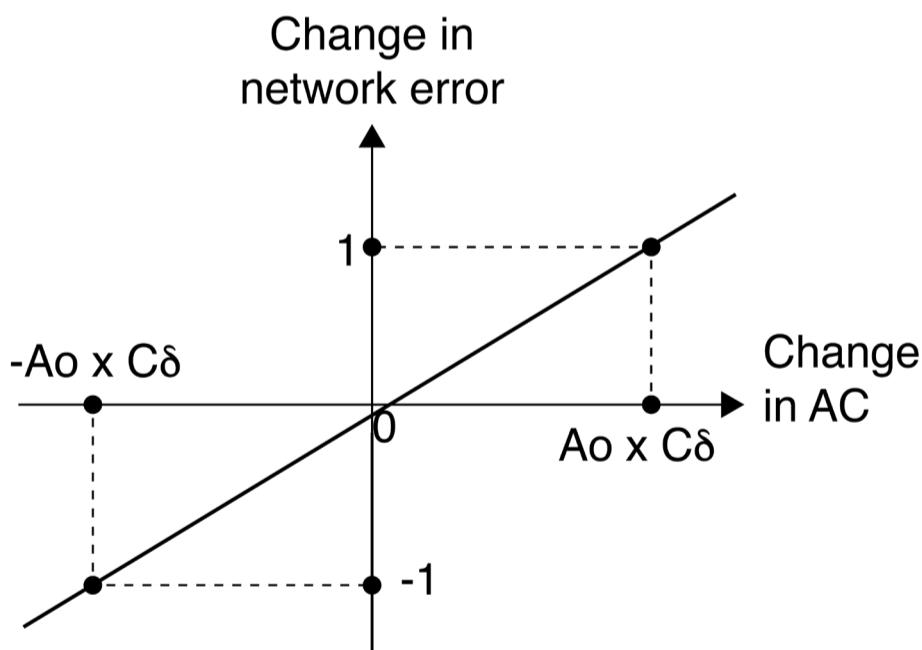


Figure 18.23: For each change in the value of the weight AC , we can look up the corresponding change in our network's error. When AC changes by $Ao \times C\delta$, the network error changes by 1 .

We can summarize this process visually with a new 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 18.24.

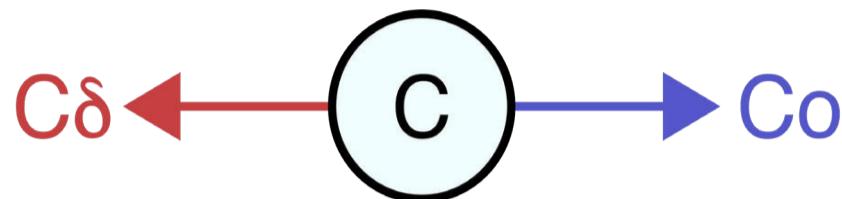


Figure 18.24: Neuron C has an output Co , drawn with an arrow pointing right, and a delta $C\delta$, drawn with an arrow pointing left.

With this convention, the whole process for finding the updated value for a weight is summarized in Figure 18.25. 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 $y-x$?). Our approach to computing $AC - (Ao \times C\delta)$ is to find $Ao \times C\delta$, multiply that by -1 , and then add that result to AC .

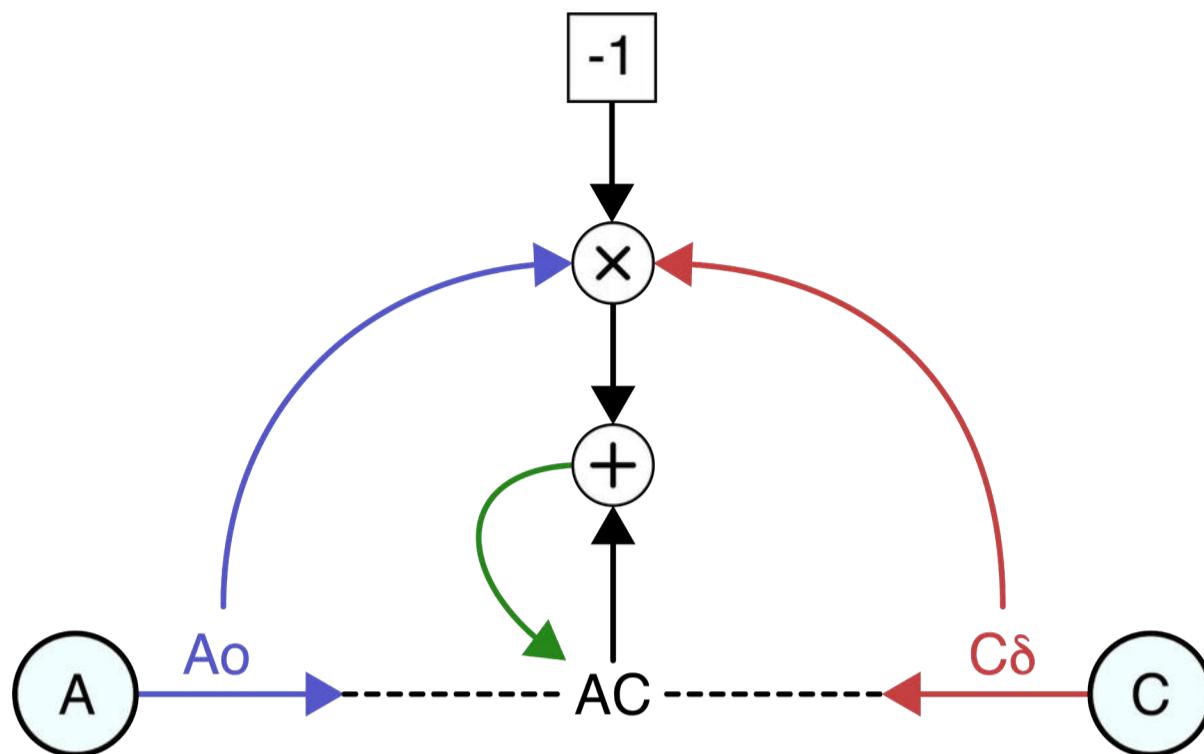


Figure 18.25: Updating the value of weight AC . We start with the output Ao from neuron A and the delta $C\delta$ from neuron C, and multiply them together. We'd like to subtract this from the current value of AC . To show this clearly in the diagram, we instead multiply the product by -1 , and then add it to AC . The green arrow is the update step, where this result becomes the new value of AC .

Figure 18.25 is the goal of this chapter, and the reason we computed $C\delta$.

This is how our network learns.

Figure 18.25 tells us how to change the weight AC to bring down the error. The diagram says that to reduce the error by -1 , we should add $-Ao \times C\delta$ to the value of AC .

Success!

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 D_o and D doesn't depend on C_o . 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 last two weights in the network, but how about all the other weights? To use this technique, we need to figure out the deltas for all the neurons in all the remaining layers. Then we can use Figure 18.25 to improve all the weights in the network.

And this brings us to the remarkable trick of backpropagation: we can use the neuron deltas at one layer to find all the neuron deltas for its preceding layer. And as we've just seen, knowing the neuron deltas and the neuron outputs tells us how to update all the weights coming into that neuron.

Let's see how to do that.

18.8 Step 3: Other Neuron Deltas

Now that we have the delta values for the output neurons, we will use them to compute the deltas for neurons on the layer just before the output layer. In our simple model, that's just neurons A and B. Let's focus for the moment just on neuron A, and its connection to neuron C.

What happens if A_o , the output of A, changes for some reason? Let's say it goes up by A_m . Figure 18.26 follows the chain of actions from this change in A_o to the change in C_o to the change in the error.

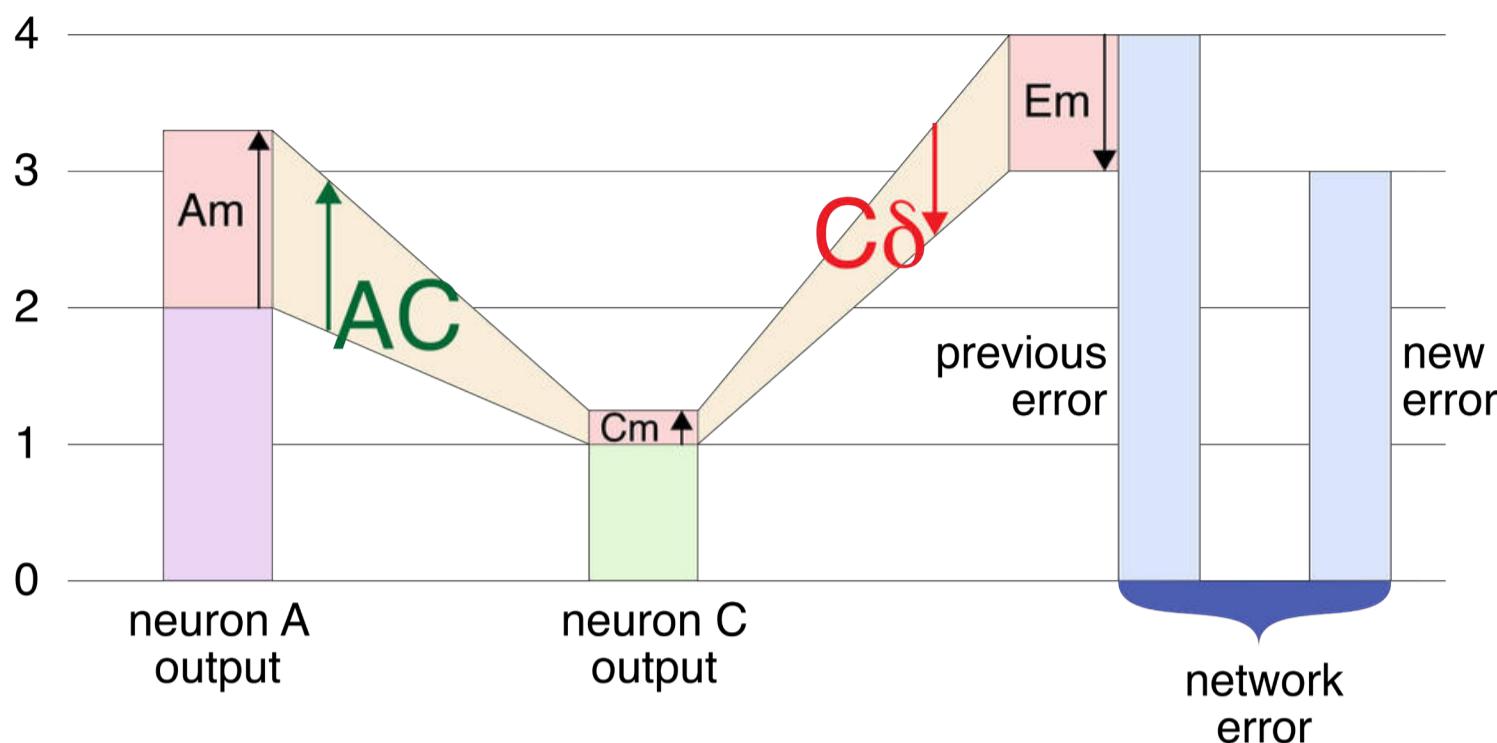


Figure 18.26: Following the results if we change the output of neuron A. Read the diagram left to right. The change to A, shown as Am , is multiplied by the weight AC and is added to the values accumulated by neuron C. This raises the output of C by Cm . As we know, this change in C can be multiplied by $C\delta$ to find the change in the network error. In this example, $Am=5/4$, and $AC=1/5$, so $Cm=5/4 \times 1/5=1/4$. The value of $C\delta$ is -4 , so the change in the error is $1/4 \times -4=-1$.

We know that the neuron C adds together its weighted inputs and then passes on the sum (since we're ignoring the activation function for now). So if nothing else changes in C except for the value coming from A, that's the only source of any change in C_o , the output of C. We represent that change to C_o as the value Cm . As we saw before, we can predict the change in the network's error by multiplying Cm by $C\delta$.

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 A_o (that is, Am) by the weight AC , we'll get Cm , the change in the output of C. And we know from above that if we multiply Cm by $C\delta$, we get the change in the error.

So mushing this all together, we find that the error due to a change Am in the output of A is $Am \times AC \times C\delta$.

In other words, if we multiply the change in A (that is, Am) by $AC \times C\delta$, we get the change in the error due to the change in neuron A. That is, $A\delta = AC \times C\delta$.

We just identified the delta of A! Since the delta is the value that we multiply a neuron's change by to find the change in the error, and we just found that value is $AC \times C\delta$, then we've found $A\delta$.

Figure 18.27 shows this visually.

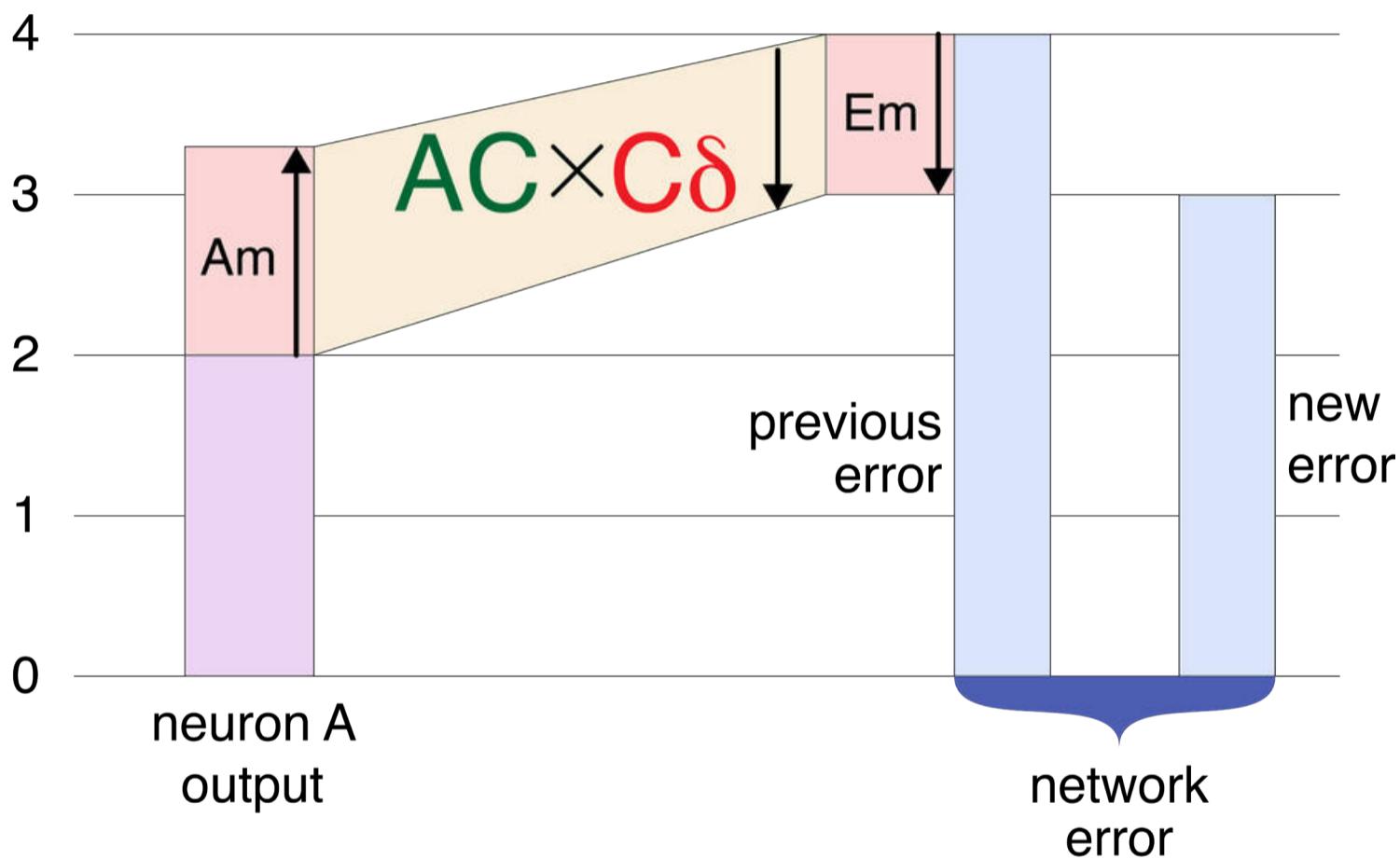


Figure 18.27: We can mush together the operations in Figure 18.26 into a more succinct diagram. In that figure, we saw that Am , the change in A, gets multiplied by the weight AC and then the value $C\delta$. So we can represent that as a single step, where we multiply Am by the two values AC and $C\delta$ multiplied together. As before, the value of $Am=5/4$, $AC=1/5$, and $C\delta$ is -4 . So $AC \times C\delta=-4/5$, and multiplying that by $Am=5/4$ gives us a change in the error of -1.0 .

This is kind of amazing. Neuron C has disappeared. It's literally out of the picture in Figure 18.27. All we needed was its delta, $C\delta$, and from that we could find $A\delta$, the delta for A. And now that we know $A\delta$, we can update all of the weights that feed into neuron A, and then... no, wait a second.

We don't really have $A\delta$ 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, we can see that neuron D also uses the output of A. If Ao changes due to Am , then the output of D will change as well, and that will also have an effect on the error.

To find the change in the error due to neuron D, caused by a change to neuron A, we can repeat the process above, just replacing neuron C with neuron D. So if Ao changes by Am , and nothing else changes, the change in the error due to the change in D is given by $AC \times D\delta$.

Figure 18.28 shows these two paths at the same time. This figure is set up slightly differently from the ones above. 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.

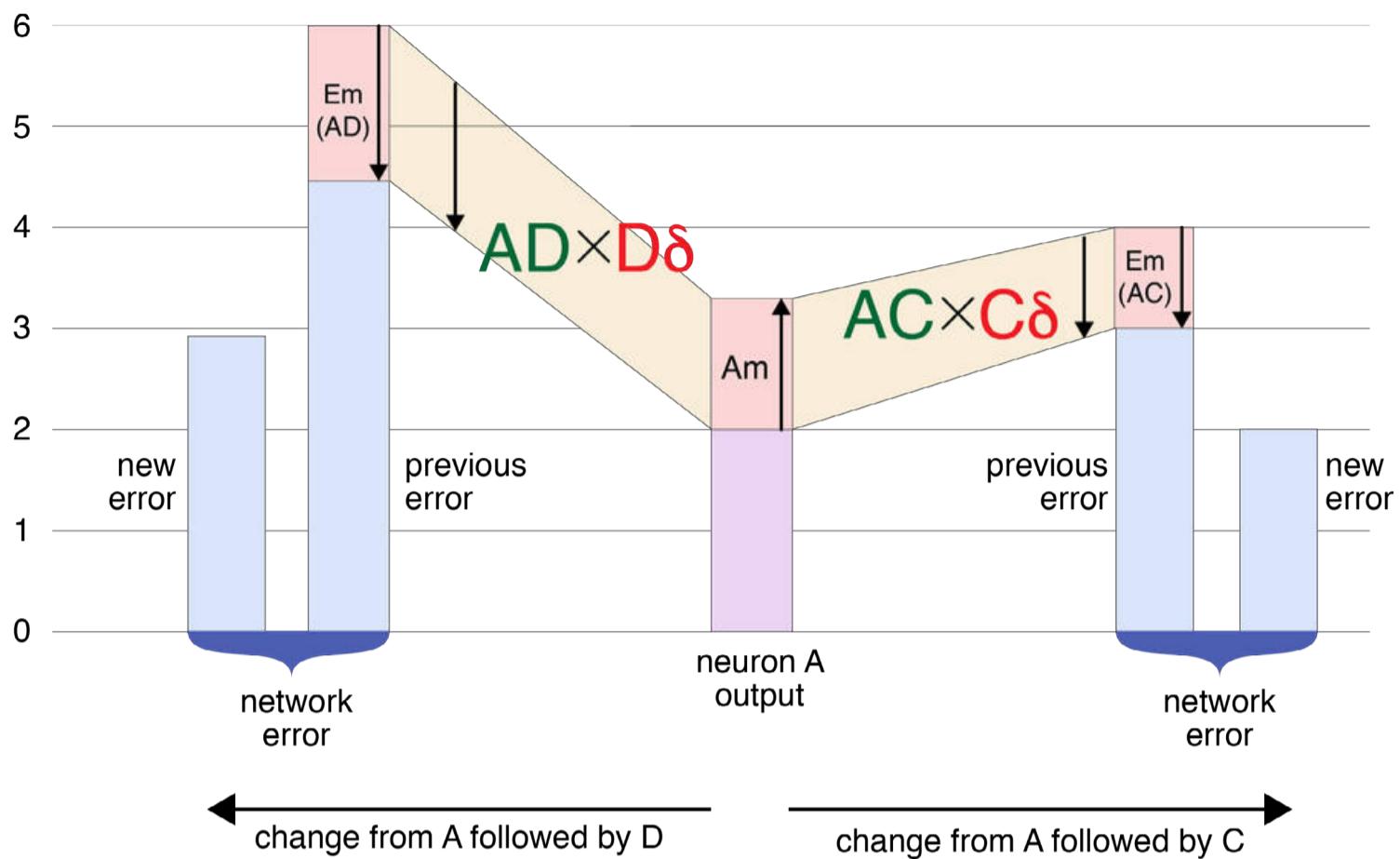


Figure 18.28: The output of neuron A is used by both neuron C and neuron D. In this figure, we've changed our left-to-right convention to show the same change in A, given by Am , affecting the final error by way of two different paths, one to the left and one to the right, each starting at neuron A in the center. The path through neuron C is shown going to the right, where Am , the change in the output of A, is scaled by $AC \times C\delta$ to get a change in the error labeled $Em(AC)$. Moving from the center to the left, Am is scaled by $AD \times D\delta$ to get another change to the error, labeled $Em(AD)$. The result is two separate changes to the error.

Figure 18.28 shows two separate changes to the error. Since neurons C and D don't influence each other, so their effects on the error are independent. To find the total change to the error, we just add up the two changes. Figure 18.29 shows the result of adding the change in error via neuron C and the change via neuron D.

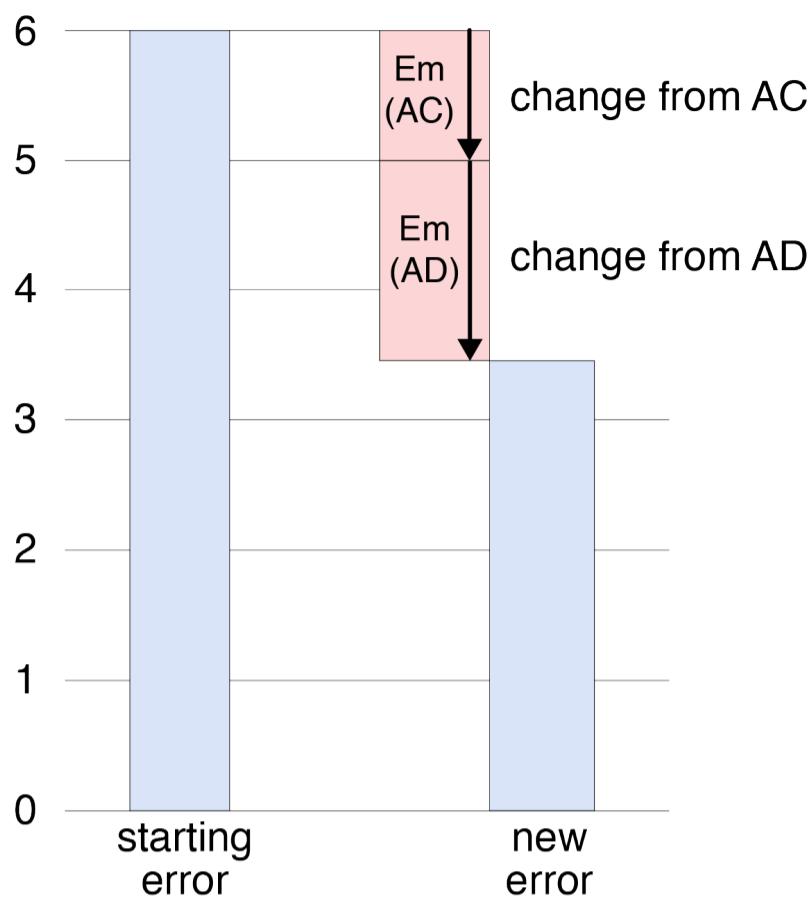


Figure 18.29: When the output of neuron A is used by both neuron C and neuron D, the resulting changes to the error add together.

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 18.29, 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.

We've actually done something far better than find 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 that, 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 18.30. We'll 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.

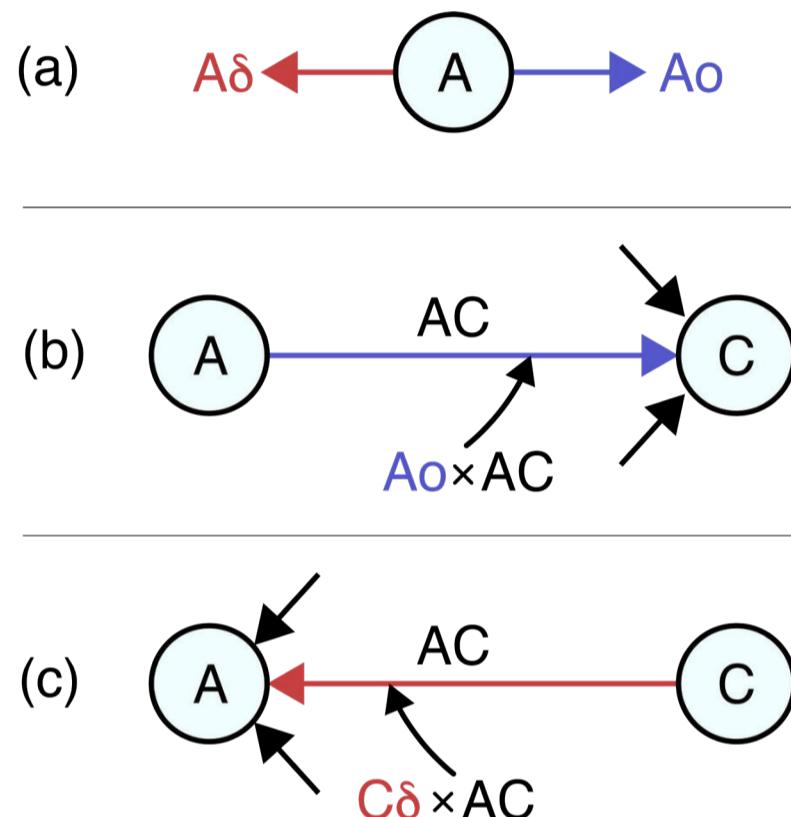


Figure 18.30: Drawing the values associated with neuron A. (a) Our convention is to draw the output Ao as an arrow coming out of the right of the neuron, and the delta $A\delta$ as an arrow coming out of the left. (b) The output of A is multiplied by AC on its way to being used by C when we're evaluating a sample. (c) The delta of C is multiplied by AC on its way to being used by A when we're computing delta values.

In other words, 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 drawing, 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 18.30(b).

When we later want to compute $A\delta$, we draw 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 18.30(c).

Now we can summarize both the processing of a sample input, and the computation of the deltas, in Figure 18.31.

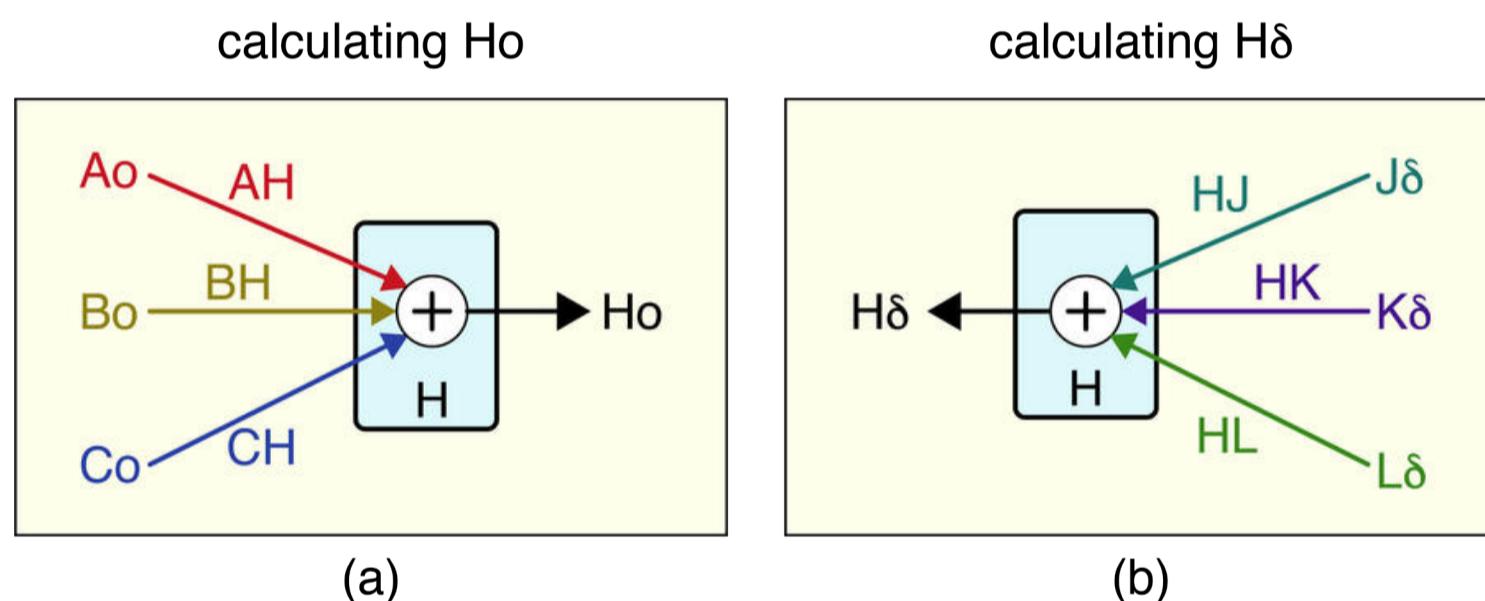


Figure 18.31: Calculating the output and delta for neuron H. Left: To calculate H_o , we scale the output of each preceding neuron by the weight of its connection and add the results together. Right: To calculate $H\delta$, we scale the delta of each following neuron by the connection's weight and add the results together.

This is pleasingly symmetrical. It also reveals an important practical result: when a neuron is connected to the same number of preceding and following neurons, calculating the delta for a neuron takes the same amount of work (and therefore the same amount of time) as calculating its output. So calculating deltas is as efficient as calculating output values. Even when the input and output counts are different, the amount of work involved is still close in both directions.

Note that Figure 18.31 doesn't require anything of neuron H except that it has inputs from a preceding layer that travel on connections with weights, and deltas from a following layer that travel on connections with weights. So we can apply the left half of Figure 18.31 and calculate the output of neuron H as soon as the outputs from the previous layer are available. And we can apply the right half of Figure 18.31 and calculate the delta of neuron H as soon as the deltas from the following layer are available.

This also tells us why we had to treat the output layer neurons as a special case: there are no “next layer” deltas to be used.

This process of finding the delta for every neuron in the network *is* the backpropagation algorithm.

18.9 Backprop in Action

In the last section we saw the backpropagation algorithm, which lets us compute the delta for every neuron that in a network.

Because that calculation depended on the deltas in the following neurons, and the output neurons don't have any of those, we had to 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 back one layer (towards the inputs), and find the deltas all the neurons on that layer, and then step back again, compute all the deltas, step back again, and so on until we reach the input.

Let's walk through the process of using backprop to find the deltas for all the neurons in a slightly larger network.

In Figure 18.32 we show a new network with four layers. There are still two inputs and outputs, but now we have 3 hidden layers of 2, 4, and 3 neurons.

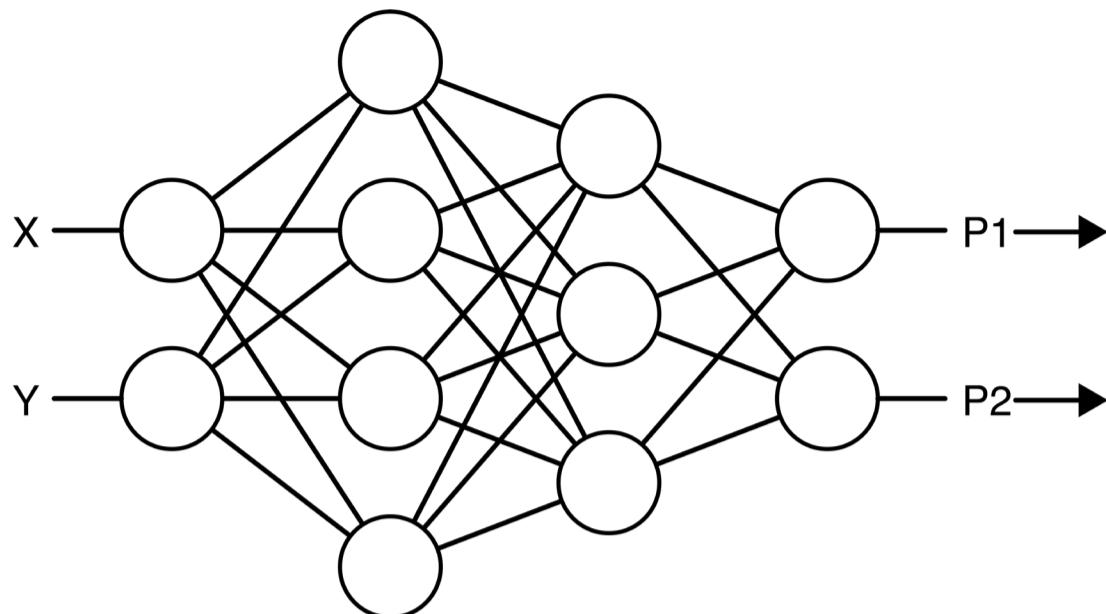


Figure 18.32: A new classifier network with 2 inputs, 2 outputs, and 3 hidden layers.

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'll start backpropagation by finding the error in the output neurons, as shown in Figure 18.33.

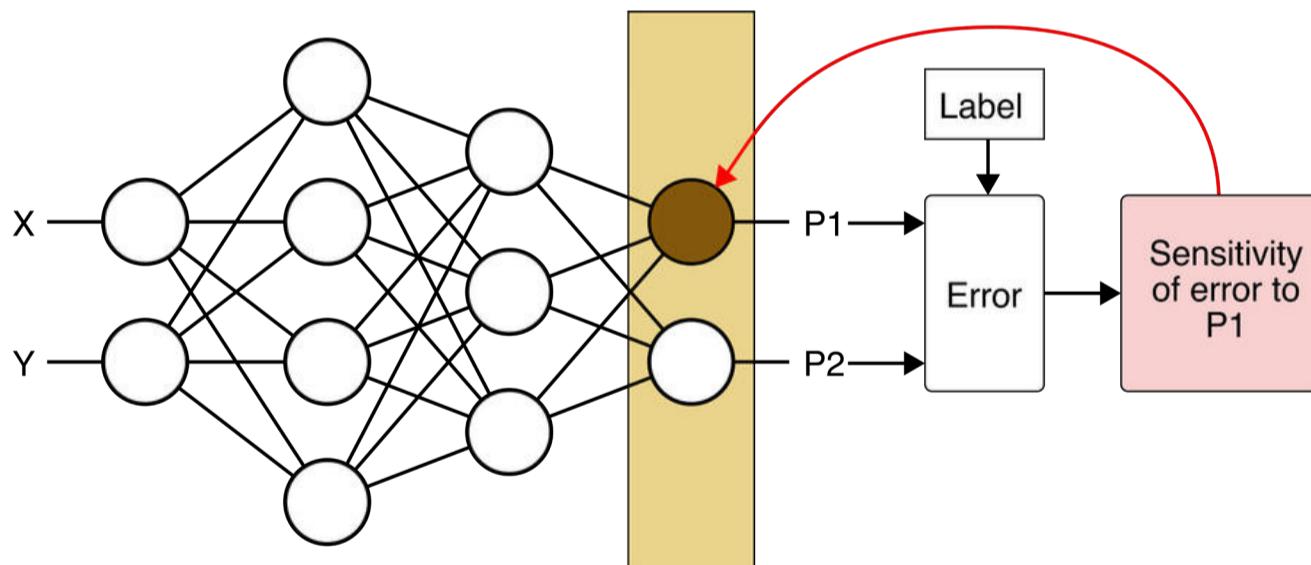


Figure 18.33: Computing the delta of the first output neuron in a new network. Using the general approach, we take the outputs of the output layer (here called P1 and P2) and compare them to the label to derive an error. From the outputs, the label, and the error value we find how much a change in P1 would change the error. That value is the delta stored at the neuron that produces P1.

We've begun arbitrarily with the upper neuron, which gives us the prediction we've labeled P_1 (the likelihood that the sample is in class 1). From the values of P_1 and P_2 and the label, we can compute the error in the network's output. Let's suppose the network didn't get this sample perfectly predicted, so the error is greater than zero.

Using the error, the label, and the values of P_1 and P_2 , 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 18.20. But if we're using some other function, it might be more complicated, so we've illustrated the general case.

Once we've computed the value of delta for this neuron, we store it with the neuron, and we're done with that neuron for now.

We'll repeat this process for all the other neurons in the output layer (here we have only one more). That finishes up the output layer, since we now have a delta for every neuron in the layer. Figure 18.34 summarizes these two neurons getting their deltas.

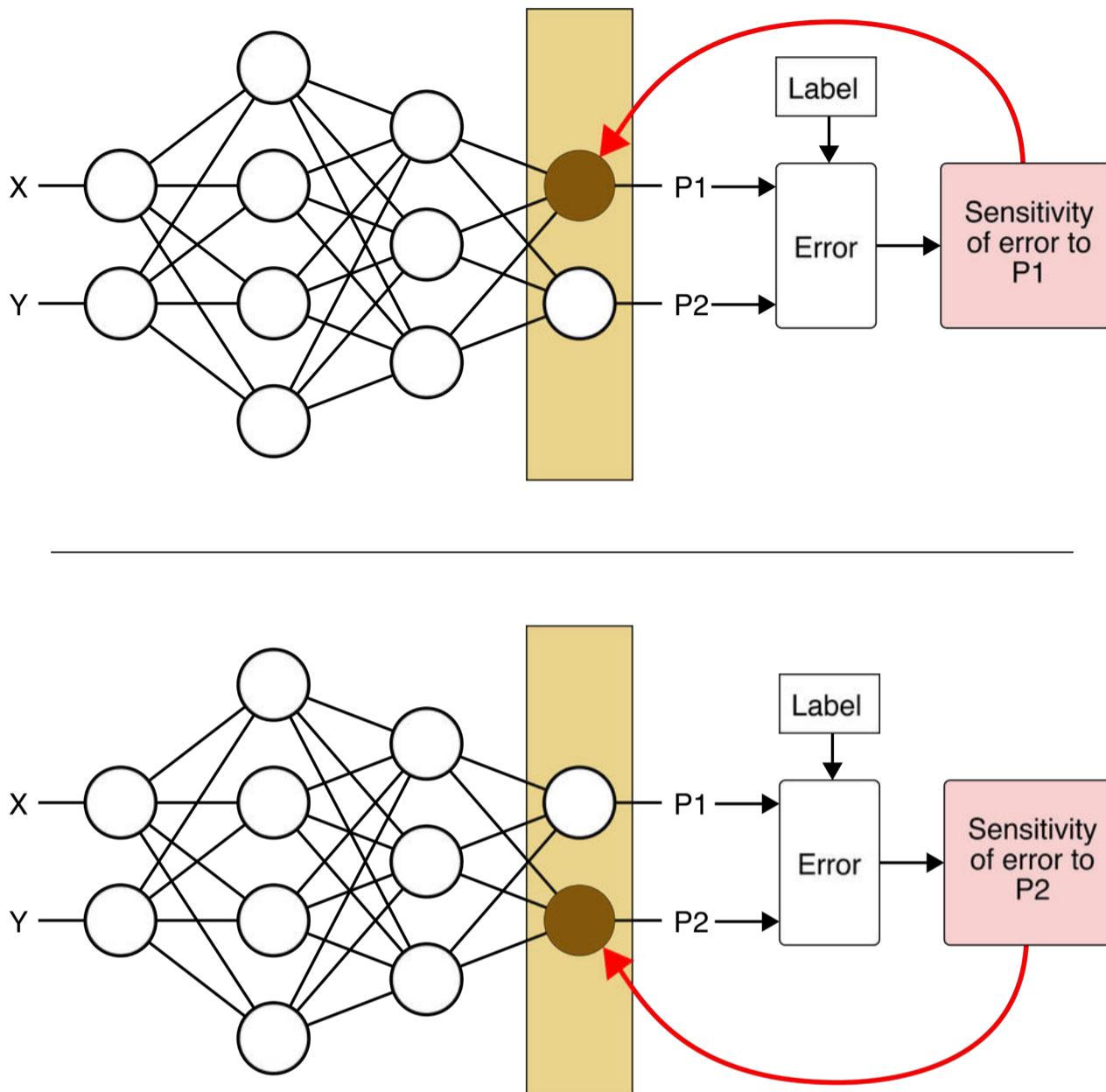


Figure 18.34: Summarizing the steps for finding the delta for both output neurons.

At this point we could start adjusting the weights coming into the output layer, but we usually break things up by first finding all the neuron deltas, and then adjusting all the weights. Let's follow that typical sequence here.

So we'll move backwards one step to the third hidden layer (the one with 3 neurons). Let's consider finding the value of delta for the top-most of these three, as in the left image of Figure 18.35.

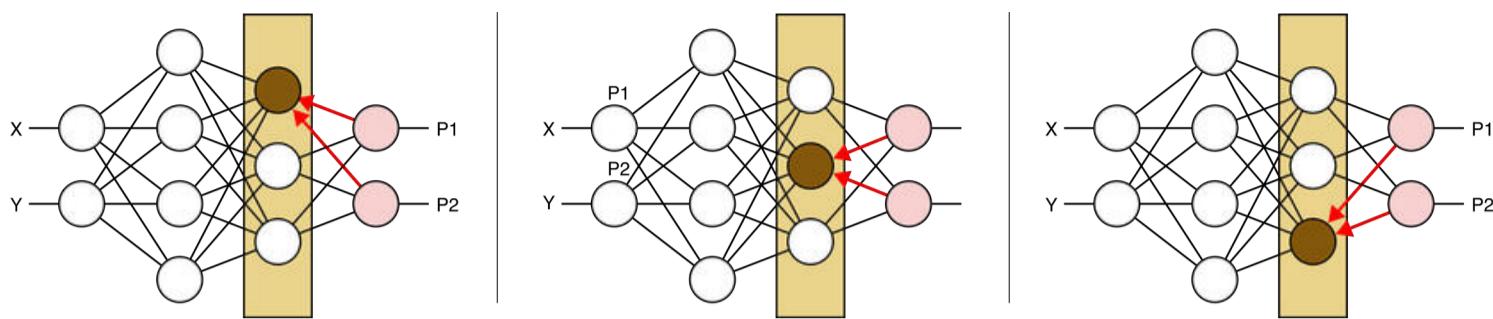


Figure 18.35: Using backpropagation to find the deltas for the next-to-last layer of neurons. To find the delta for each neuron, we find the deltas of the neurons that use its output, multiply those deltas by the corresponding weights, and add the results together.

To find the delta for this neuron, we follow the recipe of Figure 18.28 to get the individual contributions, and then the recipe of Figure 18.29 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 3-neuron layer, we take a step backwards and start on the hidden layer with 4 neurons.

And 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 this 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 any more now. All we need are the deltas in the next layer's neurons, and the weights that get us to those neurons.

Figure 18.36 shows how we compute the deltas for the four neurons in the second hidden layer.

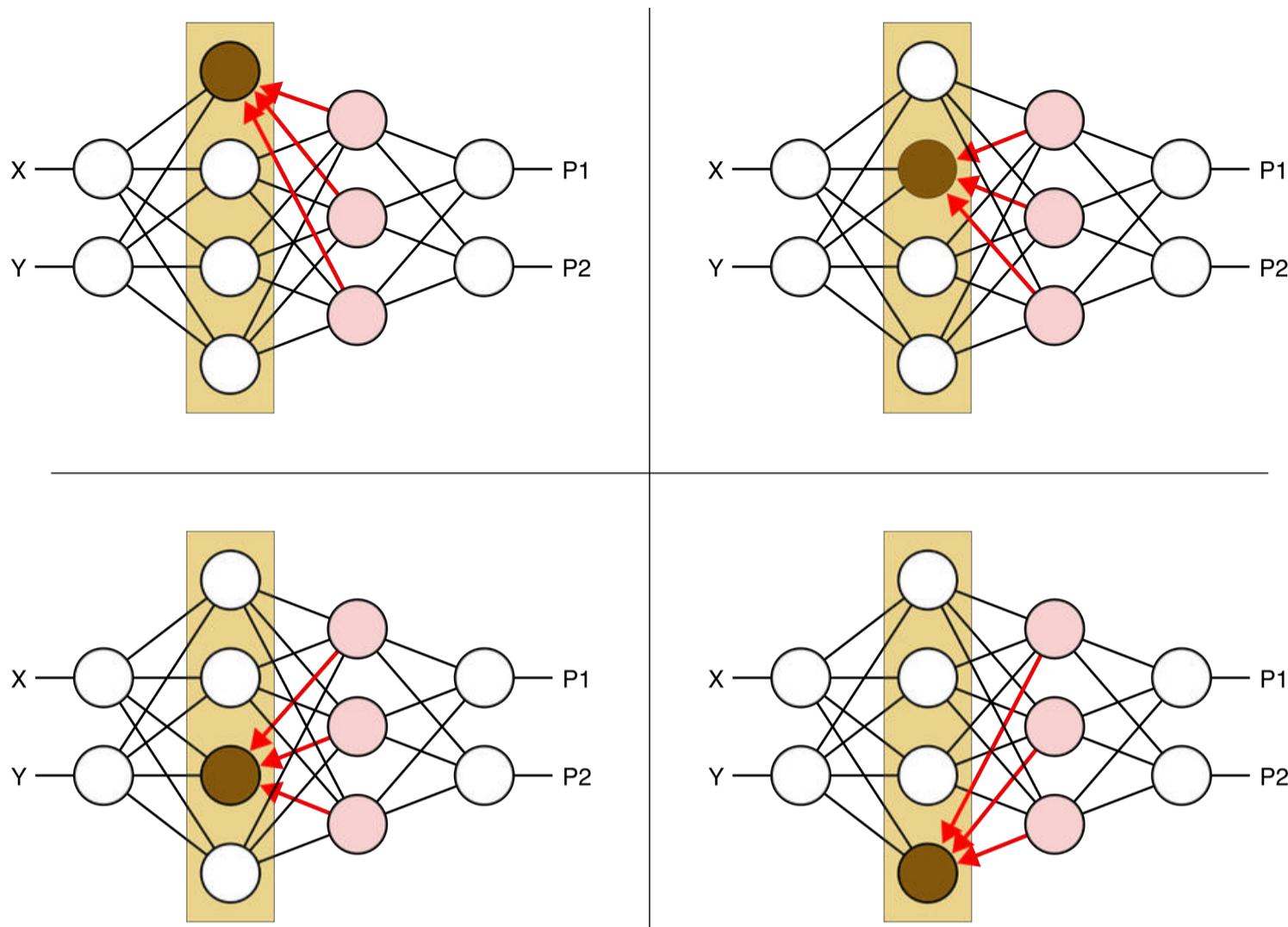


Figure 18.36: Using backprop to find the delta values for the second hidden layer.

When all 4 neurons have had deltas assigned to them, that layer is finished, and we take another step backwards.

Now we're at the first hidden layer with two neurons. Each of these connects to the 4 neurons on the next layer. Once again, all we care about now 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 18.37.

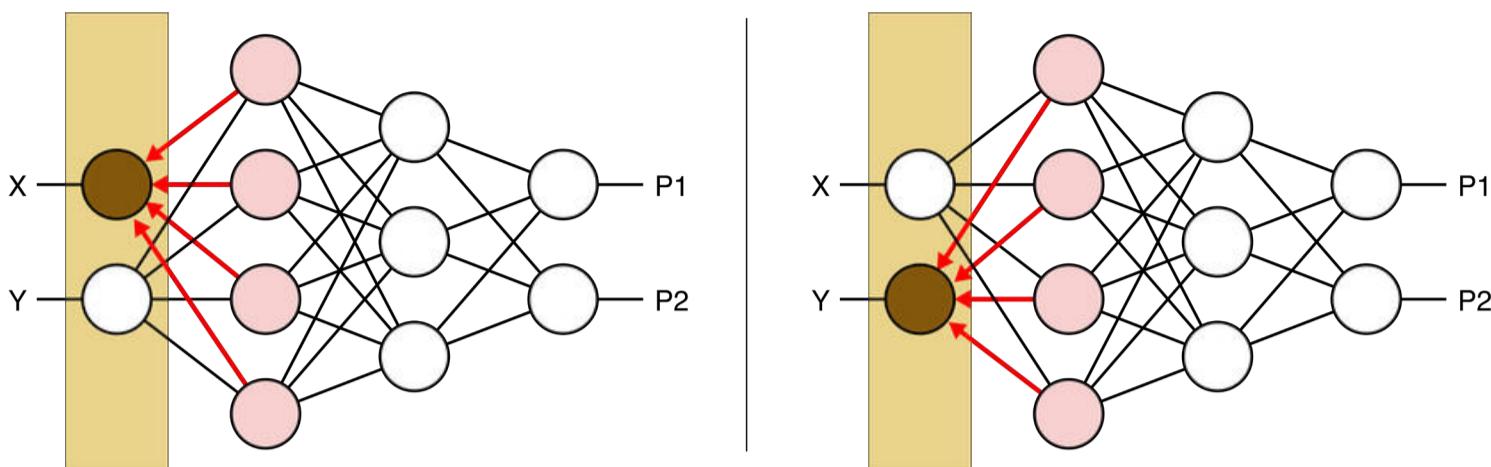


Figure 18.37: Using backprop to find the neurons for the first hidden layer.

When Figure 18.37 is complete, we've found the delta for every neuron in the network.

Now we'll adjust the weights. We'll run through the connections between neurons and use the technique we saw in Figure 18.25 to update for every weight to a new and improved value.

Figure 18.34 through Figure 18.37 show why the algorithm is called *backwards propagation*. We're taking the deltas from any layer and *propagating*, or moving, their information *backwards* one layer at a time, modifying it as we go.

As we've seen, computing each of these delta values is fast. It's just one multiplication per outgoing connection, and then adding those pieces together. That takes almost no time at all.

Backprop becomes highly efficient when we use parallel hardware like a GPU. Because the neurons on a layer of a feed-forward network don't interact, and the weights and deltas that get multiplied are already computed, we can use a GPU to multiply *all* the deltas and weights for *an entire layer* at once.

Computing an entire layer's worth of deltas in parallel saves us a lot of time. If each of our layers had 100 neurons, and we had enough hardware, computing all 400 deltas would take only the same time required to find 4 deltas.

The tremendous efficiency that comes from this parallelism is a key reason why backprop is so important.

Now we have all of the deltas, and we know how to update the weights. Once we actually do update the weights, we should re-compute all the deltas, because they're based on the weights.

We're just about done, but we need to make good on our earlier promise and put activation functions back into our neurons.

18.10 Using Activation Functions

Including the activation function in backpropagation is a small step. But understanding that step and why it's the right thing to do takes a bit of thinking. We left it off in our earlier discussion so we wouldn't get distracted, but let's now get the activation function back in there.

We'll start by thinking about a neuron during the feed-forward step, when we're evaluating a sample and data is flowing from left to right, producing neuron outputs as it goes.

When we calculate the output of a neuron, the sum of the weighted inputs goes through an activation function before it leaves the neuron, as shown in Figure 18.38.

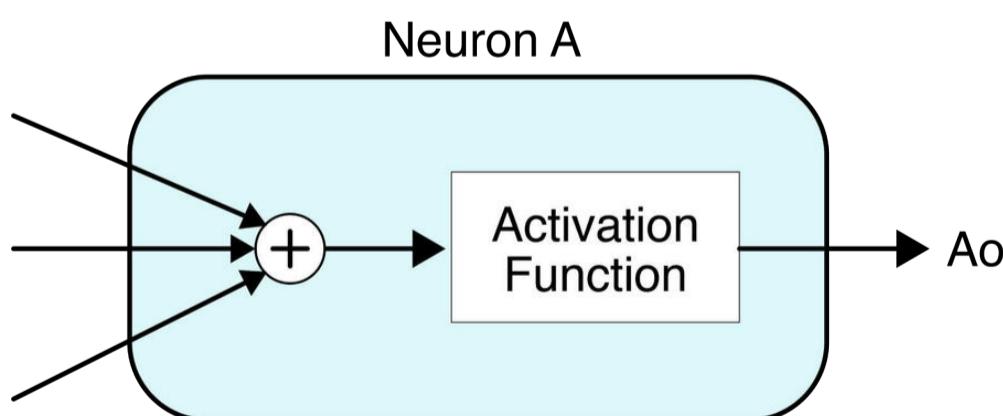


Figure 18.38: Neuron A, with its activation function.

To make things a bit more specific, let's choose an activation function. We'll pick the sigmoid, discussed in Chapter 17, because it's smooth and makes for clear demonstrations. We won't use any particular qualities of the sigmoid, so our discussion will be applicable to any activation function.

Figure 18.39 shows a plot of the sigmoid curve. Increasingly large positive values approach 1 ever more closely without quite getting there, and increasingly large negative values approach 0, but never quite get there, either. Rather than constantly refer to values with phrases like “very, very nearly 1” or “extremely close to 0,” let's say for simplicity that input values that are greater than about 7 can be considered to get an output value of 1, while those less than -7 can be considered to get an output of 0. Values in the range (-7, 7) will be smoothly blended in the S-shaped function that gives the sigmoid its name.

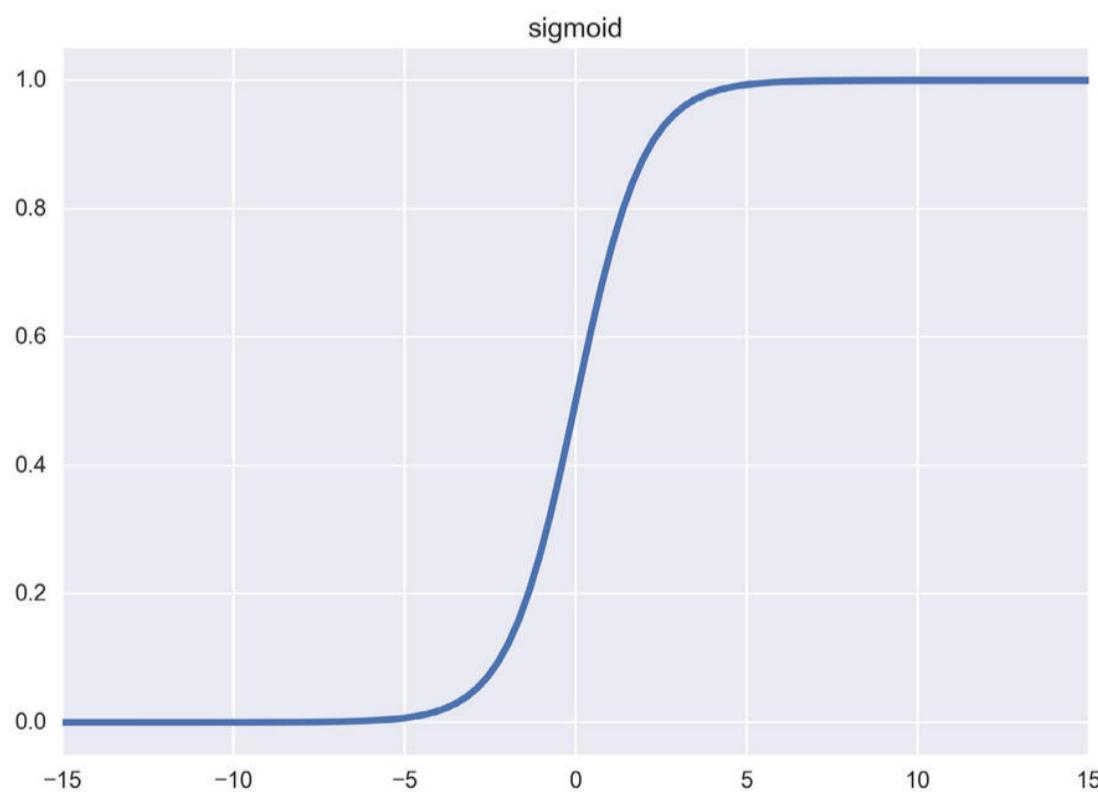


Figure 18.39: The sigmoid curve, plotted from -15 to 15. Values greater than about 7 or less than about -7 are very near to 1 and 0, respectively.

Let's look at neuron C in our original tiny four-neuron network of Figure 18.8. Neuron C gets one input from neuron A, and one from neuron B. For now, let's look just at the input from A, as in Figure 18.40. The value A_o , or the output of A, gets multiplied by the weight

AC before it gets summed with all the other inputs in C . Since we're focusing just on the pair of neurons A and C , we'll ignore any other inputs to C . The input value $Ao \times AC$ is then used as the input to the activation function to find the output value Co .

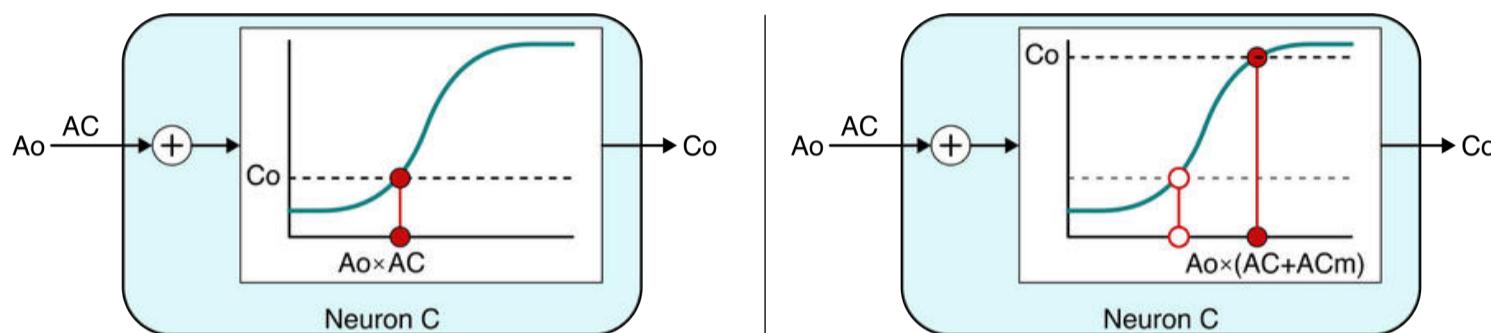


Figure 18.40: Ignoring the other inputs to A for the moment, the input to the activation function is given by multiplying the output of A and the weight AC . We can find the value of the activation function at that point, which gives us the output Co . Left: When the output of A is Ao and the weight is AC , the value into the activation function is $Ao \times AC$. Right: When the output of A is Ao and the weight is $AC + ACm$, the value into the activation function is $Ao \times (AC + ACm)$. Here we show the values from the left plot as dots with white in the center, and the new values as filled-in dots. Note that the change in the output is substantial, because we're on a steep part of the curve.

We know that to reduce the error, we'll be adding some positive or negative number ACm to the value of the weight AC . The right diagram in Figure 18.40 shows the result of adding a positive value of ACm . In this case, the output Co changes by a lot, because we're on a steep part of the curve.

So this says that by adding ACm to the weight, we're going to have a *bigger* effect on the output error than we would have had without the activation function, because that function has turned our change of ACm into something larger.

Suppose instead our starting value of $Ao \times AC$ put us near a shallow part of the curve, and we add the same ACm to AC , as in Figure 18.41.

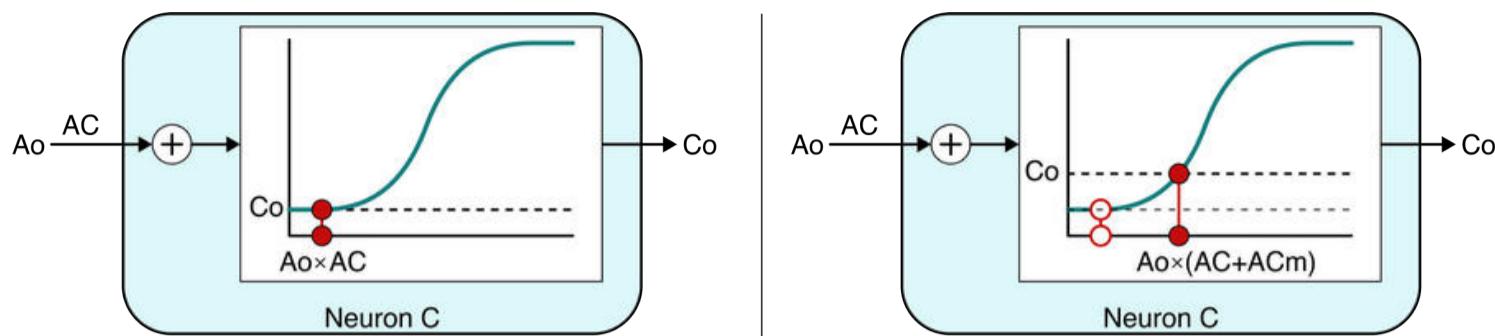


Figure 18.41: In a shallow part of the curve. Left: Before adding ACm as in Figure 18.40. Right: Adding this value results in a small change in the output of C .

Now adding the same value ACm to the weight causes a smaller change in Co than before. In this case, it's even less than ACm itself. The smaller change to the output means there will be a smaller change in the network error. In other words, adding ACm to the weight in this situation will result in a *smaller* change to the output error than we'd get if there was no activation function present.

What we'd love to have is something that can tell us, for any point on the activation function curve, how steep the curve is at that point. When we're at a point where the curve is going up steeply to the right, positive changes in the input will be amplified a lot, as in Figure 18.40. When we're at a point where the curve is going up shallowly to the right, as in Figure 18.41, such changes will be amplified by a little.

If we change the input with a negative value of ACm , and move our starting point to the left, then the amount of slope tells us how much the change in the error will decrease. We get the same situations as in Figure 18.40(b) and Figure 18.41(b), but with the starting and ending positions reversed.

Happily, we already know how to find the slope of a curve: that's just the derivative. Figure 18.42 shows the sigmoid, and its derivative.

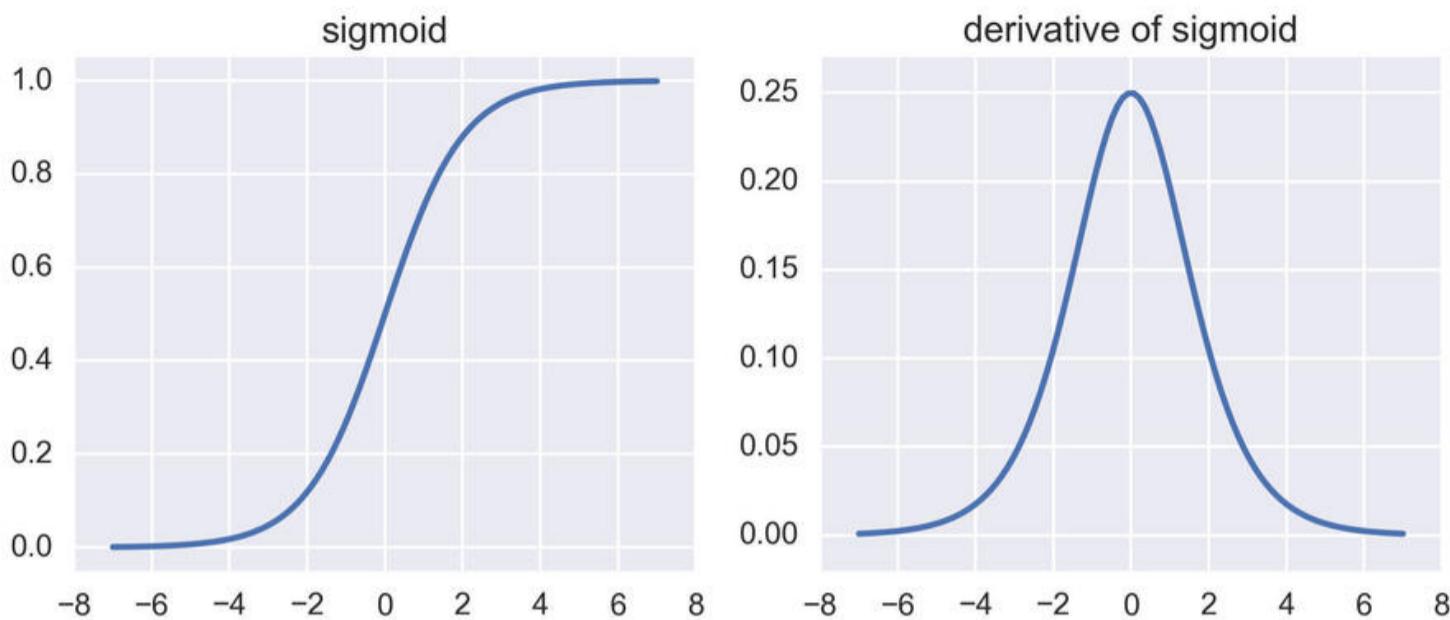


Figure 18.42: The sigmoid curve and its derivative. Note that the vertical scales are different for the two plots.

The sigmoid is flat at the left and right ends. So if we're in the flat regions and we move left or right a little bit, that will cause little to no change in the output of the function. In other words, the curve is flat, or has no slope, or has a derivative of 0. The derivative increases as the input moves from about -7 to 0 because the curve is getting steeper. Then the derivative decreases back to 0 again even as the input keeps going up because the curve becomes more shallow off to the right, approaching 1 but never quite getting there.

If we were to work through the math, we'd find that this derivative is exactly what we need to fix our prediction of the change to the error based on a change to a weight. When we're on a steep part of the curve, we want to crank up the value of delta for this neuron, because changes to its inputs will cause big changes in the activation function, and thus have big changes to the network error. When we're on a shallow part of the curve, then changes to the inputs will have little effect on the change in the output, so we want to make this neuron's delta smaller.

In other words, to account for the activation function, we just take the delta we normally compute, and multiply it by the derivative of the activation function, evaluated at the same point that we used during

the forward pass. Now the delta accounts for how the activation function will exaggerate or diminish the amount of change in the neuron's output, and hence its effect on the network error.

To keep things nicely bundled, we can perform this step immediately after computing the delta as we did before.

The whole business for one neuron is summarized in Figure 18.43. Here we imagine we have a neuron H. The top part is the forward pass, when we're evaluating a sample and computing this neuron's output. Following a common convention, we've given the name z to the result of the summation step. The value of the activation function is what we get when we look vertically upwards from the point z on the X axis. The bottom part of the figure is the backward pass, where we compute this neuron's delta. Again, we use z to find the derivative of the activation function, and we multiply our incoming sum by that before passing it on.

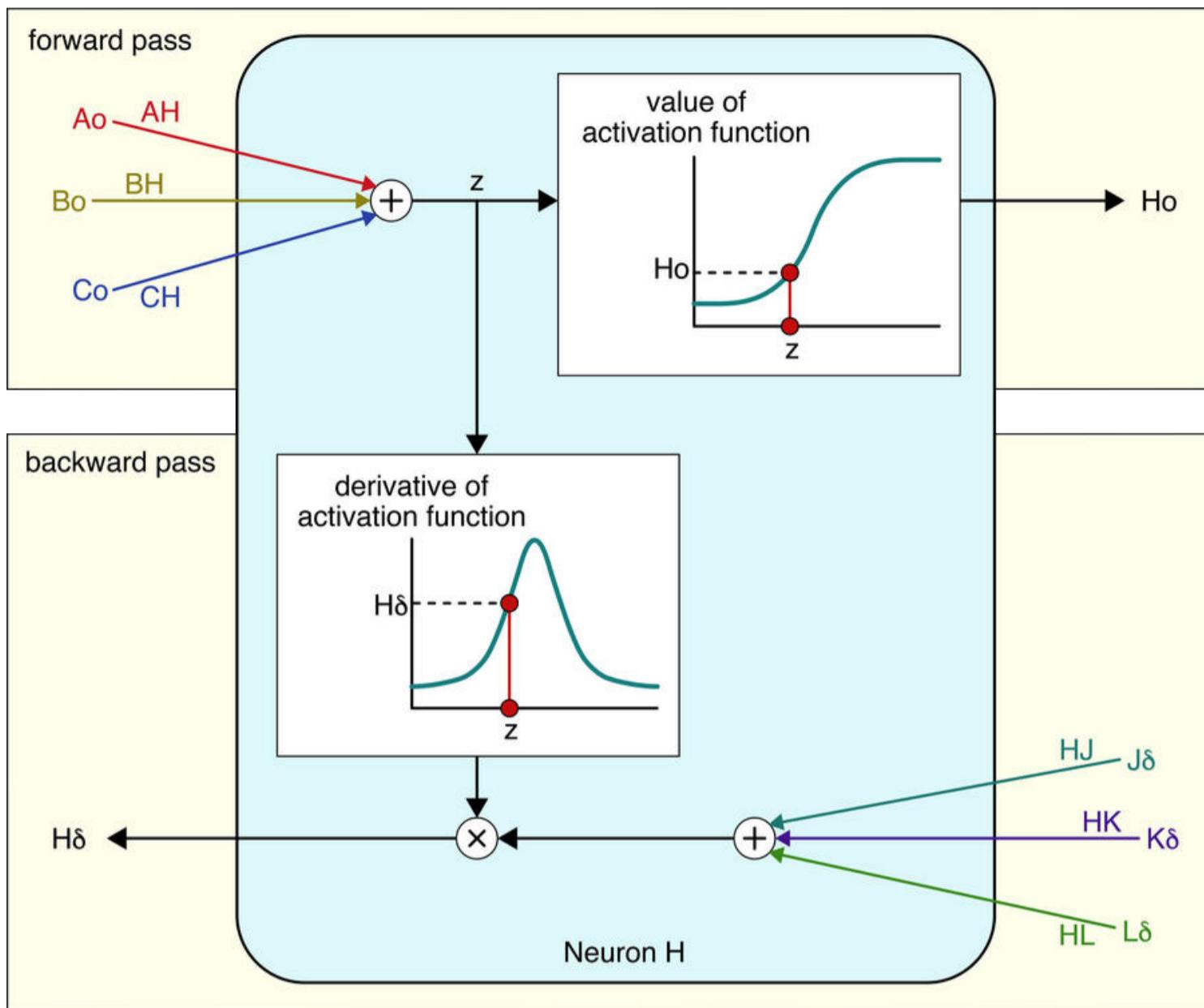


Figure 18.43: Network evaluation and backpropagation of deltas in a nutshell, for neuron H. Top: In the forward pass, the weighted inputs are added together, giving us a value we call z . The value of the activation function at z is our output Ho . Bottom: In the backward pass, the weighted deltas are added together, and we use the z from before to look up the derivative of the activation function. We multiply the sum of the weighted deltas by this value, giving us $H\delta$.

In this figure, neuron H has three inputs, coming from neurons A, B, and C, with output values Ao , Bo , and Co . During the forward pass, when we're finding the neuron's output, these are multiplied respectively by the weights AH , BH , and CH , and then added together. We've labeled this sum with the letter z . Now we look up z in the activation function, and its value is Ho , the output of this neuron.

Now when we run the backward pass to find the neuron's delta, we find the deltas of the neurons that use H_o as an input. Let's say they're neurons J, K and L. So we multiply their deltas $J\delta$, $K\delta$, and $L\delta$ by their respective weights HJ , HK , and HL , and add up those results.

Now we get the value of z from the forward pass, and use it to find the value of the derivative of the activation function. We multiply the sum we just found with this number, and the result is $H\delta$, the delta for this neuron.

This all goes for the output neurons too, if they have activation functions, only in the backward pass we use the error information rather than deltas from the next layer.

Notice how compact and local everything is. The forward pass depends only on the output values of the previous layer, and the weights that connect to them. The backward pass depends only on the deltas from the following layer, the weights that connect to them, and the activation function.

Now we can see why we were able to get away with ignoring the activation function throughout most of this chapter. We can pretend that we really did have an activation function all along: the **identity activation function**, shown in Figure 18.44. This has an output that's the same as its input. That is, it has no effect.

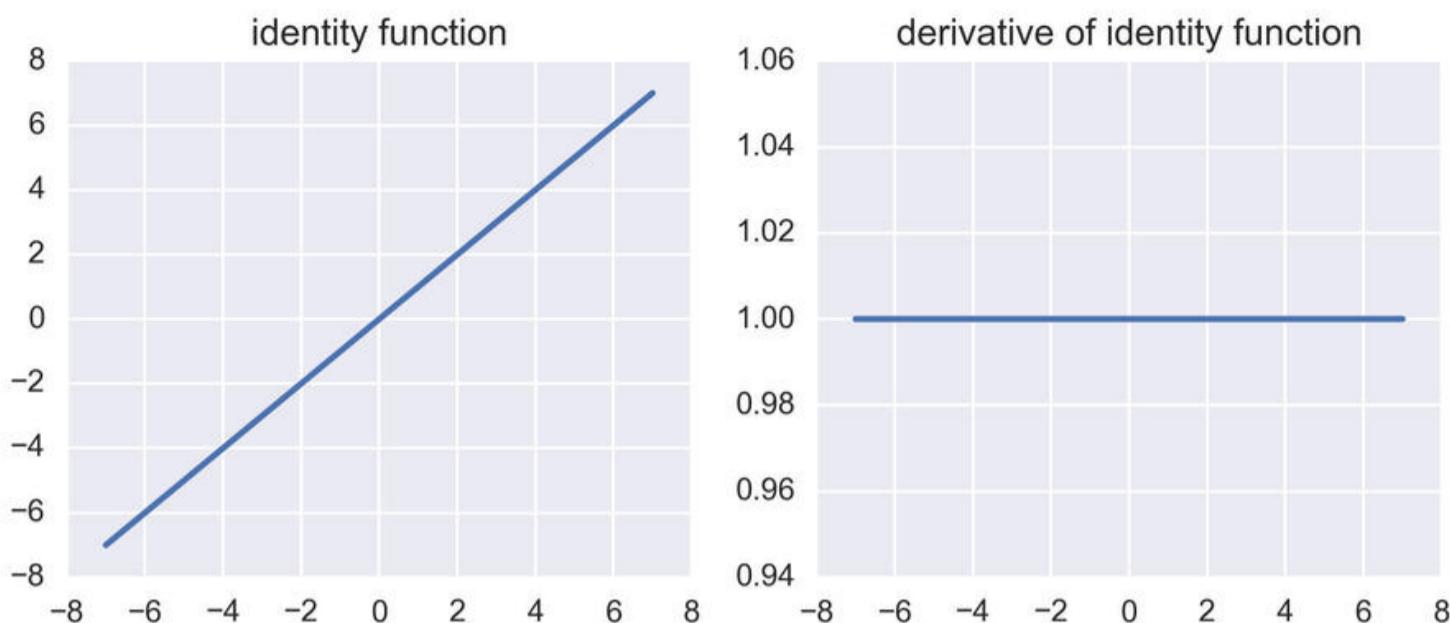


Figure 18.44: The identity function as activation function. Left: The identity produces as output the same value it received as input. Right: Its derivative is 1 everywhere, because the function has a constant slope of 1.

As Figure 18.44 shows, the derivative of the identity activation function is always 1 (that's because the function is a straight line with slope of 1 everywhere, and the derivative at any point is just the slope of the curve at that point). Let's think back on our discussion of backpropagation and include this identity activation within every neuron. During the forward pass, the outputs would be unchanged by this function, so it has no effect. During the backward pass, we'd always multiply the summed deltas by 1, again having no effect.

We said earlier that our results weren't limited to using the sigmoid. That's because we didn't use any special properties of sigmoid in our discussion, other than to assume it has a derivative everywhere. This is why activation functions are designed so that they have a derivative for every value (recall from Chapter 5 that library routines automatically apply mathematical techniques to take care of any spots that don't have a derivative).

Let's look at the popular ReLU activation function. Figure 18.45 shows the ReLU function and its derivative.

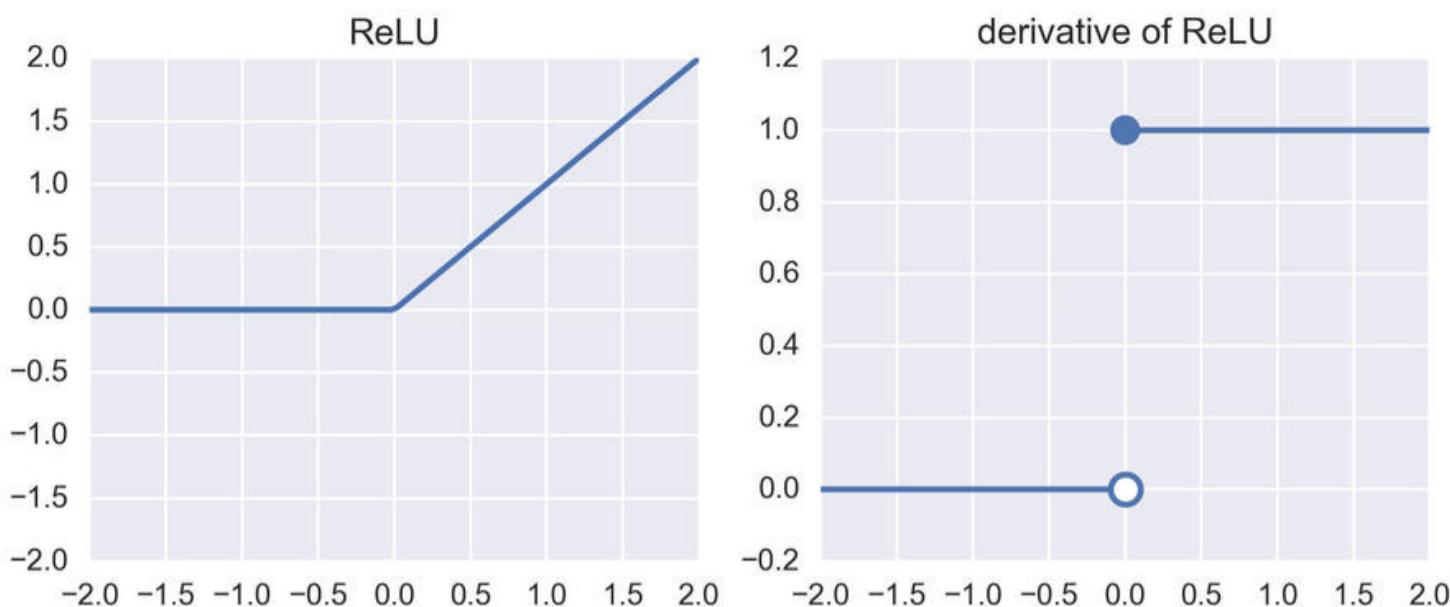


Figure 18.45: The ReLU function as an activation function. Left: ReLU returns its input when that value is 0 or larger, and otherwise returns 0. Right: Its derivative is a step function, 0 for inputs less than 0 and 1 for inputs greater than 0. The sudden jump at 0 is automatically managed for us by libraries so that we have a smooth derivative everywhere.

Everything we did with the sigmoid can be applied to the ReLU, without change. And the same goes for any other activation function.

That wraps things up. We've now put activation functions back into our neurons.

That brings us to the end of basic backpropagation.

Before we leave the discussion, though, let's look at a critical control that keeps things working well: the learning rate.

18.11 The Learning Rate

In our description of updating a weight, we multiplied the left neuron's output value and the right neuron's delta, and subtracted that from the weight (recall Figure 18.25 from much earlier).

But as we've mentioned a few times, changing a weight by a lot in a single step is often a recipe for trouble. The derivative is only accurate for very tiny changes in a 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, slowing everything down. Still, that inefficiency is usually better than a system that's constantly over-reacting to errors.

In practice, we control the amount of change to the weights during every update with a hyperparameter called the **learning rate**, often symbolized by the lower-case Greek letter η (eta). This is a number between 0 and 1, and it tells the weights how much of their newly-computed value to use when they update.

When we set the learning rate to 0, the weights don't change at all. Our system will never change and never learn. If we set the learning rate to 1, the system will apply big changes to the weights, and might overshoot the mark. 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. So we usually set the learning rate somewhere between these extremes.

Figure 18.46 shows how the learning rate is applied. We just scale the value of $-(A_o \times C\delta)$ by η before adding it back in to AC .

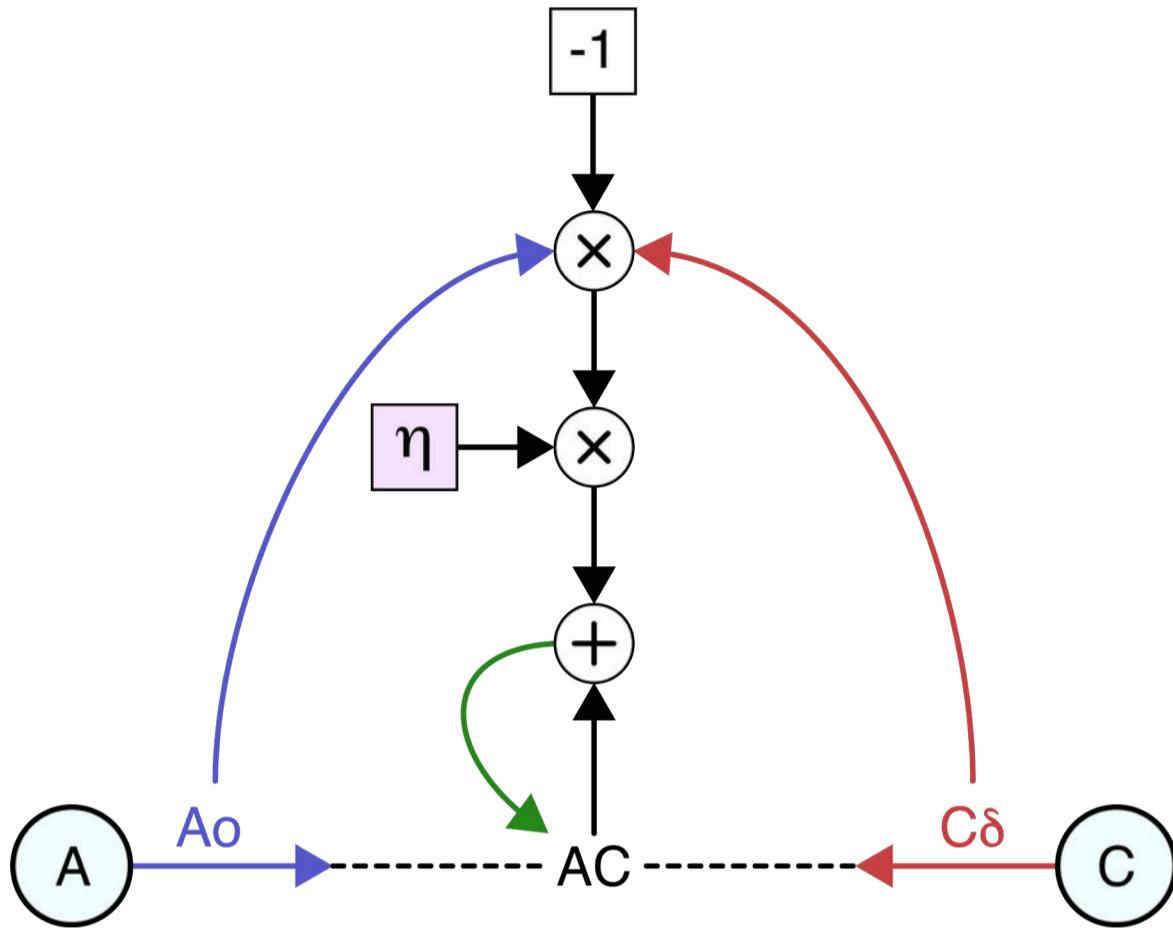


Figure 18.46: The learning rate helps us control how fast the network learns by controlling the amount by which weights change on each update. Here we see Figure 18.25 with an extra step that multiplies the value $-(A_o \times C\delta)$ by the learning rate δ before adding it to AC . When δ is a small positive number (say 0.01), then each change will be small, which often helps the network learn.

The best choice of 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 properly 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 eta using trial and error. Happily, there are algorithms that automate the search for a good starting value for the learning rate, and other algorithms that fine-tune the learning rate as learning progresses. We'll see such algorithms in Chapter 19. 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 0.01 and then train the network for a while, watching how well it learns. Then we then raise or lower it from there and train again, over and over, hunting for the value that learns most efficiently.

18.11.1 Exploring the Learning Rate

Let's see how backprop performs with different learning rates. We'll build a classifier to find the boundary between the two half-moons that we used in Chapter 15. Figure 18.47 shows our training data of about 1500 points. We pre-processed this data to give it zero mean and a standard deviation of 1 for each feature.

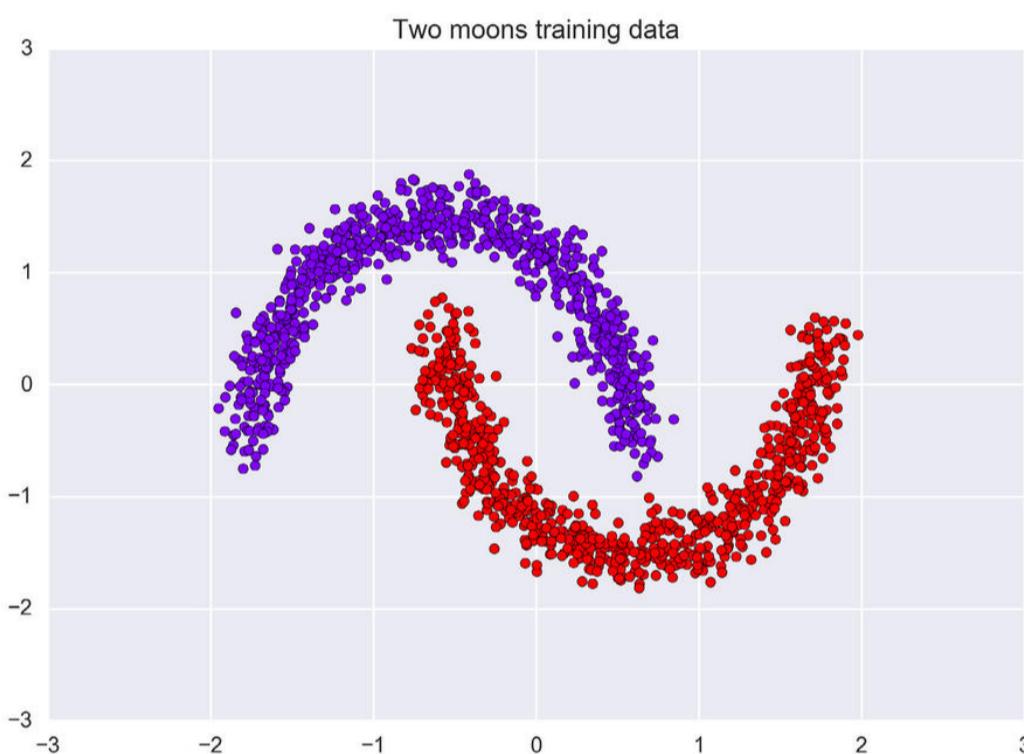


Figure 18.47: About 1500 points generated synthetically by the `make_moons()` routine in scikit-learn.

Because we have only two categories, we'll build a **binary classifier**. This lets us skip the whole one-hot encoding of labels and dealing with multiple outputs, and instead 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 2 hidden layers, each with 4 neurons. These are essentially arbitrary choices that give us a network that's just complex enough for our discussion. Both layers will be fully-connected, so every neuron in the first hidden layer will send its output to every neuron in the second hidden layer. Figure 18.48 shows the idea.

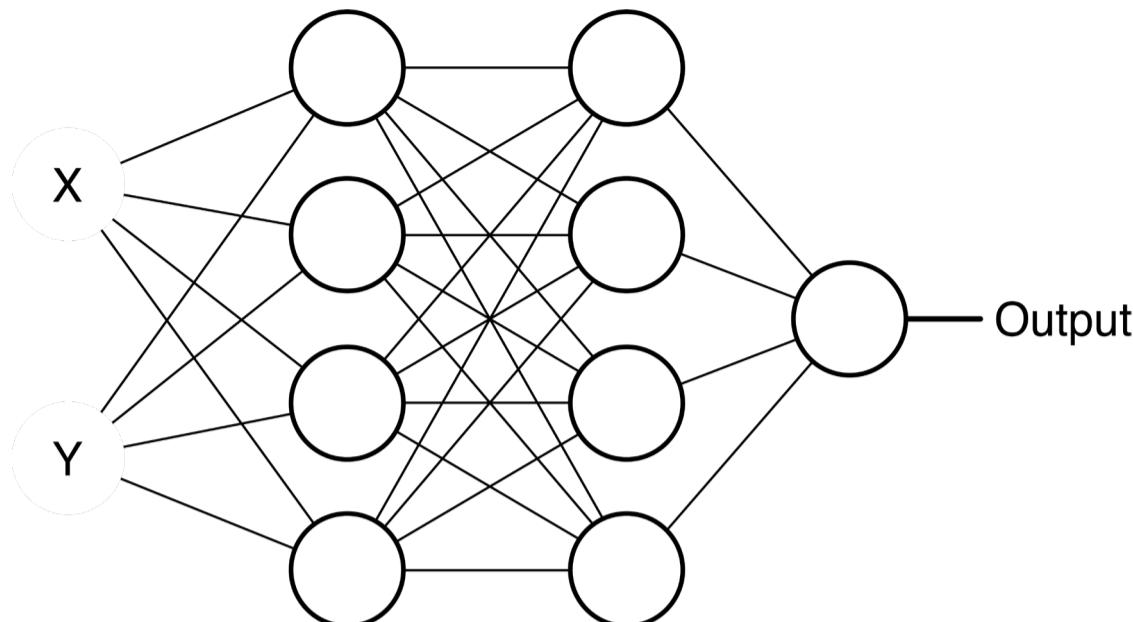


Figure 18.48: Our binary classifier takes in two values as input (the X and Y of each point). Each input goes into the 4 neurons on the first layer. Each of those 4 neurons connects to each of the 4 neurons on the next hidden layer. Then a single neuron takes the outputs of the second hidden layer and presents a single value for output. In this network, we've used ReLU activation functions for the neurons in the hidden layers, and a sigmoid activation function on the output neuron.

How many weights are in our network? There are 4 coming out of each of the 2 inputs, then 4×4 between the layers, and then 4 going into the output neuron. That gives us $(2 \times 4) + (4 \times 4) + 4 = 28$. Each of the 9 neurons also has a bias term, so our network has $28 + 9 = 37$ weights. They start with 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 above, we'll evaluate one sample, calculate the error, compute the deltas with backprop, and then update the weights using the learning rate. Then we'll move on to the next sample. Note that if the error is 0, then the weights won't change at all. Each time we process all the samples in the training set, we say we've completed one **epoch** of training.

Our discussion of backprop mentioned how much we rely on making “small changes” to the weights. There are two reasons for this. The first is that the direction of change for every weight is given by the derivative

(or gradient) of the error at that weight. But as we saw, 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 will cause changes in the outputs of neurons later on, and we've seen that we use those neuron outputs to help compute the changes to the later weights. To prevent everything from turning into a terrible snarl of conflicting moves, we change the weights only by small amounts.

But what is “small”? For every network and data set, we have to experiment to find out. As we saw above, the size of our step is controlled by the **learning rate**, or **eta** (η). The bigger this value, the more each weight will move towards its new value.

Let's start with a really large learning rate of 0.5. Figure 18.49 shows the boundaries computed by our network for our test data.

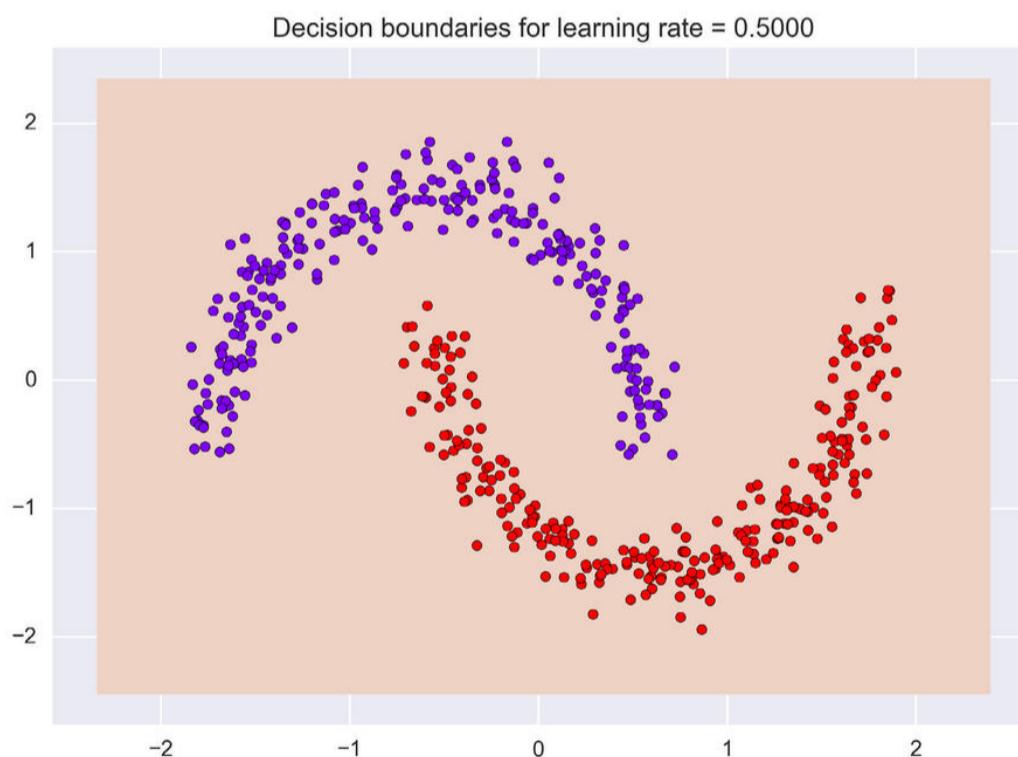


Figure 18.49: The boundaries computed by our network using a learning rate of 0.5.

This is terrible. 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 18.50.

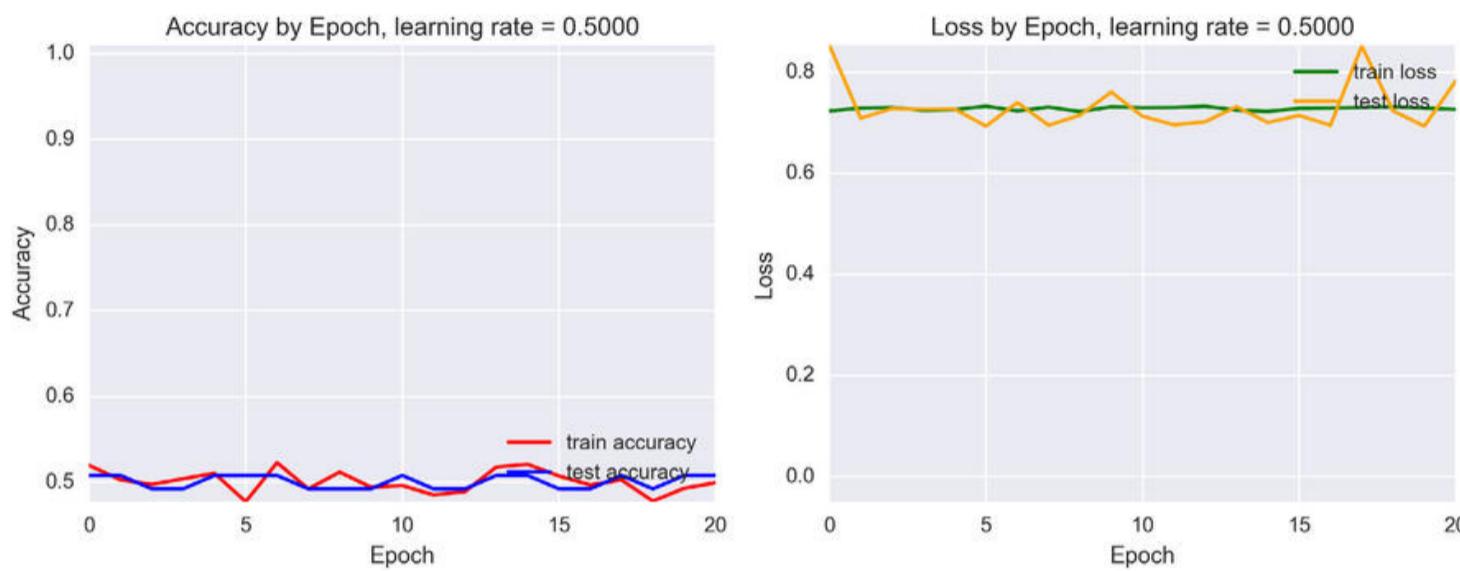


Figure 18.50: Accuracy and loss for our moons data with a learning rate of 0.5.

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 category, 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 18.51 shows the values of all 37 weights during training.

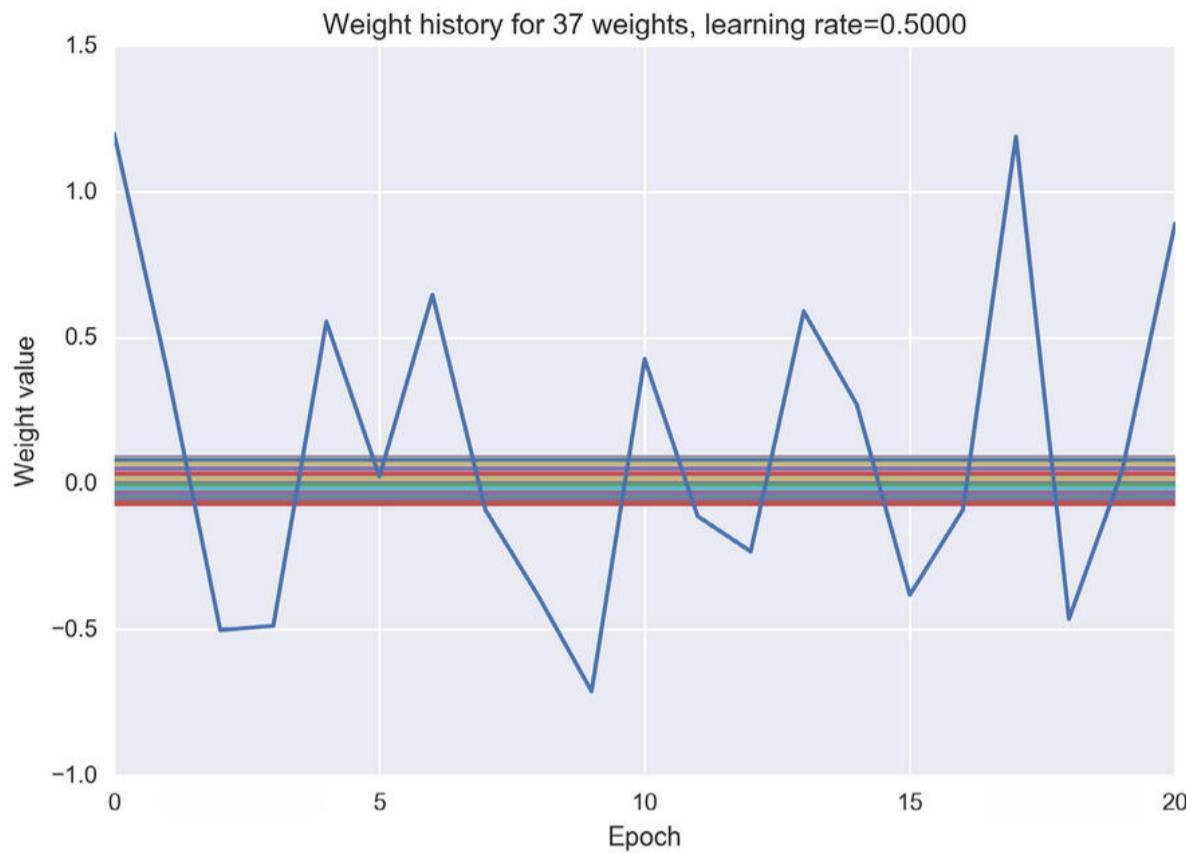


Figure 18.51: The weights of our network when using a learning rate of 0.5. One weight is constantly changing and overshooting its goal, while the others aren't making any visible changes.

Most of the weights don't seem to be moving at all, but they could be meandering a little bit. 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 every time, then over-correcting by too much, then over-correcting for that, and so on.

These results are disappointing, but they're not shocking, because a learning rate of 0.5 is *big*.

Let's reduce the training rate by a factor of 10 to a more common value of 0.05. We'll change absolutely nothing else about the network and the data, and we'll even re-use the same sequence of pseudo-random numbers to initialize the weights. The new boundaries are shown in Figure 18.52.

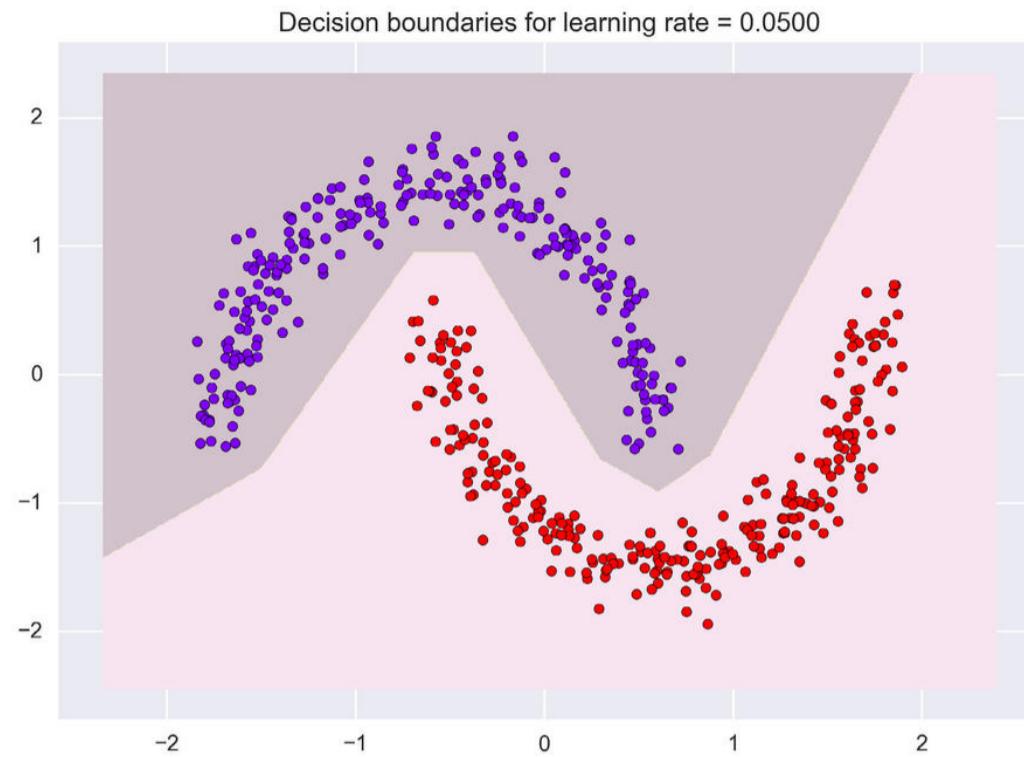


Figure 18.52: The decision boundaries when we use a learning rate of 0.05.

This is *much* better! This is great! Looking at the graphs in Figure 18.53 reveals that we've reached 100% accuracy on both the training and test sets after about 16 epochs.

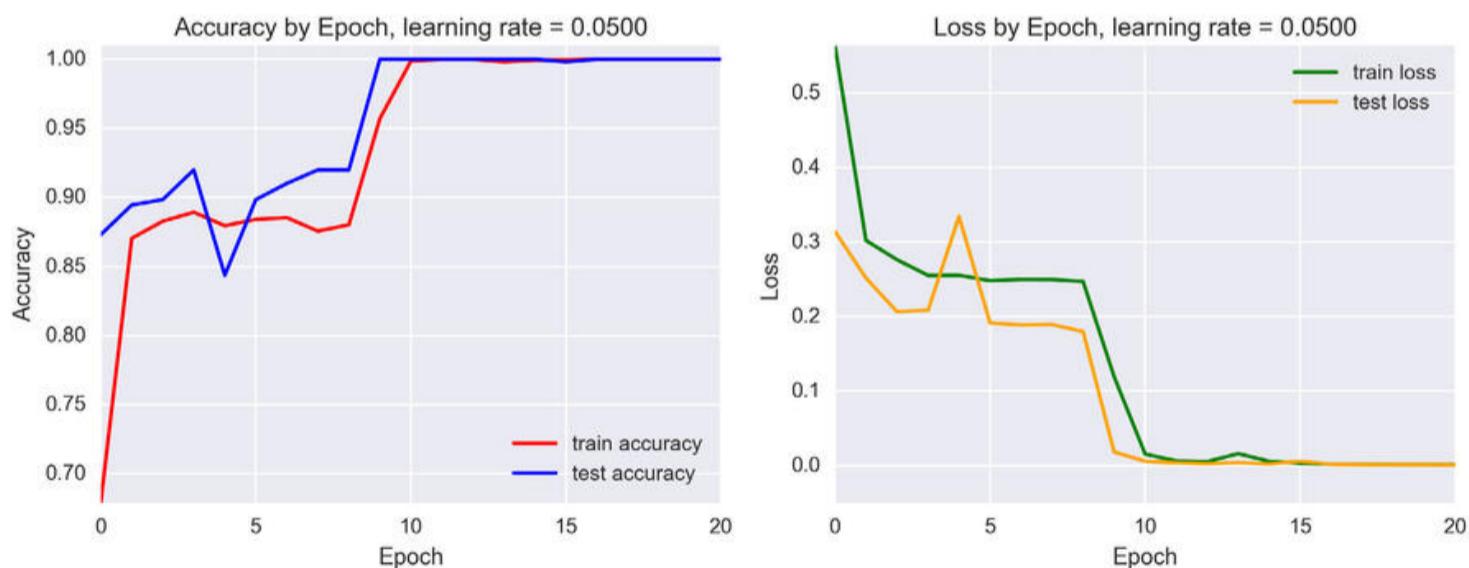


Figure 18.53: Accuracy and loss for our network when using a learning rate of 0.05.

What are the weights doing? Figure 18.54 shows us their history. Overall, this is way better, because lots of weights are changing. Some weights are getting pretty large. In Chapter 20 we'll cover regularization

techniques to keep the weights small in deep networks, and later in this chapter we'll see why it's nice to keep the weights small (say, between -1 and 1). For the moment, let's just note that the weights have each learned a good value.

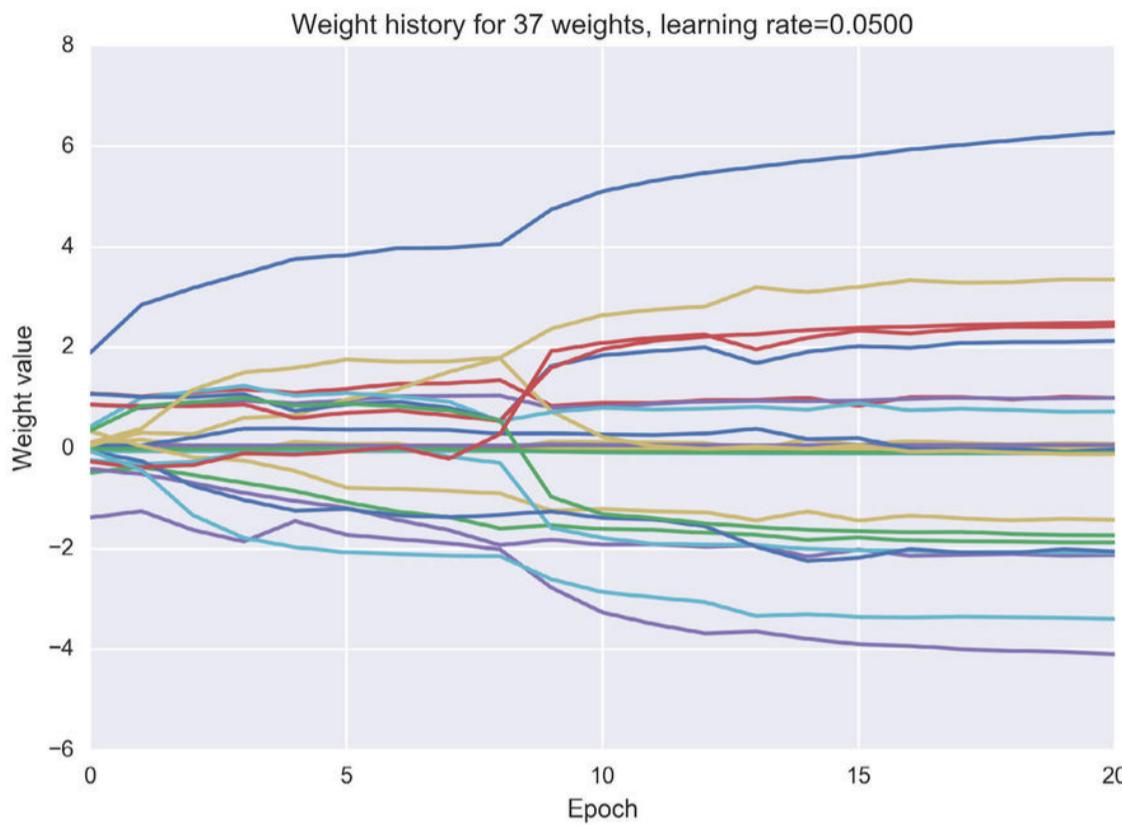


Figure 18.54: The weights in our network over time, using a learning rate of 0.05.

So that's success. Our network has learned to perfectly sort the data, and it did it in only 16 epochs, which is nice and fast. On a late 2014 iMac without GPU support, the whole training process took less than 10 seconds.

Just for fun, let's lower the learning rate down to 0.01. Now the weights will change even more slowly. Does this produce better results?

Figure 18.55 shows the decision boundary resulting from these tiny steps. The boundary seems to use more straight lines than the boundary in Figure 18.52, but both boundaries separate the sets perfectly. We might prefer the boundaries of Figure 18.52 in some situations, as they seem to better follow the shape of the data.

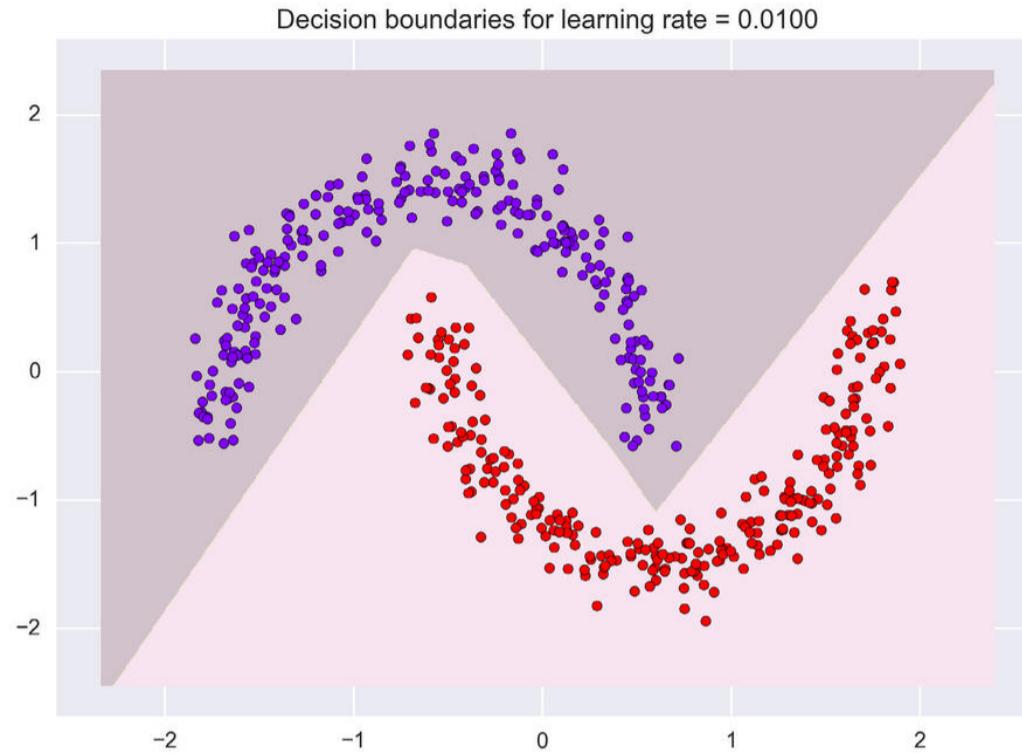


Figure 18.55: The decision boundaries for a learning rate of 0.01.

Figure 18.56 show our accuracy and loss graphs. Because our learning rate is so much slower, our network takes around 170 epochs to get to 100% accuracy, rather than the 16 in Figure 18.54.

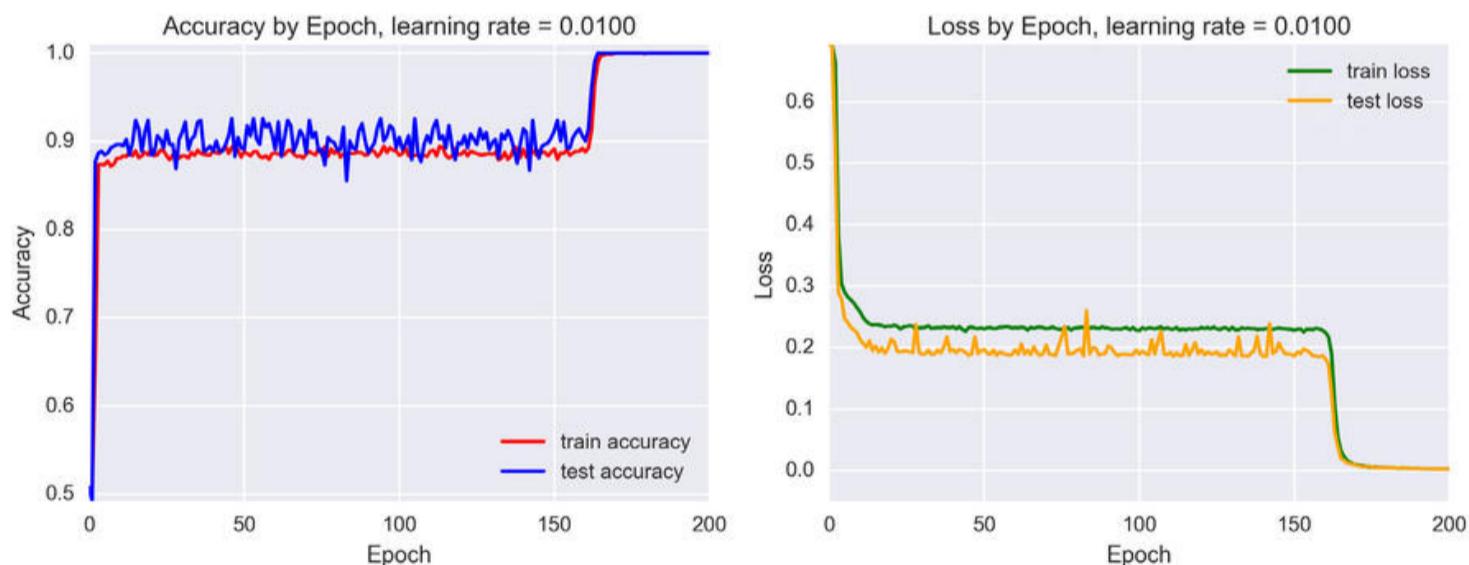


Figure 18.56: The accuracy and learning rate for our network using a learning rate of 0.01.

These graphs show an interesting learning behavior. After an initial jump, both the training and test accuracies reach about 90% and plateau there. At the same time, the losses hit about 0.2 (for the test data)

and 0.25 (for the training), and they plateau as well. Then around epoch 170, things improve rapidly again, with the accuracy climbing to 100% and the errors dropping to 0.

This pattern of alternating improvement and plateaus is not unusual, and we can even see a hint of it in Figure 18.53 where there's an imperfect plateau 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 local minima, it's more common for them to get caught in a flat region of a saddle, like those we saw in Chapter 5 [Dauphin14]. Sometimes it takes a long time for one of the weights to move into a region where the gradient (or derivative) 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 first 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 18.57. The interesting thing is that at least some of the weights are not flat, or on a plateau. They're changing, but very 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.

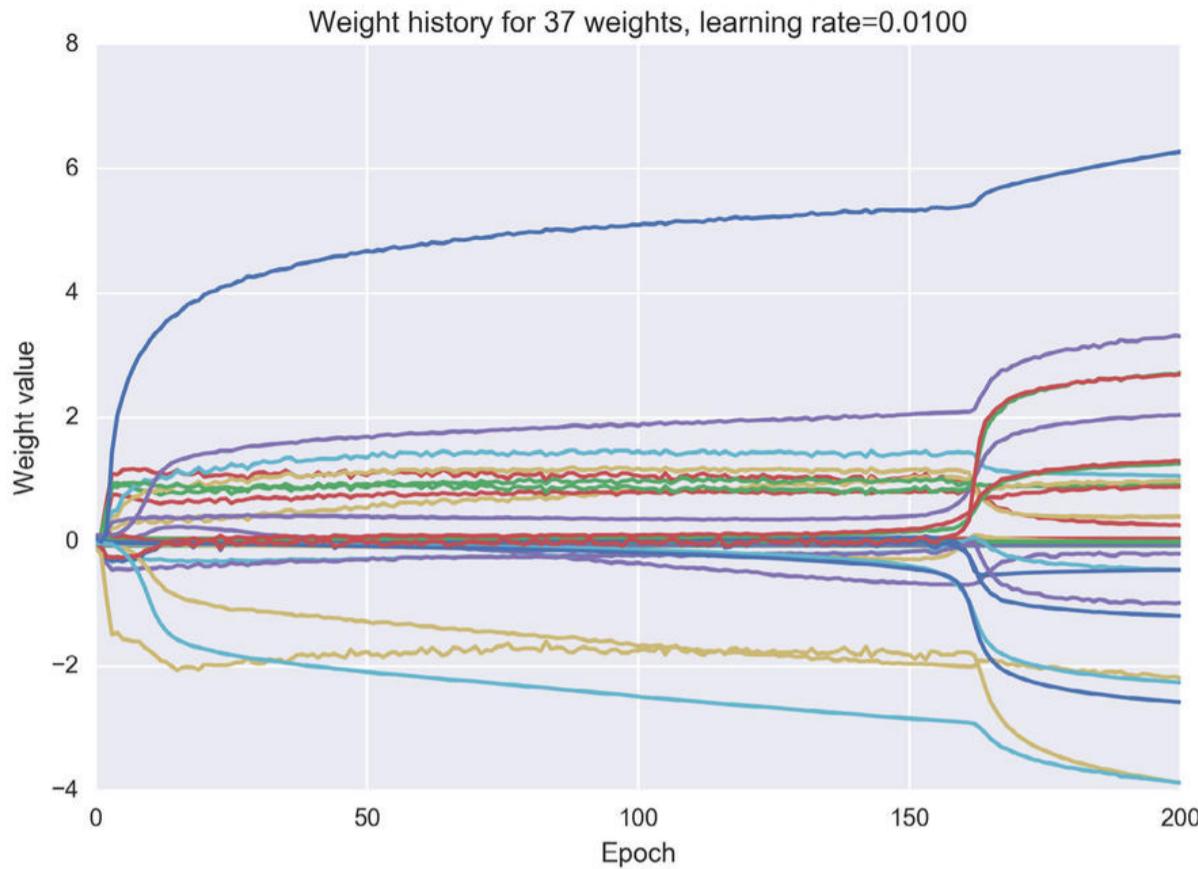


Figure 18.57: The history of our weights using a learning rate of 0.01.

The weights seem to be growing a little bit even at epoch 200. If we let the system continue, these weights will continue to slowly grow over time with no apparent effect on the accuracies or errors.

So was there any benefit to lowering the learning rate down to 0.01? Not really. Even at 0.05, the categorization was already perfect on both the training and test data. In this case, 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.

When we look for the best value for the learning rate, it can feel like we're the character of Goldilocks in recent versions of the fable *Goldilocks and the Three Bears* [Wikipedia17]. We're searching for something that's not too big, and not too small, but "just right."

When our learning rate was too big, the weights took steps that were too large, and the network never settled down or improved its performance. When the learning rate was too small, our progress was very slow, as the weights sometimes were creeping from one value to another at a glacial pace.

When the learning rate was just right, training was fast and efficient, and produced great results. In this case, they were perfect.

This kind of experimenting with the learning rate is part of developing nearly every deep learning network. The speed with which backprop changes the weights needs to be tuned to match the type of network and the type of data. Happily, we'll see in Chapter 19 that there are automatic tools that can handle the learning rate for us in sophisticated ways.

18.12 Discussion

Let's recap what we've seen, and then consider some of the implications of the backpropagation algorithm.

18.12.1 Backprop In One Place

To recap quickly, we start by running a sample through the network and calculate the output for every neuron, as in Figure 18.58(a).

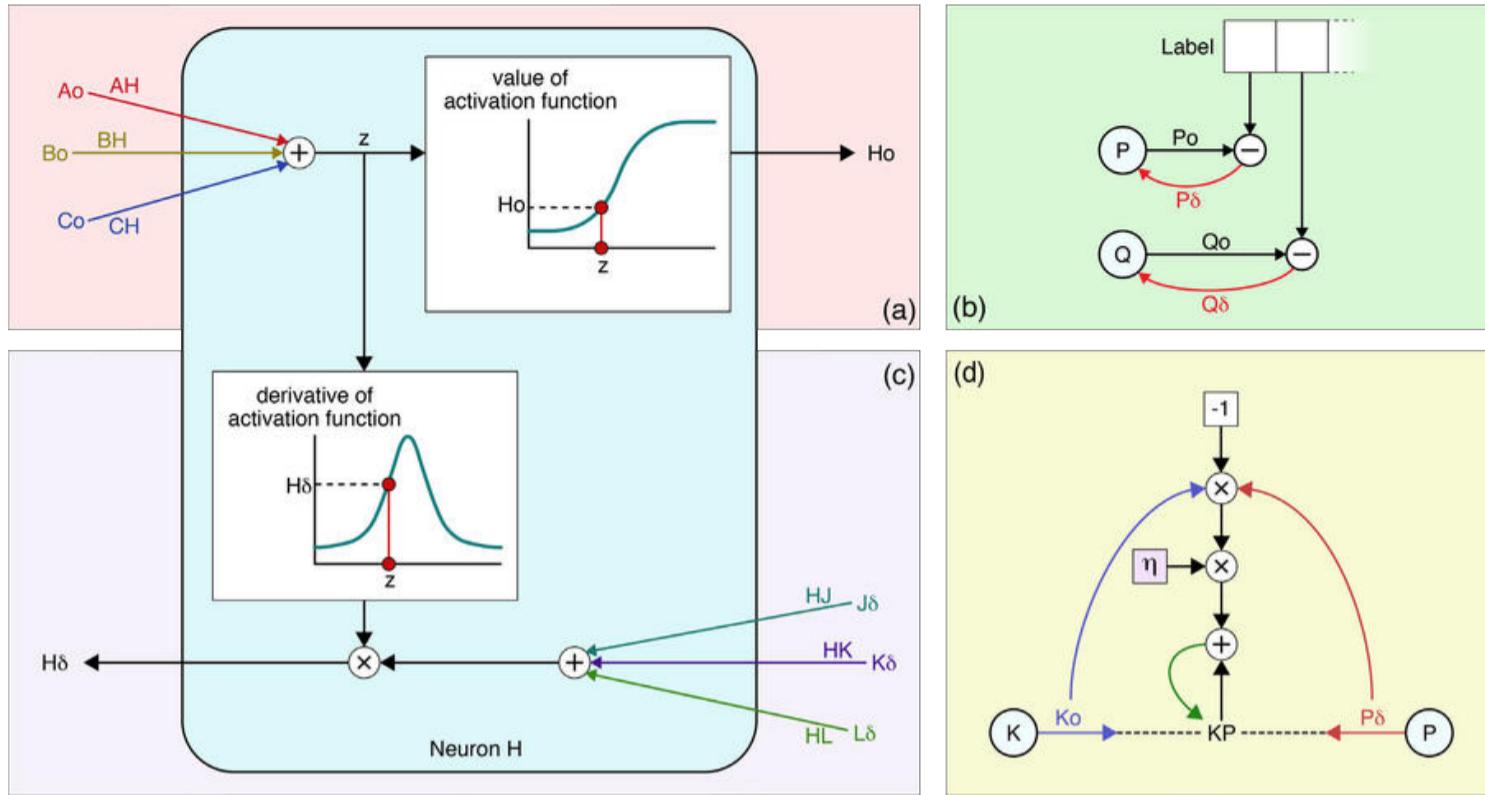


Figure 18.58: The backprop algorithm, along with weight update, in a nutshell. This figure collects Figure 18.20, Figure 18.43, and Figure 18.46 in one place. Part (a) shows the forward step, part (b) shows finding the deltas for the output neurons, part (c) steps propagate the deltas backwards, and step (d) updates the weights.

Then we kick off the backprop algorithm in Figure 18.58(b). We find the delta value for each of the output neurons, telling us that if the neuron’s output changes by a certain amount, the error will change by that amount times the neuron’s delta.

Now we take a step backwards in Figure 18.58(c) to the previous layer of the network, and find the deltas for all the neurons in that layer. We need only the deltas from the output layer, and the weights between the two layers.

Once all the deltas are assigned, we update the weights. Using the deltas and neuron outputs, we compute an adjustment to every weight. We scale that adjustment by the learning rate, and then add it to the current value of the weight to get the new, updated value of the weight, as in in Figure 18.58(d).

18.12.2 What Backprop Doesn't Do

There's a shorthand in some discussions of backprop that can be confusing. Authors sometimes say something like, "backpropagation moves the error backwards from the output layer to the input layer," or "backpropagation uses the error at each layer to find the error at the layer before."

This can be misleading because backprop is *not* "moving the error" at all. In fact, the only time we use "the error" is at the very start of the process when we find the delta values for the output neurons.

What's really going on involves the *change* in the error, which is represented by the delta values. These act as amplifiers of change in the neuron outputs, telling us that if an output value or weight changes by a given amount, we can predict the corresponding change in the error.

So backprop isn't moving "the error" backwards. But *something* is moving backwards. Let's see what that is.

18.12.3 What Backprop Does Do

The first step in the backprop process is to find the deltas for the output neurons. These are found from the **gradient** of the error. We sliced the gradient to get a look at the 2D curve for each prediction, and then we took the derivative of that curve, but that was just for ease of visualization and discussion. The derivatives are just pieces of what really matters: the gradient.

As we work our way backwards, the deltas continue to represent gradients. Every delta value represents a different gradient. For instance, the delta attached to a neuron C describes the gradient of the error with respect to the output of C, and the delta for A describes the gradient of the error with respect to changes in the output of A.

So when we change the weights, we're changing them in order to follow the gradient of the error. This is an example of **gradient descent**, which mimics the path that water takes as it runs downhill on a landscape.

We can say that backpropagation is an algorithm that lets us efficiently update our weights using gradient descent, since the deltas it computes describe that gradient.

So a nice way to summarize backprop is to say that it moves the gradient of the error backwards, modifying it to account for each neuron's contribution.

18.12.4 Keeping Neurons Happy

When we put activation functions back into the backprop algorithm, we concentrated on the region where the inputs are near 0.

That wasn't an accident. Let's return to the sigmoid function, and look at what happens when the value into the activation function (that is, the sum of the weighted inputs) becomes a very large number, say 10 or more. Figure 18.59 shows the sigmoid between values of -20 and 20, along with its derivative in the same range.

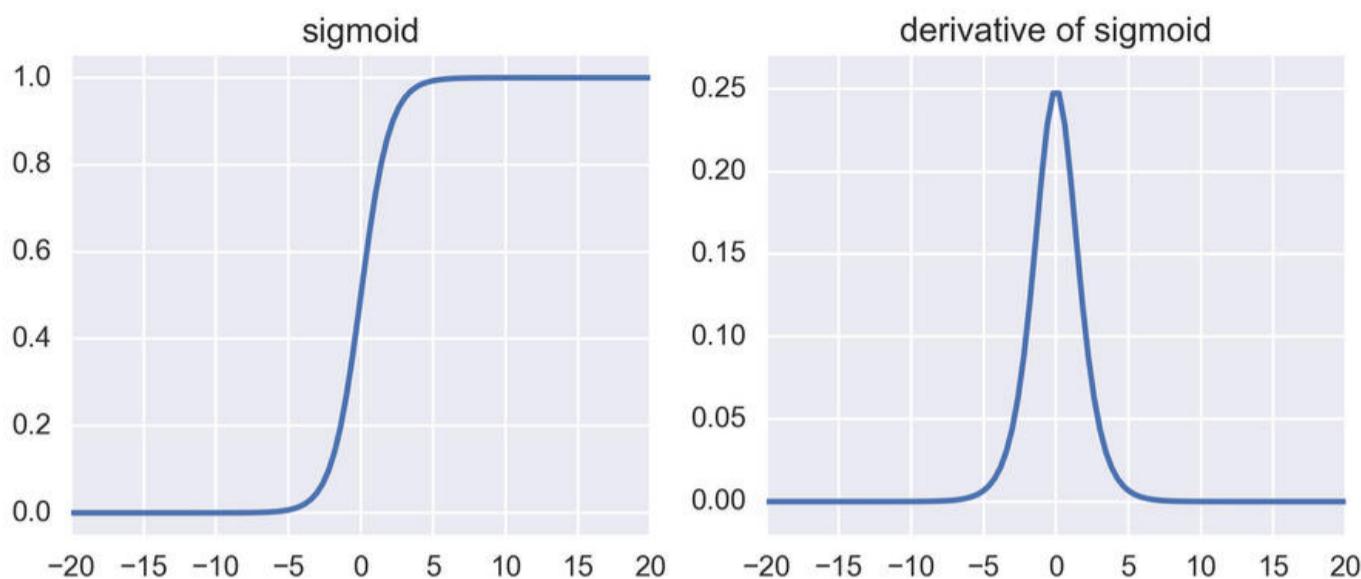


Figure 18.59: The sigmoid function becomes flat for very large positive and negative values. Left: The sigmoid for the range -20 to 20. Right: The derivative of the sigmoid in the same range. Note that the vertical scales of the two plots are different.

The sigmoid never quite reaches exactly 1 or 0 at either end, but it gets extremely close. Similarly, the value of the derivative never quite reaches 0, but as we can see from the graph it gets very close.

Figure 18.60 shows a neuron with a sigmoid activation function. The value going into the function, which we've labeled z , has the value 10, putting it in one of the curve's flat regions.

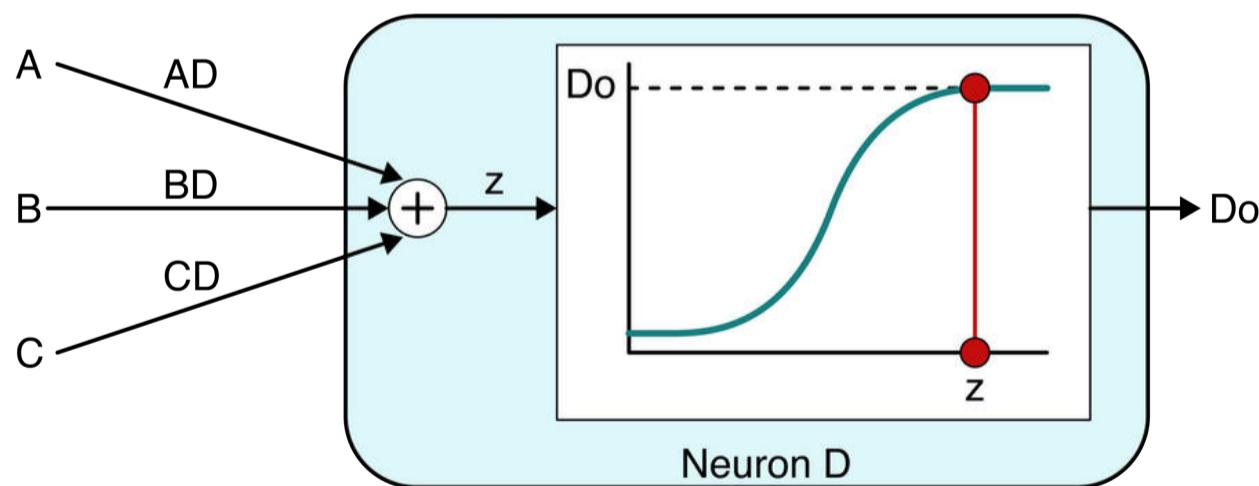


Figure 18.60: When we apply a large value (say 10) to the sigmoid, we find ourselves in a flat region and get back the value of 1.

From Figure 18.59, we can see that the output is basically 1.

Now suppose that we change one of the weights, as shown in Figure 18.61. The value z increases, so we move right on the activation function curve to find our output. Let's say that the new value of z is 15. The output is still basically 1.

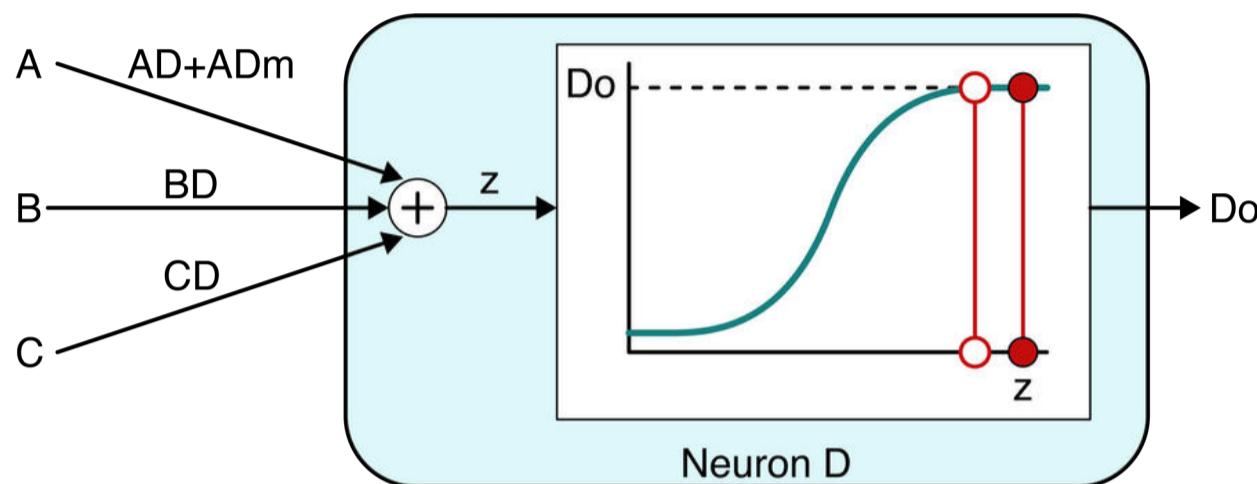


Figure 18.61: A big increase to the weight AD coming into this neuron has no effect on its output, because it just pushes us further to the right along the flat region of the sigmoid. The neuron's output was 1 before we added ADm to the weight AD , and it's still 1 afterwards.

If we increase the value of the incoming weight again, even by a lot, we'll still get back an output of 1. In other words, *changing the incoming weight has no effect on the output*. And because the output doesn't change, the error doesn't change.

We could have predicted this from the derivative in Figure 18.59. When the input is 15, the derivative is 0 (actually about 0.0000003, but our convention above says that we can call that 0). So changing the input will result no change in the output.

This is a terrible situation for any kind of learning, because we've lost the ability to improve the network by adjusting this weight. In fact, *none* of the weights coming into this neuron matter anymore (if we keep the changes small), because any changes to the weighted sum of the inputs, whether they make the sum smaller or larger, still lands us on a flat part of the function and thus there's no change to the output, and no change to the error.

The same problem holds if the input value is very negative, say less than -10 . The sigmoid curve is flat in that region also, and the derivative is also essentially zero.

In both of these cases we say that this neuron has **saturated**. Like a sponge that cannot hold any more water, this neuron cannot hold any more input. The output is 1 , and unless the weights, the incoming values, or both, move a lot closer to 0 , it's going to stay at 1 .

The result is that this neuron no longer participates in learning, which is a blow to our system. If this happens to enough neurons, the system could become crippled, learning more slowly than it should, or perhaps even not at all.

A popular way to prevent this problem is to use **regularization**. Recall from Chapter 9 that the goal of regularization is to keep the sizes of the weights small, or close to 0 . Among other benefits, this has the value of keeping the sum of the weighted inputs for each neuron also small and close to zero, which puts us in the nice S-shaped part of the activation function. This is where learning happens. In Chapters 23 and 24 we'll see techniques for regularization in deep learning networks.

Saturation can happen with any activation function where the output curve becomes flat for a while (or forever).

Other activation functions can have their own problems. Consider the popular ReLU curve, plotted from -20 to 20 in Figure 18.62.

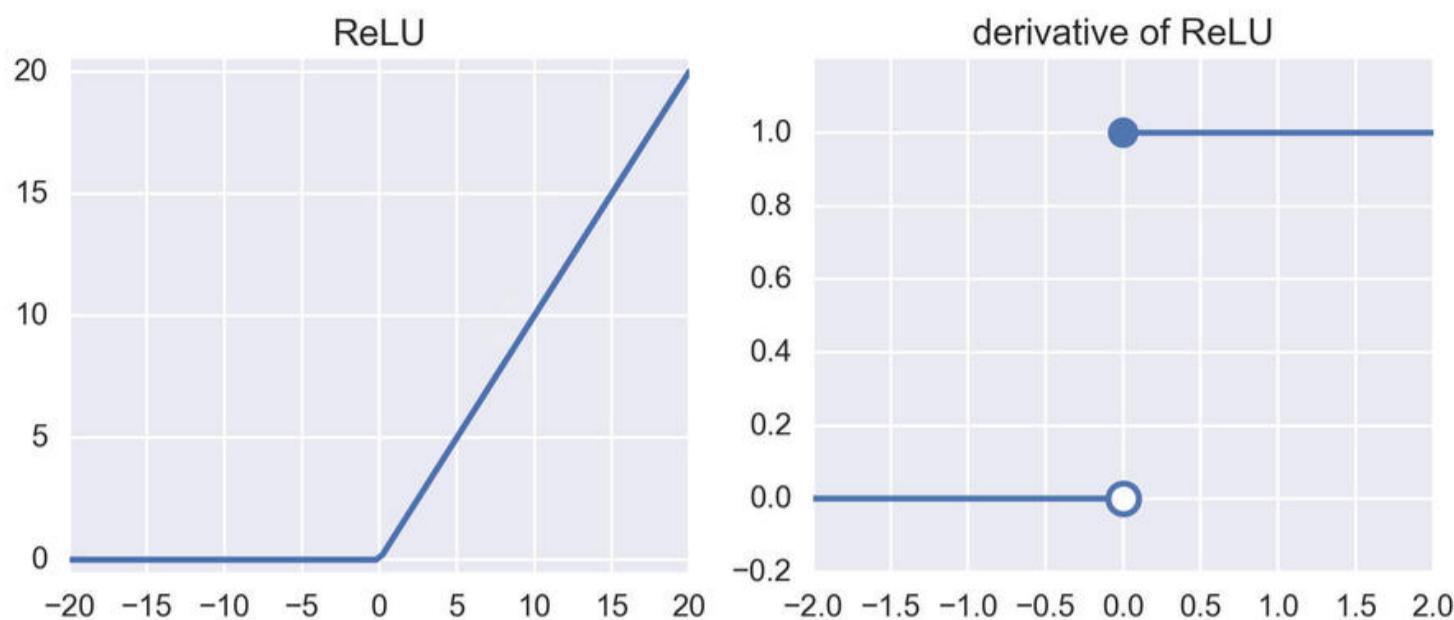


Figure 18.62: The ReLU activation function in the range -20 to 20 . Positive values won't saturate the function, but negative values can cause it to die. Left: The ReLU function. Right: The derivative of ReLU.

As long as the input is positive, this function won't saturate, because the output is the same as the input. The derivative for positive inputs is 1, so the sum of the weighted inputs will be passed directly to the output without change.

But when the input is negative, the function's output is 0, and the derivative is 0 as well. Not only do changes make no difference to the output, but the output itself has ceased to make any contribution to the error. The neuron's output is 0 and unless the weights, inputs, or both change by a lot, it's going to stay at 0.

To characterize this dramatic effect, we say that this neuron has **died**, or is now **dead**.

Depending on the initial weights and the first input sample, one or more neurons could die the very first time we perform an update step. Then as training goes on, more neurons can die.

If a lot of neurons die during training, then our network is suddenly working with just a fraction of the neurons we thought it had. That cripples our network. Sometimes even 40% of our neurons can die off during training [Karpathy16].

When we build a neural network we choose the activation functions for each layer based on experience and expectations. In many situations, sigmoids or ReLUs feel like the right function to use, and in many circumstances they work great. But when a network learns slowly, or fails to learn, it pays to look at the neurons and see if some or many are saturated, dying, or dead. If so, we can experiment with our initial starting weights and our learning rate to see if we can avoid the problem. If that doesn't work, we might need to re-structure our network, choose other activation functions, or both.

18.12.5 Mini-Batches

In our discussion above, we followed three steps for every sample: run the sample through the network, calculate all the deltas, and then adjust all the weights.

It turns out that we can save some time, and sometimes even improve our learning, by only adjusting the weights infrequently.

Recall from Chapter 8 that the full training set of samples is sometimes called a **batch** of samples. We can break up that batch into smaller **mini-batches**. Usually the size of our mini-batch is picked to match whatever parallel hardware we have available. For instance, if our hardware (say a GPU) can evaluate 16 samples simultaneously, then our mini-batch size will be 16. Common mini-batch sizes are 16, 32, and 64, though they can go higher.

The idea is that we run a mini-batch of samples through the network in parallel, and then we compute all the deltas in parallel. We'll average together all the deltas, and use those averages to then perform a single update to the weights. So instead of updating the weights after every sample, they're updated after a mini-batch of 16 samples (or 32, 64, etc.).

This gives us a big increase in speed. It can also improve learning, because the changes to the weights are smoothed out a little by the averaging over the whole mini-batch. This means if there's one weird

sample in the set, it can't pull all the weights in an unwanted direction. The deltas for that weird sample get averaged with the other 31 or 63 samples in the mini-batch, reducing its impact.

18.12.6 Parallel Updates

Since each weight depends only on values from the neurons at its two ends, every weight's update step is completely independent from every other weight's update step.

When we carry out the same steps for independent pieces of data, that's usually our cue to use parallel processing.

And indeed, most modern implementations will, if parallel hardware is available, update all the weights in the network simultaneously. As we just discussed, this update will usually happen after each mini-batch of samples.

This is an enormous time-saver, but it comes at a cost. As we've discussed, changing any weight in the network will change the output value for every neuron that's downstream from that weight. So changes to the weights near the very start of the network can have enormous ripple effects on later neurons, causing them to change their outputs by a lot.

Since the gradients represented by our deltas depend on the values in the network, changing a weight near the input means that we should really re-compute all the deltas for all the neurons that consume that value that weight modifies. That could mean almost every neuron in the network.

This would destroy our ability to update in parallel. It would also make backprop agonizingly slow, since we'd be spending all of our time re-evaluating gradients and computing deltas.

As we've seen, the way to prevent chaos is to use a "small enough" learning rate. If the learning rate is too large, things go haywire and don't settle. If it's too small, we waste a lot of time taking overly tiny

steps. Picking the “just right” value of the learning rate preserves the efficiency of backprop, and our ability to carry out its calculations in parallel.

18.12.7 Why Backprop Is Attractive

A big part of backprop’s appeal is that it’s so efficient. It’s the fastest way that anyone has thought of to figure out how to most beneficially update the weights in a neural network.

As we saw before, and summarized in Figure 18.43, running one step of backprop in a modern library usually takes about as long as evaluating a sample. In other words, consider the time it takes to start with new values in the inputs, and flow that data through the whole network and ultimately to the output layer. Running one step of backprop to compute all the resulting deltas takes about the same amount of time.

That remarkable fact is at the heart of why backprop has become a key workhorse of machine learning, even though we usually have to deal with issues like a fiddly learning rate, saturating neurons, and dying neurons.

18.12.8 Backprop Is Not Guaranteed

It’s important to note that there’s no guarantee that this scheme is going to learn anything! It’s not like the single perceptron of Chapter 10, where we have ironclad proofs that after enough steps, the perceptron will find the dividing line it’s looking for.

When we have many thousands of neurons, and potentially many millions of weights, the problem is too complicated to give a rigorous proof that things will always behave as we want.

In fact, things often do go wrong when we first try to train a new network. The network might learn glacially slowly, or even not at all. It might improve for a bit and then seem to suddenly take a wrong turn

and forget everything. All kinds of stuff can happen, which is why many modern libraries offer visualization tools for watching the performance of a network as it learns.

When things go wrong, the first thing many people try is to crank the learning rate to a very small value. If everything settles down, that's a good sign. If the system now appears to be learning, even if it's barely perceptible, that's another good sign. Then we can slowly increase the learning rate until it's learning as quickly as possible without succumbing to chaos.

If that doesn't work, then there might be a problem with the design of the network.

This is a complex problem to deal with. Designing a successful network means making a lot of good choices. For instance, we need to choose the number of layers, the number of neurons on each layer, how the neurons should be connected, what activation functions to use, what learning rate to use, and so on. Getting everything right can be challenging. We usually need a combination of experience, knowledge of our data, and experimentation to design a neural network that will not only learn, but do it efficiently.

In the following chapters we'll see some architectures that have proven to be good starting points for wide varieties of tasks. But each new combination of network and data is its own new thing, and requires thought and patience.

18.12.9 A Little History

When backpropagation was first described in the neural network literature in 1986 it completely changed how people thought about neural networks [Rumelhart86]. The explosion of research and practical benefits that followed were all made possible by this surprisingly efficient technique for finding gradients.

But this wasn't the first time that backprop had been discovered or used. This algorithm, which has been called one of the 30 "great numerical algorithms" [Trefethen15], has been discovered and re-discovered by different people in different fields since at least the 1960's. There are many disciplines that use connected networks of mathematical operations, and finding the derivatives and gradients of those operations at every step is a common and important problem. Clever people who tackled this problem have re-discovered backprop time and again, often giving it a new name each time.

Excellent capsule histories are available online and in print [Griewank12] [Schmidhuber15] [Kurenkov15] [Werbos96]. We'll summarize some of the common threads here. But history can only cover the published literature. There's no knowing how many people have discovered and re-discovered backprop, but didn't publish it.

Perhaps the earliest use of backprop in the form we know it today was published in 1970, when it was used for analyzing the accuracy of numerical calculations [Linnainmaa70], though there was no reference made to neural networks. The process of finding a derivative is sometimes called *differentiation*, so the technique was known as **reverse-mode automatic differentiation**.

It was independently discovered at about the same time by another researcher who was working in chemical engineering [Griewank12].

Perhaps its first explicit investigation for use in neural networks was made in 1974 [Werbos74], but because such ideas were out of fashion, that work wasn't published until 1982 [Schmidhuber15].

Reverse-mode automatic differentiation was used in various sciences for years. But when the classic 1986 paper re-discovered the idea and demonstrated its value to neural networks the idea immediately became a staple of the field under the name **backpropagation** [Rumelhart86].

Backpropagation is central to deep learning, and it forms the foundation for the techniques that we'll be considering in the remainder of this book.

18.12.10 Digging into the Math

This section offers some suggestions for dealing with the math of backpropagation. If you're not interested in that, you can safely skip this section.

Backpropagation is all about manipulations to numbers, hence its description as a “numerical algorithm.” That makes it a natural for presenting in a mathematical context.

Even when the equations are stripped down, they can appear formidable [Neilsen15b]. Here are a few hints for getting through the notation and into the heart of the matter.

First, it's essential to master each author's notation. There are a lot of things running around in backpropagation: errors, weights, activation functions, gradients, and so on. Everything will have a name, usually just a single letter. A good first step is to scan through the whole discussion quickly, and notice what names are given to what objects. It often helps to write these down so you don't have to search for their meanings later.

The next step is to work out how these names are used to refer to the different objects. For example, each weight might be written as something like w_{jk}^l , referring to the weight that links neuron number k on layer l to neuron j on layer $l+1$. This is a lot to pack into one symbol, and when there are several of these things in one equation it can get hard to sort out what's going on.

One way to clear the thickets is to choose values for all the subscripts, and then simplify the equations so each of these highly-indexed terms refers to just one specific thing (such as a single weight). If you think visually, consider drawing pictures showing just what objects are involved, and how their values are being used.

The heart of the backprop algorithm can be thought of, and written as, an application of the **chain rule** from calculus [Karpathy15]. This is an elegant way to describe the way different changes relate to one another, but it requires familiarity with multidimensional calculus. Luckily, there's a wealth of online tutorials and resources designed to help people come up to speed on just this topic [MathCentre09] [Khan13].

We've seen that in practice the computations for outputs, deltas, and weight updates can be performed in parallel. They can also be *written* in a parallel form using the linear algebra language of vectors and matrices. For example, it's common to write the heart of the forward pass (without each neuron's activation function) with a matrix representing the weights between two layers. Then we use that matrix to multiply a vector of the neuron outputs in the previous layer. In the same way, we can write the heart of the backward pass as the transpose of that weight matrix times a vector of the following layer's deltas.

This is a natural formalism, since these computations consist of lots of multiplies followed by additions, which is just what matrix multiplication does for us. And this structure fits nicely onto a GPU, so it's a nice place to start when writing code.

But this linear algebra formalism can obscure the relatively simple steps, because one now has to deal with not just the underlying computation, but its parallel structure in the matrix format, and the proliferation of indices that often comes along with it. We can say that compacting the equations in this form is a type of optimization, where we're aiming for simplicity in both the equations and the algorithms they describe. When learning backprop, people who aren't already very familiar with linear algebra can reasonably feel that this is a form

of *premature optimization*, because (until it is mastered) it obscures, rather than elucidates, the underlying mechanics [Hydeo9]. Arguably, only once the backprop algorithm is fully understood should it be rolled up into the more compact matrix form. Thus it may be helpful to either find a presentation that doesn't start with the matrix algebra approach, or try to pull those equations apart into individual operations, rather than big parallel multiplications of matrices and vectors.

Another potential hurdle is that the activation functions (and their derivatives) tend to get presented in different *ad hoc* ways.

To summarize, many authors start their discussions with either the chain rule or matrix forms of the basic equations, so that the equations appear tidy and compact. Then they explain why those equations are useful and correct. Such notation and equations can look daunting, but if we pull them apart to their basics we'll recognize the steps we saw in this chapter. Once we've unpacked these equations and then put them back together, we can see them as natural summaries of an elegant algorithm.

References

- [Dauphin14] Yann Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, Yoshua Bengio, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”, 2014. <http://arxiv.org/abs/1406.2572>
- [Fullér10] Robert Fullér, “The Delta Learning Rule Tutorial”, Institute for Advanced Management Systems Research, Department of Information Technologies, Åbo Adademi University, 2010. <http://uni-obuda.hu/users/fuller.robert/delta.pdf>
- [Griewank12] Andreas Griewank “Who Invented the Reverse Mode of Differentiation?”, Documenta Mathematica, Extra Volume ISMP 389–400, 2012 http://www.math.uiuc.edu/documenta/vol-ismp/52_griewank-andreas-b.pdf

[Hyde09] Randall Hyde, “The Fallacy of Premature Optimization,” ACM Ubiquity, 2009. <http://ubiquity.acm.org/article.cfm?id=1513451>

[Karpathy15] Andrej Karpathy, “Convolutional Neural Networks for Visual Recognition”, Stanford CS231n course notes, 2015. <http://cs231n.github.io/optimization-2/>

[Karpathy16] Andrej Karpathy, “Yes, You Should Understand Backprop”, Medium, 2016. <https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b>

[Khan13] Khan Academy, “Chain rule introduction”, 2013. <https://www.khanacademy.org/math/ap-calculus-ab/product-quotient-chain-rules-ab/chain-rule-ab/v/chain-rule-introduction>

[Kurenkov15] Andrey Kurenkov, “A ‘Brief’ History of Neural Nets and Deep Learning, Part 1”, 2015. <http://www.andreykurenkov.com/writing/a-brief-history-of-neural-nets-and-deep-learning/>

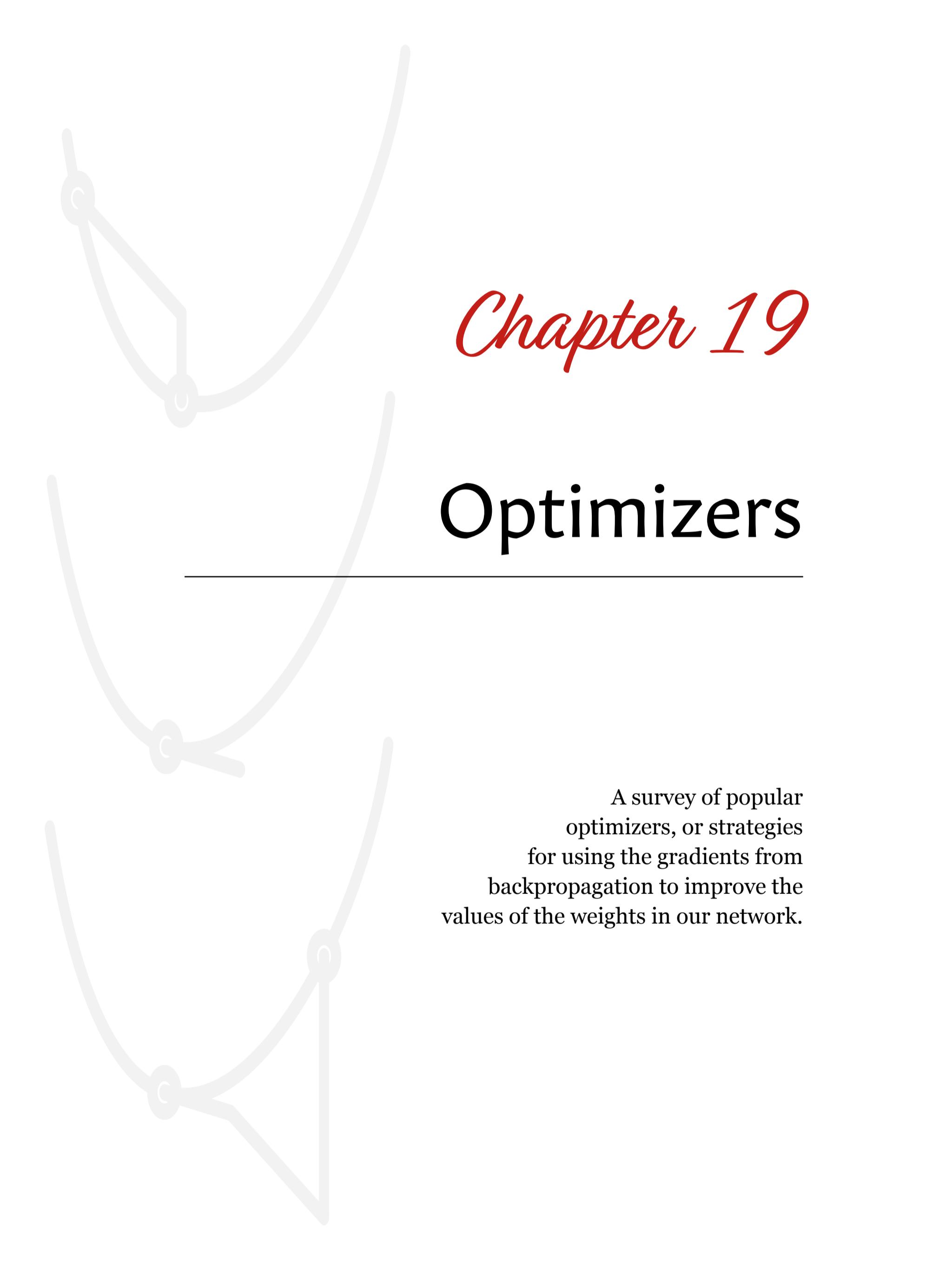
[Linnainmaa70] S. Linnainmaa, S., “The Representation of the Cumulative Rounding Error of an Algorithm as a Taylor Expansion of the Local Rounding Errors”, Master’s thesis, University of Helsinki, 1970.

[MathCentre09] Math Centre, “The Chain Rule”, Math Centre report mc-TY-chain-2009-1, 2009. <http://www.mathcentre.ac.uk/resources/uploaded/mc-ty-chain-2009-1.pdf>

[NASA12] NASA, “Astronomers Predict Titanic Collision: Milky Way vs. Andromeda”, NASA Science Blog, Production editor Dr. Tony Phillips, 2012. https://science.nasa.gov/science-news/science-at-nasa/2012/31may_andromeda

[Neilsen15a] Michael A. Nielsen, “Using Neural Networks to Recognize Handwritten Digits”, Determination Press, 2015. <http://neuralnetworksanddeeplearning.com/chap1.html>

- [Neilsen15b] Michael A. Nielsen, “Neural Networks and Deep Learning”, Determination Press, 2015. <http://neurallnetworksanddeeplearning.com/chap2.html>
- [Rumelhart86] D.E. Rumelhart, G.E. Hinton, R.J. Williams, “Learning Internal Representations by Error Propagation”, in “Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1”, pp. 318-362, 1986. <http://www.cs.toronto.edu/~fritz/absps/pdp8.pdf>
- [Schmidhuber15] Jürgen Schmidhuber, “Who Invented Backpropagation?”, Blog post, 2015. <http://people.idsia.ch/~juergen/who-invented-backpropagation.html>
- [Seung05] Sebastian Seung, “Introduction to Neural Networks”, MIT 9.641J course notes, 2005. https://ocw.mit.edu/courses/brain-and-cognitive-sciences/9-641j-introduction-to-neural-networks-spring-2005/lecture-notes/lec19_delta.pdf
- [Trefethen15] Nick Trefethen, “Who Invented the Great Numerical Algorithms?” Oxford Mathematical Institute, 2015. <https://people.maths.ox.ac.uk/trefethen/inventorstalk.pdf>
- [Werbos74] P. Werbos, “Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences”, PhD thesis, Harvard University, Cambridge, MA, 1974.
- [Werbos96] Paul John Werbos, “The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting”, Wiley-Interscience, 1994.
- [Wikipedia17] Wikipedia, “Goldilocks and the Three Bears”, 2017. https://en.wikipedia.org/wiki/Goldilocks_and_the_Three_Bears



Chapter 19

Optimizers

A survey of popular optimizers, or strategies for using the gradients from backpropagation to improve the values of the weights in our network.

Contents

19.1 Why This Chapter Is Here	807
19.2 Error as Geometry.....	807
19.2.1 Minima, Maxima, Plateaus, and Saddles...	808
19.2.2 Error as A 2D Curve	814
19.3 Adjusting the Learning Rate	817
19.3.1 Constant-Sized Updates.....	819
19.3.2 Changing the Learning Rate Over Time ..	829
19.3.3 Decay Schedules	832
19.4 Updating Strategies.....	836
19.4.1 Batch Gradient Descent	837
19.4.2 Stochastic Gradient Descent (SGD)	841
19.4.3 Mini-Batch Gradient Descent.....	844
19.5 Gradient Descent Variations.....	846
19.5.1 Momentum	847
19.5.2 Nesterov Momentum.....	856
19.5.3 Adagrad.....	862
19.5.4 Adadelta and RMSprop.....	864
19.5.5 Adam	866
19.6 Choosing An Optimizer.....	868
References	870

19.1 Why This Chapter Is Here

Training neural networks is frequently a time-consuming process. Anything that makes that go faster is a welcome addition to our toolkit. This chapter is about a family of tools that are designed to speed up learning when we use gradient descent.

In Chapter 18 we saw how backpropagation lets us efficiently apply gradient descent to adjust a network’s weights to improve its performance. We also saw how important it is to carefully choose the learning rate to balance off speed and stability.

In this chapter, we’ll look at algorithms that aim to build on the basic gradient descent algorithm. 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 creating or modifying a neural network.

19.2 Error as Geometry

It’s often helpful to think of the errors in our systems in terms of geometrical ideas. This lets us draw pictures of the errors and how we handle them, which can help us develop an intuition for what tools work best in different situations.

19.2.1 Minima, Maxima, Plateaus, and Saddles

Since optimizers are based on improving gradient descent, they also are designed to use the error gradient at each neuron to improve the network's performance.

As we saw in Chapter 18, the error gradient is found by considering the error surface, which tells us how the output error changes for a given change in a particular neuron's output. To recap, we'll briefly review the basic error gradient shapes here. We say that the neuron output values are the *inputs* to the error surface, since we use them to find the value of the error for those values.

Figure 19.1 demonstrates a simple error surface for the output values of two neurons.

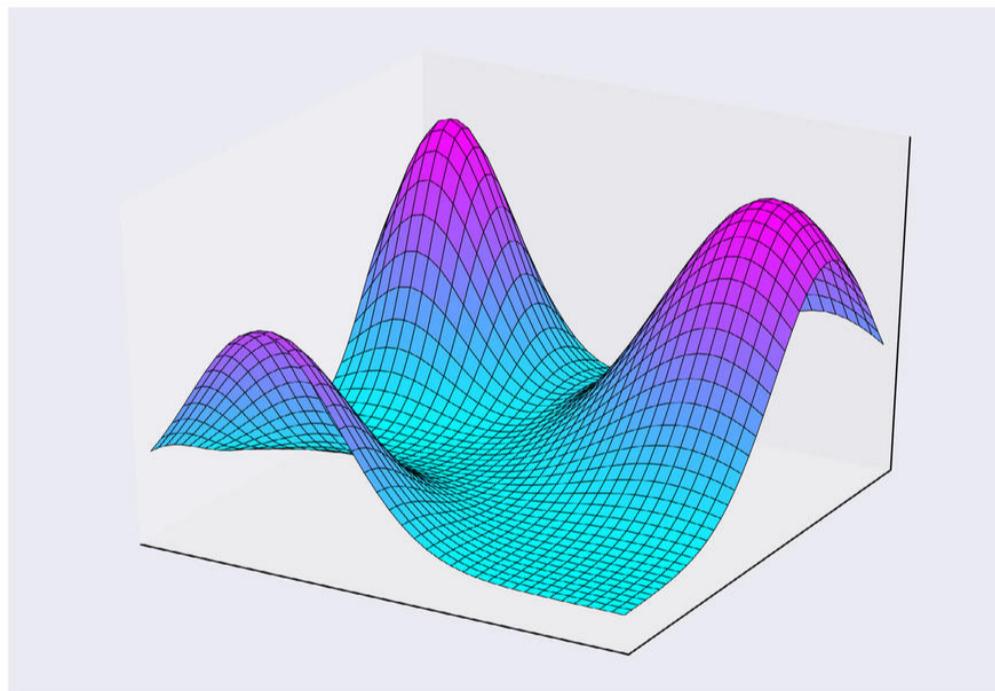


Figure 19.1: An error surface. For each combination of output values, we have an associated error, shown as the height of the surface above that point.

It's hard to draw graphs of the error with more than two neurons. For instance, 3 inputs (one for each of 3 neurons) and 1 output would require a 4-dimensional graph, which we can't draw. But we can conceive in the abstract of such surfaces for dozens or even hundreds of

dimensions. Whatever the number of dimensions, we specify a set of values, and get back a number for the error. That number is the “height” of the “surface” at that point.

We can then compute the gradient for that surface at that point. The math that calculates the gradient doesn’t care how many dimensions there are. This is nice, because it means that we can develop our intuition using a graph like Figure 19.1, confident that it will still apply for any number of dimensions.

We’ll consider four types of features common to all error surfaces: a **minimum**, a **maximum**, a **plateau**, and a **saddle**. We saw these in Chapter 5, so we’ll just briefly recap them here.

These shapes are special because they include places (sometimes just single points) where the gradient has a length of 0. We sometimes say that at these points the gradient **vanishes**. This is something we want to know about, because when there’s no gradient, the delta values in Chapter 18 become zero, which in turn means that weight updates will be zero. This means the weights don’t change and the network doesn’t improve. In short, when there’s no gradient, there’s no learning.

A **minimum** is a point on the surface where the gradient is zero, but moving in any direction causes us to go upwards. A minimum corresponds to the bottom of a bowl, as in Figure 19.2(a), or anywhere on the floor of a valley, as in Figure 19.2(b).

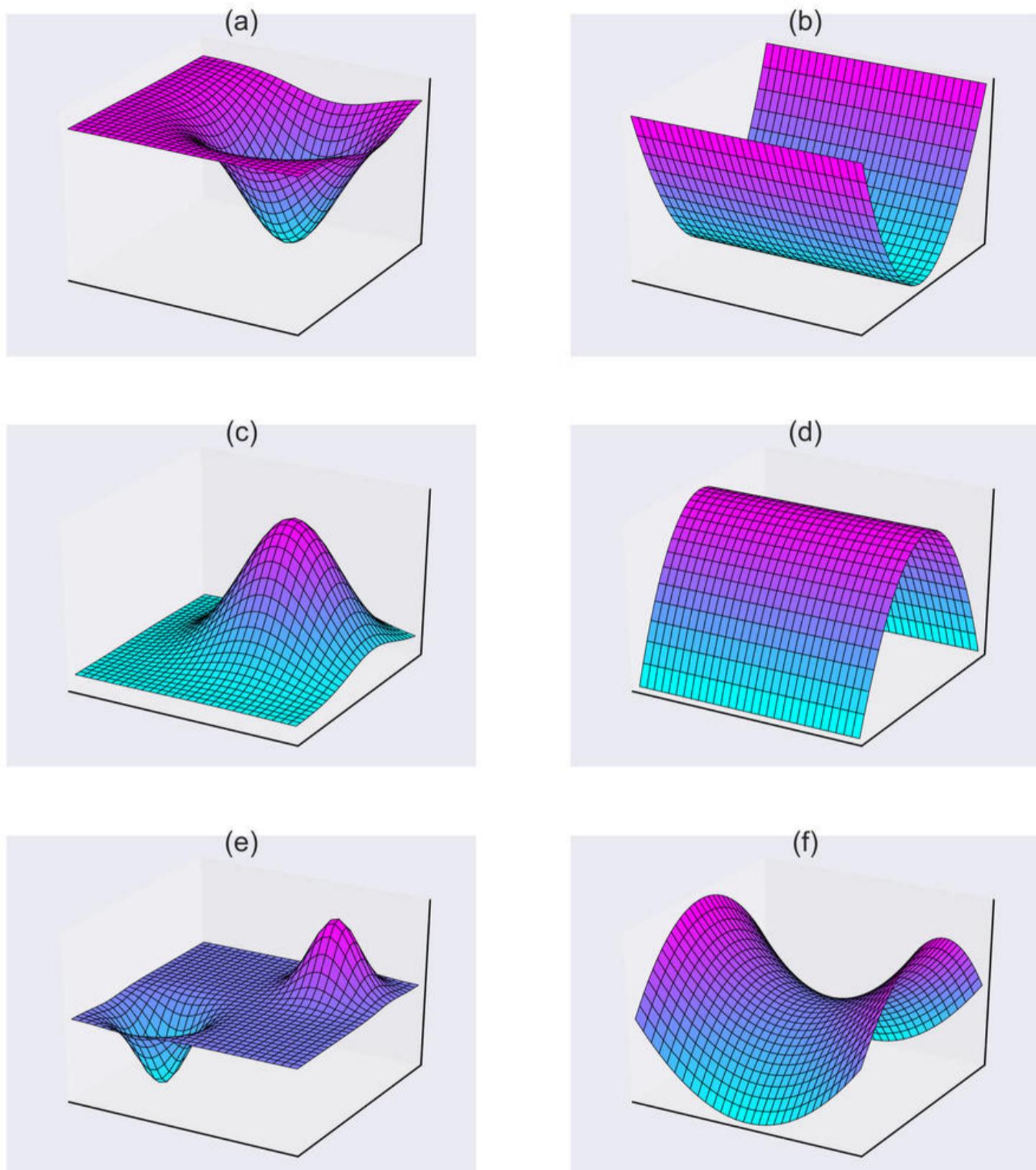


Figure 19.2: Places where the gradient can fall to 0. (a) The bottom of a minimum, or bowl. (b) The bottom of a valley. (c) The top of a maximum, or hill. (d) The top of a ridge. (e) On the flat parts of a plateau. (f) In the middle of a saddle.

The corresponding idea for a hilltop is called the **maximum**, and it's a peak where motion in any direction leads downhill. Figure 19.2(c) shows such a hill, and Figure 19.2(d) shows a ridge where all the points at the top are maxima.

A **plateau** is a region where the surface is flat all around us, so no matter how we move, we neither gain nor lose height. It's shown in Figure 19.2(e). The gradient is zero where the surface is flat.

Finally, a **saddle** is a region where moving along one or more axes increases the error, but moving along one or more other axes decreases it. This shape is shown in Figure 19.2(f). We can have saddles in any number of dimensions, where motion along some dimensions causes the error to increase, while moving in others causes the error to decrease. There's a point in the saddle where these contrary motions balance out, and the gradient is zero.

Consider a bowl with a minimum at the very bottom. The gradient falls to 0 only at the lowest point, at the bottom of the bowl. But even if we're only very close to that point, the gradient can be very small, particularly if the bowl is shallow. If the gradient is very close to 0, learning may not stop altogether, but it could slow so much that learning seems to take forever.

In practice, there can be many instances of each of these features across our error landscape, as we saw in Figure 19.1. The collective terms for multiple instances of these features are **minima**, **maxima**, **plateaus**, and **saddles**.

If there are many valleys of different depths, then one might be deeper than all the others. We say that the very deepest minimum across the whole landscape is the **global minimum**, and all the others are **local minima**. The same idea holds for the **global maximum** and the other **local maxima**. When we're minimizing the error, we want to find the global minimum, which is the smallest error possible across all possible outputs of the network. We often are satisfied, though, with a local minimum that gives us an amount of error that's acceptable for our application.

We saw in Chapter 8 that gradient descent can fall into a valley and then remain there, bouncing around from side to side, perhaps dropping to the bottom slowly or quickly, but never getting out of the valley. If our valley surrounds the global minimum, that's great, because as

we'll see later we can slow down the bouncing around. That will let us settle into the bottom of the basin, and thereby find the very best values for the output values.

But if we're in a local minimum, that behavior can be a problem, because there might be a much deeper minimum nearby, and we'll never pop out of the shallow one to reach the deep one. Figure 19.3 shows the idea.

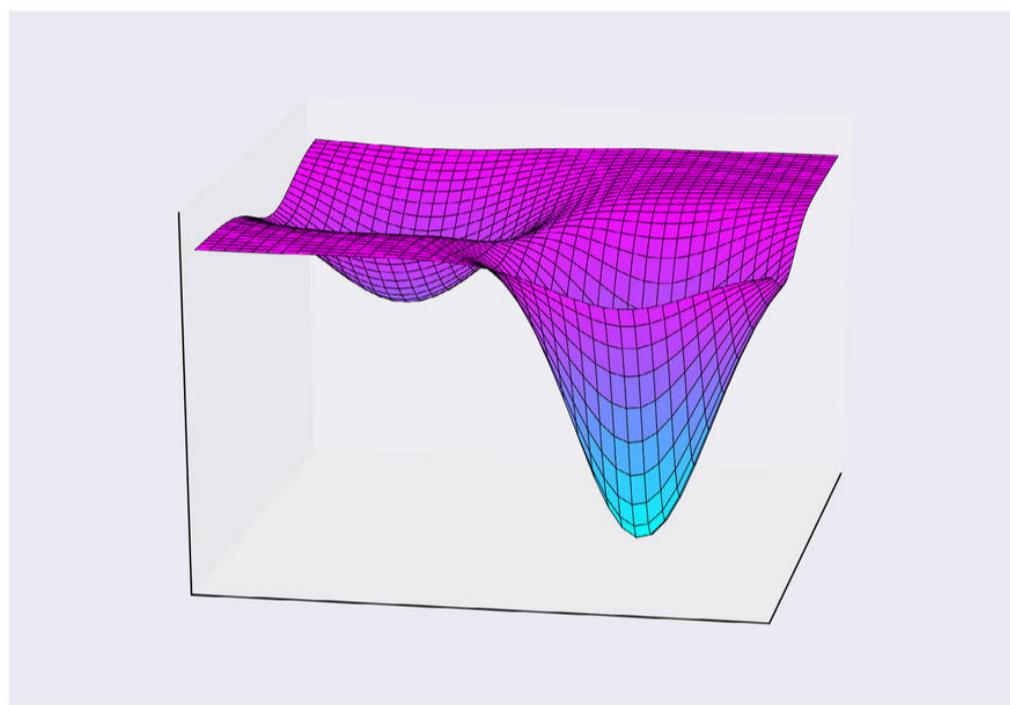


Figure 19.3: Two local minima, one deeper than the other. Algorithms that seek the lowest point don't know if they're in the deepest minimum around or not.

Since we want the lowest error, we want the deepest minimum, but as we've seen, we have no way to locate it. Gradient descent is a **local** or **greedy** algorithm, which makes it fast and robust, but also blinds it to better choices even if they're nearby.

Curiously, although minima are present in the error surfaces for neural networks, in practice a much bigger problem is posed by saddles [Dauphin14]. Many optimization algorithms can get seriously stuck in saddles, because the slopes going up in some directions might be very large, and the slopes in the other directions might be small. Figure 19.4 shows this idea for a 3D saddle with two directions.

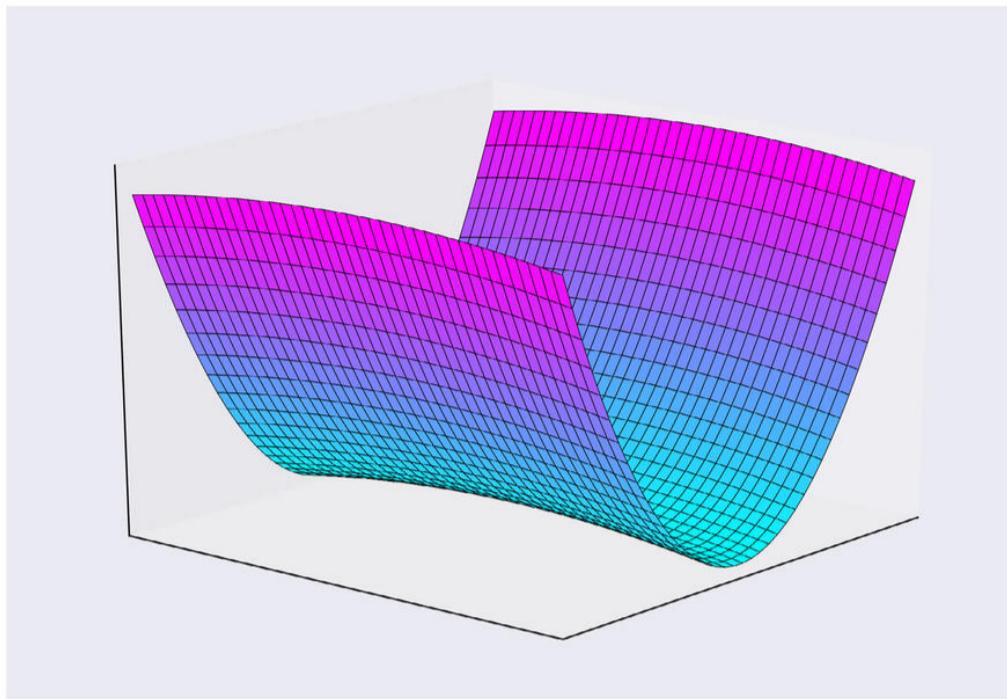


Figure 19.4: A shallow saddle. Traveling towards or away from the steep sides will cause a big change in the error. Taking similar-sized steps perpendicular to those sides results in much smaller changes to the error.

An optimizer can get focused on the steep walls, doing its best to avoid them and thereby not push the error up. Those steep sides dominate the calculation of the gradient, so the very shallow descents in the other directions get overwhelmed numerically, and the algorithm makes little or no progress in its goal of moving downwards.

The frustrating thing about getting caught in a saddle is that *we* know that the system could do better if only it could move in the downhill directions, but the gradient descent algorithm doesn't see the big picture. It just looks around where it is, sees a gradient that's zero or close to zero, and finds itself stuck near the balance point.

The purpose of this chapter is to survey some algorithms designed to reduce the problems posed by bouncing around inside local minima and getting stuck at saddles. We'll also see how to avoid getting stuck on plateaus where the gradient vanishes.

19.2.2 Error as A 2D Curve

To illustrate how different optimizers perform their updates, we'll demonstrate them using a 2D error curve. We can think of this as a cross-section of the 3D error landscapes in the previous section, which are themselves only suggestions of the much higher-dimensional error landscapes we often work with.

To get familiar with this 2D error, let's think of it as the result of trying to split two classes of samples, represented as dots arranged on a line. Dots at negative values are in one class, and all other dots are in the other, as shown in Figure 19.5.

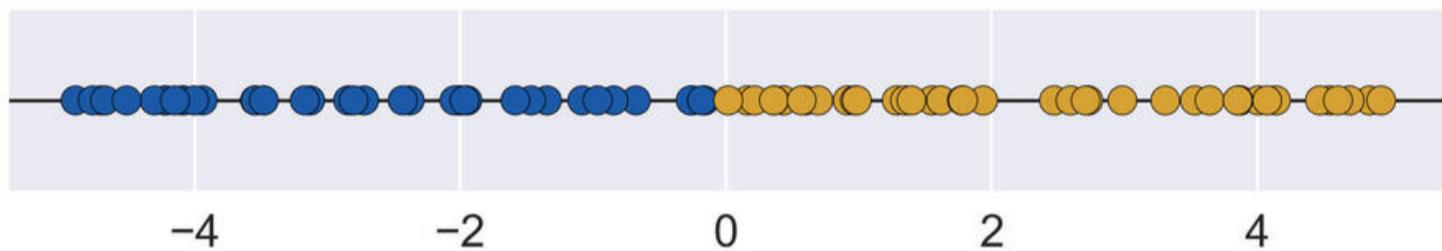


Figure 19.5: Two classes of dots on a line. Dots to the left of 0 are in class 0, shown in blue, and those to the right are in the class 1, shown in beige.

We want to 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 will be assigned to class 0, and all those to the right will be 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. Figure 19.6 shows the idea.

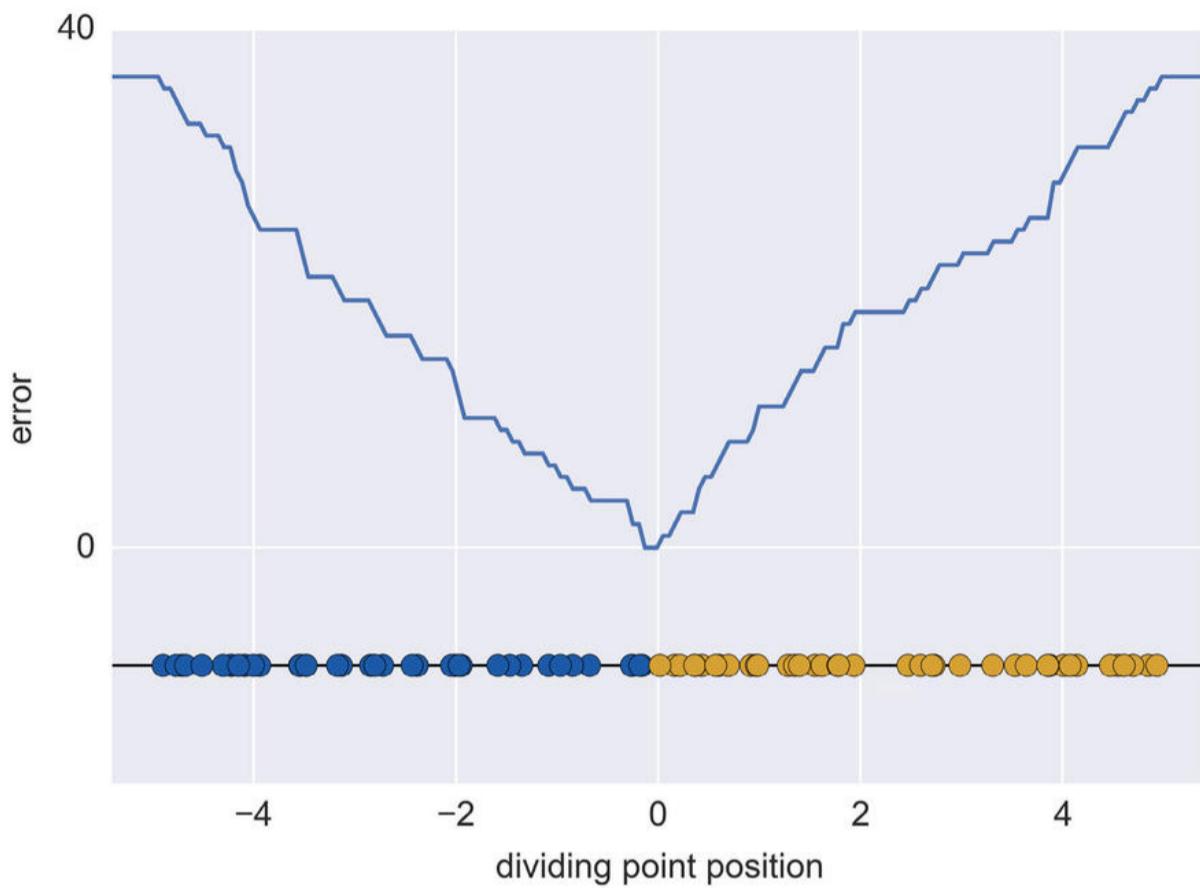


Figure 19.6: For each location on the line, we can classify all points to its left as class 0 and all those to its right as class 1. If we add up the number of misclassified points, we get the error function shown in the top part of the figure.

Unsurprisingly, since we set up our data so that the classes are split at 0, placing the dividing point at 0 gives us an error of 0. The farther we are from 0, either to the left or right, the higher our error.

As we know from Chapter 18, we want to use smooth error functions because they let us calculate their gradients (and thus drive backpropagation). So we can smooth out the error curve of Figure 19.6 as shown in Figure 19.7.

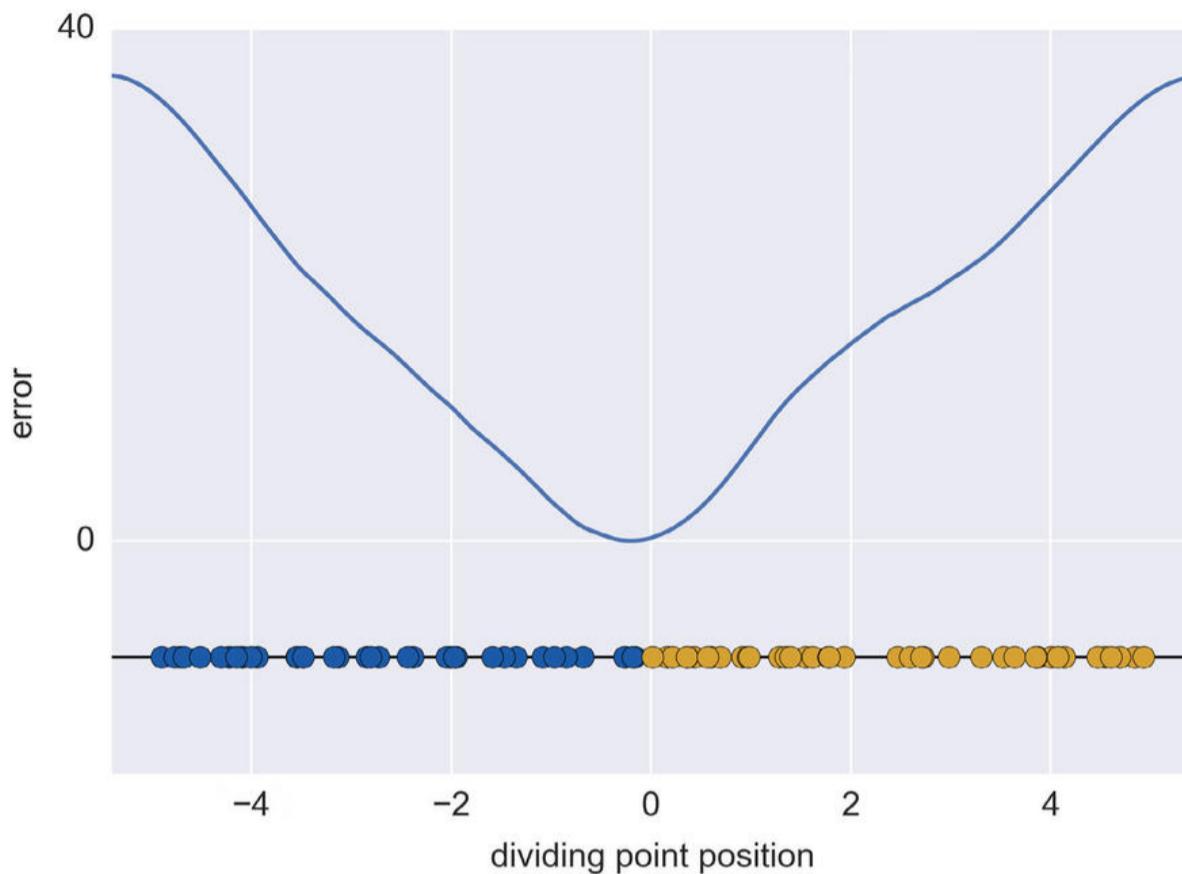


Figure 19.7: A smoothed version of Figure 19.6. We can calculate gradients on this curve, and thus drive backpropagation.

For this particular set of random data, it looks like the error is 0 when we're at 0, or just a little to the left of it. The bottom of the curve seems to be just a little left of 0. This is the minimum of the curve, where the error is the lowest. Regardless of where we start, this is where we want to end up.

Treating this as our error curve, the optimizers we'll see in this chapter are all designed to use gradient descent to find that minimum as efficiently as they can.

The critical parameter when learning with gradient descent is the **learning rate**, usually written with the lower-case Greek letter η (eta). Larger values approaching 1, lead to faster learning, but as we saw in Chapter 8 this can lead us to miss valleys by jumping right over them. Smaller values of η (nearing 0, but no smaller) lead to slower learning and can find narrow valleys, but we also saw in Chapter 8 that they can also get stuck in gentle valleys even when there are much deeper ones nearby. Figure 19.8 recaps these phenomena graphically.

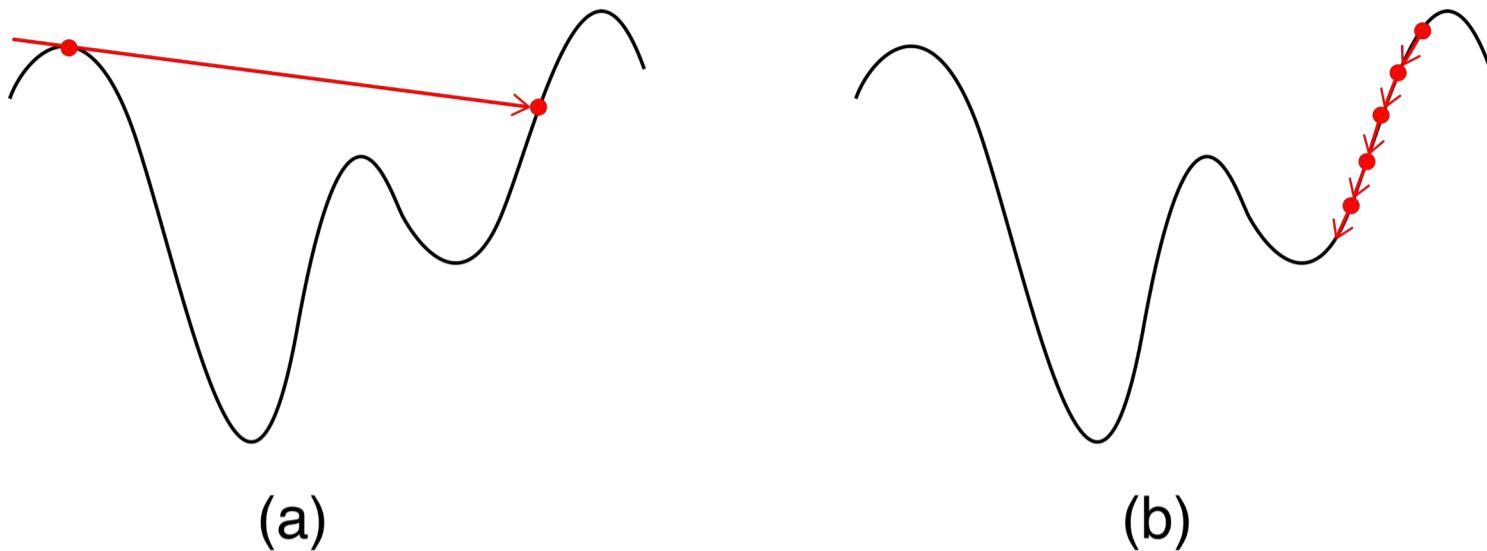


Figure 19.8: The influence of the learning rate η . (a) When η is too large, we can jump right over a deep valley and miss it. (b) When η is too small, we can slowly descend into a local minimum, and miss the deeper valley.

19.3 Adjusting the Learning Rate

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 that we can take big steps early in the learning process, while we're hunting for a nice deep valley. As time goes on, we'll presumably have found our way into a minimum, so we can take smaller and smaller steps as we approach the very bottom of the valley.

We'll illustrate our optimizers with a single isolated valley with the shape of a negative Gaussian, shown in Figure 19.9.

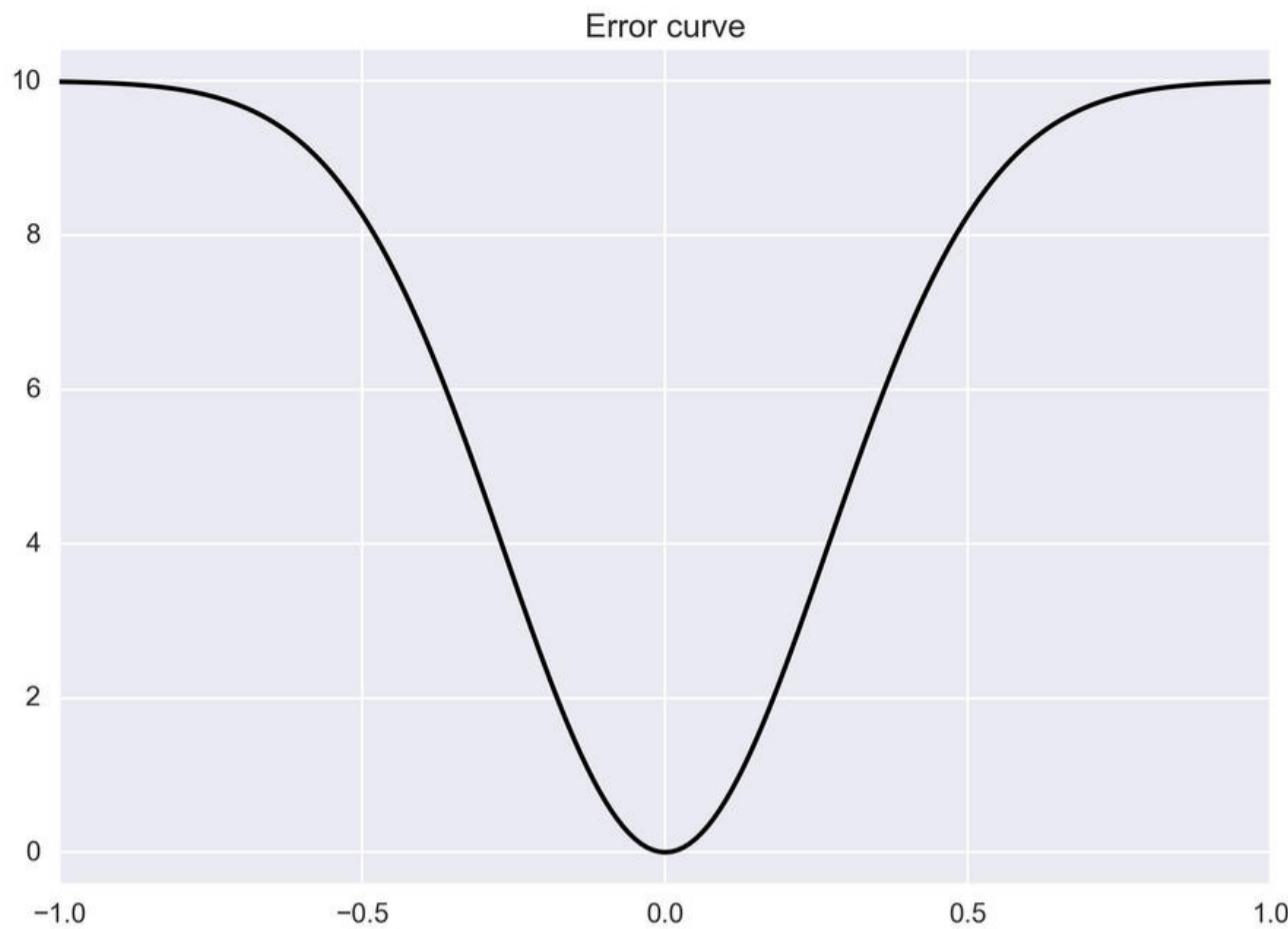


Figure 19.9: Our error curve for looking at optimizers.

Some gradients for this error curve are shown in Figure 19.10. We can see that 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 zero.

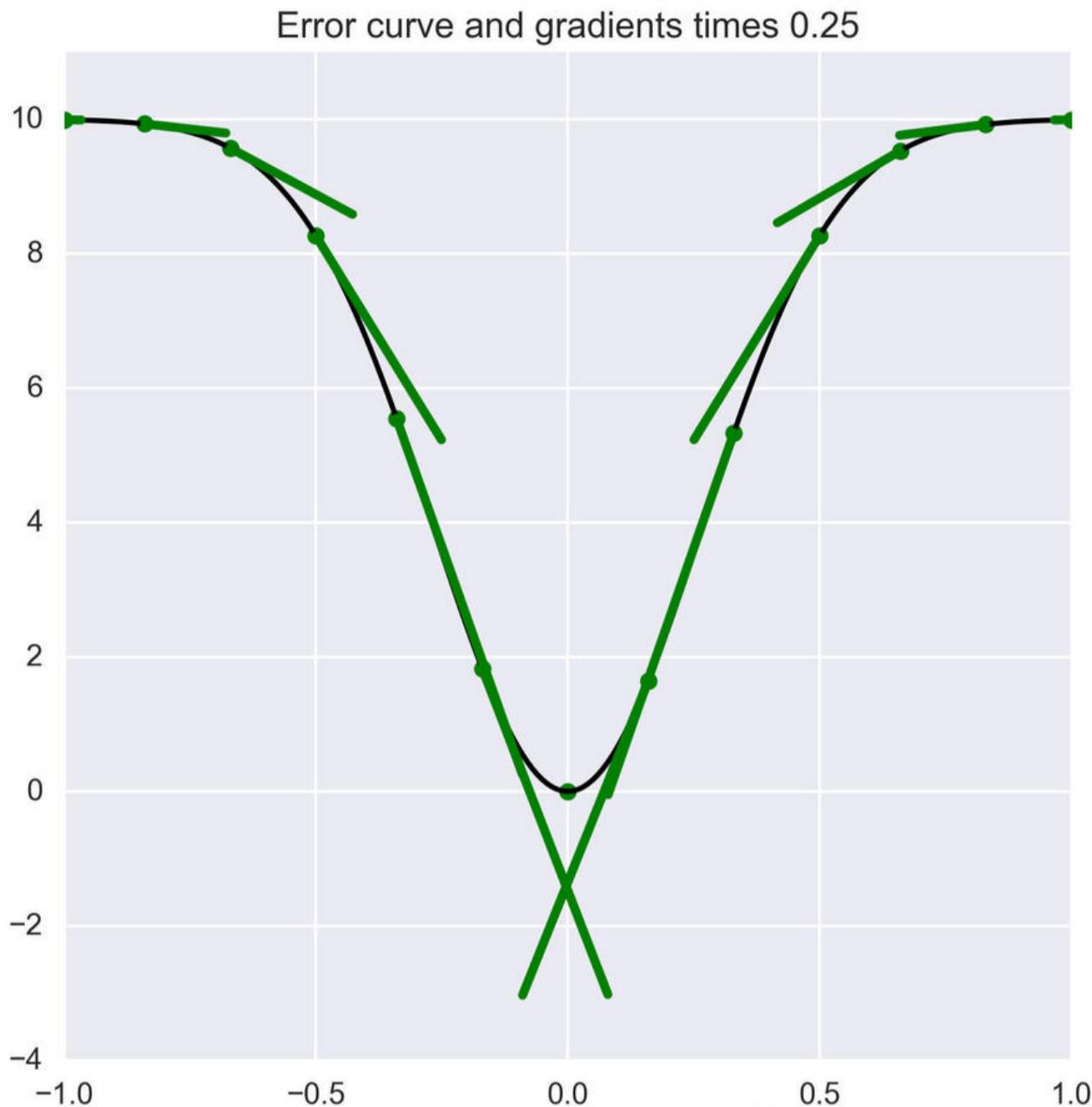


Figure 19.10: Our error curve and its gradients at some locations. The gradients have been scaled in length for clarity. Note that at the very bottom of the curve the gradient falls to a length of 0, so it's drawn as just a single dot.

19.3.1 Constant-Sized Updates

We'll start by reviewing what happens when we use a constant learning rate. In other words, we'll scale the gradient by a value of η that stays fixed, or constant, during the whole learning process.

Figure 19.11 shows the basic steps of constant-size updating. Suppose we're looking at a particular weight in a neural network. We'll pretend that the weight begins with value W_1 , and we updated it once, so it now

has the value W_2 , shown in Figure 19.11(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 which we'll call W_3 .

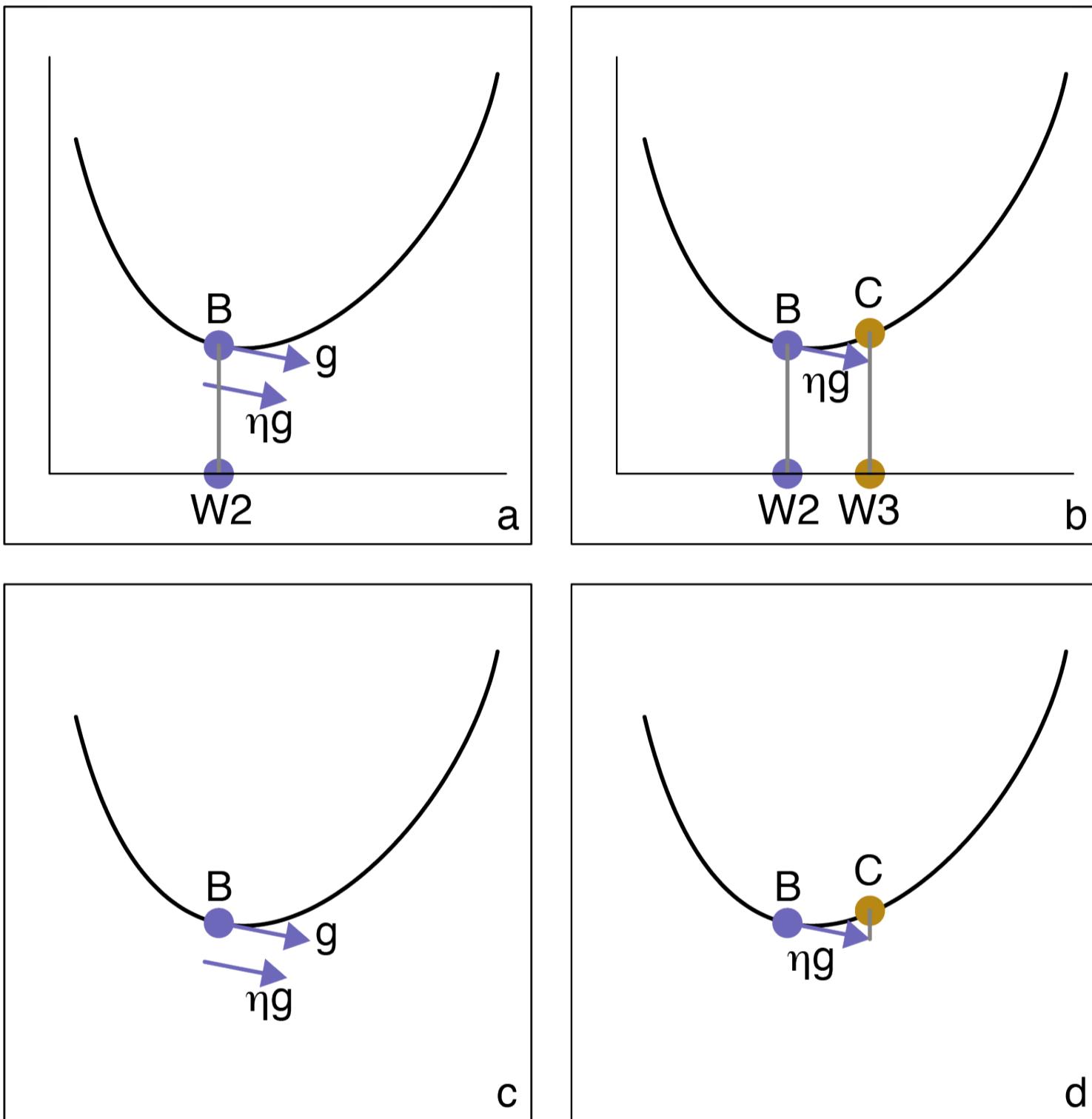


Figure 19.11: Finding the step for “vanilla” gradient descent. (a) When our weight has value W_2 , we find the point on the error curve above it, marked B. That gradient at B is the arrow labeled g . We then multiply that gradient by the learning rate η , giving us the arrow ηg . Since η is usually less than 1, ηg is shorter than g but points in the same direction. (b) Finding W_3 from W_2 and ηg . (c) A simpler version of part (a). (d) A simpler version of part (b).

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 ηg . Because η is between 0 and 1, ηg is a new arrow that points in the same direction as g , but it's either the same size as g or smaller.

To find W_3 , the new value of the weight, we add the scaled gradient to W_2 . In pictures, that means we place the tail of the arrow ηg at B, as in Figure 19.11(b). The horizontal position of the tip of that arrow is the new value of the weight, W_3 , and its value directly above it on the error surface is marked C. In this case, we stepped a bit too far and increase our error by a little.

There are a lot of labels and lines running around in those figures. We can make it all much simpler by just drawing the points that are on the error curves, so Figure 19.11(a) and (b) can be drawn as Figure 19.11(c) and (d). We just need to keep in mind that the values of the weights are the horizontal position of the points, and the error is the value of the curve above them.

Let's look at this technique in practice using our little error curve of a single valley. Figure 19.12 shows a starting point in the upper left. The gradient here is small, so we move to the right a small amount (recall that the gradient points uphill, so we move in the direction of the negative gradient to head downhill).

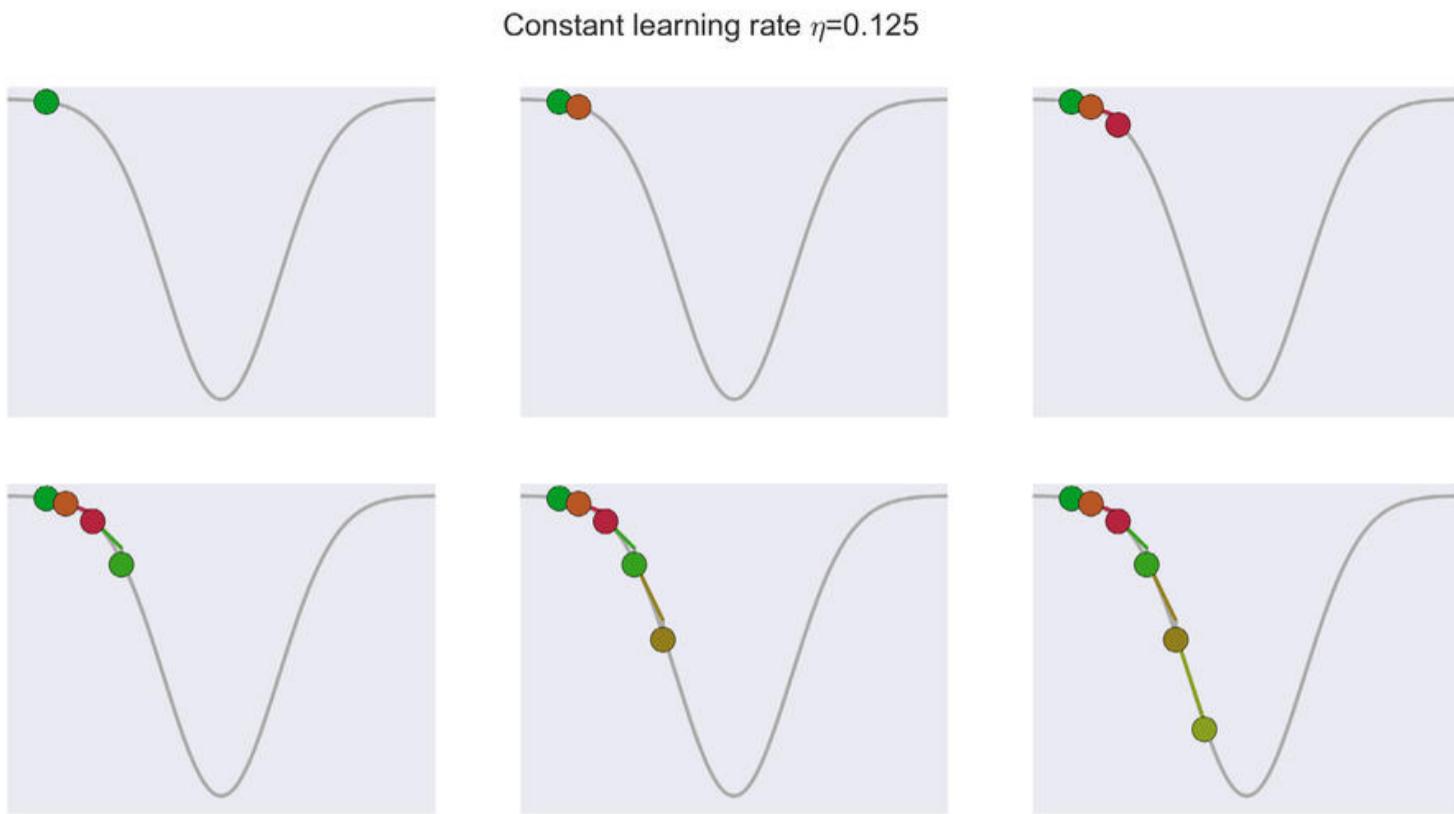


Figure 19.12: Learning with a constant learning rate.

For these figures, we've chosen $\eta = 1/8$, or 0.125. This is a pretty 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 really interested in the nature of what happens rather than the numbers (the actual graph and gradient values are shown in Figure 19.10).

So rather than move from our first point by the entire gradient, we move only 1/8 of the way.

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. The vertical gray lines are to help us see how the new point is directly above or below the end of the gradient line.

A close-up of the first six points is shown in Figure 19.13. We also show the error for each point. In these figures, we show the local derivative, scaled by the learning rate, at each point with a straight line that's the same color as the next point. We also show a vertical line from the end of that scaled derivative back to the curve. These lines are a little hard to see in these figures because they're short and follow the curve closely, but they'll be easier to see in later figures.

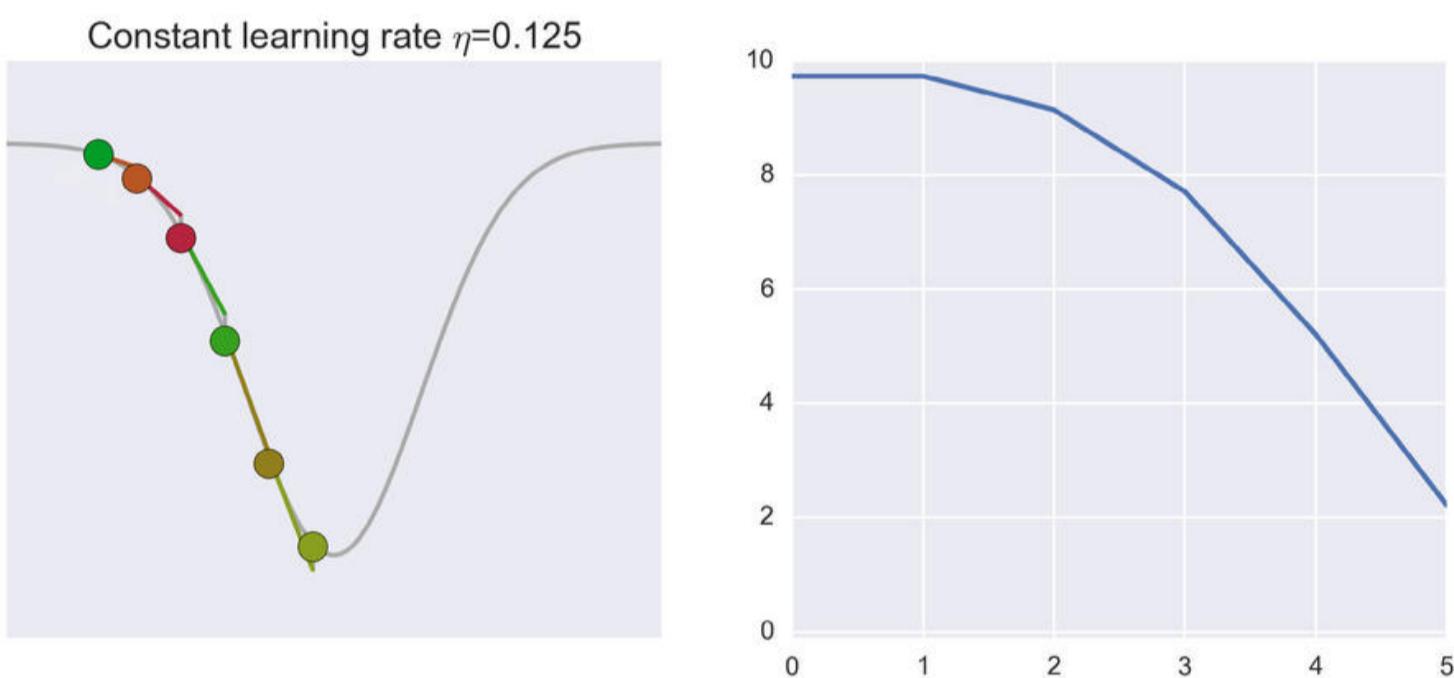


Figure 19.13: Left: A close-up of the final image in Figure 19.12. Right: The error associated with each of these six points.

Will this process ever reach the bottom of the bowl (that is, will our network ever get down to 0 error)? Figure 19.14 shows the first 15 steps in this process.

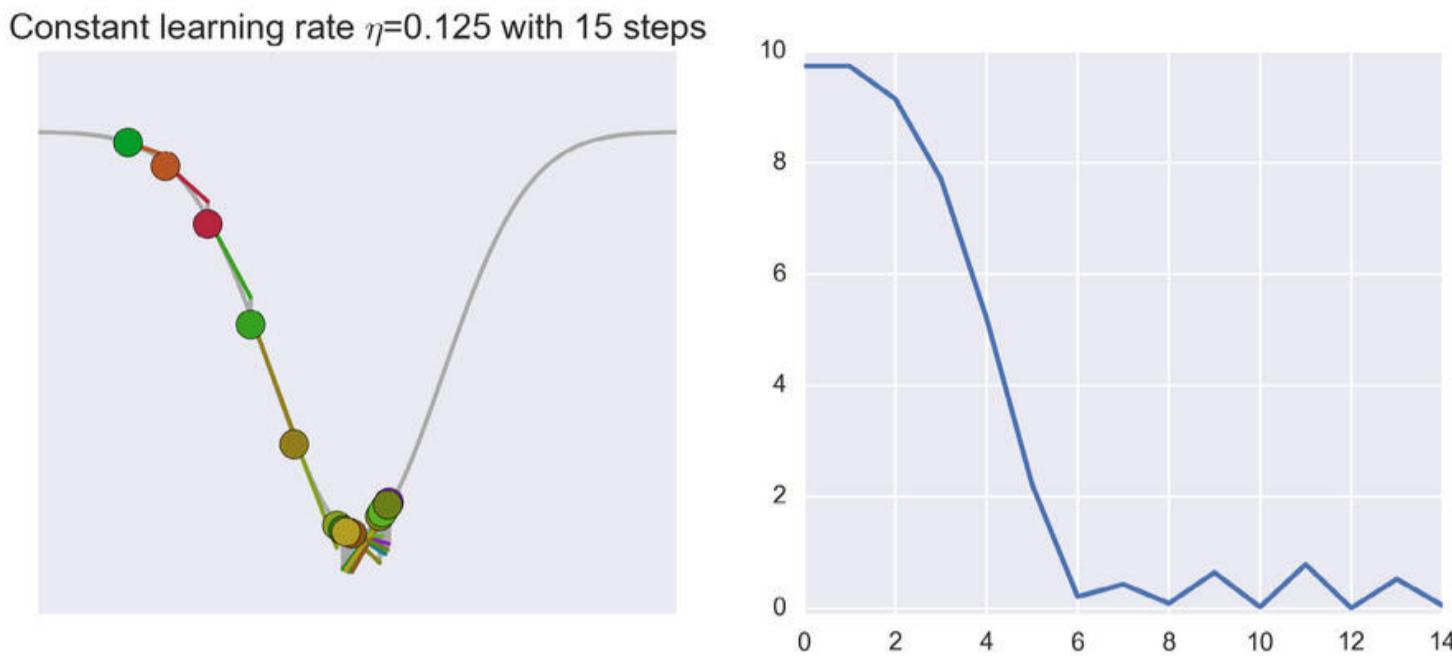


Figure 19.14: Left: The first 15 steps of learning with a constant learning rate. Our constant-sized steps mean that we spend a lot of time bouncing around the bottom of the bowl. Right: The errors of these 15 points. The points on the left get close to zero, but then jump up on the right side of the valley to a larger error.

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 overshoot again and end up on the right side, and so on, back and forth. We sometimes say that we're **bouncing around** the bottom of the bowl.

We are very gradually heading towards an error of 0, but it looks like it's going to take forever. 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.

Maybe the bouncing problem of Figure 19.14 was caused by using $1/8$ for the learning rate. Maybe other values wouldn't bounce around like that. Figure 19.15 shows how things go for the first six steps of some very small values of η .

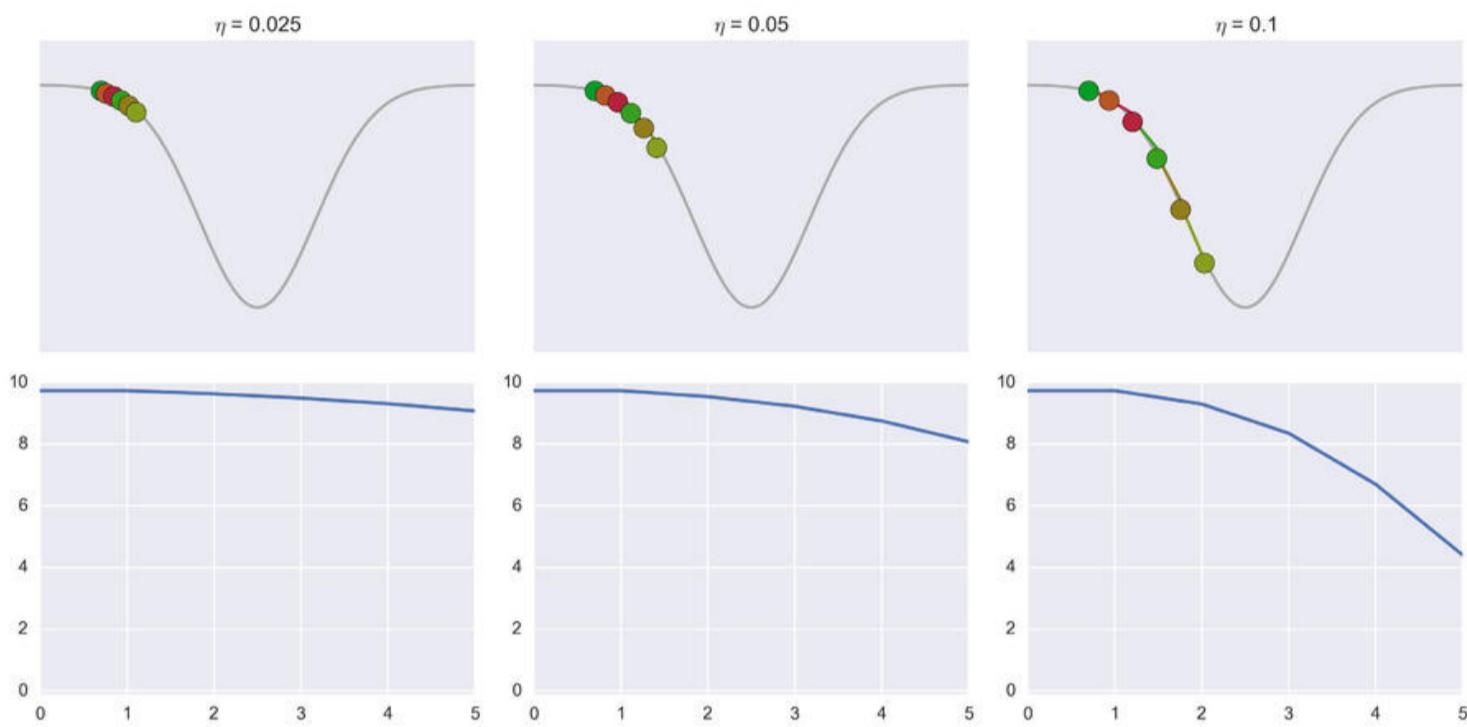


Figure 19.15: The progress down the valley for six steps of gradient descent using smaller learning rates. Top row: The first six points using learning rates of 0.025 (left column), 0.05 (middle column), and 0.1 (right column). Bottom row: The error values of the above points.

When η takes on the values 0.025, 0.05, and 0.1 as in Figure 19.15, we take tiny steps. This is likely to do well when we finally get down to the bottom of the bowl, but it's going to take forever to get there. Once we do get near the bottom, we'll get the same kind of bouncing behavior as before, but on a much smaller scale.

The first six points for some larger values of η are shown in Figure 19.16. Here we can more easily see the colored lines at each point showing the local derivative scaled by the learning rate, and the gray vertical lines running from the end of the scaled derivative to the curve.

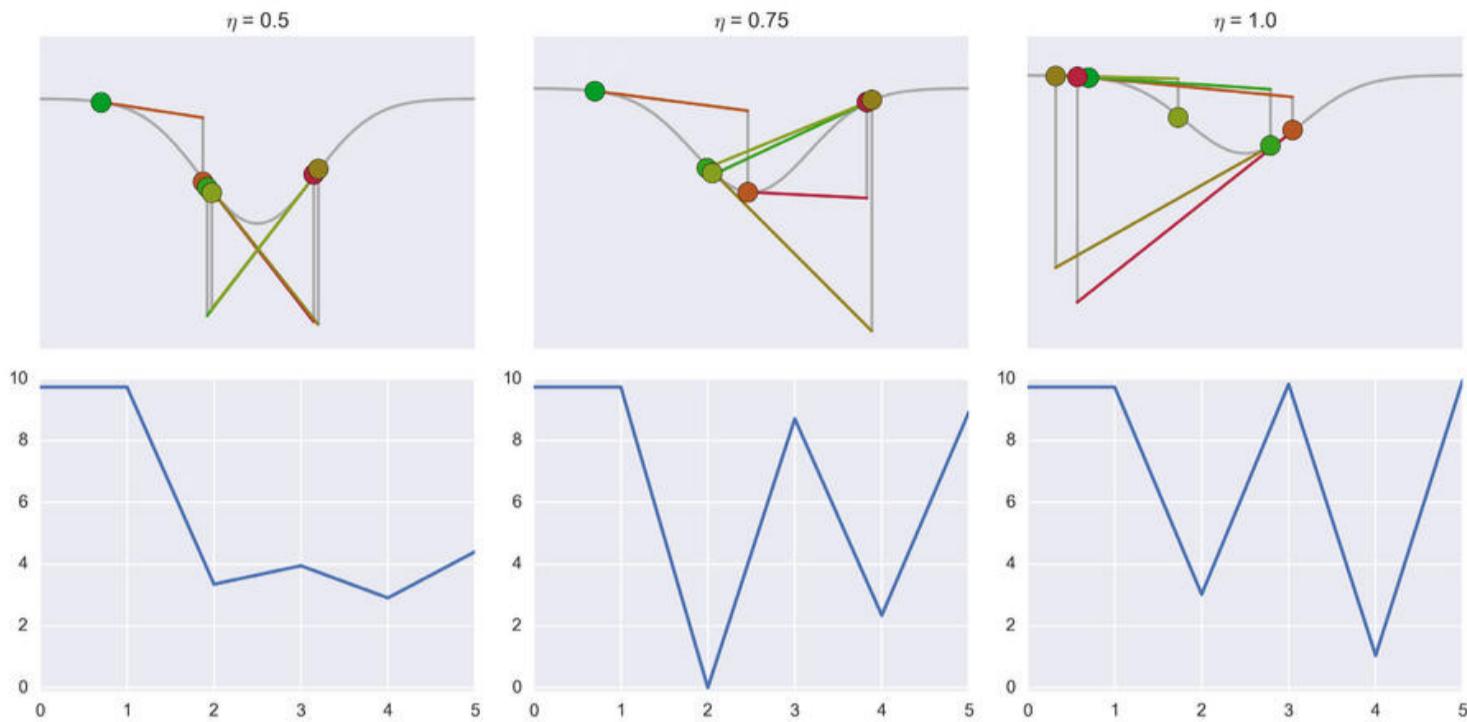


Figure 19.16: The progress down the valley for 15 steps of gradient descent using larger learning rates. Top row: The first six points using learning rates of 0.5 (left column), 0.75 (middle column), and 1.0 (right column). Bottom row: The error values of the above points.

For the large values of the learning rate in Figure 19.16, we start to see problems right away. When $\eta = 0.5$, we're bouncing around the bottom much worse than before. This will take a very long time to settle down. When $\eta = 0.75$, our first step takes us nearly to the bottom of the bowl, but then we move so far that we end up with nearly as much error as we started with, though on the right side of the valley. And when $\eta = 1.0$, the same problem is showing up but even worse.

This is typical of large learning rates. The problem can be traced back to the definitions of the derivative and gradient. These measures are only accurate very close to the point where we're evaluating them. When we move too far, they stop matching the curve and we can end up somewhere unexpected.

Six points isn't much. Let's look at the results of these learning rates for 15 points. The graphs and errors for the small learning rates are shown in Figure 19.17.

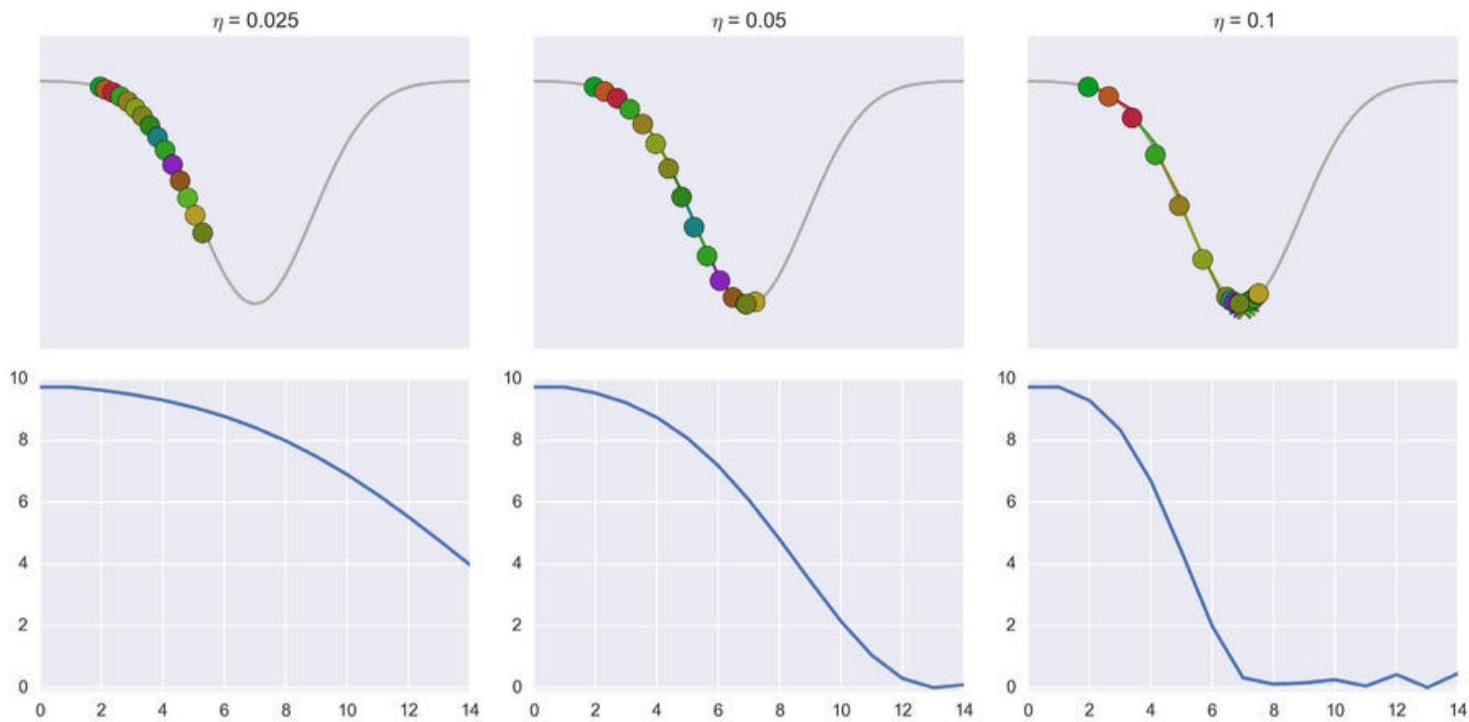


Figure 19.17: Taking 15 steps with our small learning rates. Top row: Learning rates of 0.025 (left column), 0.05 (middle column), and 0.1 (right column). Bottom row: Errors for the points in the top row.

The results for 15 steps with the high values of the learning rate are shown in Figure 19.18.

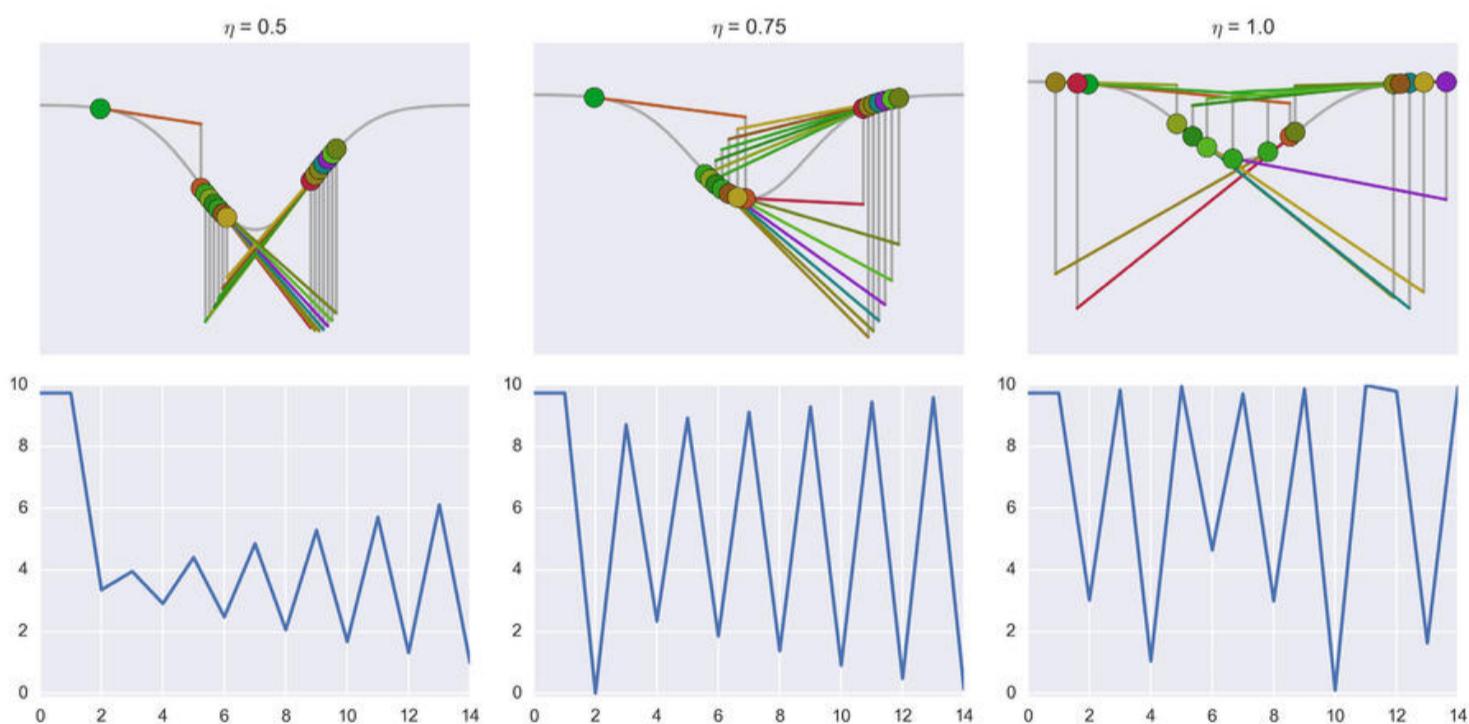


Figure 19.18: Taking 15 steps with large learning rates. Top row: Learning rates of 0.5 (left column), 0.75 (middle column), and 1.0 (right column). Bottom row: Errors for the points in the top row.

Picking a value of η that's too small can get us near the bottom, but it will take a long time. Picking a value of η that's too big can lead to the kinds of bouncing problems we see in Figure 19.18.

Larger learning rates can also cause us to jump out of a nice valley with a low minimum. In Figure 19.19 we jump right over the rest of the valley we're in, and into a new valley with a much larger minimum.

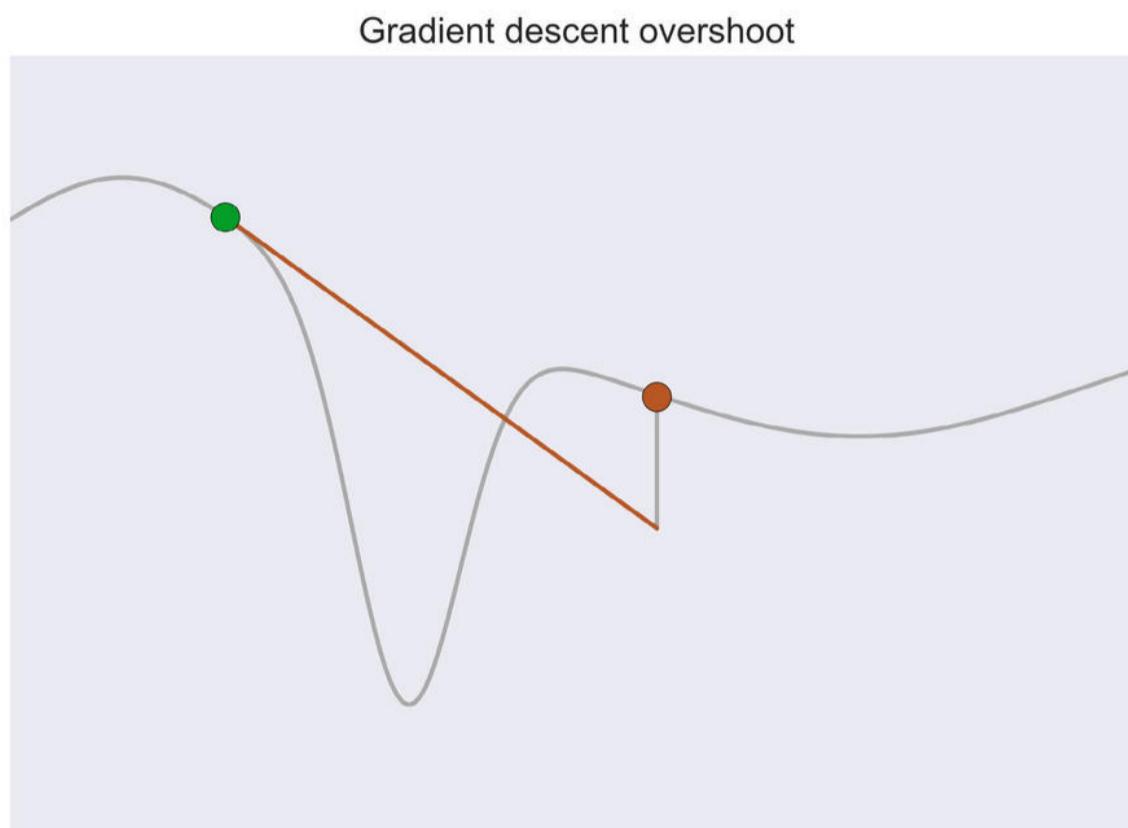


Figure 19.19: We're at the green point on the left and we take an update step, taking us to the red point on the right. The green point is within the valley we want to fall into, but this large step overshoots the valley and ends up in a different valley with a higher minimum.

It seems like a challenge to find just one learning rate that will move at a reasonable speed, but won't overshoot valleys or get trapped bouncing around in the bottom.

19.3.2 Changing the Learning Rate Over Time

What if we change the learning rate as we go? We could use a large value of η near the start of our learning, so we don't crawl along, but 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, say 0.99, after every update step. Let's 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 19.20 shows what happens to η when we do this for many steps, using a few different values for the multiplier.

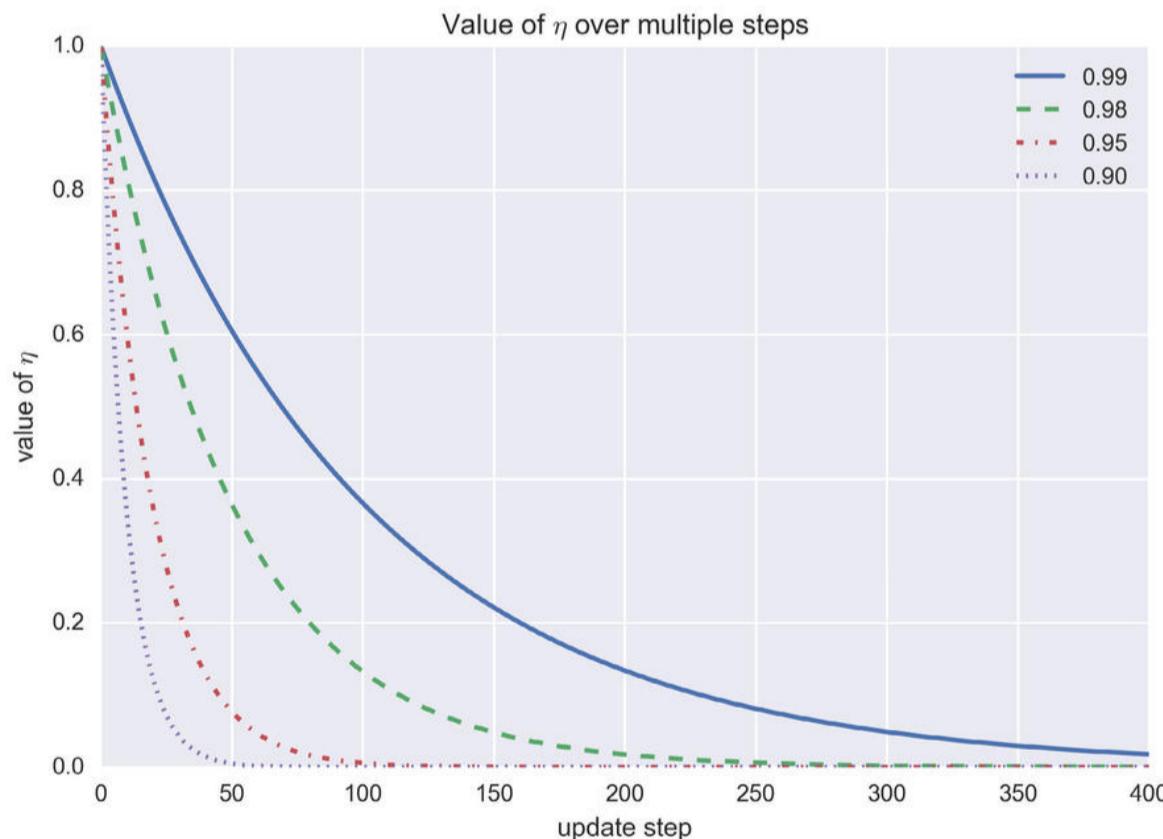


Figure 19.20: Starting with a learning rate of $\eta=1$, the various curves show how the learning rate drops after multiplying it by a given value after each update.

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. The figure shows four examples of the decay parameter.

Let's apply this gradual reduction of the learning rate to gradient descent on our error curve. Once again we'll start a learning rate of $1/8$. To make the effect of the decay parameter easily visible, we'll set it to the unusually low value of 0.8 . This means each step will only be 80% as long as the step before it. The results are shown in Figure 19.21.

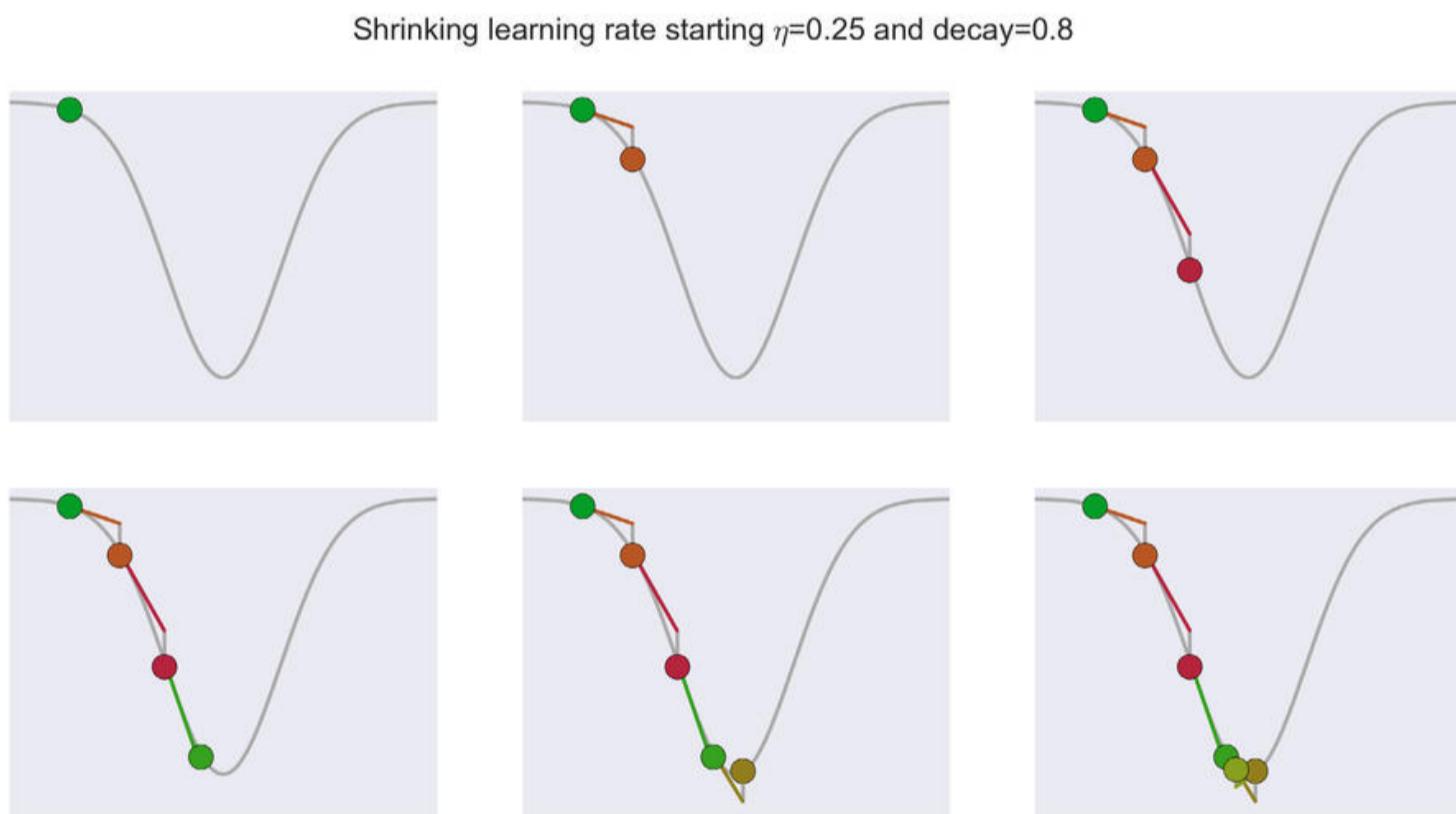


Figure 19.21: The first six steps using a shrinking learning rate.

The final image along with the error at each point along the way is shown in Figure 19.22.

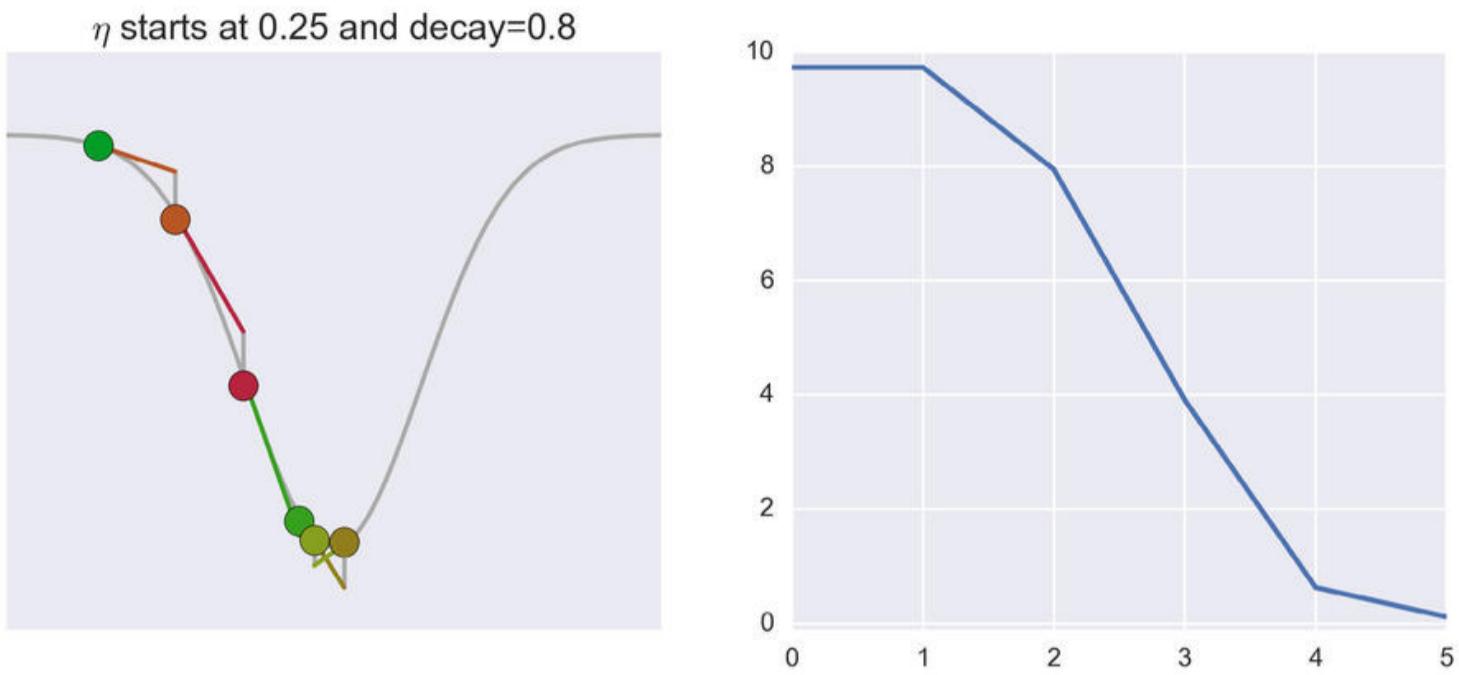


Figure 19.22: Left: The bottom-right image in Figure 19.21. Right: The error for these six points.

This is encouraging. Let's continue this process for 15 steps. The result is shown in Figure 19.23.

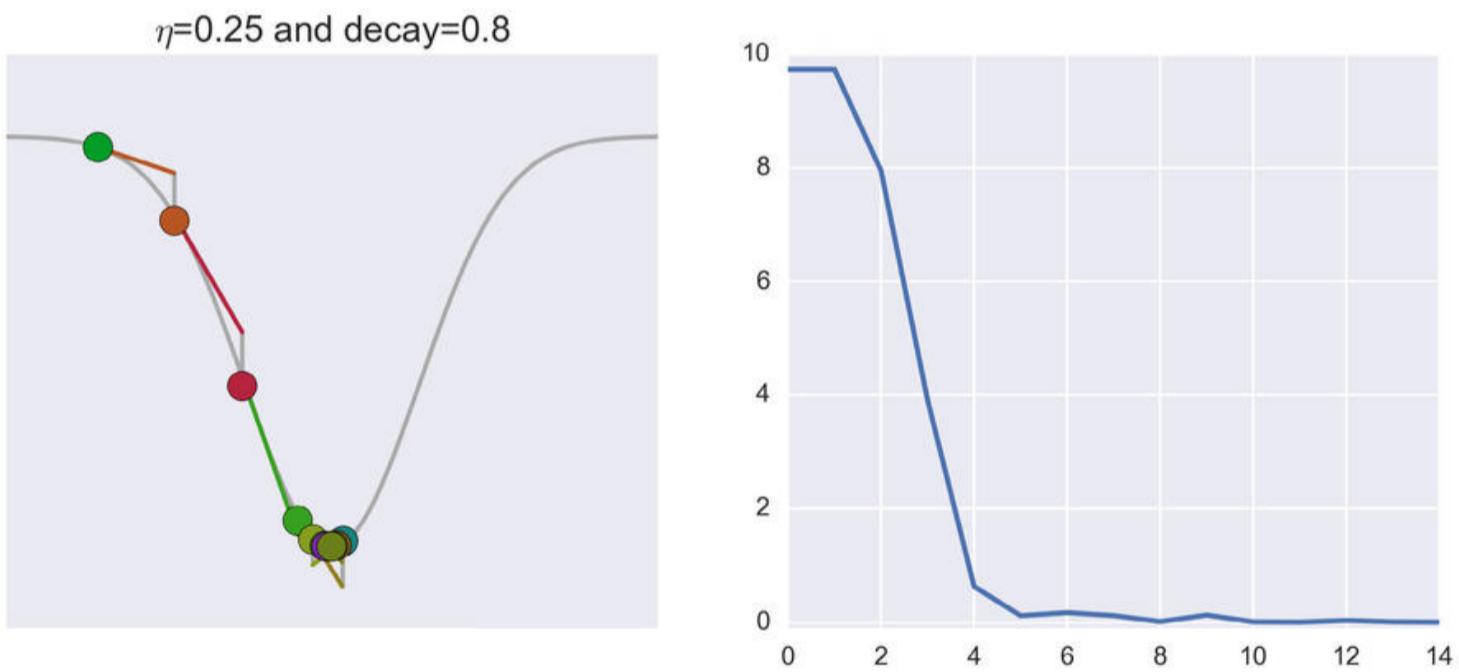


Figure 19.23: The first 15 steps using a shrinking learning rate.

Let's compare this with our "bouncing" result from using a constant step size. Figure 19.24 shows the results for the constant and shrinking step sizes together.

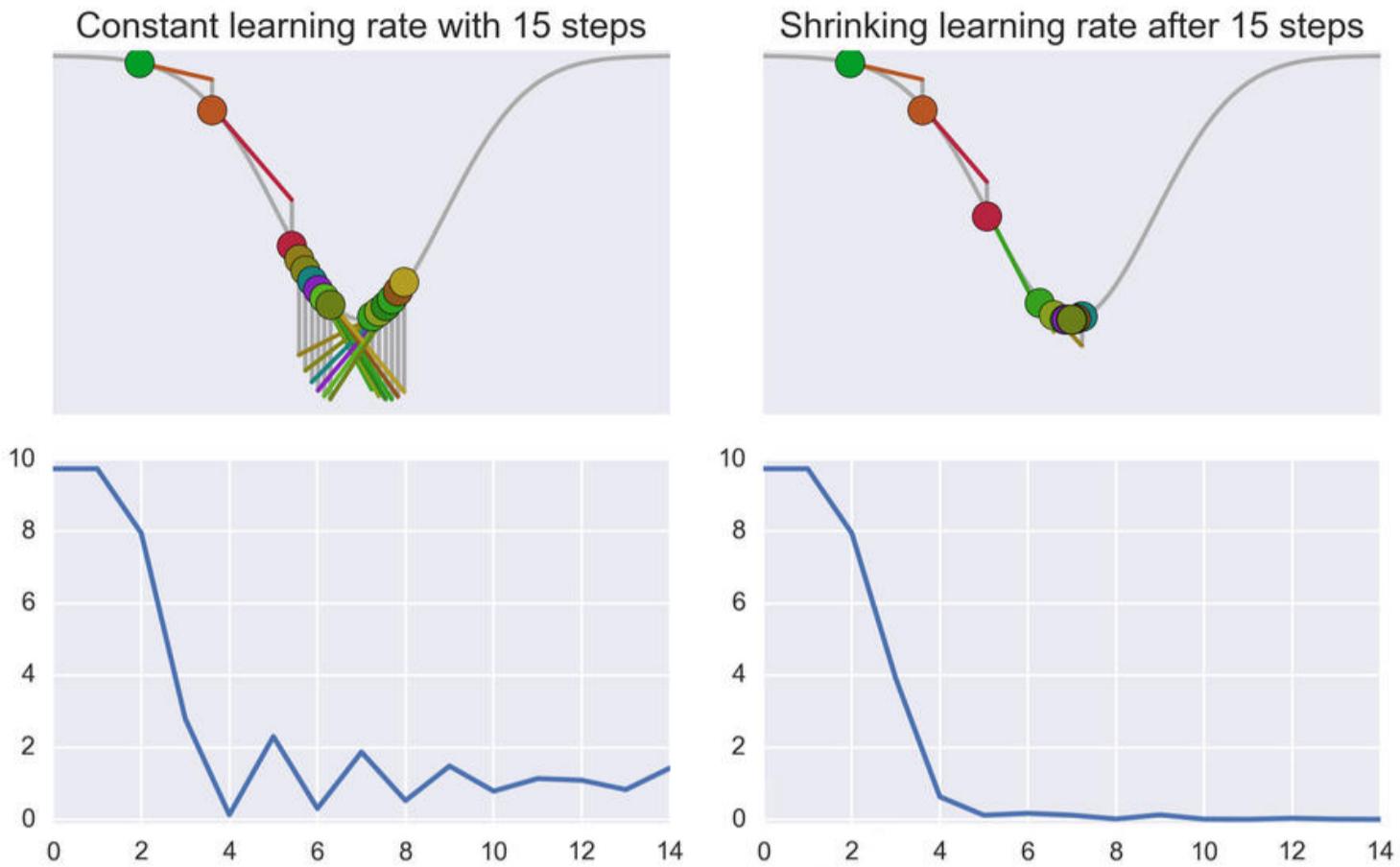


Figure 19.24: On the left is the constant step size from Figure 19.14, and on the right is the decaying step size from Figure 19.23. Notice how the shrinking learning rate helps us efficiently settle into the minimum of the valley.

The shrinking step size does a beautiful job of landing us in the bottom of the bowl and keeping us there. Even though there are 15 points in each diagram, we can really only make out the first 7 steps in the shrinking learning rate version. The rest are visually all sitting on top of one another at the bottom of the bowl. Numerically, we'd find that those points are probably moving a tiny bit, but less and less with each step.

19.3.3 Decay Schedules

The decay technique is attractive, but it comes with some new challenges.

First, of course, we have to choose a value for the decay parameter.

Second, we might not want to apply the decay after every update. The particular way we choose to change the learning rate over time is called a **decay schedule** [Bengio12].

Decay schedules are usually expressed in epochs, rather than samples. That is, 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. This schedule is shown in Figure 19.25(a).

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 19.25(b) shows this approach, putting off the exponential decay schedule of Figure 19.25(a) for a few epochs.

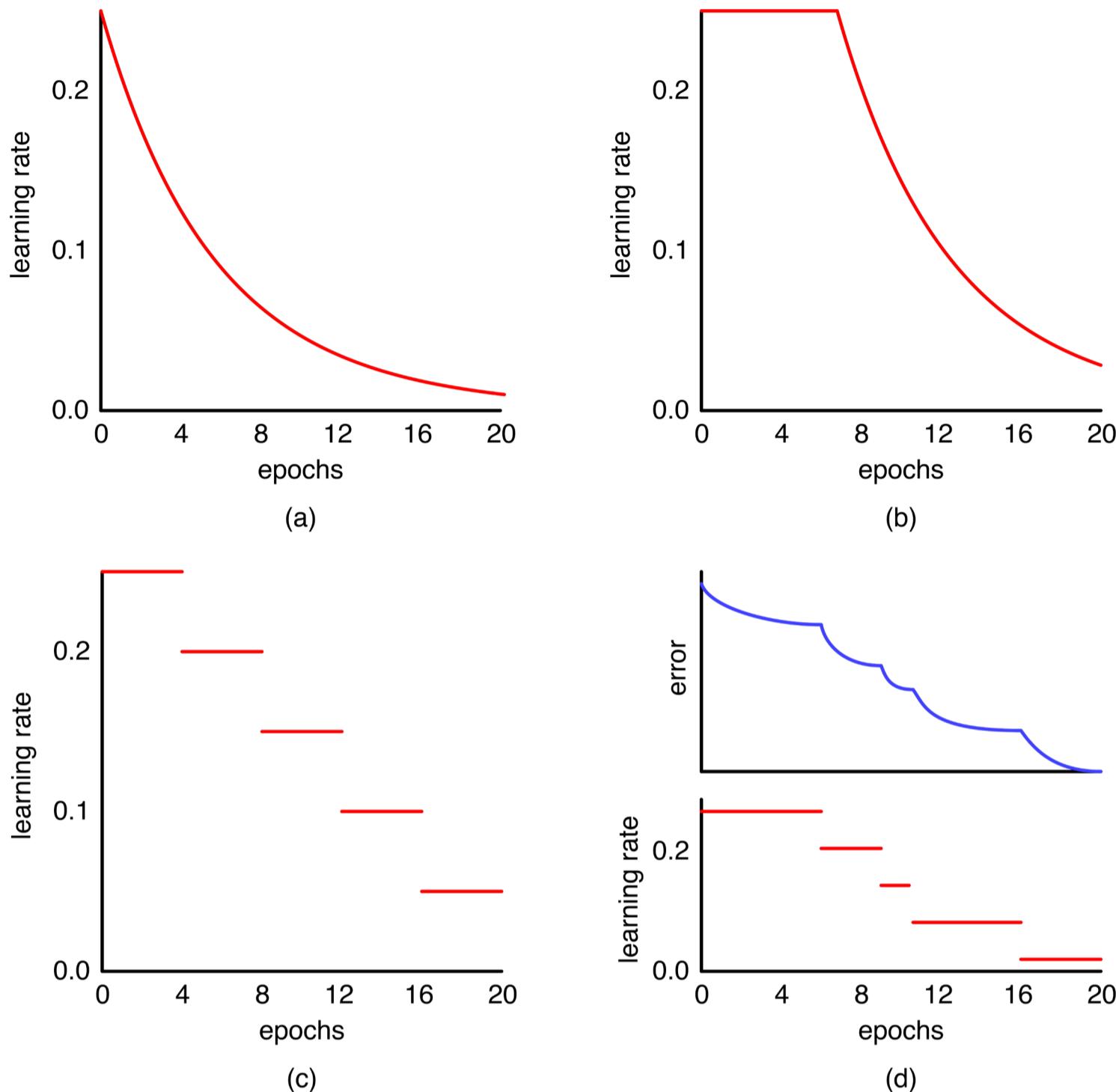


Figure 19.25: Decay schedules for reducing the size of the learning rate over time. (a) Exponential decay, where the learning rate is reduced after every epoch. (b) Delayed exponential decay. (c) Fixed-step decay, where the learning rate is reduced after every fixed number of epochs (here, 4). (d) Error-based decay, where the learning rate is reduced when the error stops dropping.

Another option is to apply the decay only every once in a while. The **interval decay** approach, shown in Figure 19.25(c), 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.

Or we might want 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, then we apply the decay so it can take smaller steps, and hopefully work its way into a deeper part of the error landscape. This approach is shown in Figure 19.25(d).

We can easily cook up lots 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.

We can even increase the learning rate if we want. The **bold driver** method looks at how the total loss is changing after each epoch [Orr99]. If the error is going down, then we *increase* the learning rate a little, say 1% to 5%. 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 [Darken92]. Sometimes we can treat these parameters as **hyperparameters** and search for the best values automatically, as discussed in Chapter 15.

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. The exponential, step, and bold driver approaches are popular [Karpathy16].

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 so that our steps become smaller, allowing 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 comes when we 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.

Below we'll look at some variations on gradient descent that address those ideas.

19.4 Updating Strategies

In Chapter 19 we mentioned three variations on using gradient descent to update a network's weights: batch gradient descent, stochastic gradient descent, and mini-batch gradient descent. In the following sections we'll see how these approaches perform on a small but real 2-class categorization problem.

Figure 19.26 shows our familiar pattern of two crescent moons. The points of each crescent belong to their own class. These 300 samples will be our reference data for the rest of this chapter.

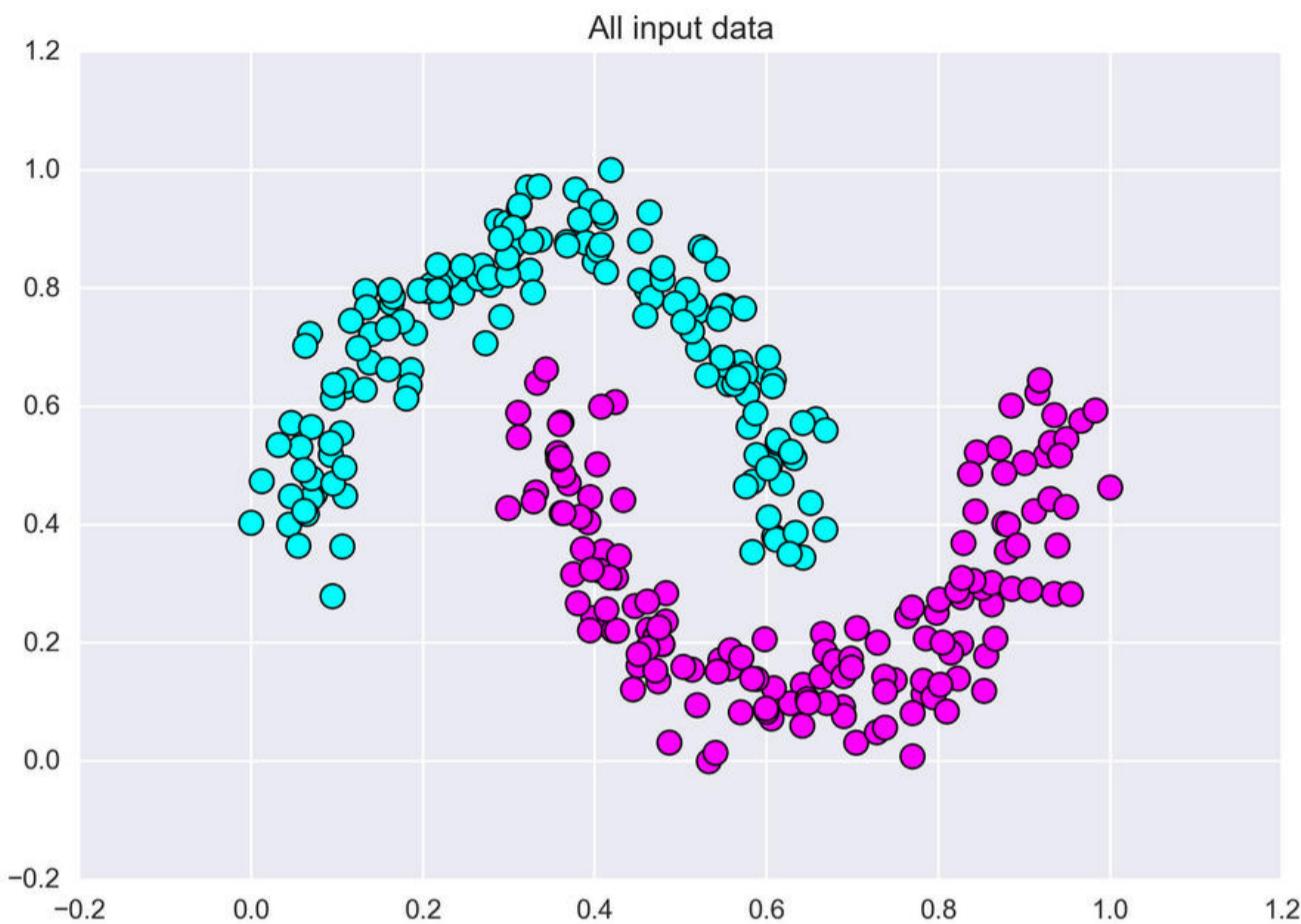


Figure 19.26: The data we'll use for the rest of this chapter. The 300 points form two classes of 150 points each.

To compare our networks, we'll train our networks until the error has reached zero, or seems to have stopped improving. We'll 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 will vary over a large range.

To classify our points we'll use a neural network with three hidden layers (of 12, 13, and 13 nodes), and an output layer of 2 nodes, giving us the likelihood for each of the two classes. Whichever class has the larger likelihood at the output is taken as our network's prediction. For consistency, when we need a constant learning rate we'll use a value of $\eta = 0.01$.

19.4.1 Batch Gradient Descent

Let's begin by updating the weights just once per epoch, after we've evaluated all the samples. This is called **batch gradient descent**.

The name references a style of computing used in the early days of centralized, mainframe processors. These systems could run only one program at a time. Each program, called a **job**, was typically saved with each line of the program on an individual punched card. When the cards were placed into a hopper and read in order, they were reconstructed in the computer to form a program. A human operator in charge of the computer would often gather together a bunch of programs, called a **batch**, and feed them to the computer in one giant stack. The system would then run the first job, produce output (typically on fan-fold paper), and then it would run the next job, and so on, without requiring a person to manage the process. The phrase **batch processing** is sometimes used today to refer to any technique where a sequence of operations, can be started and left to run [IBM10].

In our case, each update to the weights can be considered one of these operations. Instead of performing an update after each sample is evaluated, we wait for the whole collection of samples, or the whole epoch, or batch, to be evaluated. Then we update all of the weights once using the combined information from all the samples. Thus our batch is the entire training set, and we update the weights once after the entire set has been processed.

Figure 19.27 shows the error from a typical training run using batch gradient descent.

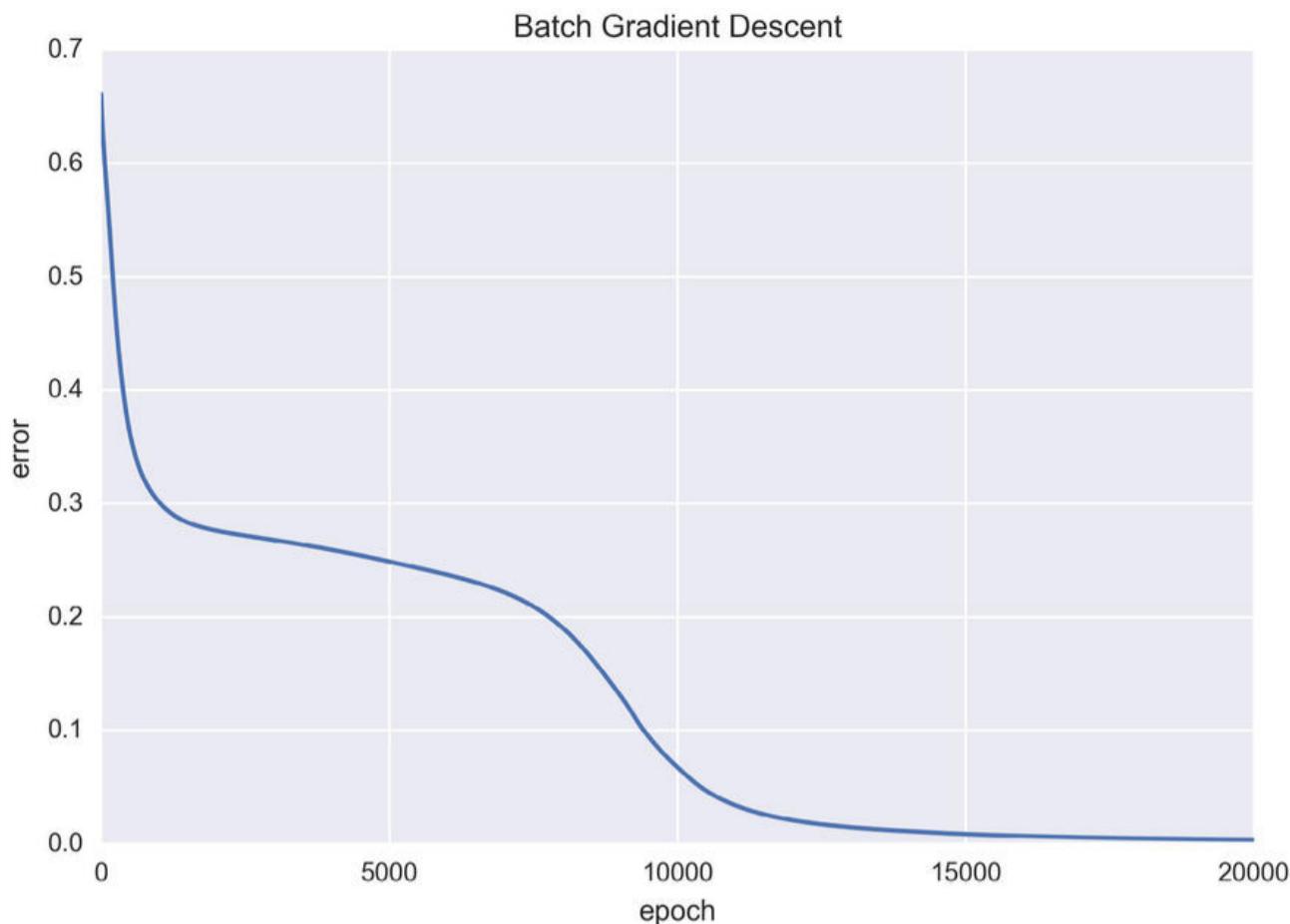


Figure 19.27: The error for a training run using batch gradient descent. The weights are updated once per epoch, using the average influence of all the samples in the batch. This graph shows 20,000 epochs of training.

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 error becomes much more shallow. The 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 we're looking at 20,000 epochs, which is a huge amount of training. To confirm that impression, let's zoom in on the first 400 epochs, shown in Figure 19.28.



Figure 19.28: A close-up of the first 400 epochs of batch gradient descent shown in Figure 19.27.

It seems that batch gradient descent really is moving smoothly. That makes sense, because it's using the average from all the samples on each update.

Batch gradient descent produces a good-looking error curve that is smooth and eventually finds its way to 0.

Batch gradient descent 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 or even crippling. This can be a problem in some real situations when we work with enormous datasets of many millions 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.

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 say that it's "offline" because all of the data is already available at the start of training. We'll see "online" methods later, where more data can arrive even while training. So we can imagine disconnecting the computer from all networks, and if it's running an offline algorithm, it will still be able to learn from all of our training data.

19.4.2 Stochastic Gradient Descent (SGD)

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." That word is in there because the samples arrive in a random order, so we can't predict how the weights are going to change from one sample to the next.

Each sample pushes the weights around a little. Since our dataset has 300 samples, that means we'll 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 won't see this small-scale wiggling. But it's still visible even epoch by epoch.

Figure 19.29 shows the error of our network learning from this data using SGD.

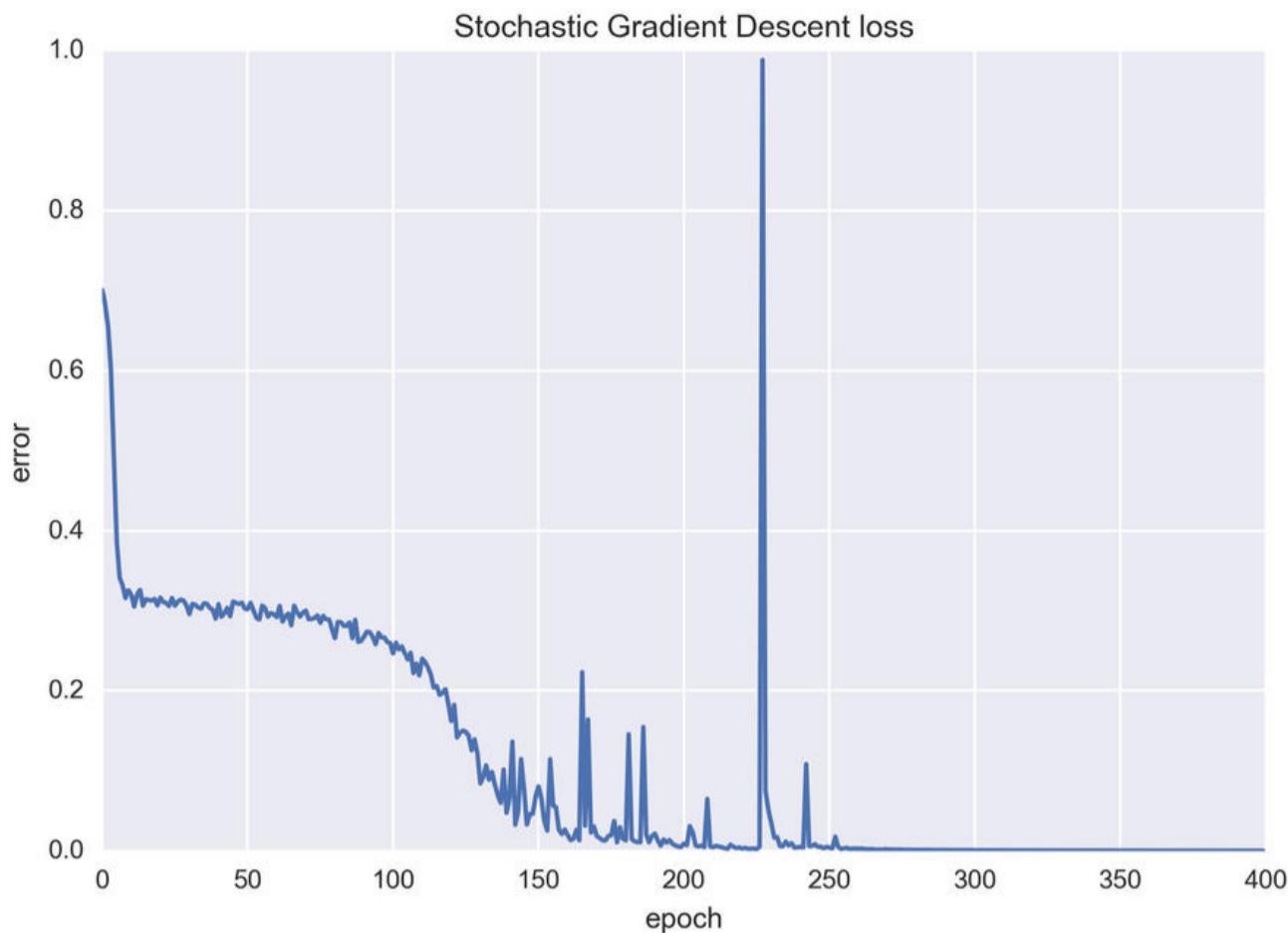


Figure 19.29: Stochastic Gradient Descent, or SGD. The plot shows the error after every epoch when we learn the data of Figure 19.26 by updating the weights after every sample. This graph shows 400 epochs of training.

The graph has the same general shape as that for batch gradient descent, which is sensible 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 categorization for almost every sample to being dead wrong on almost every sample, and then back to being right again (though that took a few epochs, as shown by the small curve to the right of the spike). If we were watching the errors as learning progressed, we might stop the training session right there. If we were using an automatic algorithm to watch the error, it might also stop it there. Yet just a few epochs after that spike we're back to nearly 0. The algorithm has definitely earned having 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 19.29 after 400 epochs, since the curve stayed at 0 from then on. Compare this to the roughly 20,000 epochs required by batch gradient descent in Figure 19.27. This increase in efficiency over batch gradient descent is typical [Ruder16].

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, 6 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 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 19.29. This is both good and bad. The good side of this is that SGD can jump from one region of the error surface to another as it searches for minima. But the bad side is that SGD can overshoot a deep minimum, and spend its time bouncing around inside of some other minimum with a larger error. Reducing the learning rate over time will definitely help the jumping problem, but the searching will still be noisy.

Noise in the error curve can be a problem, because it makes it hard for us to know when the system is learning or not, and whether it's overfitting or not. We have to consider a number of epochs to get a good average for the error. That can consume a lot of time, and we might only know that we've overshot the mark long after it happened.

19.4.3 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**. Here, we update the weights after some fixed number of samples have been evaluated. This number is almost always considerably smaller than the batch size (that is, 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 chosen to fully use the parallel capabilities of our GPU, if we have one. But that's just for speed purposes. We can use any size of mini-batch that we like.

The results of using a mini-batch of 32 samples is shown in Figure 19.30.

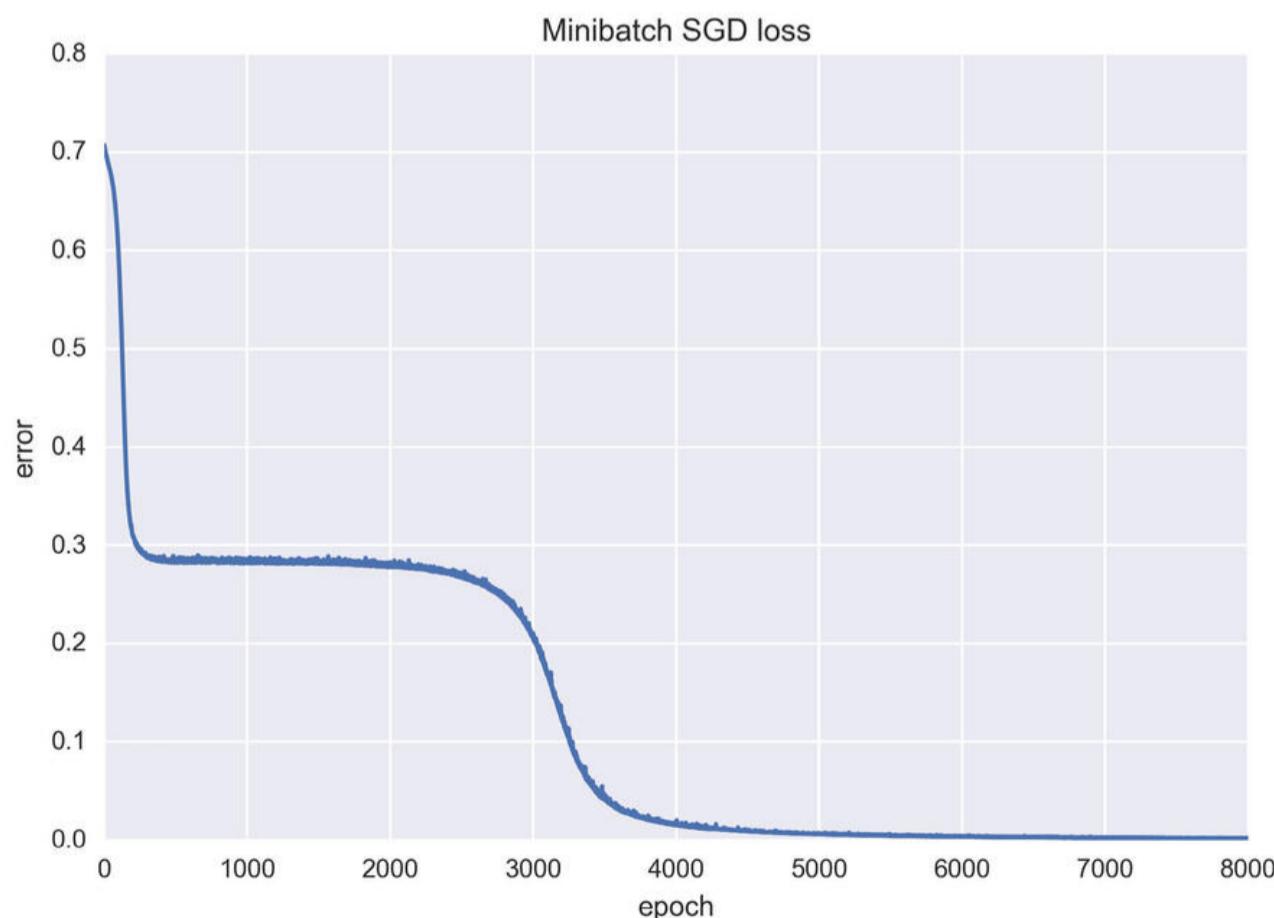


Figure 19.30: Mini-batch Gradient Descent. We're using a mini-batch size of 32 samples. We hit about zero error after about 5000 epochs.

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 5000 epochs, between the 400 needed by SGD and the 20,000 of batch gradient descent. Figure 19.31 shows a close-up of the first 400 steps.

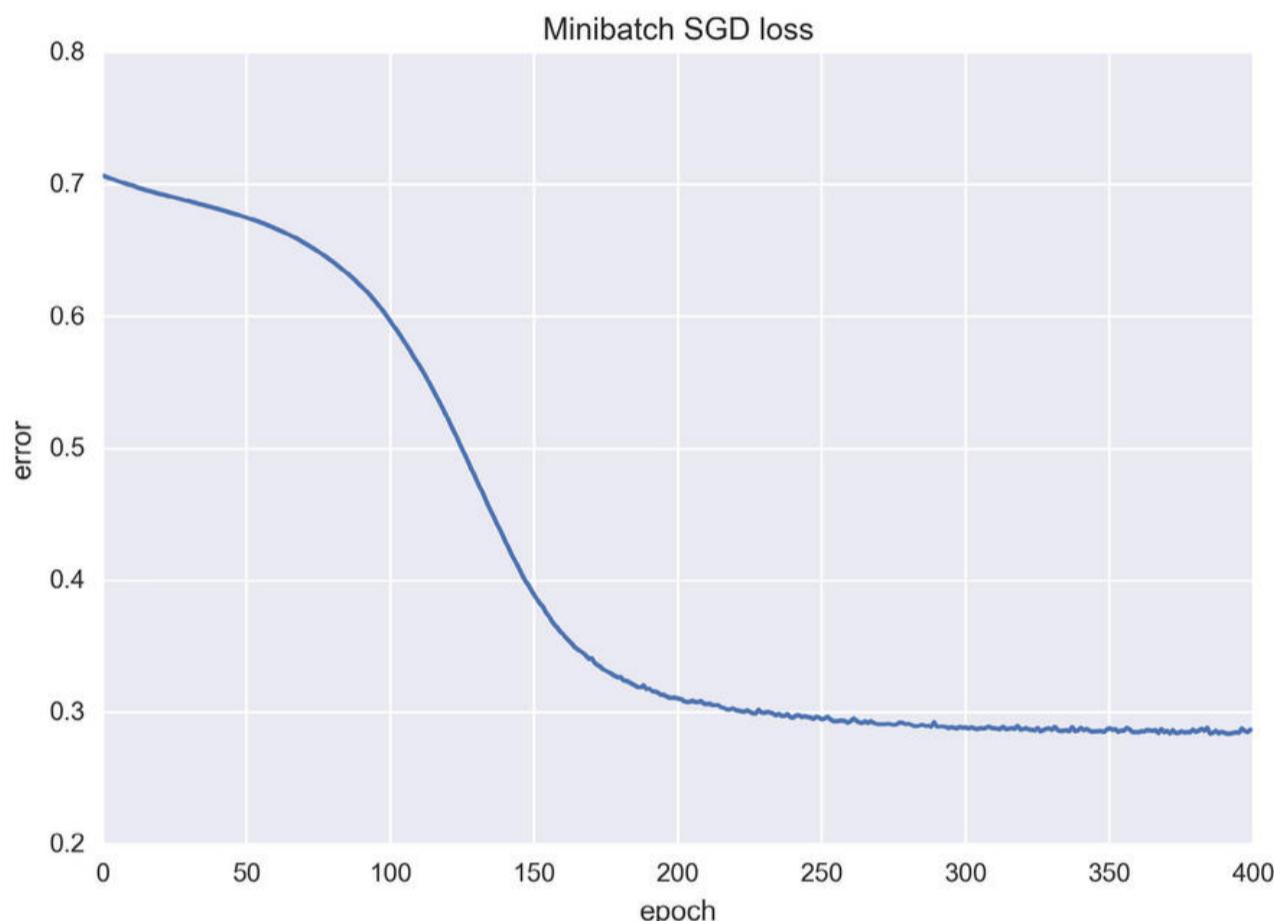


Figure 19.31: A close-up of the first 400 epochs of Figure 19.30, showing the deep plunge at the very beginning of training. Note the vertical scale which runs from 0.2 to 0.8.

How many updates did mini-batch SGD perform? We have 300 samples, and we used a mini-batch of size 32, so there were 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 data sets, we often have a partial mini-batch at the end). So 10 updates per epoch, times 5000 epochs, gives us 50,000 updates. This too is 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 what's happening with 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. So 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 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 [Ruder16].

19.5 Gradient Descent Variations

Let's review some of the challenges of mini-batch gradient descent, and ways to address them (following convention, from here on we'll often refer to mini-batch gradient descent as SGD). The organization of this section is inspired by [Ruder16].

Mini-batch gradient descent is a good algorithm, but it's not perfect. First of all, we have to specify what value of the learning rate η we want to use, and that's notoriously hard to pick ahead of time. As we've seen, a value that's too low can result in long learning times and getting stuck in shallow local minima, but a value that's too high can cause us to overshoot deep local minima, and then get stuck at 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 that schedule and its parameters.

Then we have to pick the size of the mini-batch. This is usually not really an issue, as we just use whatever value produces calculations that are most closely matched to the structure of our hardware.

Another issue is to consider that right now we're updating all the weights with a "one update rate fits all" approach. Maybe that's not the best way to go. Maybe we could 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.

Another way we might improve things is to keep in mind that, as we noted above, sometimes the region in a saddle can be shallow in all directions, so locally, it's almost (but not quite) a plateau. This can slow our progress to an excruciatingly crawl. Since we know that there will usually be plenty of saddles in our error landscape [Dauphin14], it would be nice if there was a way to get un-stuck in these situations, or better yet, 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. 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.

19.5.1 Momentum

If we think of our error surface as a landscape, then we saw in Chapter 5 that we can picture each step as a ball that's rolling down the surface, looking for the lowest point. In our 2D categorization example, the position of the ball is given by the weights, and the ball's height is given by the error at those values.

Figure 19.32 repeats a figure from Chapter 5 that shows an example of this way of thinking about the training process.

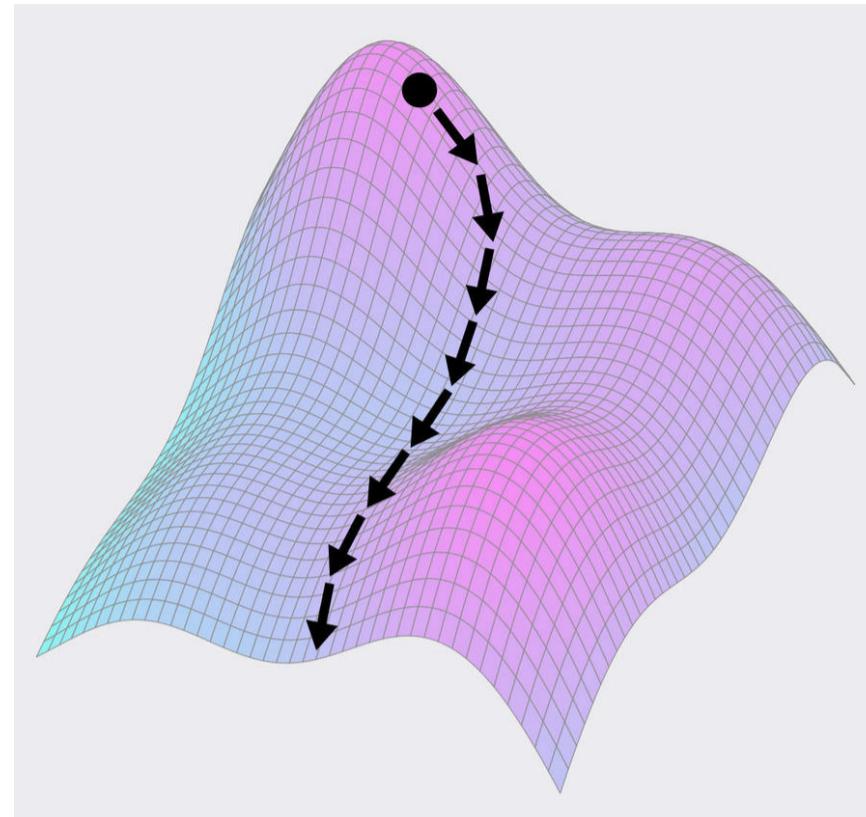


Figure 19.32: Rolling a ball down an error surface. This is a repeat of a figure from Chapter 5.

We know from the physical world that a real ball rolling down a hill in this way will have some **inertia**, which describes its resistance to a change in its motion. That is, 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 (such as friction, or someone pushing on the ball, or a change in the slope of the error landscape).

A different but related idea is the ball's **momentum**, which is a bit more abstract from a physical point of view. We sometimes casually conflate the two terms in everyday speech, treating “momentum” as an object's tendency to continue moving in the same speed and direction. We'll use that casual interpretation here.

This idea is what keeps the ball in Figure 19.32 moving across the plateau after it's come down from the peak and passed out of the saddle. If the ball's motion was given strictly by the gradient, when it hit the plateau it would stop (or if it's a near-plateau, the ball would slow to a crawl). But the ball's momentum (or more properly, its inertia) keeps it rolling onwards.

Suppose we're near the left side of Figure 19.33. As we roll down the hill, we'll reach the plateau starting at around -0.5 .

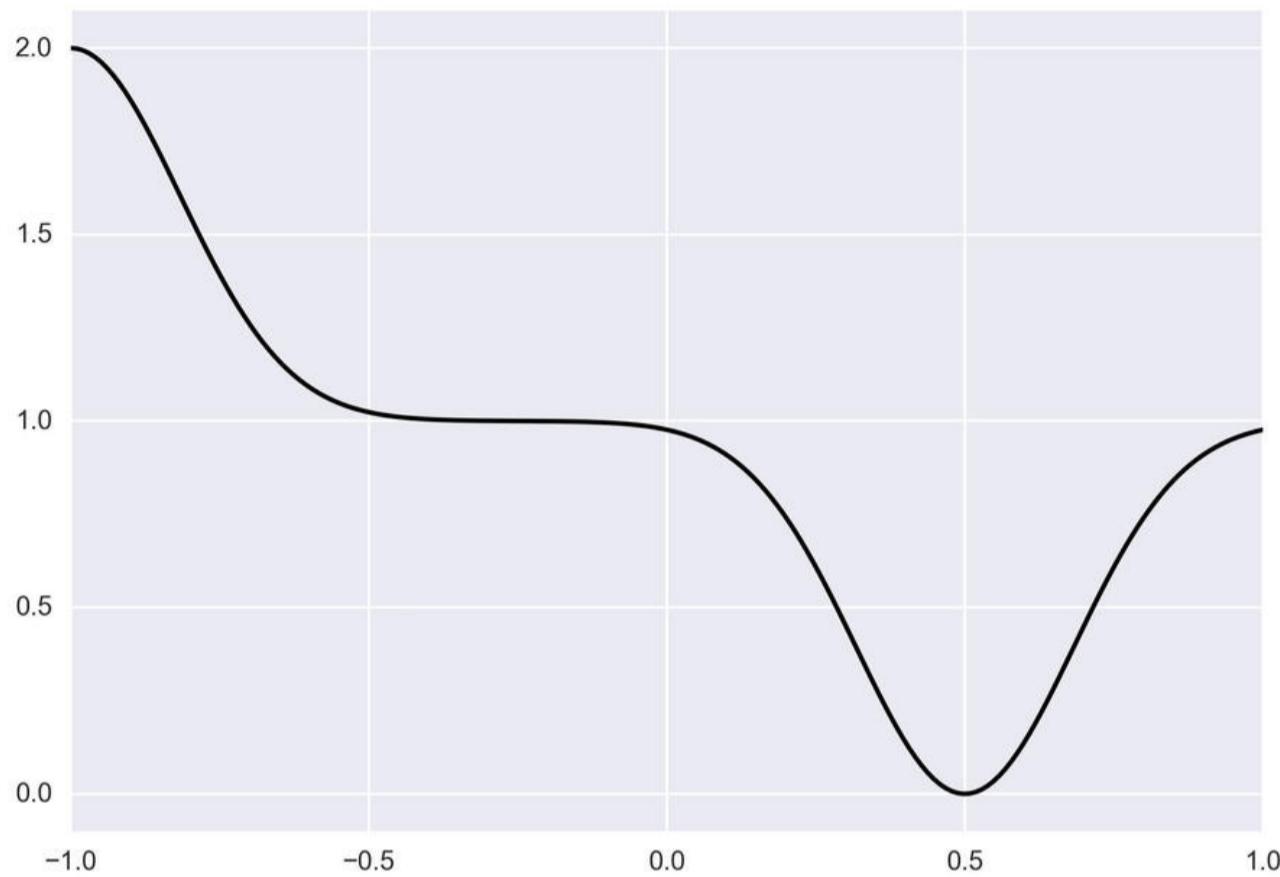


Figure 19.33: An error curve with a plateau between a hill and valley.

With regular gradient descent, we'd stop on the plateau since the gradient is zero, as shown in the left of Figure 19.34. But if we include some momentum, the ball will keep going for a while. It will slow down, but if we're lucky it will roll far enough to find the next valley.

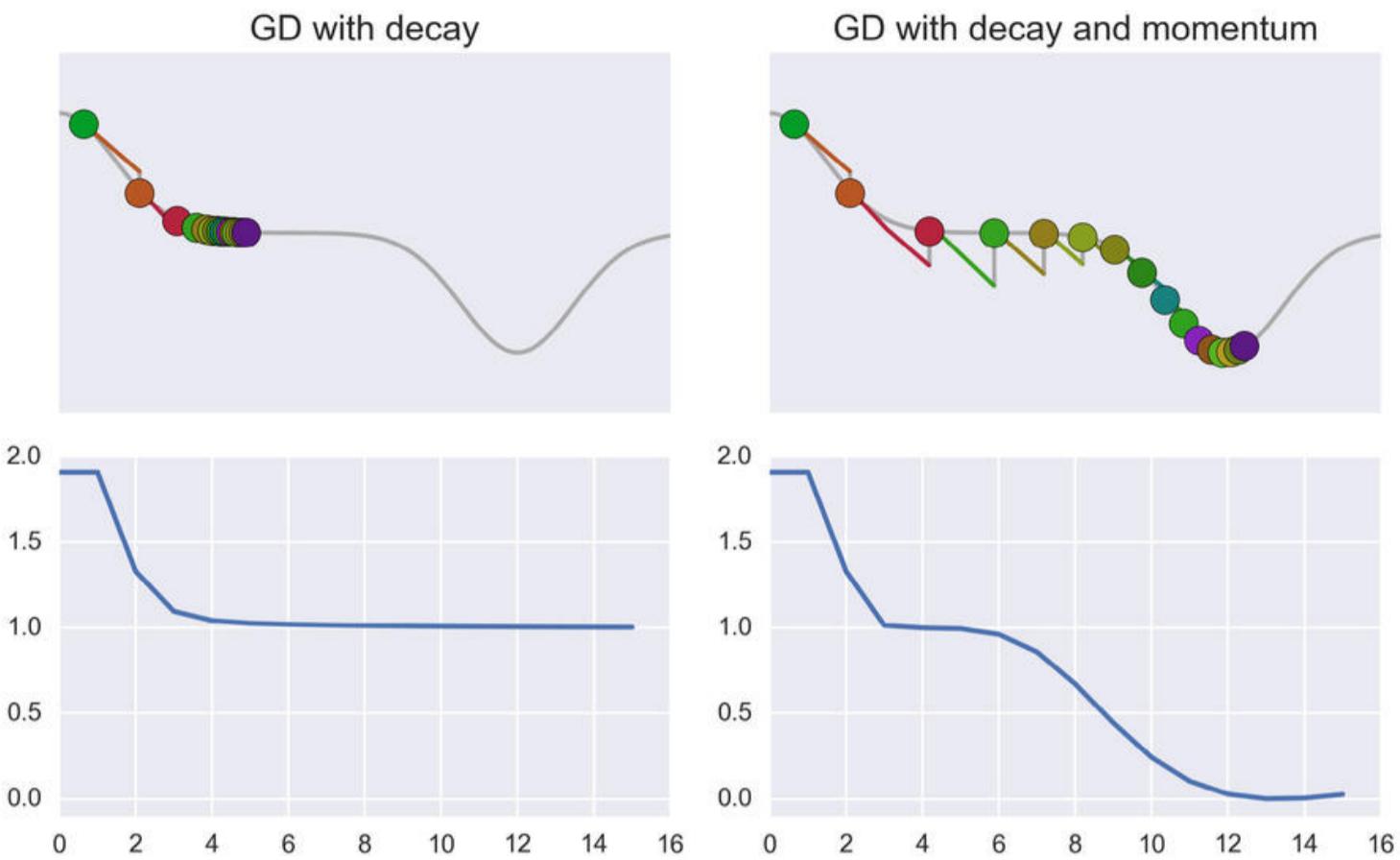


Figure 19.34: Gradient descent on the error curve of Figure 19.33. Both figures show 16 steps. Left: Gradient descent with decay. When we hit the plateau, the gradient goes to zero and progress stops. Right: Gradient descent with decay and momentum (the diagonal lines are discussed below). The ball rolls for a little while over the plateau, losing energy and slowing. But it reaches the next valley before it comes to a halt, giving it the chance to drop into the valley and get down to the minimum error.

If we could get our learning process to use this kind of momentum, that would be great, because it would help us cross over plateaus like the one in the figure.

The technique of **momentum gradient descent** [Qian99] is based on this idea. For each step, once we calculate how much we want each weight to change, we then add in a small amount of its change from the *previous* step. So if the change on a given step is 0, or nearly 0, but we had some larger change on the last step, we'll use some of that prior motion now, pushing us along over the plateau.

Figure 19.35 shows the idea visually. We suppose that some weight had a value A, which we updated to become value B. We now want to find the next value for the weight, which we'll call C. To find C, we find the change that we applied to point A that, after moving to the error surface, brought us to point B. This is the momentum, labeled m .

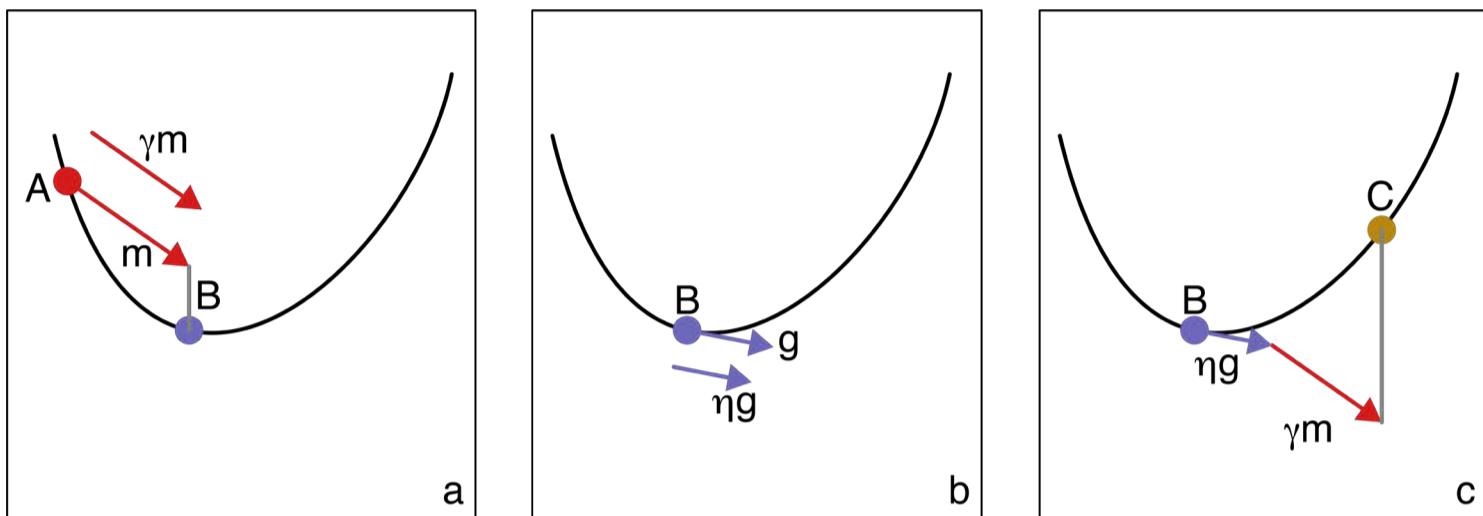


Figure 19.35: Finding the step for gradient descent with momentum. (a) When we're at point B, we look back to the previous point A and find the “momentum,” or the change we applied to point A. We call this “ m .” This momentum is scaled by γ , a number from 0 to 1, giving us the shorter arrow labeled γm . (b) As with basic gradient descent, we find the gradient at B and scale it by η . (c) Starting at point B, we add the scaled gradient ηg and the scaled momentum γm to give us the new point C.

We multiply the momentum m by a scaling factor usually referred to with the lower-case Greek letter γ (gamma). Sometimes this is called the **momentum scaling factor**. This is 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. To find point C, we add the scaled momentum γm and the scaled gradient ηg to B.

Let's see this in action. Figure 19.36 shows our symmetrical valley from before, and sequential steps of training using an exponential decay schedule and momentum. This is just like our sequence from Figure 19.21, but now the change for each step also includes momentum, or

a scaled version of the change from the previous step. We can see this by the two lines that emerge from each point (one for the gradient, the other for the momentum). This total then becomes the new change.

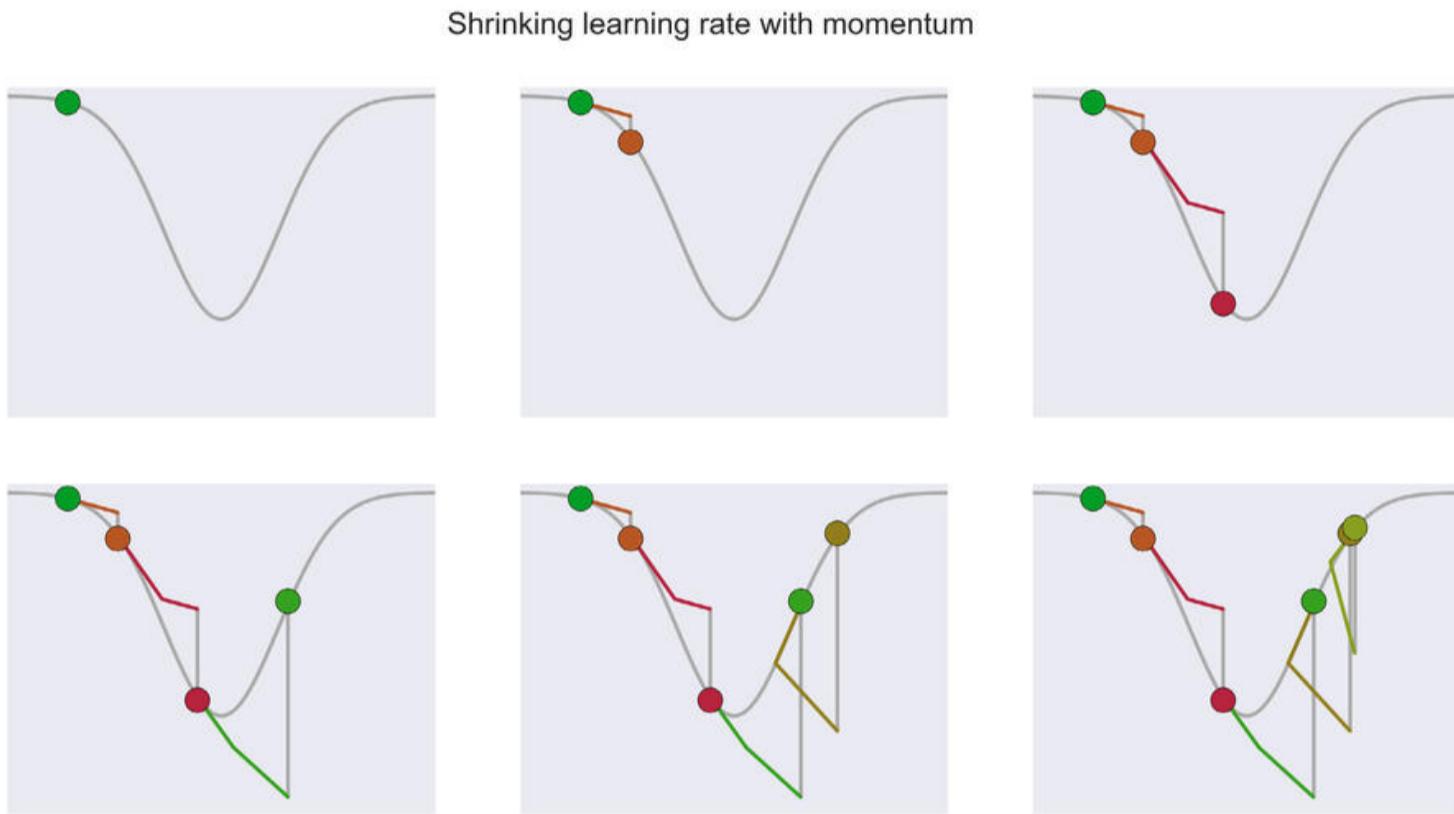


Figure 19.36: Learning with both an exponential decay schedule and momentum. The change at each step is given by the sum of the scaled gradient, shown by the first line leaving a point, and a little bit of the previous change, shown by the line following the first one.

So 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.

The 6th step in the grid, along with the error at each point along the way is shown in Figure 19.37.

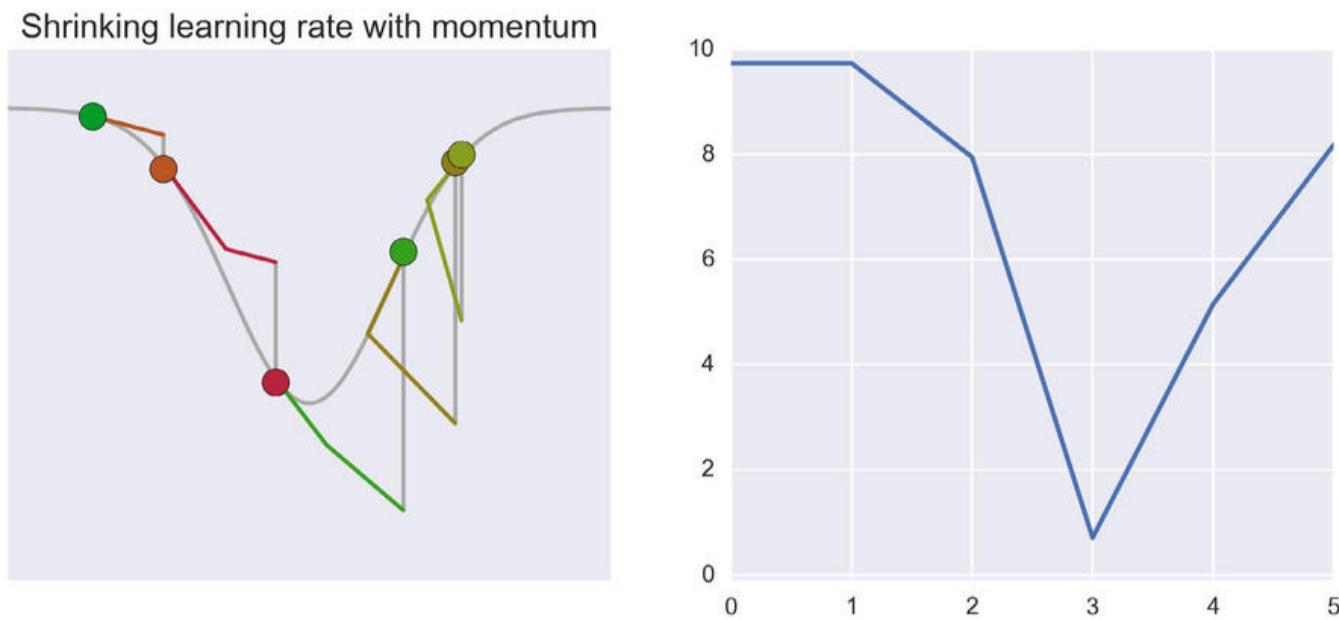


Figure 19.37: The final step in Figure 19.36, along with the error for each point.

An interesting thing happened here: as the ball “rolled up” the right side of the valley, even though the gradients pointed downwards, it continued to roll upwards. That’s just what we’d expect of a real ball. We can see it slowing down, and then eventually it will come back down the slope, overshooting the bottom but by less than before, then slowing and coming back down again, and so on, until it finally settles into the bottom of the bowl.

If we used too much momentum we could fly right up the other side and out of the bowl altogether, but too little momentum and we might not get across the plateaus we encounter along the way. Figure 19.38 shows our error curve with a plateau from Figure 19.33. Here we’re using a value of γ to scale the momentum such that we get through the plateau, but we can still settle into the minimum at the bottom of the bowl.

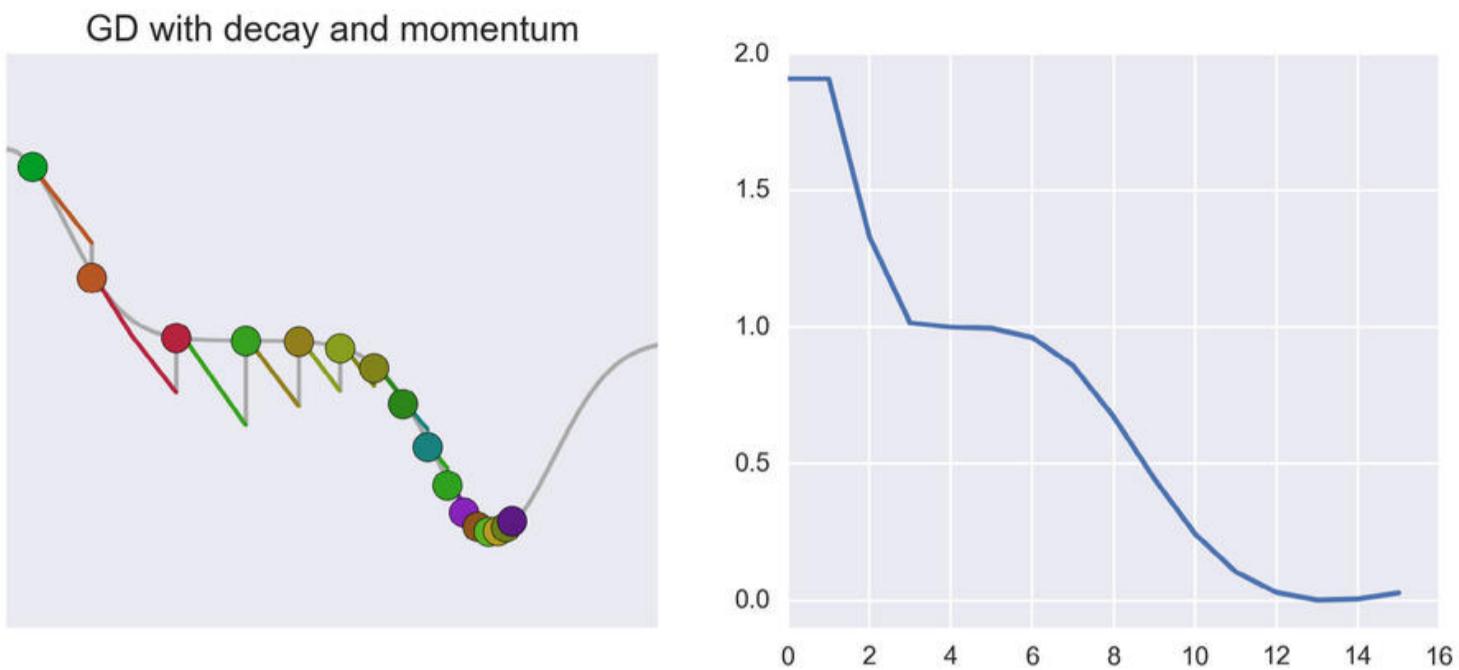


Figure 19.38: Using enough momentum to cross a plateau, but not so much that we don't settle nicely into the bottom of the minimum. We can see a little overshooting of the minimum in the error curve, but later iterations will move back towards the minimum.

Finding the right amount of momentum to use is another task where we need to use our experience and intuition, along with experiments 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.

So 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”) GD. 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 the above figures, we set gamma to 0.7 to better illustrate the process.

Figure 19.39 shows the result of learning with learning rate decay and momentum for 15 points. 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.

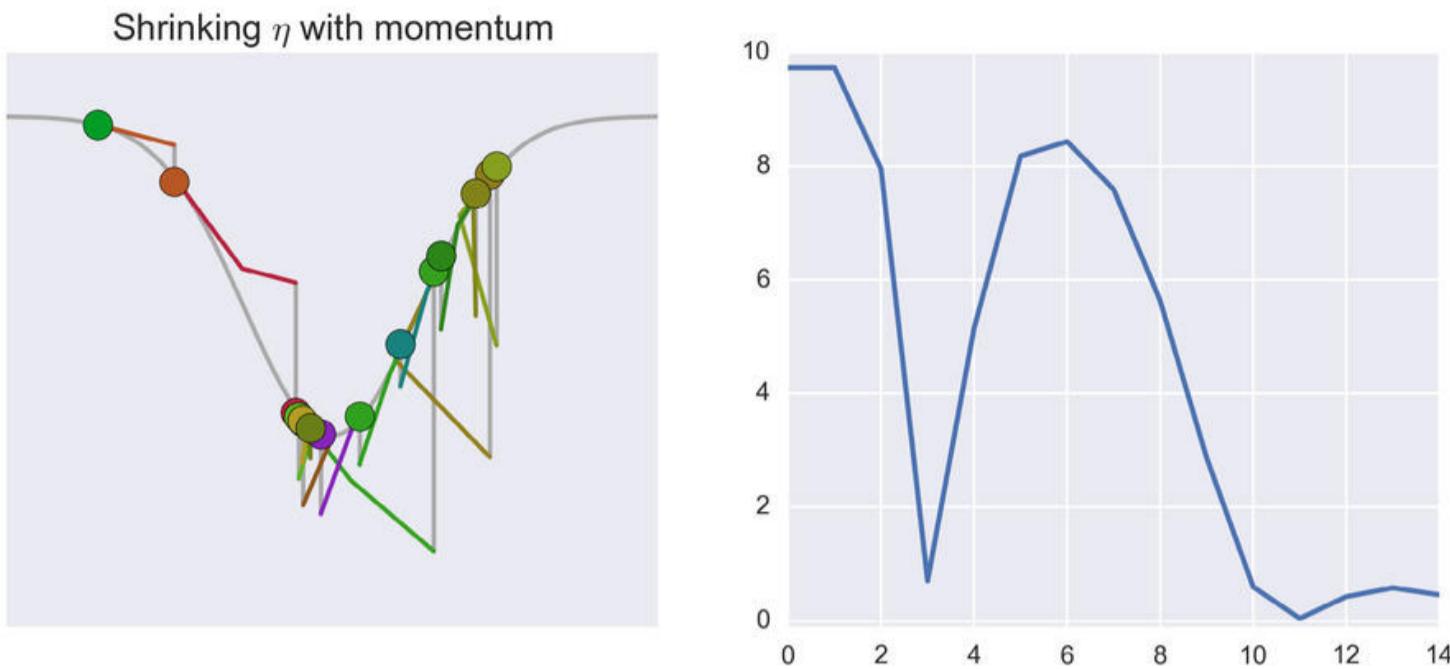


Figure 19.39: Learning with momentum and a decaying learning rate for 15 points.

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 there. There are risks to using too high a value of momentum, as we saw above, but used in moderation it's usually helpful.

Figure 19.40 shows the error for a training run on our dataset of Figure 19.26 consisting of two crescent moons. We're using mini-batch gradient descent with momentum. It's noisier than the mini-batch curve of Figure 19.30 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 19.30 for our data took about 5000 epochs to reach zero. With momentum, we get there in a little over 600 epochs.

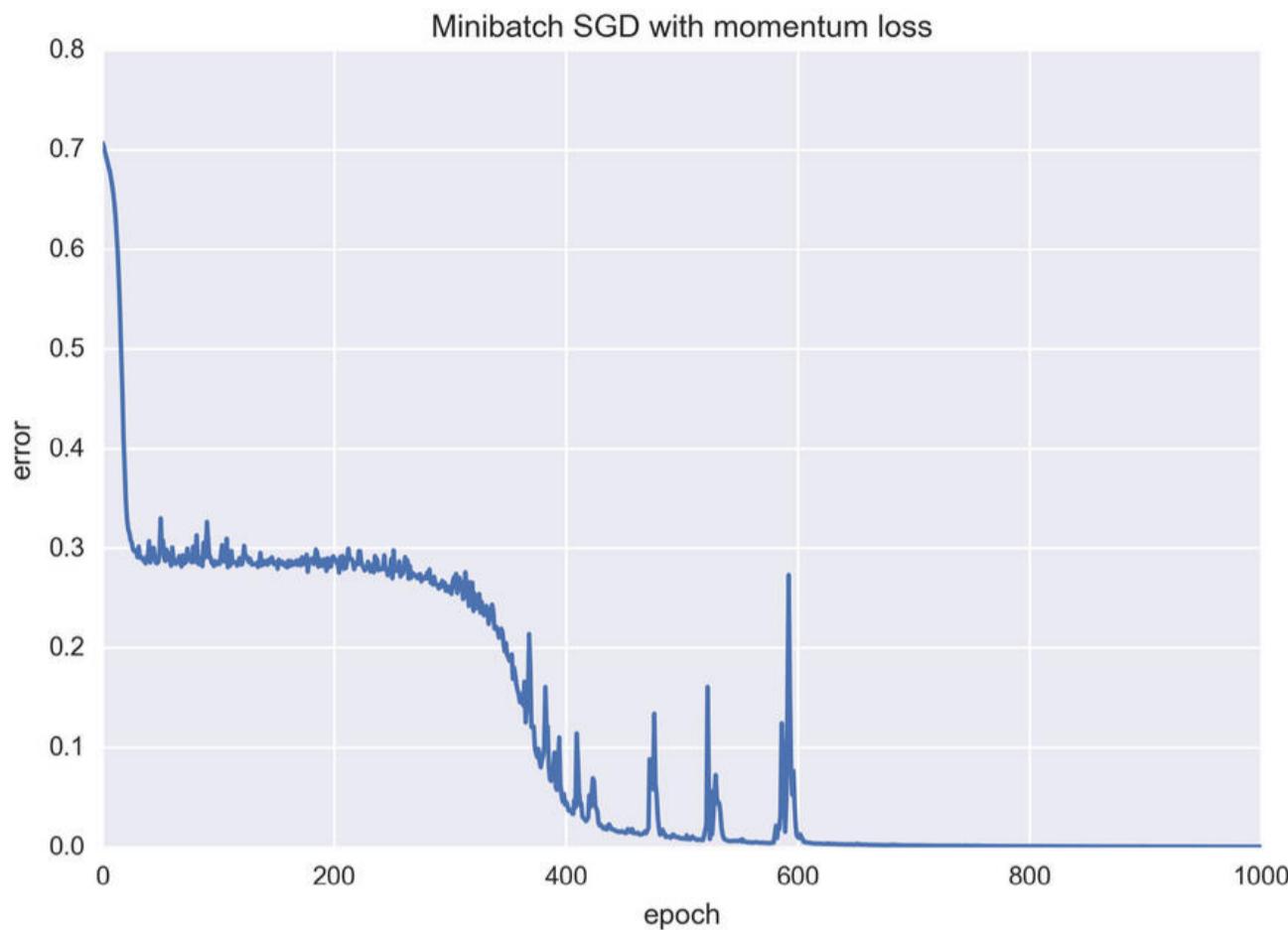


Figure 19.40: Error curve for training with our two-crescent data using mini-batch gradient descent with momentum. We got to zero error in a little more than 600 epochs.

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 treat this as a *hyperparameter* and search for the value that gives us the best results.

19.5.2 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 the gradient at the location where we currently are, we use the gradient at the location 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'll find at our estimated next position will be 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'll overshoot the bottom and end up somewhere on the other wall. As we saw before, momentum will carry 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 include some of the gradient at that point in our calculations *now*. So instead of moving so far to the right, that leftward push from the future will cause us to move a little less distance, so we won't overshoot so far and we'll 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'll take a larger step now. If the next move is going to move us backwards, we'll take a smaller step.

Let's break this down into steps so we don't get mixed up between estimates and realities. Figure 19.41 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 Figure 19.41(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 .

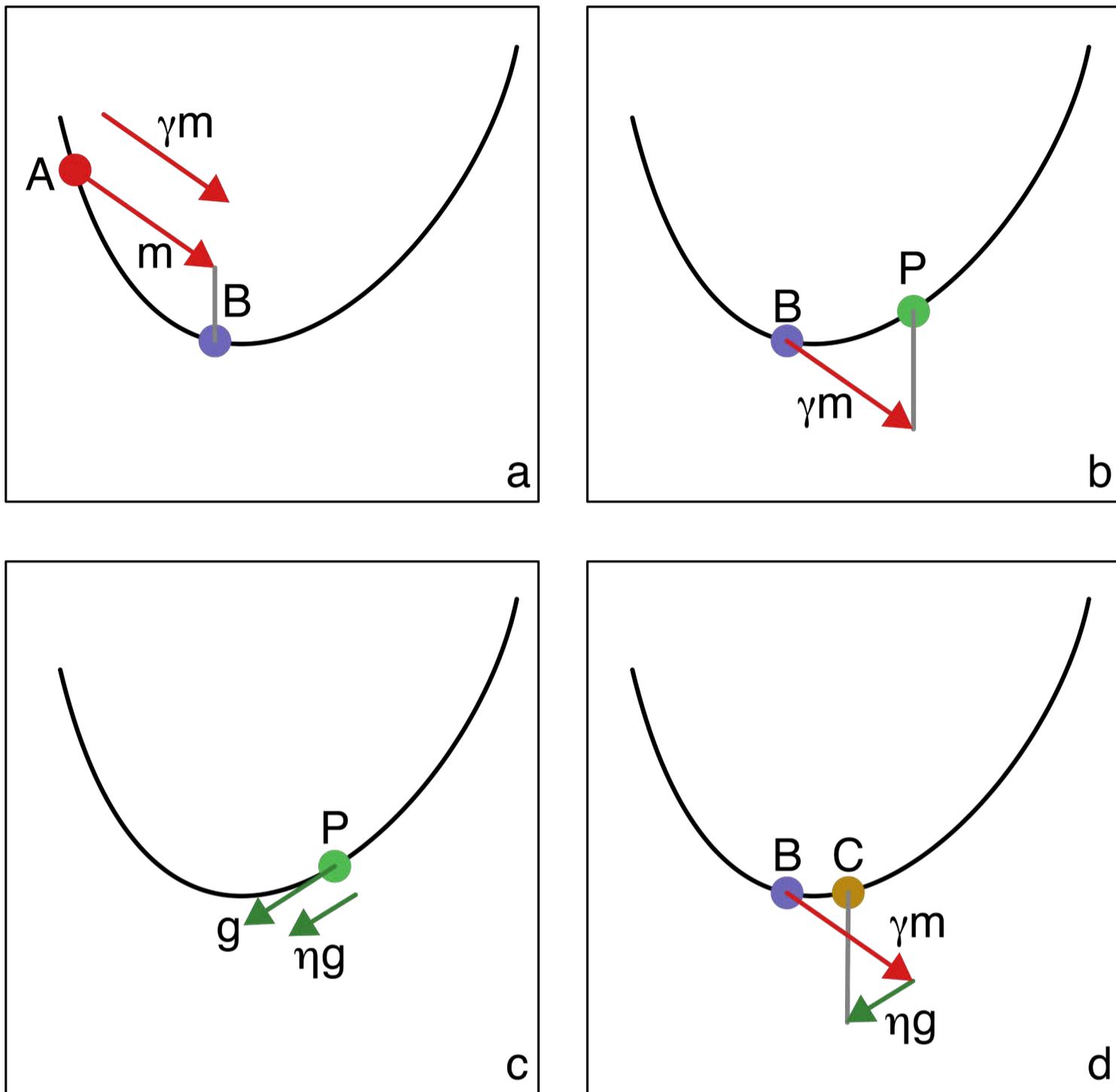


Figure 19.41: Gradient descent with Nesterov momentum. There are four steps. (a) As with gradient descent with momentum, when we're at point B we look back to the previous point A to find the change we computed at that step. That's the momentum m , which we scale by γ to get γm . (b) We add that scaled momentum to B to give us a new “predicted” point P. (c) We find the gradient at P. As usual, we'll call that g , and scale it by the learning rate η to get the smaller arrow ηg . (d) Starting at point B, we add the scaled gradient and scaled momentum to give us the new point C.

Now comes the new part, shown Figure 19.41(b). Rather than finding the gradient at B, we'll add the scaled momentum to B right now to get the “predicted” point P. This is our guess for where we'd end up on the next step. As shown Figure 19.41(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 19.41(d) by adding the scaled momentum γm and the scaled gradient ηg to B.

This is kind of crazy, because 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 that the point C in Figure 19.41(d) is closer to the bottom of the bowl than it would have been with normal momentum. By looking into the future and seeing that we'd be on the other side of the valley, we were 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** [Nesterov83]. 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 from us.

The first six steps of using Nesterov momentum on our little test case of a valley are shown in Figure 19.42.

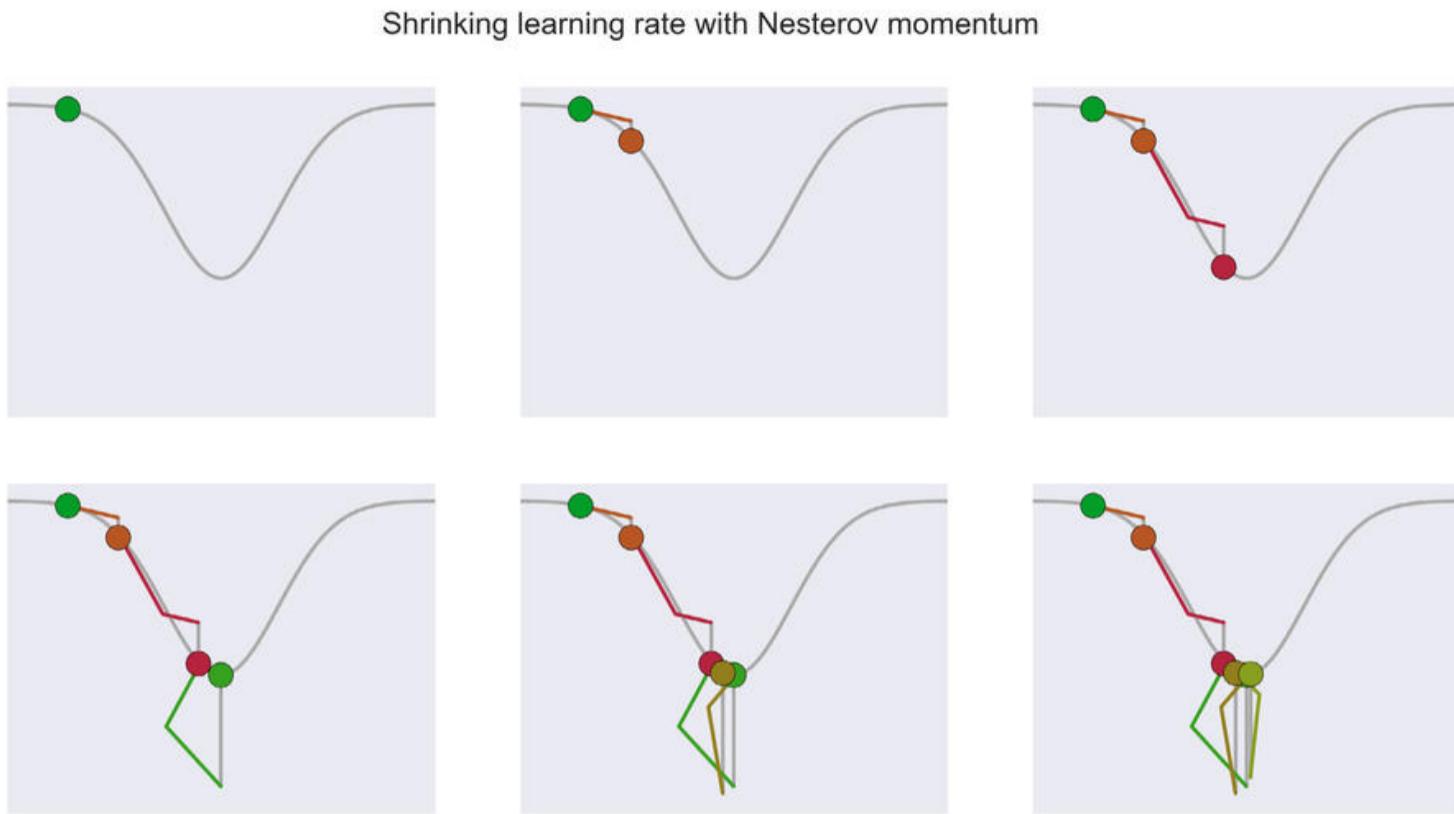


Figure 19.42: Using Nesterov momentum with learning rate decay to descend a valley. At each step we use the gradient of the point we predict we'd land on if we continued our current motion.

Figure 19.43 shows the bottom-right image from Figure 19.42 along with its error.

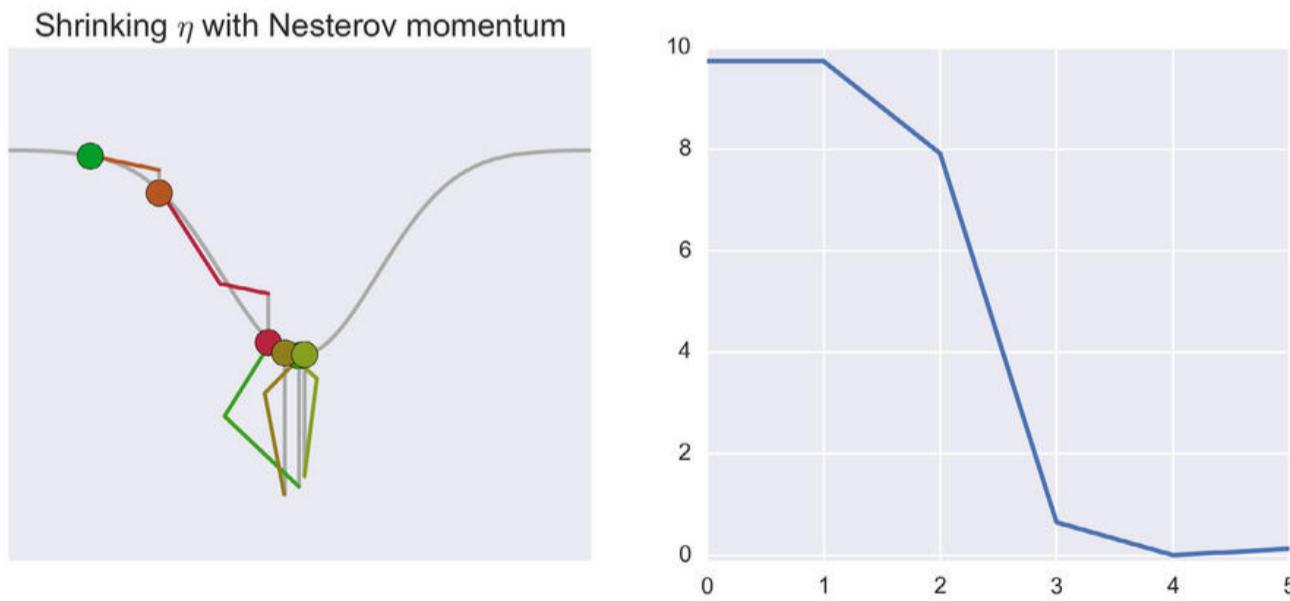


Figure 19.43: Learning with Nesterov momentum and learning rate decay. Left: The final image from Figure 19.42. Right: The error at each point along the way.

The result of this process after 15 points is shown in Figure 19.44.

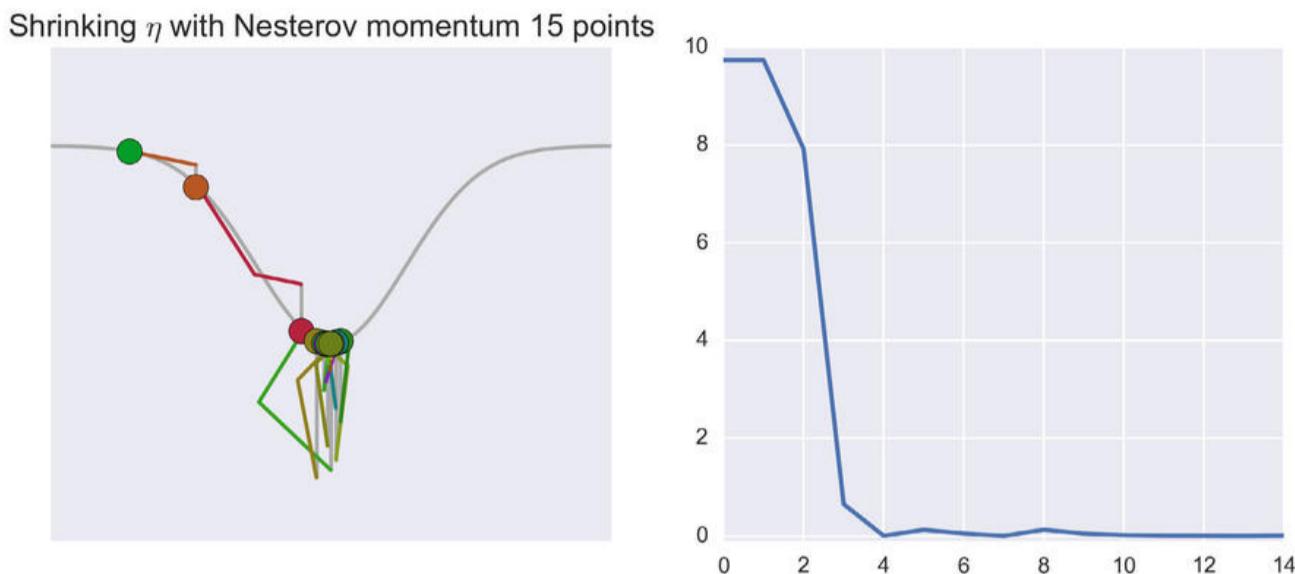


Figure 19.44: Running Nesterov momentum for 15 points. It finds the bottom of the valley in about 7 steps and then stays there.

The error curve for our standard test case using Nesterov momentum is shown in Figure 19.45. This uses the exact same model and parameters as the momentum-only results in Figure 19.40, but it's both less noisy and more efficient, getting down to about 0 error at roughly epoch 425.

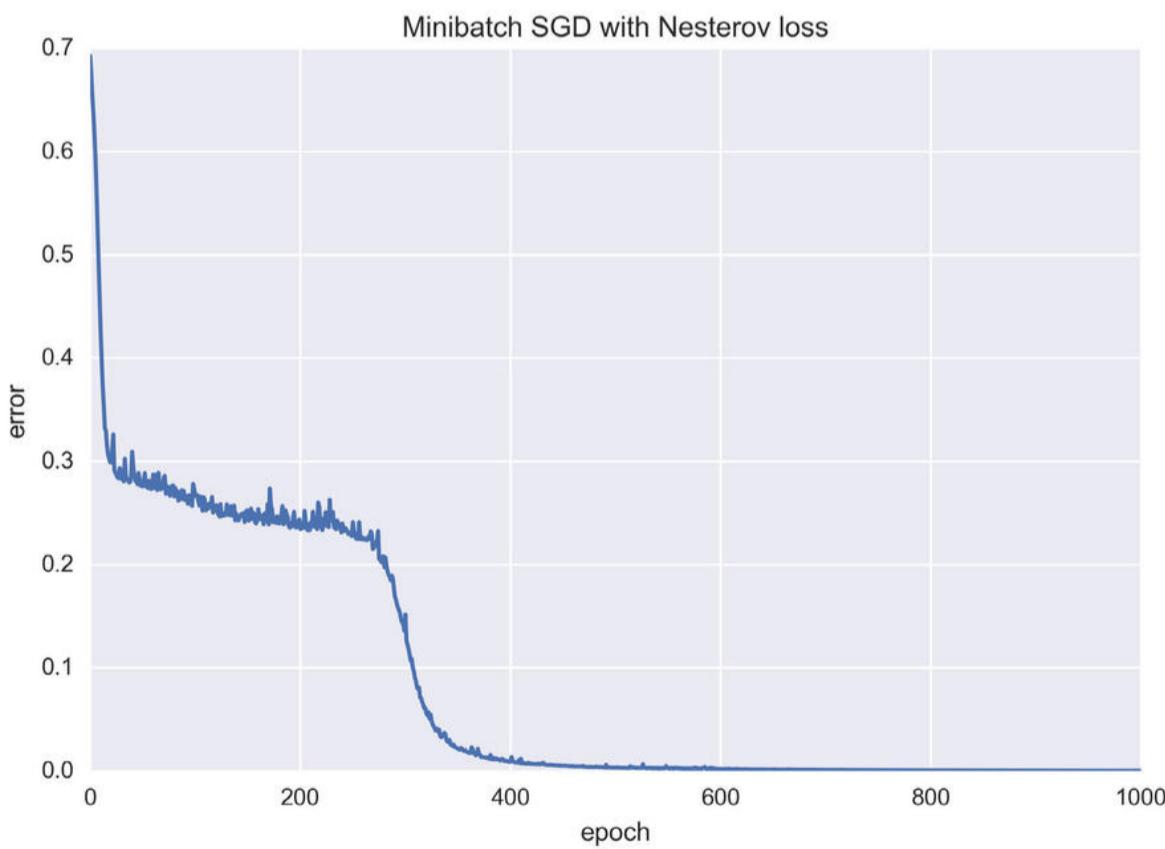


Figure 19.45: Error for mini-batch gradient descent with Nesterov momentum. The system reaches zero error around epoch 600. The graph shows 1000 epochs.

Any time we use momentum it's definitely worth considering Nesterov momentum instead. It requires no additional parameters from us, but usually learns more quickly and with less noise.

19.5.3 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.

There are several related algorithms that use this idea. Their names all begin with "Ada," standing for "adaptive."

We'll start with an algorithm called **Adagrad**, which is short for **Adaptive Gradient Learning** [Duchi11]. As the name implies, the algorithm adapts (or changes) the size of the gradient for each weight.

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 (that is, multiplies it by itself), and adds that into a running sum. 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.

Since the sum just goes larger and larger over time, and we divide each change by that growing sum, 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 19.46 shows the performance of Adagrad on our test data.

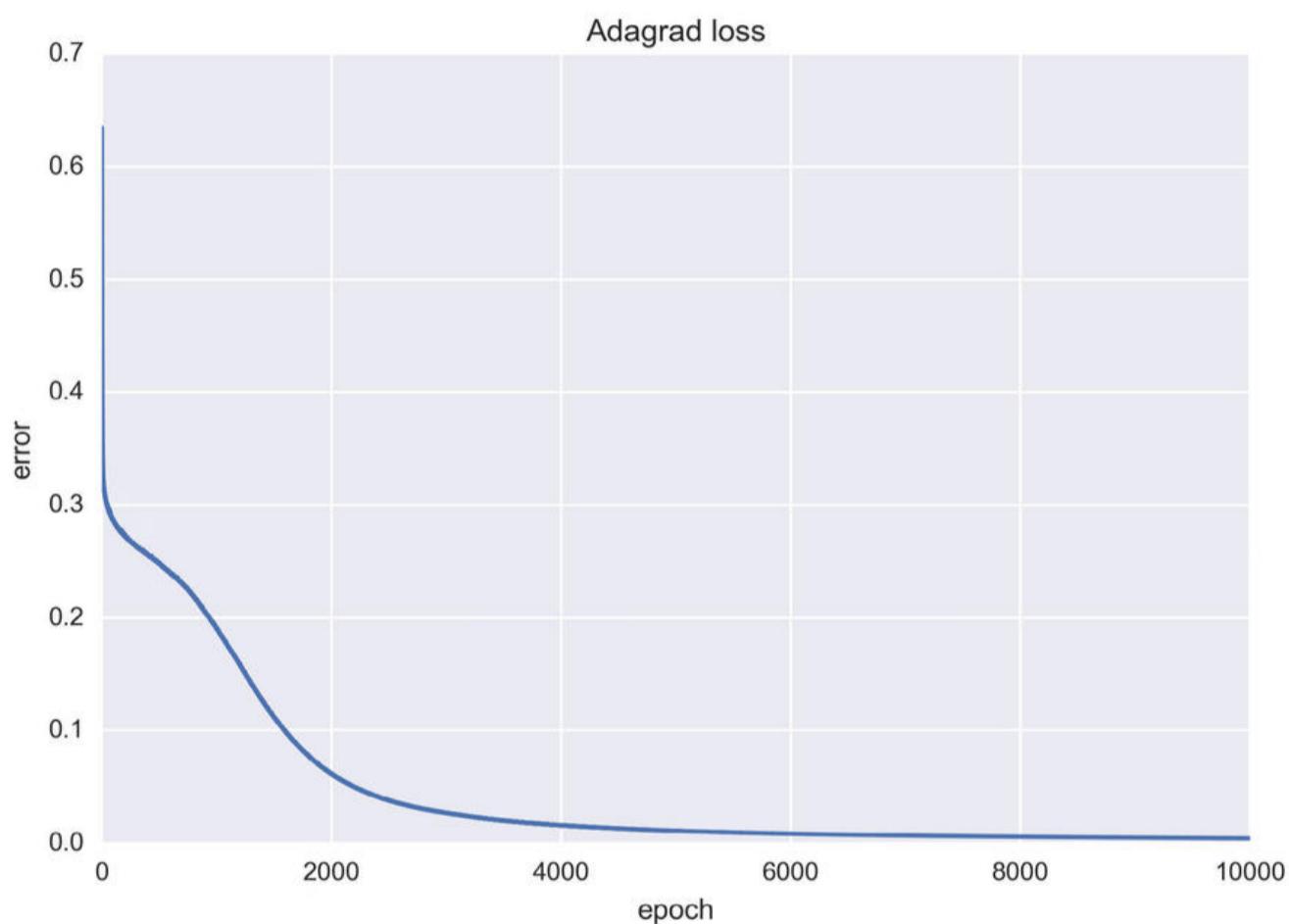


Figure 19.46: The performance of Adagrad on our test setup. It drops a lot in the very beginning, and then slowly finds a way to the global minimum of 0 around epoch 8000. The graph shows 10,000 epochs.

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. This is 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.

19.5.4 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 kept a **decaying sum** of these gradients.

We can think of this as keeping a running list of the most recent gradients. Each time we update the weights, we tack the new gradient onto the near end of the list and drop the oldest one off the far end. 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 [Ruder16].

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** [Zeiler12]. The name comes from “adaptive,” like Adagrad, and the “delta” refers to the Greek letter δ (delta), which mathematicians often use to refer to how much something is changed. So this algorithm adaptively changes how much the weights are updated on each step using this 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 19.47 shows the results of Adadelta on our test setup.

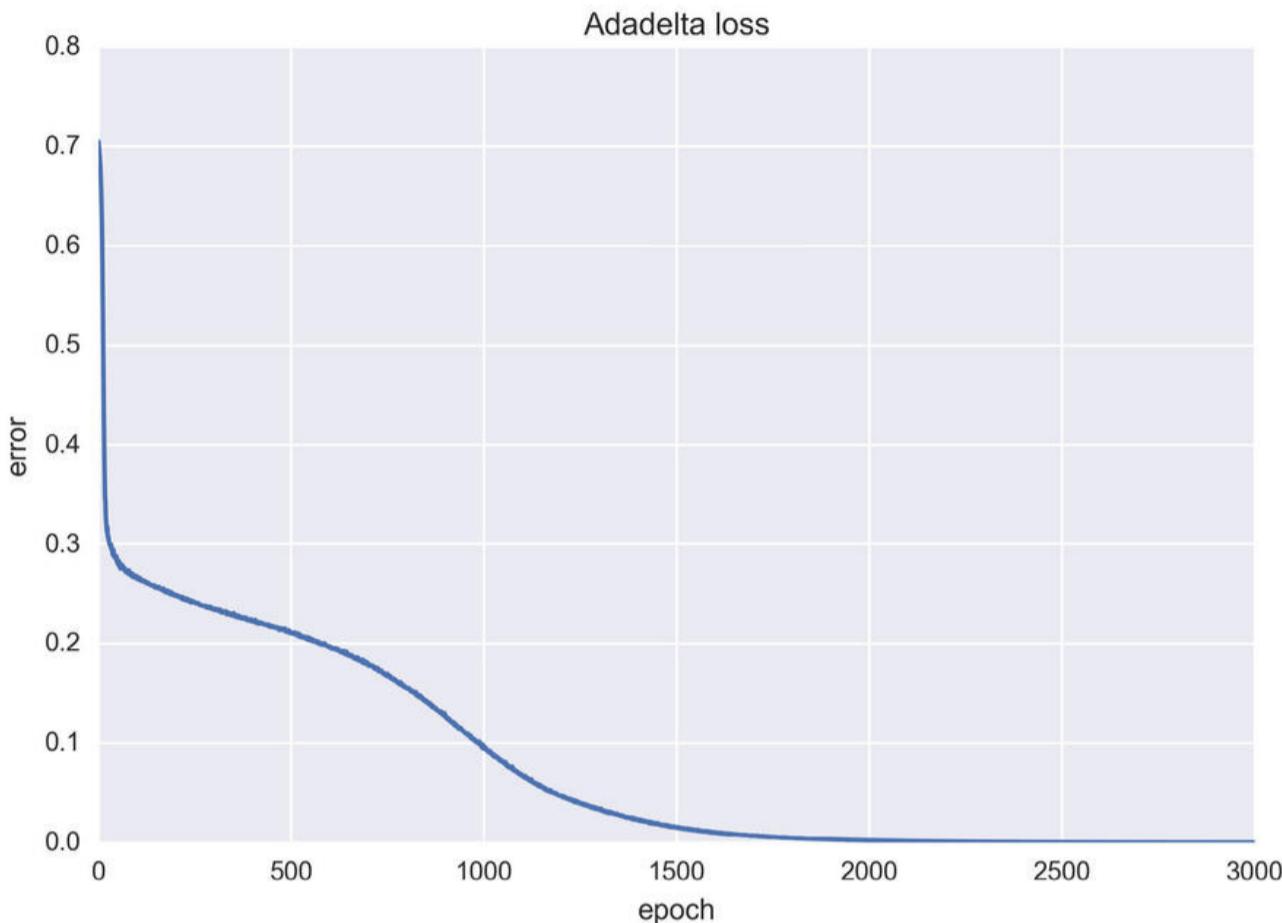


Figure 19.47: The results of training with Adadelta on our test data. We get to about zero error at epoch 2500, compared to epoch 8000 for Adagrad. The graph stops at 3000 epochs.

This compares favorably to Adagrad’s performance in Figure 19.46. It’s nice and smooth, and reaches 0 at around epoch 2500, much sooner than Adagrad’s 8000 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 will “remember” 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 by 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 uses slightly different mathematics, is called **RMSprop** [Hinton15]. 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.

19.5.5 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 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. That's useful information to have. So let's imagine keeping a second list of just 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** [Kingma15].

Figure 19.48 shows how Adam performs.

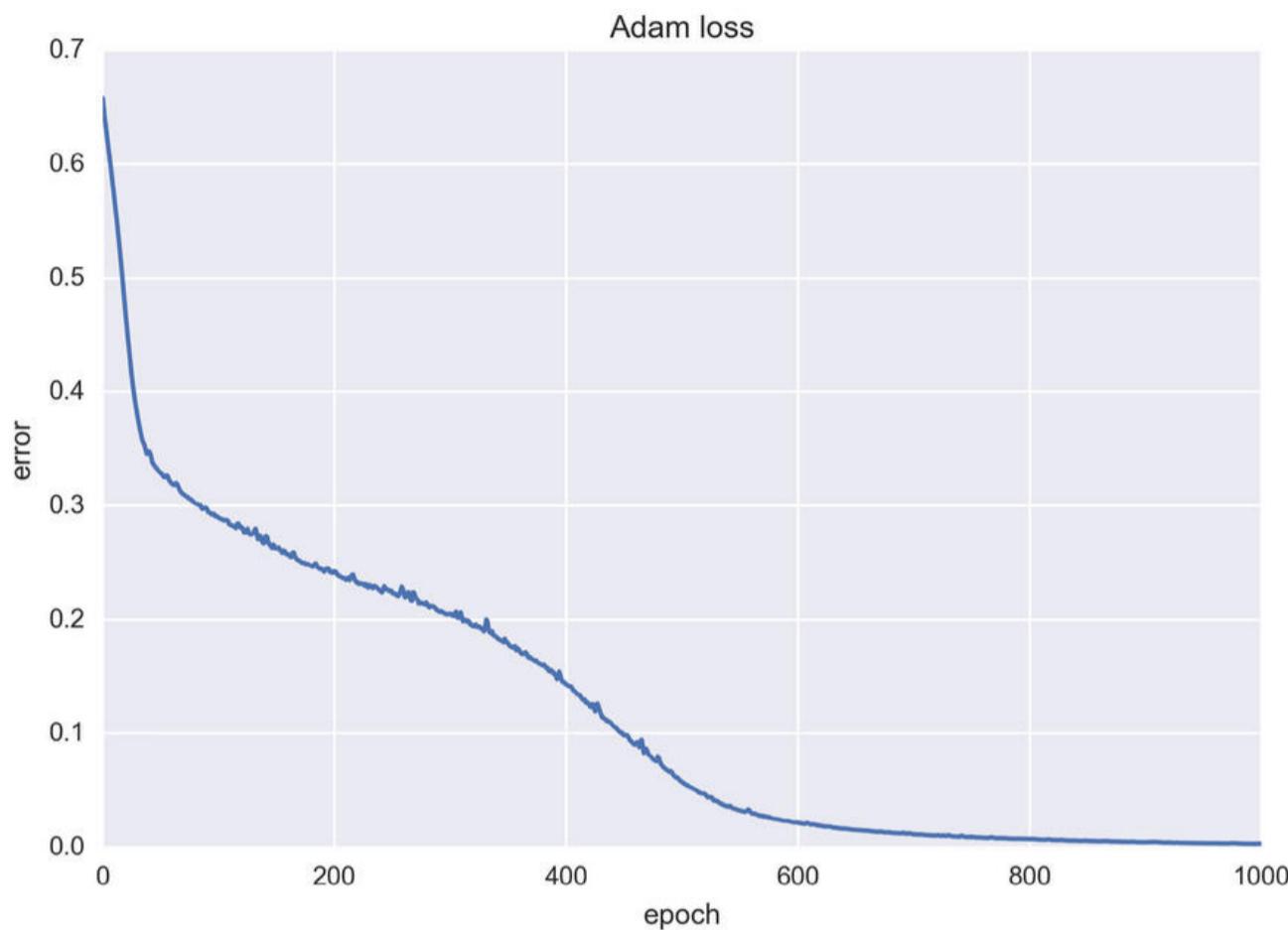


Figure 19.48: The Adam algorithm on our test set. The algorithm hits zero error at around 900 epochs. This compares favorably to Adadelta's 2500, or Adadelta's 8000. The graph shows 1000 epochs.

The output is great. It's only slightly noisy, and hits zero 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 β_1 and β_2 . The authors of the paper on Adam suggest setting β_1 to 0.9, and β_2 to 0.999, and these values indeed often work well.

19.6 Choosing An Optimizer

This has not been a complete list of all the optimizers that have been proposed and studied. There are many others, and each has their strengths and weaknesses. Our goal was to give an overview of some of the most popular techniques, and understand how they achieve their speedups.

Figure 19.49 summarizes our results for SGD with Nesterov momentum and the three adaptive algorithms of Adagrad, Adadelta, and Adam.

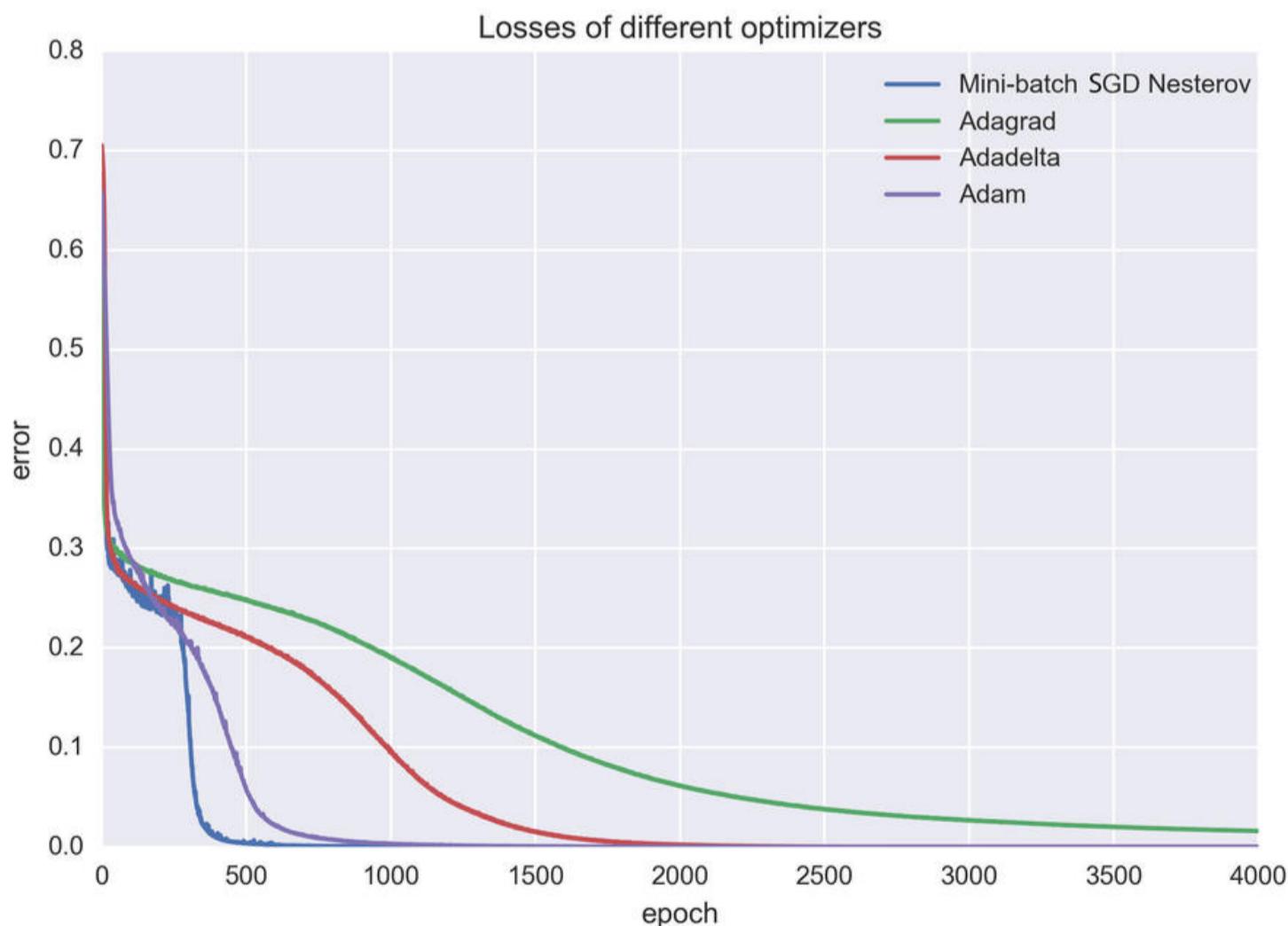


Figure 19.49: The loss, or error, over time for four of the algorithms covered above. This graph shows only the first 4000 epochs.

In this simple test case, Mini-batch gradient descent with Nesterov momentum is the clear winner, with Adam coming in a close second. In more complicated situations, the situation reverses and the adaptive algorithms usually perform better.

Across a wide variety of data sets and networks, the final three adaptive algorithms that we discussed (Adadelta, RMSprop, and Adam) often perform very similarly [Ruder16]. 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 [Kingma15].

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*. Whatever 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** [Wolpert96] [Wolpert97]. This guarantees us that there’s no optimizer that 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 there’s no one optimizer that will *always* beat the others.

We can find the best optimizer for any specific combination of network and data with an automated search that tries out multiple optimizers, perhaps with multiple choices for their parameters. 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 set of network and data to the next.

References

- [Bengio12] Yoshua Bengio, “Practical Recommendations for Gradient-Based Training of Deep Architectures”, in “Neural Networks: Tricks of the Trade: Second Edition”, editors Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller, 2012. <https://arxiv.org/abs/1206.5533v2>
- [Darken92] Darken, C., Chang, J., and Moody, J. “Learning rate schedules for faster stochastic gradient search”, Neural Networks for Signal Processing II, Proceedings of the 1992 IEEE Workshop, (September), pg 1–11, 1992.
- [Dauphin14] Dauphin, Y., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y., “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”, arXiv, 1–14, 2014. <http://arxiv.org/abs/1406.2572>
- [Duchi11] Duchi, J., Hazan, E., and Singer, Y, (2011). “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”, Journal of Machine Learning Research, 12, pgs. 2121–2159, 2011. <http://jmlr.org/papers/v12/duchi11a.html>
- [Hinton15] Geoffrey Hinton, Nitish Srivastava, Kevin Swersky, “Neural Networks for Machine Learning: Lecture 6a, Overview of mini-batch gradient descent”, 2015. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf
- [IBM10] IBM, “What Is Batch Processing?”, IBM Knowledge Center, z/OS Concepts, 2010. https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zconcepts/zconc_whatisthebatch.htm
- [Karpathy16] Andrej Karpathy, “Neural Networks Part 3: Learning and Evaluation”, Course notes for Stanford CS231n, 2016. <http://cs231n.github.io/neural-networks-2/>

- [Kingma15] Kingma, D. P., and Ba, J. L., “Adam: a Method for Stochastic Optimization”, International Conference on Learning Representations, pages 1–13, 2015.
- [Nesterov83] Nesterov, Y., “A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$ ”, Doklady ANSSSR (translated as Soviet.Math.Docl.), vol. 269, pp. 543– 547, 1983.
- [Orr99] Genevieve Orr, “CS-449: Neural Networks Course Notes, Momentum and Learning Rate Adaptation”, Willamette University, 1999. <https://www.willamette.edu/~gorr/classes/cs449/intro.html>
- [Qian99] Qian, N., “On the momentum term in gradient descent learning algorithms”, Neural Networks, 12(1), pages 145–151, 1999. <https://pdfs.semanticscholar.org/735d/4220d5579cc6afe956d9f6ea501a96ae99e2.pdf>
- [Ruder16] Sebastian Ruder, “An overview of gradient descent optimisation algorithms” arXiv preprint at <http://arxiv.org/abs/1609.04747>
- [Wolpert96] D.H. Wolpert, “The Lack of A Priori Distinctions Between Learning Theorems”, Neural Computation 8, 1996. http://www.zabaras.com/Courses/BayesianComputing/Papers/lack_of_a_priori_distinctions_wolpert.pdf
- [Wolpert97] D.H. Wolpert and W.G. Macready, “No Free Lunch Theorems for Optimization”, IEEE Transactions on Evolutionary Computation, 1(1), 1997. <https://ti.arc.nasa.gov/m/profile/dhw/papers/78.pdf>
- [Zeiler12] Zeiler, M. D. “ADADELTA: An Adaptive Learning Rate Method”, 2012. <http://arxiv.org/abs/1212.5701>