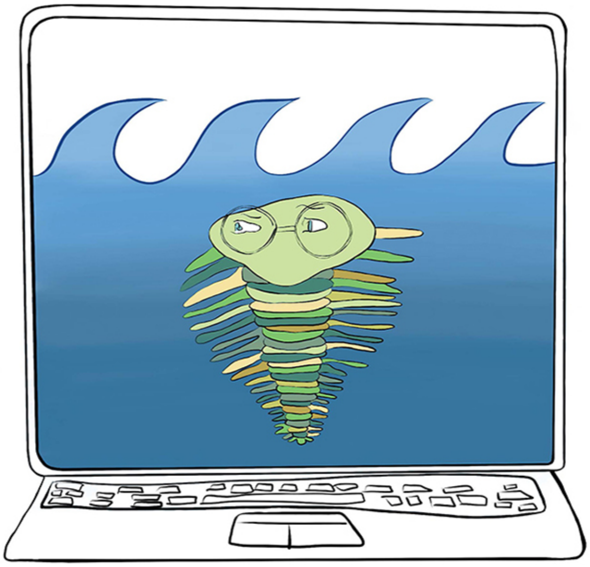
Addison Wesley Data & Analytics Series

Deep Learning Illustrated

A Visual, Interactive Guide to Artificial Intelligence



Jon Krohn
with Grant Beyleveld *and* Aglaé Bassens

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Praise for *Deep Learning Illustrated*

"Over the next few decades, artificial intelligence is poised to dramatically change almost every aspect of our lives, in large part due to today's breakthroughs in deep learning. The authors' clear visual style provides a comprehensive look at what's currently possible with artificial neural networks as well as a glimpse of the magic that's to come."
—*Tim Urban, writer and illustrator of Wait But Why*

"This book is an approachable, practical, and broad introduction to deep learning, and the most beautifully illustrated machine learning book on the market."
—*Dr. Michael Osborne, Dyson Associate Professor in Machine Learning, University of Oxford*

"This book should be the first stop for deep learning beginners, as it contains lots of concrete, easy-to-follow examples with corresponding tutorial videos and code notebooks. Strongly recommended."
—*Dr. Chong Li, cofounder, Nakamoto & Turing Labs; adjunct professor, Columbia University*

"It's hard to imagine developing new products today without thinking about enriching them with capabilities using machine learning. Deep learning in particular has many practical applications, and this book's intelligible clear and visual approach is helpful to anyone who would like to understand what deep learning is and how it could impact your business and life for years to come."
—*Helen Altshuler, engineering leader, Google*

"This book leverages beautiful illustrations and amusing analogies to make the theory behind deep learning uniquely accessible. Its straightforward example code and best-practice tips empower readers to immediately apply the transformative technique to their particular niche of interest."
—Dr. Rasmus Rothe, founder, Merantix

"This is an invaluable resource for anyone looking to understand what deep learning is and why it powers almost every automated application today, from chatbots and voice recognition tools to self-driving cars. The illustrations and biological explanations help bring to life a complex topic and make it easier to grasp fundamental concepts."
—Joshua March, CEO and cofounder, Conversocial; author of Message Me

"Deep learning is regularly redefining the state of the art across machine vision, natural language, and sequential decision-making tasks. If you too would like to pass data through deep neural networks in order to build high-performance models, then this book—with its innovative, highly visual approach—is the ideal place to begin."
—Dr. Alex Flint, roboticist and entrepreneur

Deep Learning Illustrated

Deep Learning Illustrated

A Visual, Interactive Guide to Artificial Intelligence

Jon Krohn
with Grant Beyleveld
and Aglaé Bassens

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Number: 2019938790

Copyright © 2020 Pearson Education, Inc.

Cover art by Aglaé Bassens

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/.

ISBN-13: 978-0-13-511669-2
ISBN-10: 0-13-511669-4

ScoutAutomatedPrintCode

♦

For Gigi

—Jon

♦

This page intentionally left blank

Contents

Figures xix

Tables xxvii

Examples xxix

Foreword xxxiii

Preface xxxv

Acknowledgments xxxix

About the Authors xli

I Introducing Deep Learning 1

1 Biological and Machine Vision 3

Biological Vision 3

Machine Vision 8

- The Neocognitron 8
- LeNet-5 9
- The Traditional Machine Learning Approach 12
- ImageNet and the ILSVRC 13
- AlexNet 14

TensorFlow Playground 17

Quick, Draw! 19

Summary 19

2 Human and Machine Language 21

Deep Learning for Natural Language Processing 21

- Deep Learning Networks Learn Representations Automatically 22
- Natural Language Processing 23
- A Brief History of Deep Learning for NLP 24

Computational Representations of Language 25
One-Hot Representations of Words 25
Word Vectors 26
Word-Vector Arithmetic 29
word2viz 30
Localist Versus Distributed Representations 32
Elements of Natural Human Language 33
Google Duplex 35
Summary 37

3 Machine Art 39
A Boozy All-Nighter 39
Arithmetic on Fake Human Faces 41
Style Transfer: Converting Photos into Monet (and Vice Versa) 44
Make Your Own Sketches Photorealistic 45
Creating Photorealistic Images from Text 45
Image Processing Using Deep Learning 46
Summary 48

4 Game-Playing Machines 49
Deep Learning, AI, and Other Beasts 49
Artificial Intelligence 49
Machine Learning 50
Representation Learning 51
Artificial Neural Networks 51
Deep Learning 51
Machine Vision 52
Natural Language Processing 53
Three Categories of Machine Learning Problems 53
Supervised Learning 53
Unsupervised Learning 54
Reinforcement Learning 54
Deep Reinforcement Learning 56
Video Games 57
Board Games 59
AlphaGo 59

AlphaGo Zero 62
AlphaZero 65
Manipulation of Objects 67
Popular Deep Reinforcement Learning Environments 68
OpenAI Gym 68
DeepMind Lab 69
Unity ML-Agents 71
Three Categories of AI 71
Artificial Narrow Intelligence 72
Artificial General Intelligence 72
Artificial Super Intelligence 72
Summary 72

II Essential Theory Illustrated 73

5 The (Code) Cart Ahead of the (Theory) Horse 75
Prerequisites 75
Installation 76
A Shallow Network in Keras 76
The MNIST Handwritten Digits 76
A Schematic Diagram of the Network 77
Loading the Data 79
Reformatting the Data 81
Designing a Neural Network Architecture 83
Training a Deep Learning Model 83
Summary 84

6 Artificial Neurons Detecting Hot Dogs 85
Biological Neuroanatomy 101 85
The Perceptron 86
The Hot Dog / Not Hot Dog Detector 86
The Most Important Equation in This Book 90

Modern Neurons and Activation Functions 91
    The Sigmoid Neuron 92
    The Tanh Neuron 94
    ReLU: Rectified Linear Units 94
Choosing a Neuron 96
Summary 96
Key Concepts 97

7 Artificial Neural Networks 99
The Input Layer 99
Dense Layers 99
A Hot Dog-Detecting Dense Network 101
    Forward Propagation Through the First Hidden Layer 102
    Forward Propagation Through Subsequent Layers 103
The Softmax Layer of a Fast Food-Classifying Network 106
Revisiting Our Shallow Network 108
Summary 110
Key Concepts 110

8 Training Deep Networks 111
Cost Functions 111
    Quadratic Cost 112
    Saturated Neurons 112
    Cross-Entropy Cost 113
Optimization: Learning to Minimize Cost 115
    Gradient Descent 115
    Learning Rate 117
    Batch Size and Stochastic Gradient Descent 119
    Escaping the Local Minimum 122
Backpropagation 124
Tuning Hidden-Layer Count and Neuron Count 125
An Intermediate Net in Keras 127
Summary 129
Key Concepts 130

9 Improving Deep Networks 131
Weight Initialization 131
Xavier Glorot Distributions 135
Unstable Gradients 137
Vanishing Gradients 137
Exploding Gradients 138
Batch Normalization 138
Model Generalization (Avoiding Overfitting) 140
L1 and L2 Regularization 141
Dropout 142
Data Augmentation 145
Fancy Optimizers 145
Momentum 145
Nesterov Momentum 146
AdaGrad 146
AdaDelta and RMSProp 146
Adam 147
A Deep Neural Network in Keras 147
Regression 149
TensorBoard 152
Summary 154
Key Concepts 155

III Interactive Applications of Deep Learning 157

10 Machine Vision 159
Convolutional Neural Networks 159
The Two-Dimensional Structure of Visual Imagery 159
Computational Complexity 160
Convolutional Layers 160
Multiple Filters 162
A Convolutional Example 163
Convolutional Filter Hyperparameters 168

Pooling Layers 169
LeNet-5 in Keras 171
AlexNet and VGGNet in Keras 176
Residual Networks 179
    Vanishing Gradients: The Bête Noire of Deep CNNs 179
    Residual Connections 180
    ResNet 182
Applications of Machine Vision 182
    Object Detection 183
    Image Segmentation 186
    Transfer Learning 188
    Capsule Networks 192
Summary 193
Key Concepts 193

11 Natural Language Processing 195
Preprocessing Natural Language Data 195
    Tokenization 197
    Converting All Characters to Lowercase 199
    Removing Stop Words and Punctuation 200
    Stemming 201
    Handling *n*-grams 202
    Preprocessing the Full Corpus 203
Creating Word Embeddings with word2vec 206
    The Essential Theory Behind word2vec 206
    Evaluating Word Vectors 209
    Running word2vec 209
    Plotting Word Vectors 213
The Area under the ROC Curve 217
    The Confusion Matrix 218
    Calculating the ROC AUC Metric 219
Natural Language Classification with Familiar Networks 222
    Loading the IMDb Film Reviews 222
    Examining the IMDb Data 226
    Standardizing the Length of the Reviews 228

Dense Network 229
Convolutional Networks 235
Networks Designed for Sequential Data 240
Recurrent Neural Networks 240
Long Short-Term Memory Units 244
Bidirectional LSTMs 247
Stacked Recurrent Models 248
Seq2seq and Attention 250
Transfer Learning in NLP 251
Non-sequential Architectures: The Keras Functional API 251
Summary 256
Key Concepts 257

12 Generative Adversarial Networks 259
Essential GAN Theory 259
The Quick, Draw! Dataset 263
The Discriminator Network 266
The Generator Network 269
The Adversarial Network 272
GAN Training 275
Summary 281
Key Concepts 282

13 Deep Reinforcement Learning 283
Essential Theory of Reinforcement Learning 283
The Cart-Pole Game 284
Markov Decision Processes 286
The Optimal Policy 288
Essential Theory of Deep Q-Learning Networks 290
Value Functions 291
Q-Value Functions 291
Estimating an Optimal Q-Value 291
Defining a DQN Agent 293
Initialization Parameters 295

Building the Agent's Neural Network Model 297
Remembering Gameplay 298
Training via Memory Replay 298
Selecting an Action to Take 299
Saving and Loading Model Parameters 300
Interacting with an OpenAI Gym Environment 300
Hyperparameter Optimization with SLM Lab 303
Agents Beyond DQN 306
Policy Gradients and the REINFORCE Algorithm 307
The Actor-Critic Algorithm 307
Summary 308
Key Concepts 309

IV You and AI 311

14 Moving Forward with Your Own Deep Learning Projects 313
Ideas for Deep Learning Projects 313
Machine Vision and GANs 313
Natural Language Processing 315
Deep Reinforcement Learning 316
Converting an Existing Machine Learning Project 316
Resources for Further Projects 317
Socially Beneficial Projects 318
The Modeling Process, Including Hyperparameter Tuning 318
Automation of Hyperparameter Search 321
Deep Learning Libraries 321
Keras and TensorFlow 321
PyTorch 323
MXNet, CNTK, Caffe, and So On 324
Software 2.0 324
Approaching Artificial General Intelligence 326
Summary 328

V Appendices 331

A Formal Neural Network Notation 333

B Backpropagation 335

C PyTorch 339
PyTorch Features 339
Autograd System 339
Define-by-Run Framework 339
PyTorch Versus TensorFlow 340
PyTorch in Practice 341
PyTorch Installation 341
The Fundamental Units Within PyTorch 341
Building a Deep Neural Network in PyTorch 343

Index 345

This page intentionally left blank

Figures

P.1 Santiago Cajal (1852–1934) xxxv
P.2 A hand-drawn diagram from Cajal's (1894) publication showing the growth of a neuron (a–e) and contrasting neurons from frog (A), lizard (B), rat (C), and human (D) samples xxxvi
P.3 The reading trilobite enjoys expanding your knowledge. xxxvii
P.4 This trilobite calls attention to tricky passages of text. xxxviii
1.1 The number of species on our planet began to increase rapidly 550 million years ago, during the prehistoric Cambrian period. 4
1.2 A bespectacled trilobite 4
1.3 The Nobel Prize-winning neurophysiologists Torsten Wiesel (left) and David Hubel 5
1.4 Hubel and Wiesel used a light projector to present slides to anesthetized cats while they recorded the activity of neurons in the cats' primary visual cortex. 5
1.5 A simple cell in the primary visual cortex of a cat fires at different rates, depending on the orientation of a line shown to the cat. 6
1.6 A caricature of how consecutive layers of biological neurons represent visual information in the brain of, for example, a cat or a human 7
1.7 Regions of the visual cortex. 8
1.8 Abridged timeline of biological and machine vision, highlighting the key historical moments in the deep learning and traditional machine learning approaches to vision that are covered in this section 9
1.9 Paris-born Yann LeCun is one of the preeminent figures in artificial neural network and deep learning research. 10
1.10 Yoshua Bengio is another of the leading characters in artificial neural networks and deep learning. 10
1.11 LeNet-5 retains the hierarchical architecture uncovered in the primary visual cortex by Hubel and Wiesel and leveraged by Fukushima in his neocognitron. 11
1.12 Feature engineering—the transformation of raw data into thoughtfully transformed input variables—often predominates the application of traditional machine learning algorithms. 12
1.13 Engineered features leveraged by Viola and Jones (2001) to detect faces reliably. 13
1.14 The hulking ImageNet dataset was the brainchild of Chinese-American computer science professor Fei-Fei Li and her colleagues at Princeton in 2009. 14
1.15 Performance of the top entrants to the ILSVRC by year. 15

1.16 The eminent British-Canadian artificial neural network pioneer Geoffrey Hinton, habitually referred to as "the godfather of deep learning" in the popular press. 15
1.17 AlexNet's hierarchical architecture is reminiscent of LeNet-5, with the first (left-hand) layer representing simple visual features like edges, and deeper layers representing increasingly complex features and abstract concepts. 16
1.18 This deep neural network is ready to learn how to distinguish a spiral of orange dots (negative cases) from blue dots (positive cases) based on their position on the X_1 and X_2 axes of the grid on the right. 18
1.19 The network after training 19
2.1 Venn diagram that distinguishes the traditional family from the representation learning family of machine learning techniques 22
2.2 NLP sits at the intersection of the fields of computer science, linguistics, and artificial intelligence. 23
2.3 Milestones involving the application of deep learning to natural language processing 24
2.4 One-hot encodings of words, such as this example, predominate the traditional machine learning approach to natural language processing. . . . 26
2.5 A toy-sized example for demonstrating the high-level process behind techniques like word2vec and GloVe that convert natural language into word vectors 27
2.6 Diagram of word meaning as represented by a three-dimensional vector space 28
2.7 Examples of word-vector arithmetic 30
2.8 The default screen for word2viz, a tool for exploring word vectors interactively 31
2.9 Relationships between the elements of natural human language. 34
3.1 High-level schematic of a generative adversarial network (GAN). 40
3.2 Results presented in Goodfellow and colleagues' 2014 GAN paper 41
3.3 An example of latent-space arithmetic from Radford et al. (2016) 42
3.4 A cartoon of the latent space associated with generative adversarial networks (GANs). 43
3.5 Photos converted into the styles of well-known painters by CycleGANs . 44
3.6 A mutant three-eyed cat (right-hand panel) synthesized via the pix2pix web application. 45
3.7 Photorealistic high-resolution images output by StackGAN, which involves two GANs stacked upon each other 46
3.8 A sample image (left) processed using a traditional pipeline (center) and the deep learning pipeline by Chen et al. (right) 47
3.9 Novel "hand drawings" of apples produced by the GAN architecture we develop together in Chapter 12. 48
4.1 Venn diagram showing the relative positioning of the major concepts covered over the course of this book 50
4.2 Generalization of deep learning model architectures 52

4.3 The reinforcement learning loop. 55
4.4 Demis Hassabis cofounded DeepMind in 2010 after completing his PhD in cognitive neuroscience at University College London. 58
4.5 The normalized performance scores of Mnih and colleagues' (2015) DQN relative to a professional game tester: Zero percent represents random play, and 100% represents the pro's best performance. 60
4.6 The Go board game. 61
4.7 David Silver is a Cambridge- and Alberta-educated researcher at Google DeepMind. 62
4.8 The Elo score of AlphaGo (blue) relative to Fan Hui (green) and several Go programs (red). 63
4.9 Comparing Elo scores between AlphaGo Zero and other AlphaGo variations or other Go programs. 64
4.10 Comparing Elo scores between AlphaZero and each of its opponents in chess, shogi, and Go. 66
4.11 Chelsea Finn is a doctoral candidate at the University of California, Berkeley, in its AI Research Lab. 67
4.12 Sample images from Levine, Finn, et al. (2016) exhibiting various object-manipulation actions the robot was trained to perform 68
4.13 A sampling of OpenAI Gym environments. 70
4.14 A DeepMind Lab environment, in which positive-reward points are awarded for capturing scrumptious green apples 71
5.1 A sample of a dozen images from the MNIST dataset. 77
5.2 The Danish computer scientist Corinna Cortes is head of research at Google's New York office. 78
5.3 Each handwritten MNIST digit is stored as a 28×28-pixel grayscale image. 78
5.4 A rough schematic of the shallow artificial-neural-network architecture we're whipping up in this chapter. 79
5.5 The first MNIST digit in the validation dataset (`X_valid[0]`) is a *seven*. . 81
6.1 The anatomy of a biological neuron 86
6.2 The American neurobiology and behavior researcher Frank Rosenblatt. . 87
6.3 Schematic diagram of a perceptron, an early artificial neuron. 87
6.4 First example of a hot dog-detecting perceptron: In this instance, it predicts there is indeed a hot dog. 88
6.5 Second example of a hot dog-detecting perceptron: In this instance, it predicts there is not a hot dog. 89
6.6 Third example of a hot dog-detecting perceptron: In this instance, it again predicts the object presented to it is a hot dog. 90
6.7 The general equation for artificial neurons that we will return to time and again. 91
6.8 The perceptron's transition from outputting zero to outputting one happens suddenly, making it challenging to gently tune w and b to match a desired output. 92
6.9 The sigmoid activation function 93
6.10 The tanh activation function 94

6.11 The ReLU activation function 95
7.1 A dense network of artificial neurons, highlighting the inputs to the neuron labeled a_1 101
7.2 Our hot dog-detecting network from Figure 7.1, now highlighting the activation output of neuron a_1, which is provided as an input to both neuron a_4 and neuron a_5 103
7.3 Our hot dog-detecting network, with the activations providing input to the output neuron $\hat{y}$ highlighted 105
7.4 Our food-detecting network, now with three softmax neurons in the output layer 107
7.5 A summary of the model object from our *Shallow Net in Keras* Jupyter notebook 109
8.1 Plot reproducing the tanh activation function shown in Figure 6.10, drawing attention to the high and low values of z at which a neuron is saturated 113
8.2 A trilobite using gradient descent to find the value of a parameter p associated with minimal cost, C 116
8.3 A trilobite exploring along two model parameters—p_1 and p_2—in order to minimize cost via gradient descent. 117
8.4 The learning rate (η) of gradient descent expressed as the size of a trilobite. 118
8.5 An individual round of training with stochastic gradient descent. 121
8.6 An outline of the overall process for training a neural network with stochastic gradient descent. 121
8.7 A trilobite applying vanilla gradient descent from a random starting point (top panel) is ensnared by a local minimum of cost (middle panel). 123
8.8 The speed of learning over epochs of training for a deep learning network with five hidden layers. 126
8.9 A summary of the model object from our *Intermediate Net in Keras* Jupyter notebook 128
8.10 The performance of our intermediate-depth neural network over its first four epochs of training 129
9.1 Histogram of the a activations output by a layer of sigmoid neurons, with weights initialized using a standard normal distribution 134
9.2 Histogram of the a activations output by a layer of sigmoid neurons, with weights initialized using the Glorot normal distribution 135
9.3 The activations output by a dense layer of 256 neurons, while varying activation function (tanh or ReLU) and weight initialization (standard normal or Glorot uniform). 137
9.4 Batch normalization transforms the distribution of the activations output by a given layer of neurons toward a standard normal distribution. 139
9.5 Fitting y given x using models with varying numbers of parameters. . . . 141
9.6 Dropout, a technique for reducing model overfitting, involves the removal of randomly selected neurons from a network's hidden layers in each round of training. 143

9.7 Our deep neural network architecture peaked at a 97.87 percent validation following 15 epochs of training, besting the accuracy of our shallow and intermediate-depth architectures. 149
9.8 The TensorBoard dashboard enables you to, epoch over epoch, visually track your model's cost (`loss`) and accuracy (`acc`) across both your training data and your validation (`val`) data. 153
10.1 When reading a page of a book written in English, we begin in the top-left corner and read to the right. 161
10.2 A 3×3 kernel and a 3×3-pixel window 162
10.3 This schematic diagram demonstrates how the activation values in a feature map are calculated in a convolutional layer. 164
10.4 A continuation of the convolutional example from Figure 10.3, now showing the activation for the next filter position 165
10.5 Finally, the activation for the last filter position has been calculated, and the activation map is complete. 166
10.6 A graphical representation of the input array (left; represented here is a three-channel RGB image of size 32×32 with the kernel patch currently focused on the first—i.e., top-left—position) and the activation map (right). 166
10.7 An example of a max-pooling layer being passed a 4×4 activation map. . . 170
10.8 A summary of our LeNet-5-inspired ConvNet architecture. 174
10.9 Our LeNet-5-inspired ConvNet architecture peaked at a 99.27 percent validation accuracy following nine epochs of training, thereby outperforming the accuracy of the dense nets we trained earlier in the book. . . 175
10.10 A general approach to CNN design: A block (shown in red) of convolutional layers (often one to three of them) and a pooling layer is repeated several times. 176
10.11 A schematic representation of a residual module. 181
10.12 Shown at left is the conventional representation of residual blocks within a residual network. 181
10.13 These are examples of various machine vision applications. 183
10.14 These are examples of object detection (performed on four separate images by the Faster R-CNN algorithm). 185
10.15 This is an example of image segmentation (as performed by the Mask R-CNN algorithm). 186
10.16 With convolutional neural networks, which are agnostic to the relative positioning of image features, the figure on the left and the one on the right are equally likely to be classified as Geoff Hinton's face. 192
11.1 The second sentence of Jane Austen's classic *Emma* tokenized to the word level 199
11.2 A dictionary of bigrams detected within our corpus 202
11.3 Sample of a dictionary of bigrams detected within our lowercased and punctuation-free corpus 204
11.4 Sample of a more conservatively thresholded dictionary of bigrams 205
11.5 Clean, preprocessed sentence from the Project Gutenberg corpus 206

11.6 The location of the token "dog" within the 64-dimensional word-vector space we generated using a corpus of books from Project Gutenberg . . . 211
11.7 This is a Pandas DataFrame containing a two-dimensional representation of the word-vector space we created from the Project Gutenberg corpus. . 215
11.8 Static two-dimensional word-vector scatterplot 215
11.9 Interactive bokeh two-dimensional word-vector plot 216
11.10 Clothing words from the Project Gutenberg corpus, revealed by zooming in to a region of the broader bokeh plot from Figure 11.9 217
11.11 The (orange-shaded) area under the curve of the receiving operator characteristic, determined using the TPRs and FPRs from Table 11.5 . . . 221
11.12 The first two film reviews from the training dataset of Andrew Maas and colleagues' (2011) IMDb dataset. 227
11.13 The first film review from the training dataset, now shown as a character string 227
11.14 The first film review from the training dataset, now shown in full as a character string 228
11.15 The sixth film review from the training dataset, padded with the `PAD` token at the beginning so that—like all the other reviews—it has a length of 100 tokens 229
11.16 Dense sentiment classifier model summary 231
11.17 Training the dense sentiment classifier 232
11.18 Histogram of validation data $\hat{y}$ values for the second epoch of our dense sentiment classifier 233
11.19 DataFrame of y and $\hat{y}$ values for the IMDb validation data 234
11.20 An example of a false positive: This negative review was misclassified as positive by our model. 235
11.21 An example of a false negative: This positive review was misclassified as negative by our model. 236
11.22 Convolutional sentiment classifier model summary 238
11.23 Training the convolutional sentiment classifier 239
11.24 Histogram of validation data $\hat{y}$ values for the third epoch of our convolutional sentiment classifier 239
11.25 Schematic diagram of a recurrent neural network 241
11.26 Schematic diagram of an LSTM 245
11.27 A non-sequential model architecture: Three parallel streams of convolutional layers—each with a unique filter length ($k = 2$, $k = 3$, or $k = 4$)—receive input from a word-embedding layer. 252
12.1 Highly simplified schematic diagrams of the two models that make up a typical GAN: the generator (left) and the discriminator (right) 260
12.2 This is an outline of the discriminator training loop. 261
12.3 An outline of the generator training loop. 261
12.4 Example of sketches drawn by humans who have played the Quick, Draw! game. 263
12.5 The directory structure inside the Docker container that is running Jupyter. 264

12.6 This example bitmap is the 4,243rd sketch from the apple category of the Quick, Draw! dataset. 266
12.7 A schematic representation of our discriminator network for predicting whether an input image is real (in this case, a hand-drawn apple from the Quick, Draw! dataset) or fake (produced by an image generator) 268
12.8 A schematic representation of our generator network, which takes in noise (in this case, representing 32 latent-space dimensions) and outputs a 28×28-pixel image. 270
12.9 Shown here is a summary of the whole adversarial network. 273
12.10 Fake apple sketches generated after 100 epochs of training our GAN . . . 278
12.11 Fake apple sketches after 200 epochs of training our GAN 279
12.12 Fake apple sketches after 1,000 epochs of training our GAN 279
12.13 GAN training loss over epochs 280
12.14 GAN training accuracy over epochs 280
13.1 The objective of the Cart-Pole game is to keep the brown pole balanced upright on top of the black cart for as long as possible. 285
13.2 The Cart-Pole game ends early if the pole falls toward horizontal or the cart is navigated off-screen. 285
13.3 The reinforcement learning loop (top; a rehashed version of Figure 4.3, provided again here for convenience) can be considered a Markov decision process, which is defined by the five components S, A, R, $\mathbb{P}$, and γ (bottom). 287
13.4 Based on the discount factor γ, in a Markov decision process more-distant reward is discounted relative to reward that's more immediately attainable. 289
13.5 The policy function π enables an agent to map any state s (from the set of all possible states S) to an action a from the set of all possible actions A. 289
13.6 The biggest star in the field of reinforcement learning, Richard Sutton has long been a computer science professor at the University of Alberta. . 292
13.7 As in Figure 13.4, here we use the Pac-Man environment (with a green trilobite representing a DQN agent in place of the Mr. Pac-Man character) to illustrate a reinforcement learning concept. 296
13.8 The performance of our DQN agent during its first 10 episodes playing the Cart-Pole game. 303
13.9 The performance of our DQN agent during its final 10 episodes playing the Cart-Pole game. 303
13.10 An experiment run with SLM Lab, investigating the impact of various hyperparameters (e.g., hidden-layer architecture, activation function, learning rate) on the performance of a DQN agent within the Cart-Pole environment 304
13.11 The broad categories of deep reinforcement learning agents 306
13.12 The actor-critic algorithm combines the policy gradient approach to reinforcement learning (playing the role of actor) with the Q-learning approach (playing the role of critic). 308

14.1 Following our pixel-by-pixel rendering of an MNIST digit (Figure 5.3), this is an example of an image from the Fashion-MNIST dataset. 314
14.2 The wide and deep model architecture concatenates together inputs from two separate legs. 317
14.3 A plot of training loss (red) and validation loss (blue) over epochs of model training. 320
14.4 A strictly structured grid search (shown in the left-hand panel) is less likely to identify optimal hyperparameters for a given model than a search over values that are sampled randomly over the same hyperparameter ranges (right-hand panel). 321
14.5 The relative Google search frequency (from October 2015 to February 2019) of five of the most popular deep learning libraries 322
14.6 Andrej Karpathy is the director of AI at Tesla, the California-based automotive and energy firm. 325
14.7 Trilobyte waving good-bye 329

Tables

2.1 Contrasting attributes of localist, one-hot representations of words with distributed, vector-based representations . . . 33
2.2 Traditional machine learning and deep learning representations, by natural language element . . . 34
8.1 Cross-entropy costs associated with selected example inputs . . . 115
11.1 Comparison of word2vec architectures . . . 208
11.2 The words most similar to select test words from our Project Gutenberg vocabulary . . . 212
11.3 A confusion matrix . . . 219
11.4 Four hot dog / not hot dog predictions . . . 219
11.5 Four hot dog / not hot dog predictions, now with intermediate ROC AUC calculations . . . 220
11.6 Comparison of the performance of our sentiment classifier model architectures . . . 256
14.1 Fashion-MNIST categories . . . 314

This page intentionally left blank

Examples

5.1 Software dependencies for shallow net in Keras 80
5.2 Loading MNIST data 80
5.3 Flattening two-dimensional images to one dimension 82
5.4 Converting pixel integers to floats 82
5.5 Converting integer labels to one-hot 82
5.6 Keras code to architect a shallow neural network 83
5.7 Keras code to train our shallow neural network 83
8.1 Keras code to architect an intermediate-depth neural network 127
8.2 Keras code to compile our intermediate-depth neural network 127
8.3 Keras code to train our intermediate-depth neural network 128
9.1 Weight initialization with values sampled from standard normal distribution 133
9.2 Architecture for a single dense layer of sigmoid neurons 133
9.3 Weight initialization with values sampled from Glorot normal distribution 135
9.4 Additional dependencies for deep net in Keras 148
9.5 Deep net in Keras model architecture 148
9.6 Deep net in Keras model compilation 149
9.7 Regression model dependencies 150
9.8 Regression model network architecture 150
9.9 Compiling a regression model 151
9.10 Fitting a regression model 151
9.11 Predicting the median house price in a particular suburb of Boston 152
9.12 Using TensorBoard while fitting a model in Keras 154
10.1 Dependencies for LeNet in Keras 171
10.2 Retaining two-dimensional image shape 172
10.3 CNN model inspired by LeNet-5 172
10.4 CNN model inspired by AlexNet 176
10.5 CNN model inspired by VGGNet 178
10.6 Loading the VGGNet19 model for transfer learning 189
10.7 Adding classification layers to transfer-learning model 190
10.8 Defining data generators 190
10.9 Train transfer-learning model 192
11.1 Converting a sentence to lowercase 200
11.2 Removing stop words and punctuation with a list comprehension 200
11.3 Adding word stemming to our list comprehension 201
11.4 Detecting collocated bigrams 202
11.5 Removing capitalization and punctuation from Project Gutenberg corpus 204
11.6 Detecting collocated bigrams with more conservative thresholds 205

11.7 Creating a “clean” corpus that includes bigrams 205
11.8 Running word2vec 210
11.9 t-SNE for dimensionality reduction 214
11.10 Static two-dimensional scatterplot of word-vector space 214
11.11 Interactive bokeh plot of two-dimensional word-vector data 216
11.12 Loading sentiment classifier dependencies 222
11.13 Setting dense sentiment classifier hyperparameters 223
11.14 Loading IMDb film review data 225
11.15 Printing the number of tokens in six reviews 226
11.16 Printing a review as a character string 227
11.17 Print full review as character string 228
11.18 Standardizing input length by padding and truncating 228
11.19 Dense sentiment classifier architecture 229
11.20 Compiling our sentiment classifier 231
11.21 Creating an object and directory for checkpointing model parameters after each epoch 231
11.22 Fitting our sentiment classifier 232
11.23 Loading model parameters 232
11.24 Predicting $\hat{y}$ for all validation data 232
11.25 Printing a full validation review 233
11.26 Plotting a histogram of validation data $\hat{y}$ values 233
11.27 Calculating ROC AUC for validation data 234
11.28 Creating a `ydf` DataFrame of y and $\hat{y}$ values 234
11.29 Ten cases of negative validation reviews with high $\hat{y}$ scores 235
11.30 Ten cases of positive validation reviews with low $\hat{y}$ scores 235
11.31 Additional CNN dependencies 236
11.32 Convolutional sentiment classifier hyperparameters 236
11.33 Convolutional sentiment classifier architecture 237
11.34 RNN sentiment classifier hyperparameters 242
11.35 RNN sentiment classifier architecture 243
11.36 LSTM sentiment classifier hyperparameters 246
11.37 LSTM sentiment classifier architecture 247
11.38 Bidirectional LSTM sentiment classifier architecture 248
11.39 Stacked recurrent model architecture 249
11.40 Multi-ConvNet sentiment classifier hyperparameters 253
11.41 Multi-ConvNet sentiment classifier architecture 253
12.1 Generative adversarial network dependencies 264
12.2 Loading the Quick, Draw! data 265
12.3 Discriminator model architecture 266
12.4 Compiling the discriminator network 269
12.5 Generator model architecture 270
12.6 Adversarial model architecture 273
12.7 Compiling the adversarial network 274
12.8 GAN training 275

12.9 Plotting our GAN training loss 280
12.10 Plotting our GAN training accuracy 281
13.1 Cart-Pole DQN hyperparameters 293
13.2 A deep Q-learning agent 294
13.3 DQN agent interacting with an OpenAI Gym environment 300
14.1 Dependencies for building a Keras layer-based deep net in TensorFlow without loading the Keras library 323

This page intentionally left blank

Foreword

Machine learning is considered by many to be the future of statistics and computer engineering as it reshapes customer service, design, banking, medicine, manufacturing, and hosts of other disciplines and industries. It is hard to overstate its impact on the world so far and the changes it will bring about in the coming years and decades. Of the multitude of machine learning methods applied by professionals, such as penalized regression, random forests, and boosted trees, perhaps the most excitement-inducing is deep learning.

Deep learning has revolutionized computer vision and natural language processing, and researchers are still finding new areas to transform with the power of neural networks. Its most profound impact is often seen in efforts to replicate the human experience, such as the aforementioned vision and language processing, and also audio synthesis and translations. The math and concepts underlying deep learning can seem daunting, unnecessarily deterring people from getting started.

The authors of *Deep Learning Illustrated* challenge the traditionally perceived barriers and impart their knowledge with ease and levity, resulting in a book that is enjoyable to read. Much like the other books in this series—*R for Everyone*, *Pandas for Everyone*, *Programming Skills for Data Science,* and *Machine Learning with Python for Everyone*—this book is welcoming and accessible to a broad audience from myriad backgrounds. Mathematical notation is kept to a minimum and, when needed, the equations are presented alongside understandable prose. The majority of insights are augmented with visuals, illustrations, and Keras code, which is also available as easy-to-follow Jupyter notebooks.

Jon Krohn has spent many years teaching deep learning, including a particularly memorable presentation at the New York Open Statistical Programming Meetup—the same community from which he launched his Deep Learning Study Group. His mastery of the subject shines through in his writing, giving readers ample education while at the same time inviting them to be excited about the material. He is joined by Grant Beyleveld and Aglaé Bassens who add their expertise in applying deep learning algorithms and skillful drawings.

Deep Learning Illustrated combines theory, math where needed, code, and visualizations for a comprehensive treatment of deep learning. It covers the full breadth of the subject, including densely connected networks, convolutional neural nets, recurrent neural nets, generative adversarial networks, and reinforcement learning, and their applications. This makes the book the ideal choice for someone who wants to learn about neural networks with practical guidance for implementing them. Anyone can, and should, benefit from, as well as enjoy, their time spent reading along with Jon, Grant, and Aglaé.

—Jared Lander
Series Editor

This page intentionally left blank

Preface

Commonly called *brain cells*, billions of interconnected *neurons* make up your nervous system, and they enable you to sense, to think, and to take action. By meticulously staining and examining thin slices of brain tissue, the Spanish physician Santiago Cajal (Figure P.1), was the first[1] to identify neurons (Figure P.2), and in the early half of the twentieth century, researchers began to shed light on how these biological cells work. By the 1950s, scientists inspired by our developing understanding of the brain were experimenting with computer-based *artificial* neurons, linking these together to form *artificial neural networks* that loosely mimic the operation of their natural namesake.

Armed with this brief history of neurons, we can define the term *deep learning* deceptively straightforwardly: Deep learning involves a network in which artificial neurons—typically thousands, millions, or many more of them—are stacked at least several layers deep. The artificial neurons in the first layer pass information to the second, the second to the third, and so on, until the final layer outputs some values. That said, as we literally illustrate throughout this book, this simple definition does not satisfactorily capture deep learning's remarkable breadth of functionality nor its extraordinary nuance.

As we detail in Chapter 1, with the advent of sufficiently inexpensive computing power, sufficiently large datasets, and a handful of landmark theoretical advances, the first wave of the deep learning tsunami to hit the proverbial shore was a standout performance in a leading machine vision competition in 2012. Academics and technologists took note, and in the action-packed years since, deep learning has facilitated countless now-everyday

Figure P.1 Santiago Cajal (1852–1934)

1. Cajal, S.-R. (1894). *Les Nouvelles Idées sur la Structure du Système Nerveux chez l'Homme et chez les Vertébrés*. Paris: C. Reinwald & Companie.

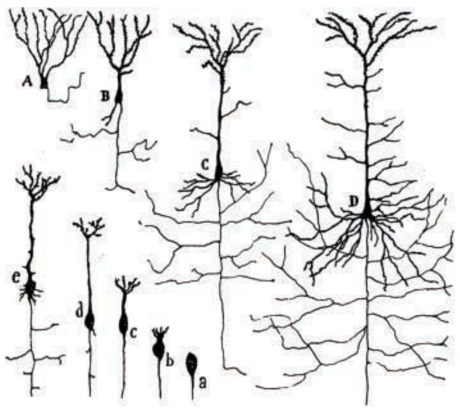


Figure P.2 A hand-drawn diagram from Cajal's (1894) publication showing the growth of a neuron (a–e) and contrasting neurons from frog (A), lizard (B), rat (C), and human (D) samples

applications. From Tesla's Autopilot to the voice recognition of Amazon's Alexa, from real-time translation between languages to its integration in hundreds of Google products, deep learning has improved the accuracy of a great number of computational tasks from 95 percent to 99 percent or better—the tricky few percent that can make an automated service feel as though it works by magic. Although the concrete, interactive code examples throughout this book will dispel this apparent wizardry, deep learning has indeed imbued machines with superhuman capability on complex tasks as diverse as face recognition, text summarization, and elaborate board games.[2] Given these prominent advances, it is unsurprising that "deep learning" has become synonymous with "artificial intelligence" in the popular press, the workplace, and the home.

These are exciting times, because, as you'll discover over the course of this book, perhaps only once in a lifetime does a single concept disrupt so widely in such a short period of time. We are delighted that you too have developed an interest in deep learning and we can't wait to share our enthusiasm for this unprecedentedly transformative technique with you.

How to Read This Book

This book is split into four parts. Part I, "Introducing Deep Learning," is well suited to any interested reader. This part serves as a high-level overview that establishes what deep learning is, how it evolved to be ubiquitous, and how it is related to concepts like AI, machine learning, and reinforcement learning. Replete with vivid bespoke illustrations, straightforward analogies, and character-focused narratives, Part I should be illuminating for anyone, including individuals with no software programming experience.

2. See `bit.ly/aiindex18` for a review of machine performance relative to humans.

In contrast, Parts II through IV are intended for software developers, data scientists, researchers, analysts, and others who would like to learn how to apply deep learning techniques in their field. In these parts of the book, essential underlying theory is covered in a manner that minimizes mathematical formulas, relying instead on intuitive visuals and hands-on examples in Python. Alongside this theory, working code run-throughs available in accompanying Jupyter notebooks[3] facilitate a pragmatic understanding of the principal families of deep learning approaches and applications: machine vision (Chapter 10), natural language processing (Chapter 11), image generation (Chapter 12), and game playing (Chapter 13). For clarity, wherever we refer to code, we will provide it in `fixed-width font, like this`. For further readability, in code chunks we also include the default Jupyter styling (e.g., numbers in green, strings in red, etc.).

If you find yourself yearning for more detailed explanations of the mathematical and statistical foundations of deep learning than we offer in this book, our two favorite options for further study are:

1. Michael Nielsen's e-book *Neural Networks and Deep Learning*,[4] which is short, makes use of fun interactive applets to demonstrate concepts, and uses mathematical notation similar to ours
2. Ian Goodfellow (introduced in Chapter 3), Yoshua Bengio (Figure 1.10), and Aaron Courville's book *Deep Learning*,[5] which comprehensively covers the math that underlies neural network techniques

Scattered throughout this book, you will find amiable trilobites that would like to provide you with tidbits of unessential reading that they think you may find interesting or helpful. The *reading trilobite* (as in Figure P.3) is a bookworm who enjoys expanding your knowledge. The *trilobite calling for your attention*, meanwhile (as in Figure P.4), has noticed a passage of text that may be problematic, and so would like to clarify the situation. In addition to trilobites habituated within sidebars, we made liberal use of footnotes. These

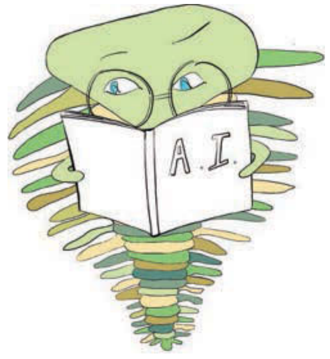


Figure P.3 The reading trilobite enjoys expanding your knowledge.

3. `github.com/the-deep-learners/deep-learning-illustrated`
4. Nielsen, M. (2015). *Neural Networks and Deep Learning*. Determination Press. Available for free at: `neuralnetworksanddeeplearning.com`
5. Goodfellow, I., et al. (2016). *Deep Learning*. MIT Press. Available for free at: `deeplearningbook.org`

Figure P.4 This trilobite calls attention to tricky passages of text. Look out for it!

are likewise not essential reading but provide quick explanations of new terms and abbreviations, as well as citations of seminal papers and other references for you to follow up with if you're so inclined.

For much of this book's content, corresponding video tutorials are also available. Although the book provided us with an opportunity to flesh out theoretical concepts more thoroughly, the videos enable you to become familiar with our Jupyter notebooks from a different perspective, in which the importance of each line of code is described verbally as it is typed out.[6] The video tutorial series is spread across three titles, each of which parallels particular chapters of the book:

1. *Deep Learning with TensorFlow LiveLessons*:[7] Chapter 1 and Chapters 5 through 10
2. *Deep Learning for Natural Language Processing LiveLessons*:[8] Chapters 2 and 11
3. *Deep Reinforcement Learning and GANs LiveLessons*:[9] Chapters 3, 4, 12, and 13

Register your copy of *Deep Learning Illustrated* on the InformIT site for convenient access to updates and corrections as they become available. To start the registration process, go to `informit.com/register` and log in or create an account. Enter the product ISBN (9780135116692) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

6. Many of the Jupyter notebooks covered in this book are derived directly from the videos, which were all recorded prior to writing. In some places, we decided to update the code for the book, so while the video version and the book version of a given code notebook align quite closely, they may not always be strictly identical.
7. Krohn, J. (2017). *Deep Learning with TensorFlow LiveLessons: Applications of Deep Neural Networks to Machine Learning Tasks* (video course). Boston: Addison-Wesley.
8. Krohn, J. (2017). *Deep Learning for Natural Language Processing LiveLessons: Applications of Deep Neural Networks to Machine Learning Tasks* (video course). Boston: Addison-Wesley.
9. Krohn, J. (2018). *Deep Reinforcement Learning and GANs LiveLessons: Advanced Topics in Deep Learning* (video course). Boston: Addison-Wesley.

Acknowledgments

We're grateful to the team at untapt, particularly Andrew Vlahutin, Sam Kenny, and Vince Petaccio II, who supported us while we wrote this book, with extra-special mention to the neural-network-loving Ed Donner, who ceaselessly encourages us to pursue our passions in the field of deep learning.

Additionally, we're grateful to the dozens of members of the Deep Learning Study Group[10] who regularly attend our stimulating and vibrant sessions at untapt's New York offices. Because the book is derived from our Study Group discussions, it is difficult to imagine how the book would otherwise have been conceived.

Thanks to our technical reviewers for invaluable feedback that noticeably improved the book's content: Alex Lipatov, Andrew Vlahutin, Claudia Perlich, Dmitri Nesterenko, Jason Baik, Laura Graesser, Michael Griffiths, Paul Dix, and Wah Loon Keng. Thanks to the book's editors and managers—Chris Zahn, Betsy Hardinger, Anna Popick, and Julie Nahil—whose meticulousness and thoughtfulness ensured the book's quality, clarity, and presentation. Thanks to Jared Lander, who leads the New York Open Statistical Programming community that both seeded our Deep Learning Study Group and facilitated meeting Pearson's Debra Williams Cauley. Special thanks are due to Debra herself, who has backed our colorful publication ideas from the day we met her and who has been instrumental in ensuring their success. Thanks as well to the scientists and machine learning experts who mentored us academically, and who continue to inspire us, particularly Jonathan Flint, Felix Agakov, and Will Valdar.

Finally, countless thanks are due to our families and our friends, who not only put up with us working through our time with them on vacations and weekends, but also selflessly motivate us as we do.

10. `deeplearningstudygroup.org`

This page intentionally left blank

About the Authors

Jon Krohn is Chief Data Scientist at the machine learning company untapt. He presents an acclaimed series of tutorials published by Addison-Wesley, including *Deep Learning with TensorFlow LiveLessons* and *Deep Learning for Natural Language Processing LiveLessons*. Jon teaches his deep learning curriculum in-classroom at the New York City Data Science Academy and guest lectures at Columbia University. He holds a doctorate in neuroscience from the University of Oxford and, since 2010, has been publishing on machine learning in leading peer-reviewed journals, including *Advances in Neural Information Processing Systems*.

Grant Beyleveld is a data scientist at untapt, where he works on natural language processing using deep learning. He holds a doctorate in biomedical science from the Icahn School of Medicine at New York City's Mount Sinai hospital, having studied the relationship between viruses and their hosts. He is a founding member of `deeplearningstudygroup.org`.

Aglaé Bassens is a Belgian artist based in Paris. She studied Fine Arts at The Ruskin School of Drawing and Fine Art, Oxford University, and University College London's Slade School of Fine Arts. Along with her work as an illustrator, her practice includes still life painting and murals.

This page intentionally left blank

8 Training Deep Networks

In the preceding chapters, we described artificial neurons comprehensively and we walked through the process of forward propagating information through a network of neurons to output a prediction, such as whether a given fast food item is a hot dog, a juicy burger, or a greasy slice of pizza. In those culinary examples from Chapters 6 and 7, we fabricated numbers for the neuron parameters—the neuron weights and biases. In real-world applications, however, these parameters are not typically concocted arbitrarily: They are learned by training the network on data.

In this chapter, you will become acquainted with two techniques—called *gradient descent* and *backpropagation*—that work in tandem to learn artificial neural network parameters. As usual in this book, our presentation of these methods is not only theoretical: We provide pragmatic best practices for implementing the techniques. The chapter culminates in the application of these practices to the construction of a neural network with more than one hidden layer.

Cost Functions

In Chapter 7, you discovered that, upon forward propagating some input values all the way through an artificial neural network, the network provides its estimated output, which is denoted $\hat{y}$. If a network were perfectly calibrated, it would output $\hat{y}$ values that are exactly equal to the true label y. In our binary classifier for detecting hot dogs, for example (Figure 7.3), $y = 1$ indicated that the object presented to the network is a hot dog, while $y = 0$ indicated that it's something else. In an instance where we have in fact presented a hot dog to the network, therefore, ideally it would output $\hat{y} = 1$.

In practice, the gold standard of $\hat{y} = y$ is not always attained and so may be an excessively stringent definition of the "correct" $\hat{y}$. Instead, if $y = 1$ we might be quite pleased to see a $\hat{y}$ of, say, 0.9997, because that would indicate that the network has an extremely high confidence that the object is a hot dog. A $\hat{y}$ of 0.9 might be considered acceptable, $\hat{y} = 0.6$ to be disappointing, and $\hat{y} = 0.1192$ (as computed in Equation 7.7) to be awful.

To quantify the spectrum of output-evaluation sentiments from "quite pleased" all the way down to "awful," machine learning algorithms often involve *cost functions* (also known as *loss functions*). The two such functions that we cover in this book are called quadratic cost and cross-entropy cost. Let's cover them in turn.

Quadratic Cost

Quadratic cost is one of the simplest cost functions to calculate. It is alternatively called *mean squared error*, which handily describes all that there is to its calculation:

$$C = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{8.1}$$

For any given instance i, we calculate the difference (the *error*) between the true label y_i and the network's estimated $\hat{y}_i$. We then *square* this difference, for two reasons:

1. Squaring ensures that whether y is greater than $\hat{y}$ or vice versa, the difference between the two is stated as a positive value.
2. Squaring penalizes large differences between y and $\hat{y}$ much more severely than small differences.

Having obtained a *squared error* for each instance i by using $(y_i - \hat{y}_i)^2$, we can then calculate the *mean* cost C across all n of our instances by:

1. Summing up cost across all instances using $\sum\limits_{i=1}^{n}$
2. Dividing by however many instances we have using $\frac{1}{n}$

By taking a peek inside the *Quadratic Cost* Jupyter notebook from the book's GitHub repo, you can play around with Equation 8.1 yourself. At the top of the notebook, we define a function to calculate the squared error for an instance i:

```
def squared_error(y, yhat):
    return (y - yhat)**2
```

By plugging a true `y` of `1` and the ideal `yhat` of `1` in to the function by using `squared_error(1, 1)`, we observe that—as desired—this perfect estimate is associated with a cost of `0`. Likewise, minor deviations from the ideal, such as a `yhat` of `0.9997`, correspond to an extremely small cost: `9.0e-08`.[1] As the difference between `y` and `yhat` increases, we witness the expected exponential increase in cost: Holding `y` steady at `1` but lowering `yhat` from `0.9` to `0.6`, and then to `0.1192`, the cost climbs increasingly rapidly from `0.01` to `0.16` and then to `0.78`. As a final bit of amusement in the notebook, we note that had `y` truly been `0`, our `yhat` of `0.1192` would be associated with a small cost: `0.0142`.

Saturated Neurons

While quadratic cost serves as a straightforward introduction to loss functions, it has a vital flaw. Consider Figure 8.1, in which we recapitulate the tanh activation function from Figure 6.10. The issue presented in the figure, called *neuron saturation*, is common across all activation functions, but we'll use tanh as our lone exemplar. A neuron is

1. `9.0e-08` is equivalent to 9.0×10^{-8}.

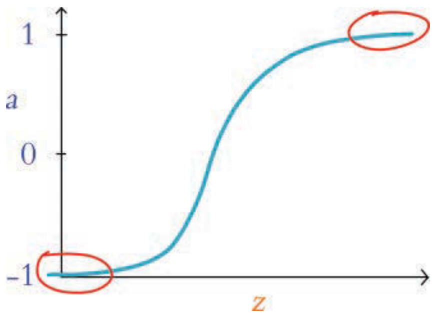


Figure 8.1 Plot reproducing the tanh activation function shown in Figure 6.10, drawing attention to the high and low values of z at which a neuron is saturated

considered saturated when the combination of its inputs and parameters (interacting as per "the most important equation," $z = w \cdot x + b$, which is captured in Figure 6.10) produces extreme values of z—the areas encircled with red in the plot in Figure 8.1. In these areas, changes in z (via adjustments to the neuron's underlying parameters w and b) cause only teensy-weensy changes in the neuron's activation a.[2]

Using methods that we cover later in this chapter—namely, gradient descent and backpropagation—a neural network is able to learn to approximate y through the tuning of the parameters w and b associated with all of its constituent neurons. In a saturated neuron, where changes to w and b lead to only minuscule changes in a, this learning slows to a crawl: If adjustments to w and b make no discernible impact on a given neuron's activation a, then these adjustments cannot have any discernible impact downstream (via forward propagation) on the network's $\hat{y}$, its estimate of y.

Cross-Entropy Cost

One of the ways[3] to minimize the impact of saturated neurons on learning speed is to use *cross-entropy cost* in lieu of quadratic cost. This alternative loss function is configured to enable efficient learning anywhere within the activation function curve of Figure 8.1. Because of this, it is a far more popular choice of cost function and it is the selection that predominates the remainder of this book.[4]

You need not preoccupy yourself with the equation for cross-entropy cost, but for the sake of completeness, here it is:

$$C = -\frac{1}{n}\sum_{i=1}^{n}[y_i \ln \hat{y}_i + (1 - y_i)\ln(1 - \hat{y}_i)] \tag{8.2}$$

2. Recall from Chapter 6 that $a = \sigma(z)$, where σ is some activation function—in this example, the tanh function.
3. More methods for attenuating saturated neurons and their negative effects on a network are covered in Chapter 9.
4. Cross-entropy cost is well suited to neural networks solving classification problems, and such problems dominate this book. For regression problems (covered in Chapter 9), quadratic cost is a better option than cross-entropy cost.

The most pertinent aspects of the equation are:

- Like quadratic cost, divergence of $\hat{y}$ from y corresponds to increased cost.
- Analogous to the use of the square in quadratic cost, the use of the natural logarithm ln in cross-entropy cost causes larger differences between $\hat{y}$ and y to be associated with exponentially larger cost.
- Cross-entropy cost is structured so that the larger the difference between $\hat{y}$ and y, *the faster the neuron is able to learn.*[5]

To make it easier to remember that the greater the cost, the more quickly a neural network incorporating cross-entropy cost learns, here's an analogy that would absolutely never involve any of your esteemed authors: Let's say you're at a cocktail party leading the conversation of a group of people that you've met that evening. The strong martini you're holding has already gone to your head, and you go out on a limb by throwing a risqué line into your otherwise charming repartee. Your audience reacts with immediate, visible disgust. With this response clearly indicating that your quip was well off the mark, you learn pretty darn quickly. It's exceedingly unlikely you'll be repeating the joke anytime soon.

Anyway, that's plenty enough on disasters of social etiquette. The final item to note on cross-entropy cost is that, by including $\hat{y}$, the formula provided in Equation 8.2 applies to only the output layer. Recall from Chapter 7 (specifically the discussion of Figure 7.3) that $\hat{y}$ is a special case of a: It's actually just another plain old a value—except that it's being calculated by neurons in the output layer of a neural network. With this in mind, Equation 8.2 could be expressed with a_i substituted in for $\hat{y}_i$ so that the equation generalizes neatly beyond the output layer to neurons in any layer of a network:

$$C = -\frac{1}{n}\sum_{i=1}^{n}[y_i \ln a_i + (1 - y_i)\ln(1 - a_i)] \tag{8.3}$$

To cement all of this theoretical chatter about cross-entropy cost, let's interactively explore our aptly named *Cross Entropy Cost* Jupyter notebook. There is only one dependency in the notebook: the `log` function from the *NumPy* package, which enables us to compute the natural logarithm ln shown twice in Equation 8.3. We load this dependency using `from numpy import log`.

Next, we define a function for calculating cross-entropy cost for an instance i:

```
def cross_entropy(y, a):
    return -1*(y*log(a) + (1-y)*log(1-a))
```

5. To understand how the cross-entropy cost function in Equation 8.2 enables a neuron with larger cost to learn more rapidly, we require a touch of partial-derivative calculus. (Because we endeavor to minimize the use of advanced mathematics in this book, we've relegated this calculus-focused explanation to this footnote.) Central to the two computational methods that enable neural networks to learn—gradient descent and backpropagation—is the comparison of the rate of change of cost C relative to neuron parameters like weight w. Using partial-derivative notation, we can represent these relative rates of change as $\frac{\partial C}{\partial w}$. The cross-entropy cost function is deliberately structured so that, when we calculate its derivative, $\frac{\partial C}{\partial w}$ is related to $(\hat{y} - y)$. Thus, the larger the difference between the ideal output y and the neuron's estimated output $\hat{y}$, the greater the rate of change of cost C with respect to weight w.

Table 8.1 Cross-entropy costs associated with selected example inputs

y	a	C
1	0.9997	0.0003
1	0.9	0.1
1	0.6	0.5
1	0.1192	2.1
0	0.1192	0.1269
1	1−0.1192	0.1269

Plugging the same values in to our `cross_entropy()` function as we did the `squared_error()` function earlier in this chapter, we observe comparable behavior. As shown in Table 8.1, by holding y steady at `1` and gradually decreasing a from the nearly ideal estimate of `0.9997` downward, we get exponential increases in cross-entropy cost. The table further illustrates that—again, consistent with the behavior of its quadratic cousin—cross-entropy cost would be low, with an a of `0.1192`, if y happened to in fact be `0`. These results reiterate for us that the chief distinction between the quadratic and cross-entropy functions is not the particular cost value that they calculate per se, but rather it is the rate at which they learn within a neural net—especially if saturated neurons are involved.

Optimization: Learning to Minimize Cost

Cost functions provide us with a quantification of how incorrect our model's estimate of the ideal y is. This is most helpful because it arms us with a metric we can leverage to reduce our network's incorrectness.

As alluded to a couple of times in this chapter, the primary approach for minimizing cost in deep learning paradigms is to pair an approach called gradient descent with another one called backpropagation. These approaches are *optimizers* and they enable the network to *learn*. This learning is accomplished by adjusting the model's parameters so that its estimated $\hat{y}$ gradually converges toward the target of y, and thus the cost decreases. We cover gradient descent first and move on to backpropagation immediately afterward.

Gradient Descent

Gradient descent is a handy, efficient tool for adjusting a model's parameters with the aim of minimizing cost, particularly if you have a lot of training data available. It is widely used across the field of machine learning, not only in deep learning.

In Figure 8.2, we use a nimble trilobite in a cartoon to illustrate how gradient descent works. Along the horizontal axis in each frame is some parameter that we've denoted as p. In an artificial neural network, this parameter would be either a neuron's weight w or bias b. In the top frame, the trilobite finds itself on a hill. Its goal is to *descend* the gradient, thereby finding the location with the minimum cost, C. But there's a twist: The trilobite

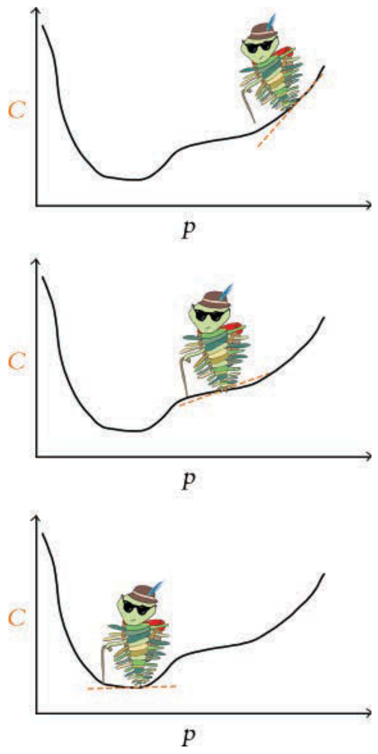


Figure 8.2 A trilobite using gradient descent to find the value of a parameter *p* associated with minimal cost, *C*

is blind! It cannot see whether deeper valleys lie far away somewhere, and so it can only use its cane to investigate the slope of the terrain in its immediate vicinity.

The dashed orange line in Figure 8.2 indicates the blind trilobite's calculation of the slope at the point where it finds itself. According to that slope line, if the trilobite takes a step to the left (i.e., to a slightly lower value of *p*), it would be moving to a location with smaller cost. On the hand, if the trilobite takes a step to the right (a slightly *higher* value of *p*), it would be moving to a location with *higher* cost. Given the trilobite's desire to descend the gradient, it chooses to take a step to the left.

By the middle frame, the trilobite has taken several steps to the left. Here again, we see it evaluating the slope with the orange line and discovering that, yet again, a step to the left will bring it to a location with lower cost, and so it takes another step left. In the lower frame, the trilobite has succeeded in making its way to the location—the value of the parameter *p*—corresponding to the minimum cost. From this position, if it were to take a step to the left *or* to the right, cost would go up, so it gleefully remains in place.

In practice, a deep learning model would not have only one parameter. It is not uncommon for deep learning networks to have millions of parameters, and some industrial applications have billions of them. Even our *Shallow Net in Keras*—one of the smallest models we build in this book—has 50,890 parameters (see Figure 7.5).

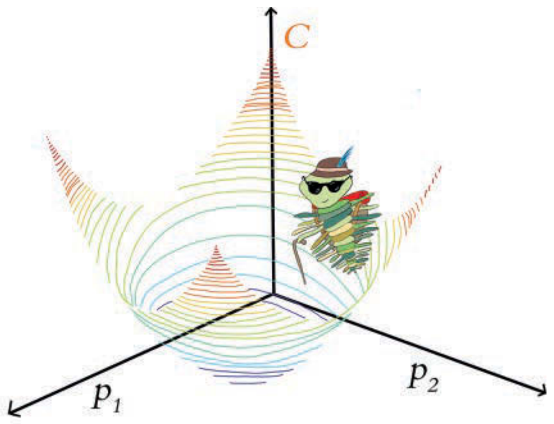


Figure 8.3 A trilobite exploring along two model parameters—p_1 and p_2—in order to minimize cost via gradient descent. In a mountain-adventure analogy, p_1 and p_2 could be thought of as latitude and longitude, and altitude represents cost.

Although it's impossible for the human mind to imagine a billion-dimensional space, the two-parameter cartoon shown in Figure 8.3 provides a sense of how gradient descent scales up to minimize cost across multiple parameters simultaneously. Across however many trainable parameters there are in a model, gradient descent iteratively evaluates slopes[6] to identify the adjustments to those parameters that correspond to the steepest reduction in cost. With two parameters, as in the trilobite cartoon in Figure 8.3, for example, this procedure can be likened to a blind hike through the mountains, where:

- Latitude represents one parameter, say p_1.
- Longitude represents the other parameter, p_2.
- Altitude represents cost—the lower the altitude, the better!

The trilobite randomly finds itself at a location in the mountains. From that point, it feels around with its cane to identify the direction of the step it can take that will reduce its altitude the most. It then takes that single step. Repeating this process many times, the trilobite may eventually find itself at the latitude and longitude coordinates that correspond to the lowest-possible altitude (the minimum cost), at which point the trilobite's surreal alpine adventure is complete.

Learning Rate

For conceptual simplicity, in Figure 8.4, let's return to a blind trilobite navigating a single-parameter world instead of a two-parameter world. Now let's imagine that we have a ray-gun that can shrink or enlarge trilobites. In the middle panel, we've used our ray-gun to make our trilobite very small. The trilobite's steps will then be correspondingly small, and so it will take our intrepid little hiker a long time to find its way to the

6. Using partial-derivative calculus.

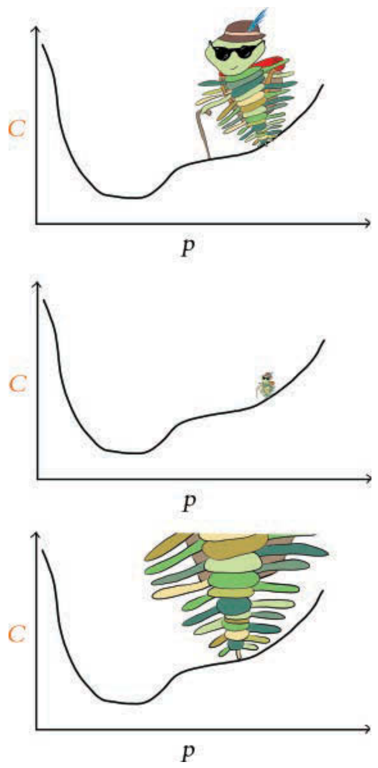


Figure 8.4 The learning rate (η) of gradient descent expressed as the size of a trilobite. The middle panel has a small learning rate, and the bottom panel, a large one.

legendary valley of minimum cost. On the other hand, consider the bottom panel, in which we've used our ray-gun to make the trilobite very large. The situation here is even worse! The trilobite's steps will now be so large that it will step right over the valley of minimum cost, and so it never has any hope of finding it.

In gradient descent terminology, step size is referred to as *learning rate* and denoted with the Greek letter η (eta, pronounced "ee-ta"). Learning rate is the first of several model *hyperparameters* that we cover in this book. In machine learning, including deep learning, hyperparameters are aspects of the model that we configure before we begin training the model. So hyperparameters such as η are preset while, in contrast, parameters—namely, $\boldsymbol{w}$ and $\boldsymbol{b}$—are learned during training.

Getting your hyperparameters right for a given deep learning model often requires some trial and error. For the learning rate η, it's something like the fairy tale of "Goldilocks and the Three Bears": Too small and too large are both inadequate, but there's a sweet spot in the middle. More specifically, as we portray in Figure 8.4, if η is too small, then it will take many, many iterations of gradient descent (read: an unnecessarily long time) to reach the minimal cost. On the other hand, selecting a value for η

that is too large means we might never reach minimal cost at all: The gradient descent algorithm will act erratically as it jumps right over the parameters associated with minimal cost.

Coming up in Chapter 9, we have a clever trick waiting for you that will circumnavigate the need for you to manually select a given neural network's η hyperparameter. In the interim, however, here are our rules of thumb on the topic:

- Begin with a learning rate of about 0.01 or 0.001.
- If your model is able to learn (i.e., if cost decreases consistently epoch over epoch) but training happens very slowly (i.e., each epoch, the cost decreases only a small amount), then increase your learning rate by an order of magnitude (e.g., from 0.01 to 0.1). If the cost begins to jump up and down erratically epoch over epoch, then you've gone too far, so rein in your learning rate.
- At the other extreme, if your model is unable to learn, then your learning rate may be too high. Try decreasing it by orders of magnitude (e.g., from 0.001 to 0.0001) until cost decreases consistently epoch over epoch. For a visual, interactive way to get a handle on the erratic behavior of a model when its learning rate is too high, you can return to the TensorFlow Playground example from Figure 1.18 and dial up the value within the "Learning rate" dropdown box.

Batch Size and Stochastic Gradient Descent

When we introduced gradient descent, we suggested that it is efficient for machine learning problems that involve a large dataset. In the strictest sense, we outright lied to you. The truth is that if we have a very large quantity of training data, *ordinary* gradient descent would not work at all because it wouldn't be possible to fit all of the data into the memory (RAM) of our machine.

Memory isn't the only potential snag; compute power could cause us headaches, too. A relatively large dataset might squeeze into the memory of our machine, but if we tried to train a neural network containing millions of parameters with all those data, vanilla gradient descent would be highly *in*efficient because of the computational complexity of the associated high-volume, high-dimensional calculations.

Thankfully, there's a solution to these memory and compute limitations: the *stochastic* variant of gradient descent. With this variation, we split our training data into *mini-batches*—small subsets of our full training dataset—to render gradient descent both manageable and productive.

Although we didn't focus on it at the time, when we trained the model in our *Shallow Net in Keras* notebook back in Chapter 5 we were already using stochastic gradient descent by setting our `optimizer` to `SGD` in the `model.compile()` step. Further, in the subsequent line of code when we called the `model.fit()` method, we set `batch_size` to `128` to specify the size of our mini-batches—the number of training data points that we use for a given iteration of SGD. Like the learning rate η presented earlier in this chapter, *batch size* is also a model hyperparameter.

Let's work through some numbers to make the concepts of batches and stochastic gradient descent more tangible. In the MNIST dataset, there are 60,000 training images.

With a batch size of 128 images, we then have $\lceil 468.75 \rceil = 469$ batches[7,8] of gradient descent per epoch:

$$\begin{aligned} \text{number of batches} &= \left\lceil \frac{\text{size of training dataset}}{\text{batch size}} \right\rceil \\ &= \left\lceil \frac{60,000 \text{ images}}{128 \text{ images}} \right\rceil \\ &= \lceil 468.75 \rceil \\ &= 469 \end{aligned} \tag{8.4}$$

Before carrying out any training, we initialize our network with random values for each neuron's parameters $\boldsymbol{w}$ and $\boldsymbol{b}$.[9] To begin the first epoch of training:

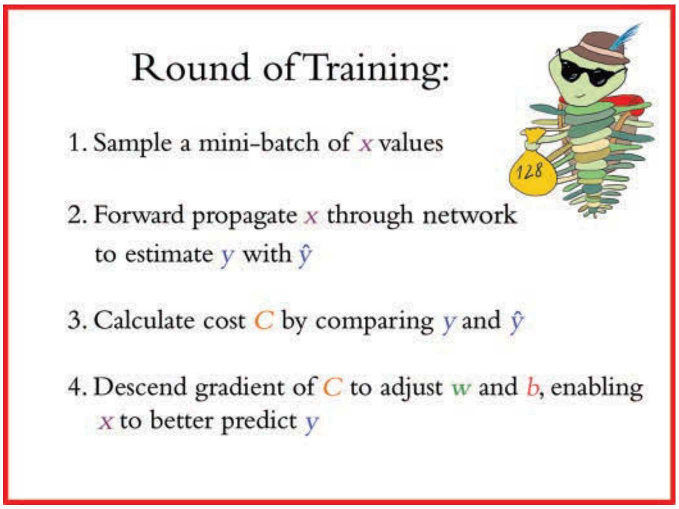
1. We shuffle and divide the training images into mini-batches of 128 images each. These 128 MNIST images provide 784 pixels each, which all together constitute the inputs $\boldsymbol{x}$ that are passed into our neural network. It's this shuffling step that puts the *stochastic* (which means *random*) in "stochastic gradient descent."
2. By forward propagation, information about the 128 images is processed by the network, layer through layer, until the output layer ultimately produces $\hat{\boldsymbol{y}}$ values.
3. A cost function (e.g., cross-entropy cost) evaluates the network's $\hat{\boldsymbol{y}}$ values against the true $\boldsymbol{y}$ values, providing a cost $\boldsymbol{C}$ for this particular mini-batch of 128 images.
4. To minimize cost and thereby improve the network's estimates of $\boldsymbol{y}$ given $\boldsymbol{x}$, the gradient descent part of stochastic gradient descent is performed: Every single $\boldsymbol{w}$ and $\boldsymbol{b}$ parameter in the network is adjusted proportional to how much each contributed to the error (i.e., the cost) in this batch (note that the adjustments are scaled by the learning rate hyperparameter η).[10]

These four steps constitute a *round of training*, as summarized by Figure 8.5.

Figure 8.6 captures how rounds of training are repeated until we run out of training images to sample. The sampling in step 1 is done *without replacement*, meaning that at the end of an epoch each image has been seen by the algorithm only once, and yet between different epochs the mini-batches are sampled randomly. After a total of 468 rounds, the final batch contains only 96 samples.

This marks the end of the first epoch of training. Assuming we've set our model up to train for further epochs, we begin the next epoch by replenishing our pool with all 60,000 training images. As we did through the previous epoch, we then proceed through a further 469 rounds of stochastic gradient descent.[11] Training continues in this way until the total desired number of epochs is reached.

7. Because 60,000 is not perfectly divisible by 128, that 469th batch would contain only $0.75 \times 128 = 96$ images.
8. The square brackets we use here and in Equation 8.4 that appear to be missing the horizontal element from the bottom are used to denote the calculation of an integer-value ceiling. The whole-integer ceiling of 468.75, for example, is 469.
9. We delve into the particulars of parameter initialization with random values in Chapter 9.
10. This error-proportional adjustment is calculated during backpropagation. We haven't covered backpropagation explicitly yet, but it's coming up in the next section, so hang on tight!
11. Because we're sampling randomly, the order in which we select training images for our 469 mini-batches is completely different for every epoch.



Round of Training:

1. Sample a mini-batch of x values
2. Forward propagate x through network to estimate y with $\hat{y}$
3. Calculate cost C by comparing y and $\hat{y}$
4. Descend gradient of C to adjust w and b, enabling x to better predict y

Figure 8.5 An individual round of training with stochastic gradient descent. Although mini-batch size is a hyperparameter that can vary, in this particular case, the mini-batch consists of 128 MNIST digits, as exemplified by our hike-loving trilobite carrying a small bag of data.

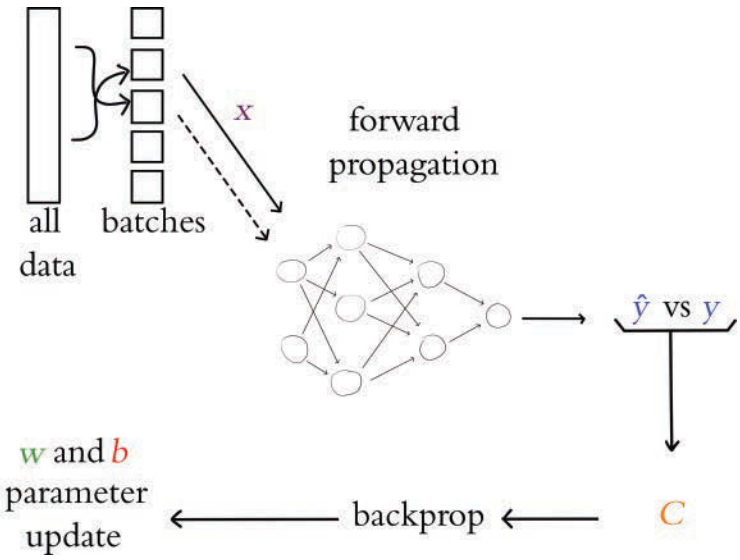


Figure 8.6 An outline of the overall process for training a neural network with stochastic gradient descent. The entire dataset is shuffled and split into batches. Each batch is forward propagated through the network; the output $\hat{y}$ is compared to the ground truth y and the cost C is calculated; backpropagation calculates the gradients; and the model parameters w and b are updated. The next batch (indicated by a dashed line) is forward propagated, and so on until all of the batches have moved through the network. Once all the batches have been used, a single epoch is complete and the process starts again with a reshuffling of the full training dataset.

The total *number of epochs* that we set our network to train for is yet another hyperparameter, by the way. This hyperparameter, though, is one of the easiest to get right:

- If the cost on your validation data is going down epoch over epoch, and if your final epoch attained the lowest cost yet, then you can try training for additional epochs.
- Once the cost on your validation data begins to creep upward, that's an indicator that your model has begun to *overfit* to your training data because you've trained for too many epochs. (We elaborate much more on overfitting in Chapter 9.)
- There are methods[12] you can use to automatically monitor training and validation cost and stop training early if things start to go awry. In this way, you could set the number of epochs to be arbitrarily large and know that training will continue until the validation cost stops improving—and certainly before the model begins overfitting!

Escaping the Local Minimum

In all of the examples of gradient descent thus far in the chapter, our hiking trilobite has encountered no hurdles on its journey toward minimum cost. There are no guarantees that this would be the case, however. Indeed, such smooth sailing is unusual.

Figure 8.7 shows the mountaineering trilobite exploring the cost of some new model that is being used to solve some new problem. With this new problem, the relationship between the parameter p and cost C is more complex. To have our neural network estimate y as accurately as possible, gradient descent needs to identify the parameter values associated with the lowest-attainable cost. However, as our trilobite makes its way from its random starting point in the top panel, gradient descent leads it to getting trapped in a *local minimum*. As shown in the middle panel, while our intrepid explorer is in the local minimum, a step to the left or a step to the right both lead to an increase in cost, and so the blind trilobite stays put, completely oblivious of the existence of a deeper valley—the *global minimum*—lying yonder.

All is not lost, friends, for stochastic gradient descent comes to the rescue here again. The sampling of mini-batches can have the effect of smoothing out the cost curve, as exemplified by the dashed curve shown in the bottom panel of Figure 8.7. This smoothing happens because the estimate is noisier when estimating the gradient from a smaller mini-batch (versus from the entire dataset). Although the actual gradient in the local minimum truly is zero, estimates of the gradient from small subsets of the data don't provide the complete picture and might give an inaccurate reading, causing our trilobite to take a step left thinking there is a gradient when there really isn't one. This noisiness and inaccuracy is paradoxically a good thing! The incorrect gradient may result in a step that is large enough for the trilobite to escape the local valley and continue making its way down the mountain. Thus, by estimating the gradient many times on these mini-batches, the noise is smoothed out and we are able to avoid local minima. In summary, although each mini-batch on its own lacks complete information about the cost curve, in the long run—over a large number of mini-batches—this tends to work to our advantage.

12. See `keras.io/callbacks/#earlystopping`.

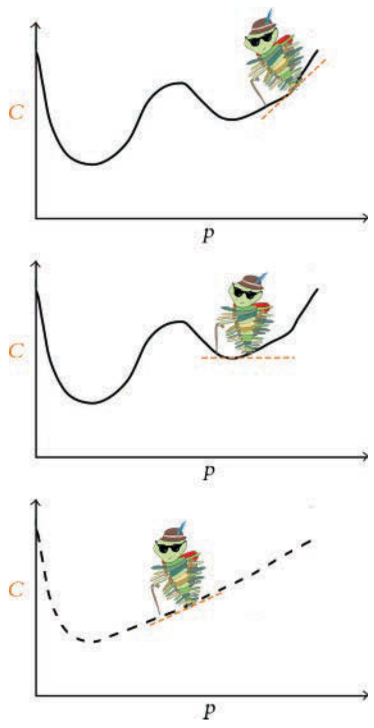


Figure 8.7 A trilobite applying vanilla gradient descent from a random starting point (top panel) is ensnared by a local minimum of cost (middle panel). By turning to stochastic gradient descent in the bottom panel, the daring trilobite is able to bypass the local minimum and make its way toward the global minimum.

Like the learning rate hyperparameter η, there is also a Goldilocks-style sweet spot for batch size. If the batch size is too large, the estimate of the gradient of the cost function is far more accurate. In this way, the trilobite has a more exacting impression of the gradient in its immediate vicinity and is able to take a step (proportional to η) in the direction of the steepest possible descent. However, the model is at risk of becoming trapped in local minima as described in the preceding paragraph.[13] Besides that, the model might not fit in memory on your machine, and the compute time per iteration of gradient descent could be very long.

On the other hand, if the batch size is too small, each gradient estimate may be excessively noisy (because a very small subset of the data is being used to estimate the gradient of the entire dataset) and the corresponding path down the mountain will be unnecessarily circuitous; training will take longer because of these erratic gradient descent steps. Furthermore, you're not taking advantage of the memory and compute resources on your

13. It's worth noting that the learning rate η plays a role here. If the size of the local minimum was *smaller* than the step size, the trilobite would likely breeze right past the local minimum, akin to how we step over cracks in the sidewalk.

machine.[14] With that in mind, here are our rules of thumb for finding the batch-size sweet spot:

- Start with a batch size of 32.
- If the mini-batch is too large to fit into memory on your machine, try decreasing your batch size by powers of 2 (e.g., from 32 to 16).
- If your model trains well (i.e., cost is going down consistently) but each epoch is taking very long and you are aware that you have RAM to spare,[15] you could experiment with increasing your batch size. To avoid getting trapped in local minima, we don't recommend going beyond 128.

Backpropagation

Although stochastic gradient descent operates well on its own to adjust parameters and minimize cost in many types of machine learning models, for deep learning models in particular there is an extra hurdle: We need to be able to efficiently adjust parameters *through multiple layers* of artificial neurons. To do this, stochastic gradient descent is partnered up with a technique called *backpropagation*.

Backpropagation—or backprop for short—is an elegant application of the "chain rule" from calculus.[16] As shown along the bottom of Figure 8.6 and as suggested by its very name, backpropagation courses through a neural network in the opposite direction of forward propagation. Whereas forward propagation carries information about the input $\boldsymbol{x}$ through successive layers of neurons to approximate $\boldsymbol{y}$ with $\hat{\boldsymbol{y}}$, backpropagation carries information about the cost C *backwards* through the layers in reverse order and, with the overarching aim of reducing cost, adjusts neuron parameters throughout the network.

Although the nitty-gritty of backpropagation has been relegated to Appendix B, it's worth understanding (in broad strokes) what the backpropagation algorithm does: Any given neural network model is randomly initialized with parameter ($\boldsymbol{w}$ and $\boldsymbol{b}$) values (such initialization is detailed in Chapter 9). Thus, prior to any training, when the first $\boldsymbol{x}$ value is fed in, the network outputs a random guess at $\hat{\boldsymbol{y}}$. This is unlikely to be a good guess, and the cost associated with this random guess will probably be high. At this point, we need to update the weights in order to minimize the cost—the very essence of machine learning. To do this within a neural network, we use backpropagation to calculate the *gradient* of the cost function with respect to each weight in the network.

14. Stochastic gradient descent with a batch size of 1 is known as *online learning*. It's worth noting that this is not the fastest method in terms of compute. The matrix multiplication associated with each round of mini-batch training is highly optimized, and so training can be several orders of magnitude quicker when using moderately sized mini-batches relative to online learning.

15. On a Unix-based operating system, including macOS, RAM usage may be assessed by running the `top` or `htop` command within a Terminal window.

16. To elucidate the mathematics underlying backpropagation, a fair bit of partial-derivative calculus is necessary. While we encourage the development of an in-depth understanding of the beauty of backprop, we also appreciate that calculus might not be the most appetizing topic for everyone. Thus, we've placed our content on backprop mathematics in Appendix B.

Recall from our mountaineering analogies earlier that the cost function represents a hiking trail, and our trilobite is trying to reach basecamp. At each step along the way, the trilobite finds the gradient (or the slope) of the cost function and moves *down* that gradient. That movement corresponds to a weight update: By adjusting the weight in proportion to the cost function's gradient *with respect to that weight*, backprop adjusts that weight in a direction that reduces the cost.

Reflecting back on the "most important equation" from Figure 6.7 ($\boldsymbol{w} \cdot \boldsymbol{x} + \boldsymbol{b}$), and remembering that neural networks are stacked with information forward propagating through their layers, we can grasp that any given weight in the network contributes to the final $\hat{\boldsymbol{y}}$ output, and thus the cost C. Using backpropagation, we move layer-by-layer backwards through the network, starting at the cost in the output layer, and we find the gradients of every single parameter. A given parameter's gradient can then be used to adjust the parameter up or down (by an increment corresponding to the learning rate η)—whichever of the two directions is associated with a reduction in cost.

We appreciate that this is not the lightest section of this book. If there's only one thing you take away, let it be this: Backpropagation uses cost to calculate the relative contribution by every single parameter to the total cost, and then it updates each parameter accordingly. In this way, the network iteratively reduces cost and, well . . . learns!

Tuning Hidden-Layer Count and Neuron Count

As with learning rate and batch size, the number of hidden layers you add to your neural network is also a hyperparameter. And as with the previous two hyperparameters, there is yet again a Goldilocks sweet spot for your network's count of layers. Throughout this book, we've reiterated that with each additional hidden layer within a deep learning network, the more abstract the representations that the network can represent. That is the primary advantage of adding layers.

The *dis*advantage of adding layers is that backpropagation becomes less effective: As demonstrated by the plot of learning speed across the layers of a five-hidden-layer network in Figure 8.8, backprop is able to have its greatest impact on the parameters of the hidden layer of neurons closest to the output $\hat{\boldsymbol{y}}$.[17] The farther a layer is from $\hat{\boldsymbol{y}}$, the more diluted the effect of that layer's parameters on the overall cost. Thus, the fifth layer, which is closest to the output $\hat{\boldsymbol{y}}$, learns most rapidly because those weights are associated with larger gradients. In contrast, the third hidden layer, which is several layers away from the output layer's cost calculation, learns about an order of magnitude more slowly than the fifth hidden layer.

Given the above, our rules of thumb for selecting the number of hidden layers in a network are:

- The more abstract the ground-truth value $\boldsymbol{y}$ you'd like to estimate with your network, the more helpful additional hidden layers may be. With that in mind, we recommend starting off with about two to four hidden layers.

17. If you're curious as to how we made Figure 8.8, check out our *Measuring Speed of Learning* Jupyter notebook.

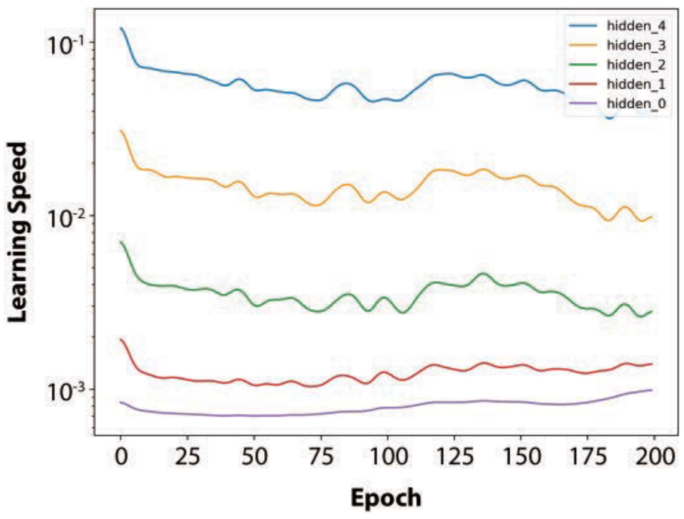


Figure 8.8 The speed of learning over epochs of training for a deep learning network with five hidden layers. The fifth hidden layer, which is closest to the output $\hat{y}$, learns about an order of magnitude more quickly than the third hidden layer.

- If reducing the number of layers does not increase the cost you can achieve on your validation dataset, then do it. Following the problem-solving principle called *Occam's razor*, the simplest network architecture that can provide the desired result is the best; it will train more quickly and require fewer compute resources.
- On the other hand, if increasing the number of layers decreases the validation cost, then you should pile up those layers!

Not only is network depth a model hyperparameter, but the number of neurons in a given layer is, too. If you have many layers in your network, then there are many layers you could be fine-tuning your neuron count in. This may seem intimidating at first, but it's nothing to be too concerned about: A few too many neurons, and your network will have a touch more computational complexity than is necessary; a touch too few neurons, and your network's accuracy may be held back imperceptibly.

As you build and train more and more deep learning models for more and more problems, you'll begin to develop a sense for how many neurons might be appropriate in a given layer. Depending on the particular data you're modeling, there may be lots of low-level features to represent, in which case you might want to have more neurons in the network's early layers. If there are lots of higher-level features to represent, then you may benefit from having additional neurons in its later layers. To determine this empirically, we generally experiment with the neuron count in a given layer by varying it by powers of 2. If doubling the number of neurons from 64 to 128 provides an appreciable improvement in model accuracy, then go for it. Rehashing Occam's razor, however, consider this: If halving the number of neurons from 64 to 32 doesn't detract from model accuracy, then that's probably the way to go because you're reducing your model's computational complexity with no apparent negative effects.

An Intermediate Net in Keras

To wrap up this chapter, let's incorporate the new theory we've covered into a neural network to see if we can outperform our previous *Shallow Net in Keras* model at classifying handwritten digits.

The first few stages of our *Intermediate Net in Keras* Jupyter notebook are identical to those of its *Shallow Net* predecessor. We load the same Keras dependencies, load the MNIST dataset in the same way, and preprocess the data in the same way. As shown in Example 8.1, the situation begins to get interesting when we design our neural network architecture.

Example 8.1 Keras code to architect an intermediate-depth neural network

```
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(784,)))
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

The first line of this code chunk, `model = Sequential()`, is the same as before (refer to Example 5.6); this is our instantiation of a neural network model object. It's in the second line that we begin to diverge. In it, we specify that we'll substitute the sigmoid activation function in the first hidden layer with our most-highly-recommended neuron from Chapter 6, the `relu`. Other than this activation function swap, the first hidden layer remains the same: It still consists of `64` neurons, and the dimensionality of the `784`-neuron input layer is unchanged.

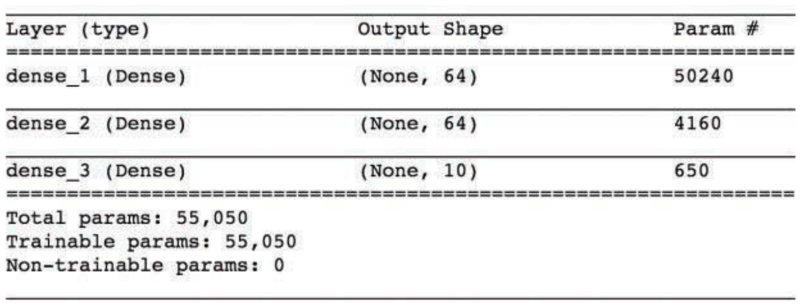
The other significant change in Example 8.1 relative to the shallow architecture of Example 5.6 is that we specify a second hidden layer of artificial neurons. By calling the `model.add()` method, we nearly effortlessly add a second `Dense` layer of `64 relu` neurons, providing us with the notebook's namesake: an intermediate-depth neural network. With a call to `model.summary()`, you can see from Figure 8.9 that this additional layer corresponds to an additional 4,160 trainable parameters relative to our shallow architecture (refer to Figure 7.5). We can break these parameters down into:

- 4,096 weights, corresponding to each of the 64 neurons in the second hidden layer densely receiving input from each of the 64 neurons in the first hidden layer ($64 \times 64 = 4{,}096$)
- Plus 64 biases, one for each of the neurons in the second hidden layer
- Giving us a total of 4,160 parameters: $n_{parameters} = n_w + n_b = 4{,}096 + 64 = 4{,}160$

In addition to changes to the model architecture, we've also made changes to the parameters we specify when compiling our model, as shown in Example 8.2.

Example 8.2 Keras code to compile our intermediate-depth neural network

```
model.compile(loss='categorical_crossentropy',
              optimizer=SGD(lr=0.1),
              metrics=['accuracy'])
```



```
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 64)                50240
_________________________________________________________________
dense_2 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_3 (Dense)              (None, 10)                650
=================================================================
Total params: 55,050
Trainable params: 55,050
Non-trainable params: 0
_________________________________________________________________
```

Figure 8.9 A summary of the model object from our *Intermediate Net in Keras* Jupyter notebook

With these lines from Example 8.2, we:

- Set our loss function to cross-entropy cost by using `loss='categorical_crossentropy'` (in *Shallow Net in Keras*, we used quadratic cost by using `loss='mean_squared_error'`)
- Set our cost-minimizing method to stochastic gradient descent by using `optimizer=SGD`
- Specify our SGD learning rate hyperparameter η by setting `lr=0.1`[18]
- Indicate that, in addition to the Keras default of providing feedback on `loss`, by setting `metrics=['accuracy']`, we'd also like to receive feedback on model accuracy[19]

Finally, we train our intermediate net by running the code in Example 8.3.

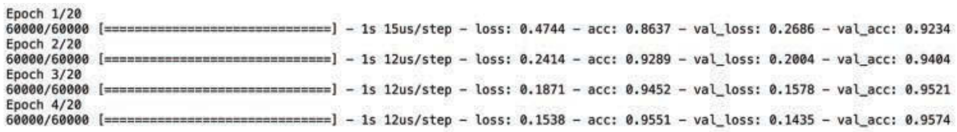
Example 8.3 Keras code to train our intermediate-depth neural network

```
model.fit(X_train, y_train,
          batch_size=128, epochs=20,
          verbose=1,
          validation_data=(X_valid, y_valid))
```

Relative to the way we trained our shallow net (see Example 5.7), the only change we've made is reducing our `epochs` hyperparameter from `200` down by an order of magnitude

18. On your own time, you can play around with increasing this learning rate by several orders of magnitude as well as decreasing it by several orders of magnitude, and observing how it impacts training.

19. Although loss provides the most important metric for tracking a model's performance epoch over epoch, its particular values are specific to the characteristics of a given model and are not generally interpretable or comparable between models. Because of this, other than knowing that we would like our loss to be as close to zero as possible, it can be an esoteric exercise to interpret how close to zero loss should be for any particular model. Accuracy, on the other hand, is highly interpretable and highly generalizable: We know exactly what it means (e.g., "The shallow neural network correctly classified 86 percent of the handwritten digits in the validation dataset"), and we can compare this classification accuracy to any other model ("The accuracy of 86 percent is worse than the accuracy of our deep neural network").



```
Epoch 1/20
60000/60000 [==============================] - 1s 15us/step - loss: 0.4744 - acc: 0.8637 - val_loss: 0.2686 - val_acc: 0.9234
Epoch 2/20
60000/60000 [==============================] - 1s 12us/step - loss: 0.2414 - acc: 0.9289 - val_loss: 0.2004 - val_acc: 0.9404
Epoch 3/20
60000/60000 [==============================] - 1s 12us/step - loss: 0.1871 - acc: 0.9452 - val_loss: 0.1578 - val_acc: 0.9521
Epoch 4/20
60000/60000 [==============================] - 1s 12us/step - loss: 0.1538 - acc: 0.9551 - val_loss: 0.1435 - val_acc: 0.9574
```

Figure 8.10 The performance of our intermediate-depth neural network over its first four epochs of training

to `20`. As you'll see, our much-more-efficient intermediate architecture required far fewer epochs to train.

Figure 8.10 provides the results of the first three epochs of training the network. Recalling that our shallow architecture plateaued as it approached 86 percent accuracy on the validation dataset after 200 epochs, our intermediate-depth network is clearly superior: The `val_acc` field shows that we attained 92.34 percent accuracy *after a single epoch of training.* This accuracy climbs to more than 95 percent by the third epoch and appears to plateau around 97.6 percent by the twentieth. My, how far we've come already!

Let's break down the verbose `model.fit()` output shown in Figure 8.10 in further detail:

- The progress bar shown next fills in over the course of the 469 "rounds of training" (Figure 8.5):
  `60000/60000 [==============================]`
- `1s 15us/step` indicates that all 469 rounds in the first epoch required 1 second to train, at an average rate of 15 microseconds per round.
- `loss` shows the average cost on our training data for the epoch. For the first epoch this is `0.4744`, and, epoch over epoch, this cost is reliably minimized via stochastic gradient descent (SGD) and backpropagation, eventually diminishing to `0.0332` by the twentieth epoch.
- `acc` is the classification accuracy on training data for the epoch. The model correctly classified 86.37 percent for the first epoch, increasing to more than 99 percent by the twentieth. Because a model can overfit to the training data, one shouldn't be overly impressed by high accuracy on the training data.
- Thankfully, our cost on the validation dataset (`val_loss`) does generally decrease as well, eventually plateauing as it approaches 0.08 over the final five epochs of training.
- Corresponding to the decreasing cost of the validation data is an increase in accuracy (`val_acc`). As mentioned, validation accuracy plateaued at about 97.6 percent, which is a vast improvement over the 86 percent of our shallow net.

Summary

We covered a lot of ground in this chapter. Starting from an appreciation of how a neural network with fixed parameters processes information, we developed an understanding of the cooperating methods—cost functions, stochastic gradient descent, and backpropagation—that enable network parameters to be learned so that we can approximate any y that has a continuous relationship to some input x. Along the way, we introduced several network hyperparameters, including learning rate, mini-batch size, and number of epochs of training—as well as our rules of thumb for configuring each of these. The chapter concluded by applying your newfound knowledge to develop an intermediate-depth neural network that greatly outperformed our previous, shallow network on the same handwritten-digit-classification task. Up next, we have techniques for improving the stability of artificial neural networks as they deepen, enabling you to architect and train a bona fide deep learning model for the first time.

Key Concepts

Here are the essential foundational concepts thus far. New terms from the current chapter are highlighted in purple.

- parameters:
  - weight w
  - bias b
- activation a
- artificial neurons:
  - sigmoid
  - tanh
  - ReLU
- input layer
- hidden layer
- output layer
- layer types:
  - dense (fully connected)
  - softmax
- cost (loss) functions:
  - quadratic (mean squared error)
  - cross-entropy
- forward propagation
- backpropagation
- optimizers:
  - stochastic gradient descent
- optimizer hyperparameters:
  - learning rate η
  - batch size

Index

A

`act()` method, defining DQN agent, 299–300, 301

Action potential, of biological neurons, 85–86

Action(s)
- deep Q-learning network theory, 290–292
- DeepMind DQN and, 59
- DQN agent, 298–300
- Markov decision processes and, 286
- reinforcement learning problems and, 55–56

Activation functions
- calculus behind backpropagation, 335–336
- choosing neuron type, 96
- convolutional example, 164–166
- Glorot distributions, 136–137
- nonlinear nature of in deep learning architectures, 95
- ReLU neuron, 94–95
- saturated neurons, 112–113
- sigmoid neuron example, 92–94
- softmax layer of fast food-classifying network, 106–108
- tanh neuron, 94

Activation maps
- convolutional networks and, 238
- in discriminator network, 267–268
- Faster R-CNN, 184–185
- in generator network, 269, 272
- LeNet-5 ConvNet architecture, 173–175
- as output from convolutional kernels, 163–167

Activation maps (*continued*)
with padding, 168–169
pooling layers spatially reducing, 169–170
U-Net, 187–188
Actor-critic algorithm, RL agent, 307–308
AdaDelta optimizer, 146–147
AdaGrad optimizer, 146
Adaptive moment estimation (Adam) optimizer, 147, 148–149
Adversarial network, GANs, 272–274
Agent(s)
beyond DQN, 306–308
deep Q-learning network theory, 290–292
deep reinforcement learning and, 57
DeepMind DQN, 58
DQN. *See* DQN agents
optimal policy in deep reinforcement learning, 289–290
reinforcement learning problems of machine learning, 54–56
reinforcement learning theory, 283
SLM Lab, 304
AGI (artificial general intelligence), 72, 326–328
AI. *See* Artificial intelligence (AI)
AlexNet
CNN model inspired by, 176–177
history of deep learning for NLP, 25
overview of, 14–17
ReLU neurons in, 95
Algorithms, development of AGI and, 327
AlphaZero, 53–54, 65–66
AlphaGo, 59–62, 63–64
AlphaGo Master, 64
AlphaGo Zero, 62–65
Amazon review polarity, NLP training/validation samples, 316
ANI (artificial narrow intelligence), 72, 326–327
Architecture
adversarial model, 273–274
AlexNet hierarchical, 16
bidirectional LSTM sentiment classifier, 248
convolutional sentiment classifier, 237–238
deep learning model, 52
deep net in Keras model, 148–149
dense sentiment classifier, 229–231
discriminator model, 266–269
generalist neural network as single network, 52
generator model, 270–272
intermediate-depth neural network, 127–128
Keras functional API, 251–256
LeNet-5 hierarchical, 9–11, 172–176
LSTM, 247
multi-ConvNet sentiment classifier, 253–254
regression model network, 150–151
residual network, 182
RNN sentiment classifier, 243–244
shallow neural network, 78–79, 83
stacked recurrent model, 249
TensorFlow Playground, 17
U-Net, 47, 187
weight initialization, 133–135
word2vec, 207–213
Arithmetic
on fake human faces, 41–44
word-vector, 29–30
Art. *See* Machine art
Artificial general intelligence (AGI), 72, 326–328
Artificial intelligence (AI)
categories of, 71–72
deep learning for NLP relevant to, 53
driven by deep learning, 52
general-purpose learning algorithms for, 58
history of chess and, 65
machine learning as subset of, 50
OpenAI Gym environments as, 68–70
overview of, 49–50
Artificial narrow intelligence (ANI), 72, 326–327

Artificial neural networks (ANNs). ***See also*** **Artificial neurons, constituting ANNs**
AlphaGo Zero development, 63
AlphaZero development, 65–66
architecture for shallow networks, 83–84
birth of GANs via, 40–41
building model for DQN agent, 297–298
deep reinforcement learning using, 56–57
dense layers, 99–100
dominating representation learning, 51
hot dog-detecting dense network, 101–106
input layer, 99
key concepts, 110
manipulation of objects via, 67–68
schematic diagram of Jupyter network, 77–79
shallow networks and, 108–110
softmax layer of fast food-classifying network, 106–108
summary, 110
Artificial neurons
deep learning and, 22
deep learning model architectures, 51–52
Artificial neurons, constituting ANNs
biological neuroanatomy, 85–86
choosing neuron type, 96
hot dog/not hot dog detector, 86–90
key concepts, 97
modern neurons/activation functions, 91–95
most important equation in this book, 90–91
overview of, 85
perceptrons as early, 86
ReLU neuron, 94–95
sigmoid neuron, 92–94
summary, 96
tanh neuron, 94
Artificial super intelligence (ASI), 72, 327
`astype()` method, LeNet-5 in Keras, 172
Atari Games, DeepMind, 58–59
Attention, seq2seq and, 250
Automatic differentiation, PyTorch, 342–343

B

Backpropagation
of bidirectional LSTMs, 247
cross-entropy costs and, 114
enabling neural networks to learn, 113
LeNet-5 model, 10–12
minimizing cost, 115
overview of, 124–125
partial-derivative calculus behind, 335–337
training recurrent neural networks, 241
tuning hidden-layer and neuron counts, 125–126
BAIR (Berkeley Artificial Intelligence Research) Lab, 44–45
Batch normalization
deep neural networks in Keras, 148
improving deep networks, 138–139
network architecture regression model, 150–151
Batch size
of 1, also known as online learning, 124
building own project and, 320
escaping local minimum of cost, 122–124
as hyperparameter, 119
and stochastic gradient descent, 119–122
Bazinska, Julia, 30–31
Behavioral cloning, 307
Benchmarking performance, SLM lab, 304
Bengio, Yoshua
LeNet-5 model, 9–12
Turing Award for deep learning, 15
weight initialization and Glorot normal distribution, 135–137
Berkeley Artificial Intelligence Research (BAIR) Lab, 44–45

BERT (bi-directional encoder representations from transformers), NLP, 251
beta (β) hyperparameter
  batch normalization adding, 139
  optimizing SGD, 145–147
Bi-directional encoder representations from transformers (BERT), NLP, 251
bias (*b*)
  adding to convolutional layers, 162
  in convolutional example, 164
  minimizing cost via gradient descent, 115–116
  notation for neural networks, 333
  in perceptron equation, 90–91
Bidirectional LSTMs (Bi-LSTMs), 247–249
Bigram collocation, 202–206
Binary-only restriction, of perceptrons, 91–92
Biological neurons
  anatomy of, 85–86
  creating perceptron algorithm with, 86
  ReLU neuron activation function, 94–95
Biological vision, 8, 20
Board games
  AlphaZero and, 65–66
  AlphaGo and, 59–62
  AlphaGo Zero and, 62–65
  overview of, 59
`boston_housing` dataset, 149–150
Bostrom, Nick, 72
Bounding boxes, developing YOLO, 185–186
`build_discriminator` function, 266–268
`_build_model()` method, DQN agent, 296–298
Burges, Chris, MNIST dataset, 77–78

C

Caffe, deep learning library, 324
Calculus, in backpropagation, 335–337
`callbacks` argument
  dense sentiment classifier, 232
  TensorBoard, 152, 154
Cambrian explosion, 3
Capsule networks, machine vision and, 192
Cart-Pole game
  defining DQN agent for. *See* DQN agents
  DQN agent interaction with OpenAI Gym, 300–303
  estimating optimal Q-value, 292
  hyperparameter optimization using SLM Lab, 304
  Markov decision processes in, 288–289
  as reinforcement learning problem, 284–286
CartPole, OpenAI Gym environment, 70
CBOW (continuous bag of words), word2vec, 207, 208
Cell body, biological neurons, 85–86
Cell state, LSTM, 244–245
Cerebral cortex, processing visual information, 3–4
cGAN (conditional GAN), 45
Chain rule of calculus, backpropagation and, 124
Chatbots, natural language processing in, 23–24
Checkpoints, dense sentiment classifier, 231
Chen, Chen, deep learning image processing, 47–48
Chess
  AlphaZero and, 65–66
  vs. Go board complexity, 59, 61
Classification
  adding layers to transfer learning model, 190
  convolutional sentiment classifier, 235–239
  of film reviews by sentiment, 229–235
  natural language. *See* Natural language classification
  as supervised learning problem, 53–54
CNNs. *See* Convolutional neural networks (CNNs)

CNTK, deep learning library, 324

Coding shallow network in Keras

designing neural network architecture, 83

installation, 76

loading MNIST data, 80–81

MNIST handwritten digits, 76–77

overview of, 75, 76

prerequisites, 75–76

reformatting data, 81–83

schematic diagram of network, 77–79

software dependencies for shallow net, 80

summary, 84

training deep learning model, 83–84

Color, visual cortex detects, 7–8

Compiling

adversarial network, 274

dense sentiment classifier, 231

discriminator network, 269

network model for DQN agent, 298

Complex neurons

forming primary visual cortex, 6–7

neocognition and, 9

Computational complexity

minimizing number of kernels to avoid, 163

from piping images into dense networks, 160

Computational homogeneity, with Software 2.0, 325

Computing power, AGI and development of, 327

Conditional GAN (cGAN), 45–46

Conditional imitation learning algorithms, 307

Confusion matrix, 218–219

Content generation, building socially beneficial projects, 318

Context words, running word2vec, 207–209

Continuous bag of words (CBOW), word2vec, 207, 208

Continuous variable, supervised learning problem, 54

Contracting path, U-Net, 187–188

Conv2D dependency, LeNet-5 in Keras, 171–174

Convolutional filter hyperparameters, CNNs, 168–169

Convolutional layers

convolutional neural networks (CNNs) and, 160–162

general approach to CCN design, 176

working with pooling layers, 169–170

Convolutional layers, GANs

birth of GANs, 41

convolutional neural networks (CNNs) and, 52–53

multiple filters in, 162–163

results of latent space arithmetic, 42–44

Convolutional neural networks (CNNs)

computational complexity, 160

contemporary machine vision and, 52–53

convolutional filter hyperparameters, 168–169

convolutional layers, 160–162

DeepMind DQN using, 58

detecting spatial patterns among words, 235–239

developing Faster R-CNN, 184–185

developing YOLO, 185–186

example of, 163–167

general approach to CCN design, 176

image segmentation with Mask R-CNN, 187

LeNet-5 in Keras, 171–176

manipulation of objects via, 67–68

model inspired by AlexNet, 176–178

model inspired by VGGNet, 178–179

multiple filters, 162–163

object detection with Fast R-CNN, 184

object detection with R-CNN, 183–184

overview of, 159

transfer learning model of, 188–192

two-dimensional structure of visual imagery, 159–160

Convolutional sentiment classifier, 235–239, 252–256

`convTranspose` layers, in generator networks, 270

Corpus

one-hot encoding of words within, 25–26

preprocessing full, 203–206

word vectors within, 27–29

word2vec architectures for, 208

Cortes, Corinna, curating MNIST dataset, 77–78

Cost (loss) functions

building own project, 319

cross-entropy cost, 113–115

quadratic cost, 112–113

in stochastic gradient descent, 120

training deep networks and, 111

using backpropagation to calculate gradient of, 124–125, 335–337

Cost, minimizing via optimization

batch size and stochastic gradient descent, 119–122

escaping local minimum, 122–124

gradient descent, 115–117

learning rate, 117–119

training deep networks and, 115

Count based, word2vec as, 208

Cross-entropy cost

essential GAN theory, 262

minimizes impact of neuron saturation, 113–115, 131

pairing with weight initialization, 131–135

CycleGANs, style transfer of well-known painters, 44–45

D

Dahl, George, 24–25

Data

augmentation, training deep networks, 145

development of AGI and, 327

Data generators, training, 190–191

DataFrame, IMDb validation data, 234

Datasets, deep reinforcement learning using larger, 57

De-convolutional layers, generator networks, 269–270, 272

deCNN, generator network as, 270

Deep Blue, history of chess, 65

Deep learning

code. *See* Coding shallow network in Keras

computational representations of language. *See* Language, computational representations of

definition of, 22

elements of natural human language in, 33–35

Google Duplex as NLP based on, 35–37

model architectures, 51–52

natural language processing and, 23–25, 37

networks learn representations automatically, 22–23

reinforcement learning combined with. *See* Reinforcement learning, deep

training deep networks. *See* Training deep networks

Deep learning, introduction

biological vision, 3–8

machine vision. *See* Machine vision

Quick, Draw! game, 19

summary, 20

TensorFlow Playground, 17–19

traditional machine learning vs., 11–12

Deep learning projects, building own

artificial general intelligence approach, 326–328

converting existing machine learning project, 316–317

deep learning libraries, 321–324

deep reinforcement learning, 316

machine vision and GANs, 313–315

modeling process, including hyperparameter tuning, 318–321

natural language processing, 315–316
overview of, 313
resources for further projects, 317–318
Software 2.0, 324–326
summary, 328–329
Deep networks, improving
deep neural network in Keras, 147–149
fancy optimizers, 145–147
key concepts, 154–155
model generalization (avoiding overfitting), 140–145
overview of, 131
regression, 149–152
summary, 154
TensorBoard, 152–154
unstable gradients, 137–139
weight initialization, 131–135
Xavier Glorot distributions, 135–137
Deep Q-learning networks (DQNs)
DeepMind video game and, 58–60
defining DQN agent. *See* DQN agents
essential theory of, 290–292
SLM Lab using, 304–306
Deep reinforcement learning. *See* Reinforcement learning, deep
Deep RL agents, 306–308
DeepMind
AlphaGo board game, 61–62
AlphaGo Zero board game, 62–65
Google acquiring, 59
video games, 58–60
DeepMind Lab
building own deep learning project with, 316
deep reinforcement learning, 69, 71
Dendrites, and biological neurons, 85–86
Denormalization, in batch normalization, 139
Dense layers
architecting intermediate net in Keras, 127–128
artificial neural networks with, 99–100
CNN model inspired by AlexNet, 177–178
computational complexity and, 160
convolutional layers vs., 168
deep learning and, 51
Fast R-CNN and, 184
in GANs, 271–272
general approach to CCN design, 176
LeNet-5 in Keras and, 172–173, 175–176
multi-ConvNet model architecture, 253–255
in natural language processing, 224–225, 230–231, 236–238
networks designed for sequential data, 243
in shallow networks, 109
using weight initialization for deep networks, 132–133, 137
in wide and need model architecture, 317
Dense network
architecture, 229–235
building socially beneficial projects, 318
defined, 100
hot dog-detecting, 101–106
revisiting shallow network, 108–110
softmax layer of fast food-classifying network, 106–108
Dense sentiment classifier, 229–235
***Dense Sentiment Classifier* Jupyter notebook. *See* Natural language classification**
Dependencies
Cart-Pole DQN agent, 293
convolutional sentiment classifier, 236
LeNet-5 in Keras, 171
loading GAN for Quick, Draw! game, 264–265
loading IMDb film reviews, 222–223
preprocessing natural language, 197
regression model, 150
TensorFlow with Keras layers, 323
Dimensionality reduction, plotting word vectors, 213–217
Discount factor (decay), Markov decision processes, 288–289

Discounted future reward
expected, 290
maximizing, 290
Discriminator network, GANs
code for training, 277
defined, 40–41
overview of, 266–269
training, 259–262
Distributed representations, localist representations vs., 32
Dot product notation, perceptron equation, 90–91
DQN agents
agents beyond, 306–308
building neural network model for, 297–298
drawbacks of, 306
hyperparameter optimization using SLM Lab, 304
initialization parameters, 295–297
interacting with OpenAI Gym environment, 300–303
overview of, 293–295
remembering gameplay, 298
selecting action to take, 299–300
training via memory replay, 298–299
DQNs. *See* Deep Q-learning networks (DQNs)
Dropout
for AlexNet in Keras, 177
for deep neural network in Keras, 148
for LeNet-5 in Keras, 171–174
network architecture regression model and, 150–151
preventing overfitting with, 142–145

E

Eager mode, TensorFlow, 322–323
Ease of use, Software 2.0, 326
Efros, Alexei, 44
ELMo (embeddings from language models), transfer learning, 251
Elo scores
AlphaGo game, 62–63
AlphaGo Zero game, 64–65
AlphaZero game, 66
Encoder-decoder structure, NMT, 250
Environment(s)
DeepMind DQN, 58
OpenAI Gym, 300–303
popular deep reinforcement learning, 68
reinforcement learning problems of machine learning, 54–56
reinforcement learning theory, 283
training agents simultaneously via SLM Lab in multiple, 304
Epochs of training, checkpointing model parameters after, 231–232
Essential theory. *See* Theory, essential
exp function, softmax layer of fast food-classifying network, 106–108
Expanding path, U-Net, 187–188
Experiment graph, SLM Lab, 304
Expertise, subject-matter
AutoNet reducing requirement for, 17
deep learning easing requirement for, 22–23
Exploding gradients, ANNs, 138
Extrinsic evaluations, evaluating word vectors, 209

F

Face detection
arithmetic on fake human faces, 41–44
birth of generative adversarial networks, 39–41
engineered features for robust real-time, 12–13
in visual cortex, 8
Facebook, fastText library, 33
False negative, IMDb reviews, 236
False positive, IMDb reviews, 235
Fan Hui, AlphaGo match, 62
Fancy optimizers, deep network improvement, 145–147

Fashion-MNIST dataset, deep learning project, 313–315
Fast food-classifying network, softmax layer of, 106–108
Fast R-CNN, object detection, 184
Faster R-CNN, object detection, 184–185
FastText, 33, 209
Feature engineering
AlexNet automatic vs. expertise-driven, 17
defined, 11
traditional machine learning and, 12–13
traditional machine learning vs. deep learning, 11–12
Feature maps
convolutional example of, 163–167
image segmentation with U-Net, 188
transfer learning model and, 188–192
Feedforward neural networks, training, 241
FetchPickAndPlace, OpenAI Gym, 70
Figure Eight
image-classification model, 315
natural language processing model, 316
Filters. *See* Kernels (filters)
Finn, Chelsea, 67
`fit_generator()` method, transfer learning, 191–192
Fitting, dense sentiment classifier, 232
`Flatten` layer, LeNet-5 in Keras, 171–174
`FloatTensor`, PyTorch, 339
`for` loop, GAN training, 275–281
Formal notation, neural networks, 333–334
Forward propagation
backpropagation vs., 124
defined, 103
in hot dog-detecting dense network, 101–106
notation for neural networks, 334
in softmax layer of fast food-classifying network, 106–108
in stochastic gradient descent, 120, 121
Frozen Lake game, 316
Fukushima, Kunihiko, LeNet-5, 9–12
Fully connected layer (as dense layer), 99
Functional API, non-sequential architectures and Keras, 251–256
Fusiform face area, detecting in visual cortex, 8

G

Game-playing machines
artificial intelligence, 49–50
artificial neural networks (ANNs), 51
board games, 59–66
categories of AI, 71–72
categories of machine learning problems, 53–56
deep learning, 51–52
deep reinforcement learning, 56–57
machine learning, 50
machine vision, 52–53
manipulation of objects, 67–68
natural language processing, 53
overview of, 49–50
popular deep reinforcement learning environments, 68–71
representation learning, 51
Software 2.0 and, 326
summary, 72
video games, 57–59
Gameplay, 298–300
gamma (γ), batch normalization adding, 139
GANs. *See* Generative adversarial networks (GANs)
Gated recurrent units (GRUs), 249–250
`gberg_sents`, tokenizing natural language, 199
Generative adversarial networks (GANs)
actor-critic algorithm reminiscent of, 308
adversarial network component, 272–274
arithmetic on fake human faces, 41–44
birth of, 39–41
building and tuning own, 315
creating photorealistic images from text, 45–46

Generative adversarial networks (GANs) (*continued*)
- discriminator network component, 266–269
- essential theory, 259–262
- generator network component, 269–272
- high-level concepts behind, 39
- image processing using deep learning, 47–48
- key concepts, 281–282
- making photos photorealistic, 45
- Quick, Draw! game dataset, 263–266
- reducing computational complexity with, 170
- Software 2.0 and, 326
- style transfer, 44–45
- summary, 281
- training, 275–281

Generator network, GANs
- code for training, 277–278
- defined, 40–41
- overview of, 269–272
- training, 259–262

Geoff Hinton, 94–95

Girshick, Ross, 183–184

GitHub repository, Quick, Draw! game dataset, 263

Global minimum of cost, training deep networks for, 122–124

Glorot normal distribution, improving deep networks, 135–137

GloVe
- converting natural words to word vectors, 28
- as major alternative to word2vec, 208

Go board game, 59–66

Goodfellow, Ian
- arithmetic on fake human faces and, 41–44
- birth of GANs, 39–41
- MNIST dataset used by, 76–77

Google Duplex technology, deep-learning-based NLP, 35–37

GPUs (graphics processing units), deep reinforcement learning, 57

Gradient descent
- batch size and stochastic, 119–122
- cross-entropy costs and, 114
- enabling neural networks to learn, 113
- escaping local minimum using, 122–124
- learning rate in, 117–119
- minimizing cost with, 115–117
- training deep networks with batch size/stochastic, 119–122

Graesser, Laura, 304

Graphics processing units (GPUs), deep reinforcement learning, 57

GRUs (gated recurrent units), 249–250

Gutenberg, Johannes, 197

H

HandManipulateBlock, OpenAI Gym, 70

Handwritten digits, MNIST, 76–78

Hassabis, Demis, 58–59

Hidden layers
- artificial neural network with, 99
- building network model for DQN agent, 297
- calculus behind backpropagation, 337
- deep learning model architectures, 51–52
- dense layers within. *See* Dense layers
- forward propagation in dense network through, 102–106
- hot dog-detecting dense network, 101–106
- neural network notation, 333–334
- schematic diagram of shallow network, 79
- TensorFlow Playground demo, 100
- tuning neuron count and number of, 125–126

Hidden state, LSTM, 245

Hierarchical softmax, training word2vec, 208

Hinton, Geoffrey
developing capsule networks, 192
developing t-distributed stochastic neighbor embedding, 213–214
as godfather of deep learning, 14–15
Histogram of validation data
convolutional sentiment classifier, 239
dense sentiment classifier, 233–234
Hochreiter, Sepp, 244
Hot dog-detecting dense network, 101–106
Hot dog/not hot dog detector, perceptrons, 86–90
Hubel, David
LeNet-5 model built on work of, 10–12
machine vision approach using work of, 8–9
research on visual cortex, 4–7
Human and machine language. *See also* Language, computational representations of
deep learning for natural language processing, 21–25
elements of natural human language in, 33–35
Google Duplex technology, 35–37
summary, 37
Humanoid, OpenAI Gym environment, 70
Hyperparameters. *See also* Parameters
in artificial neural networks, 130
automating search for, 321
batch size, 119
Cart-Pole DQN agent, 293–295
convolutional filter, 163, 167
convolutional sentiment classifier, 236–237
learning rate, 118
for loading IMDb film reviews, 223–225
LSTM, 246–247
multi-ConvNet sentiment classifier, 253
network depth, 125–126
number of epochs of training, 122
optimizing with SLM Lab, 303–306
reducing model overfitting with dropout, 144–145
RMSProp and AdaDelta, 147
RNN sentiment classifier, 242–243
tuning own project, 318–321
understanding in this book, 118–119

I

IMDb (Internet Movie Database) film reviews. *See* Natural language classification
ILSVRC (ImageNet Large Scale Visual Recognition Challenge)
AlexNet and, 14–17
ResNet, 182
traditional ML vs. deep learning entrants, 13–14
Image classification
building socially beneficial projects using, 318
ILSVRC competition for, 182
machine vision datasets for deep learning, 313–315
object detection vs., 183
Image segmentation applications, machine vision, 186–188
`ImageDataGenerator` class, transfer learning, 190–191
ImageNet, and ILSVRC, 13–14
Images
creating photorealistic. *See* Machine art
processing using deep learning, 46–48
Imitation learning, agents beyond DQN optimizing, 307
Infrastructure, rapid advances in, 327
Initialization parameters, DQN agent, 295–297
Input layer
artificial neural networks with, 99
of deep learning model architectures, 51–52
hot dog-detecting dense network, 101–106
LSTM, 245
notation for neural networks, 333
of perceptrons, 86–88

Input layer (*continued*)
- schematic diagram of shallow network, 79
- TensorFlow Playground demo, 100

Installation
- of code notebooks, 76
- PyTorch, 341

Integer labels, converting to one-hot, 82–83

***Intermediate Net in Keras* Jupyter notebook, 127–129**

Internal covariate shift, batch normalization, 138–139

Internet Movie Database (IMDb) film reviews. *See* Natural language classification

Intrinsic evaluations, word vectors, 209

`iter` argument, running word2vec, 210

J

Jones, Michael, real-time face detection, 12–13

K

Kaggle
- image-classification model, 315
- natural language processing model, 316

Karpathy, Andrej, 324–326

Kasparov, Garry, 65

Keng, Wah Loon, 304

Keras
- AlexNet and VGGNet in, 176–179
- coding in. *See* Coding shallow network in Keras
- deep learning library in, 321–323
- deep neural network in, 147–149
- functional API, non-sequential architectures and, 251–256
- implementing LSTM, 246–247
- implementing RNN, 242
- intermediate-depth neural network in, 127–129
- LeNet-5 model in, 171–176
- loading IMDb film reviews in, 225–226
- parameter-adjustment in, 144
- TensorBoard dashboard in, 152–154
- transfer learning in, 188–192
- weight initialization in, 132–135

Kernels (filters)
- convolutional example of, 164–167
- of convolutional layers, 160–162
- number in convolutional layer, 162–163
- pooling layers using, 169–170
- size, convolutional filter hyperparameter, 167

Key concepts
- artificial neural networks (ANNs), 110
- artificial neurons that constitute ANNs, 97
- deep reinforcement learning, 308–309
- generative adversarial networks (GANs), 281–282
- improving deep networks, 154–155
- machine vision, 193
- natural language processing (NLP), 256–257
- training deep networks, 130

Krizhevsky, Alex, 14, 16

L

L1 vs. L2 regularization, reducing model overfitting, 141–142

Language. *See* Human and machine language

Language, computational representations of
- localist vs. distributed representations, 32–33
- one-hot representations of words, 25–26
- overview of, 25
- word vector-arithmetic, 29–30
- word vectors, 26–29
- word2viz tool for exploring, 30–32

LASSO regression, reducing model overfitting, 141–142

Latent space
- arithmetic on fake human faces in, 42–44
- birth of generative adversarial networks, 40–41

Layers
- building own project, 319–320
- deep learning model architectures, 51–52

Leaky ReLU activation function, 96

***Learn Python the Hard Way* (Shaw), 75**

Learning rate
- batch normalization allowing for higher, 139
- building own project, 320
- shortcomings of improving SGD with momentum, 146
- as step size in gradient descent, 117–119

LeCun, Yan
- on fabricating realistic images, 39
- LeNet-5 model, 9–12
- MNIST handwritten digits curated by, 76–78
- PyTorch development, 323–324
- Turing Award for deep learning, 15

Legg, Shane, 58

Lemmatization, as sophisticated alternative to stemming, 196

LeNet-5 model
- AlexNet vs., 15–17
- in Keras, 171–176
- machine vision, 9–12

Les 3 Brasseurs bar, 39

Levine, Sergey, 67

Li, Fei-Fei, 13–14

Libraries, deep learning, 321–324

Linear regression, object detection with R-CNN, 183–184

List comprehension
- adding word stemming to, 201
- removing stop words and punctuation, 200–201

`load()` method, neural network model for DQN agent, 300

Loading data
- coding shallow network in Keras, 79–81
- for shallow net, 80–81

`load_weights()` method, loading model parameters, 232

Local minimum of cost, escaping, 122–124

Localist representations, distributed representations vs., 32–33

Long short-term memory (LSTM) cells
- bidirectional (Bi-LSTMs), 247–248
- implementing with Keras, 246–247
- as layer of NLP, 53
- overview of, 244–246

Long-term memory, LSTM, 245–246

Lowercase
- converting all characters in NLP to, 195–196, 199–200
- processing full corpus, 204–206

LSTM. *See* Long short-term memory (LSTM) cells

LunarLander, OpenAI Gym environment, 70

M

Maaten, Laurens van der, 213–214

Machine art
- arithmetic on fake human faces, 41–44
- boozy all-nighter, 39–41
- creating photorealistic images from text, 45–46
- image processing using deep learning, 46–48
- make your own sketches photorealistic, 45
- overview of, 39
- style transfer, 44–45
- summary, 48

Machine language. *See* Human and machine language

Machine learning (ML). *See also* Traditional machine learning (ML) approach
- overview of, 50
- reinforcement learning problems of, 54–56

Machine learning (ML) (*continued*)
representation learning as branch of, 51
supervised learning problems of, 53–54
traditional machine vs. representation learning techniques, 22
unsupervised learning problems of, 54
Machine translation, NLP in, 23–24
Machine vision
AlexNet, 14–17
AlexNet and VGGNet in Keras, 176–179
CNNs. *See* Convolutional neural networks (CNNs)
converting existing project, 316–317
datasets for deep learning image-classification models, 313–315
ImageNet and ILSVRC, 13–14
key concepts, 193
LeNet-5, 9–12
LeNet-5 in Keras, 171–176
neocognition, 8–9
object recognition tasks, 52–53
overview of, 8, 159
pooling layers, 169–170
Quick, Draw! game, 19
residual networks, 179–182
Software 2.0 and, 326
summary, 20, 193
TensorFlow Playground, 17–19
traditional machine learning approach, 12–13
Machine vision, applications of
capsule networks, 192
Fast R-CNN, 184
Faster R-CNN, 184–185
image segmentation, 186–187
Mask R-CNN, 187
object detection, 183
overview of, 182
R-CNN, 183–184
transfer learning, 188–192
U-Net, 187–188
YOLO, 185–186
Magnetic resonance imaging (MRI), and visual cortex, 7–8
Manipulation of objects, 67–68
Markov decision process (MDP), 286–290
Mask R-CNN, image segmentation with, 186–187
Mass, Andrew, 203
matplotlib, weight initialization, 132
`max` operation, pooling layers, 170
Max-pooling layers
AlexNet and VGGNet in Keras, 176–179
LeNet-5 in Keras, 170–174
`MaxPooling2D` dependency, LeNet-5 in Keras, 171–174
MCTS (Monte Carlo tree search) algorithm, 61, 66
MDP (Markov decision process), 286–290
Mean squared error, 112, 298
Memory
batch size/stochastic gradient descent and, 119–122
DQN agent gameplay, 298
Software 2.0 and, 326
training DQN agent via replay of, 298–299
Metrics, SLM Lab performance, 305–306
Milestones, deep learning for NLP, 24–25
`min_count` argument, word2vec, 210–211
Minibatches, splitting training data into, 119–122
ML. *See* Machine learning (ML)
Mnih, Volodymyr, 58–60
MNIST handwritten digits
calculus for backpropagation, 337
coding shallow network in Keras, 76–78
computational complexity in dense networks, 160
Fashion-MNIST dataset deep learning project, 313–315
loading data for shallow net, 80–81
loss of two-dimensional imagery in dense networks, 159–160

reformatting data for shallow net, 81–83
schematic diagram of shallow network, 77–79
software dependencies for shallow net, 80
in stochastic gradient descent, 120
training deep networks with data augmentation, 145
using in Keras, 171–176
Model generalization. *See* Overfitting, avoiding
Model optimization, agents beyond DQN using, 307
`ModelCheckpoint()` object, dense sentiment classifier, 231–232
Modeling process, building own project, 318–321
Momentum, 145–146
Monet, Claude, 44–45
Monte Carlo tree search (MCTS) algorithm, 61, 66
Morphemes, natural human language, 34
Morphology, natural human language, 34–35
`most_similar()` method, word2vec, 212–213
Motion, detecting in visual cortex, 7–8
Mountain Car game, 316
MRI (magnetic resonance imaging), and visual cortex, 7–8
Müller, Vincent, 72
***Multi ConvNet Sentiment Classifier* Jupyter notebook, 320**
MXNet, deep learning library, 324

N

***n*-dimensional spaces, 42–43, 339**
***n*-grams, 196, 202–203**
Nair, Vinod, 94–95
Natural human language, elements of, 33–35
Natural language classification
dense network classifier architecture, 229–235
examining IMDb data, 227–228
with familiar networks, 222
loading IMDb film reviews, 222–226
processing in document, 23–24
standardizing length of reviews, 228–229
***Natural Language Preprocessing* Jupyter notebook, 197**
Natural language processing (NLP)
area under ROC curve, 217–222
building own deep learning project, 315–316
building socially beneficial projects, 318
computational representations of. *See* Language, computational representations of
deep learning approaches to, 53
examples, 23–24
Google Duplex as deep-learning, 35–37
history of deep learning, 24–25
key concepts, 256–257
learning representations automatically, 22–23
natural human language elements of, 33–35
natural language classification in. *See* Natural language classification
networks designed for sequential data, 240–251
non-sequential architectures, 251–256
overview of, 195
preprocessing. *See* Preprocessing natural language data
Software 2.0 and, 326
summary, 256
transfer learning in, 251
word embedding with word2vec. *See* word2vec
`n_components`, plotting word vectors, 214
Negative rewards, reinforcement learning problems and, 56
Negative sampling, training word2vec, 208
Neocognition
LeNet-5 advantages over, 13–14
LeNet-5 model and, 9–12
machine vision and, 8–9

Nesterov momentum optimizer, stochastic gradient descent, 146

Network architecture, regression model, 150–151

Network depth, as hyperparameter, 125–126

Neural Information Processing Systems (NIPS) conference, 41

Neural machine translation (NMT), seq2seq models, 250

Neural networks
- building deep in PyTorch, 343–344
- coding shallow in Keras, 83
- formal notation for, 333–334

Neuron saturation. *See* Saturated neurons

Neurons
- AlexNet vs. LeNet-5, 17
- behaviors of biological, 85–86
- forming primary visual cortex, 4–7
- neocognition and, 8–9
- regions processing visual stimuli in visual cortex, 7–8
- TensorFlow Playground and, 17–19
- tuning hidden-layer count and number of, 126

`next_state`, DQN agent gameplay, 298

NIPS (Neural Information Processing Systems) conference, 41

`n_iter`, plotting word vectors, 214

NLP. *See* Natural language processing (NLP)

NMT (neural machine translation), seq2seq models, 250

Noë, Alva, 39

Non-sequential model architecture, 251–256

Non-trainable params, model object, 109–110

Nonlinearity, of ReLU neuron, 95

Notation, formal neural network, 333–334

Number of epochs of training
- as hyperparameter, 122
- rule of thumb for learning rate, 119
- stochastic gradient descent and, 119–122
- training deep learning model, 83–84

NumPy
- PyTorch tensors and, 324, 339
- selecting action for DQN agent, 299–300
- weight initialization, 132, 134

O

Object detection
- with Fast R-CNN, 184
- as machine vision application, 182–183
- with R-CNN, 183–184
- understanding, 183
- with YOLO, 185–186

Objective function (π), maximizing reward with, 290

Objects
- manipulation of, 67–68
- recognition tasks of machine vision, 52–53

Occam's razor, neuron count and, 126

Oliveira, Luke de, 315, 316

On-device processing, machine learning for, 46–48

One-hot format
- computational representations of language via, 25–26
- converting integer labels to, 82–83
- localist vs. distributed representations, 32–33

Online resources
- building deep learning projects, 317–318
- pretrained word vectors, 230

OpenAI Gym
- building deep learning projects, 316
- Cart-Pole game, 284–286
- deep reinforcement learning, 68–70
- interacting with environment, 300–303

Optimal policy
- building neural network model for, 288–290
- estimating optimal action via Q-learning, 290–292

Optimal Q-value (Q*), estimating, 291–292

Optimization

agents beyond DQN using, 306–307

fancy optimizers for stochastic gradient descent, 145–147

hyperparameter optimizers, 130, 303–306

minimizing cost via. *See* Cost, minimizing via optimization

stochastic gradient descent. *See* Stochastic gradient descent (SGD)

Output layer

artificial neural network with, 99

batch normalization and, 139

building network model for DQN agent, 298

calculus behind backpropagation, 335, 337

deep learning model architectures, 51–52

LSTM, 245

notation for neural networks, 334

perceptrons, 86–87, 89

schematic diagram of shallow network, 79

softmax layer for multiclass problems, 106–108

softmax layer of fast food-classifying network, 106–107

TensorFlow Playground demo, 100

Overfitting, avoiding

building your own project, 320

data augmentation, 145

dropout, 142–145

L1 and L2 regularization, 141–142

model generalization and, 140–141

P

Pac-Man

discount factor (decay) and, 288–289

DQN agent initialization and, 296

Padding

convolutional example of, 163–167

as convolutional filter hyperparameter, 167–168

standardizing length of IMDb film reviews, 228–229

Parameter initialization, building own project, 319

Parameters. *See also* Hyperparameters

Cart-Pole DQN agent initialization, 295–297

creating dense network classifier architecture, 230–232

escaping local minimum, 122–124

gradient descent minimizing cost across multiple, 116–117

pooling layers reducing overall, 169–170

saving model, 300

weight initialization, 132–135

Parametric ReLU activation function, 96

Partial-derivative calculus, cross-entropy cost, 114–115

Patches, in convolutional layers, 160

PCA (principal component analysis), 213

Perceptrons

choosing, 96

hot dog/not hot dog detector example, 86–90

modern neurons vs., 91

as most important equation in this book, 90–91

overview of, 86

Performance

hyperparameter optimization using SLM Lab, 303–306

Software 2.0 and, 326

PG. *See* Policy gradient (PG) algorithm

Phonemes, natural human language and, 34

Phonology, natural human language and, 34–35

Photorealistic images, creating. *See* Machine art

`Phraser()` method, NLP, 202–203, 204–205

Phrases() method, NLP, 202–203, 204–205
Pichai, Sundar, 35–36
pix2pix web application, 45–46
Pixels
computational complexity and, 160
converting integers to floats, 82
convolutional example of, 163–167
convolutional layers and, 160–162
handwritten MNIST digits as, 77–78
kernel size hyperparameter of convolutional filters, 167
reformatting data for shallow net, 81–83
schematic diagram of shallow network, 78–79
two-dimensional imagery and, 159–160
Plotting
GAN training accuracy, 281
GAN training loss, 280–281
word vectors, 213–217
Policy function (π), discounted future reward, 288–290
Policy gradient (PG) algorithm
actor-critic using Q-learning with, 307–308
in deep reinforcement learning, 68
REINFORCE algorithm as, 307
Policy networks, AlphaGo, 61
Policy optimization
agents beyond DQN using, 307
building neural network model for, 288–290
estimating optimal action via Q-learning, 290–292
RL agent using actor-critic with, 307–308
Pooling layers, 169–170, 176
Positive rewards, deep reinforcement learning, 56, 57
Prediction
selecting action for DQN agent, 300
training dense sentiment classifier, 232
training DQN agent via memory replay, 299
word2vec using predictive models, 208
Preprocessing natural language data
converting all characters to lowercase, 199–200
full corpus, 203–206
handling *n-grams, 202–203*
overview of, 195–197
removing stop words and punctuation, 200–201
stemming, 201
tokenization, 197–199
Principal component analysis (PCA), 213
Probability distribution, Markov decision processes, 288
Processing power, AlexNet vs. LeNet-5, 16–17
Project Gutenberg. *See* Preprocessing natural language data
Punctuation
processing full corpus, 204–206
removing, 196, 200
Python, for example code in this book, 75–76
PyTorch
building deep neural network in, 343–344
deep learning library, 323–324
features, 339–340
installation, 341
in practice, 341–343
TensorFlow vs., 340–341

Q

Q-learning networks
actor-critic combining PG algorithms with, 307–308
DQNs. *See* Deep Q-learning networks (DQNs)
Q-value functions
agents beyond DQN optimizing, 306
drawbacks of DQN agents, 306

estimating optimal, 291–292
training DQN agent via memory replay, 299

Quadratic cost, 112–113

Quake III Arena, DeepMind Lab built on, 69

Quick, Draw! game
GANs and, 263–266
for hundreds of machine-drawn sketches, 48
introduction to deep learning, 19

R

R-CNN
Fast R-CNN, 184
Faster R-CNN, 184–185
Mask R-CNN, 186–187
object detection application, 183–184

Radford, Alec, 41–44

RAM (memory), batch size/stochastic gradient descent and, 119–122

`rand` function, DQN agent action selection, 299–300

`randrange` function, DQN agent action selection, 300

Rectified linear unit neurons. *See* ReLU (rectified linear unit) neurons

Recurrent neural networks (RNNs)
bidirectional LSTM, 247–248
LSTM, 244–247
LSTM cell as layer of NLP in, 53
overview of, 240–244
stacked recurrent models, 248–250

Redmon, Joseph, 185–186

Reformatting data, coding shallow network, 81–83

Regions of interest (ROIs)
developing Faster R-CNN, 184–185
image segmentation with Mask R-CNN, 187
object detection with Fast R-CNN, 184
object detection with R-CNN, 183–184

Regression, improving deep networks, 149–152

REINFORCE algorithm, agents beyond DQN using, 307

Reinforcement learning
building socially beneficial projects, 318
essential theory of, 283–286
overview of, 49
problems of machine learning, 54–56
as sequential decision-making problems, 284

***Reinforcement Learning: An Introduction* (Barto), 292**

Reinforcement learning, deep
agents beyond DQN, 306–308
board games. *See* Board games
building own project. *See* Deep learning projects, building own
Cart-Pole game, 284–286
DeepMind DQN using, 58–59
defining DQN agent, 293–300
essential theory of deep Q-learning networks, 290–292
essential theory of reinforcement learning, 283–286
game-playing applications. *See* Game-playing machines
hyperparameter optimization with SLM Lab, 303–306
interacting with OpenAI Gym environment, 300–303
key concepts, 308–309
manipulation of objects, 67–68
Markov decision processes, 286–288
optimal policy, 288–290
overview of, 56–57, 283
popular learning environments for, 68–71
summary, 308
video games, 57–60

ReLU (rectified linear unit) neurons
with Glorot distributions, 136–137
neural network model for DQN agent, 297
overview of, 94–95
as preferred neuron type, 96
TensorFlow Playground demo, 100

Representation learning, 22, 51
`requires_grad` argument, PyTorch, 342
Residual connections, 180–182
Residual modules, 180–182
Residual networks (ResNets), 180–182
Resources, building deep learning projects, 317–318
`return_sequencesTrue`, stacking recurrent layers, 248
Reward(s)
  deep Q-learning network theory, 290–292
  DeepMind DQN and, 59
  DeepMind Lab, 69, 71
  DQN agent gameplay, 298
  Markov decision processes (MDPs), 287–289
  optimal policy, 288–290
  reinforcement learning problems and, 56
  theory of reinforcement learning, 283
  training DQN agent via memory replay, 298–299
Ridge regression, reducing model overfitting, 141–142
RMSProp, 147
RMSProp optimizer, 147
ROC AUC metric
  as area under ROC curve, 217–218
  calculating, 219–222, 234
  confusion matrix, 218–219
  for sentiment classifier model architectures, 256
ROIs. *See* Regions of interest (ROIs)
Rosenblatt, Frank, 86–90
Round of training, stochastic gradient descent, 120–121
Running time, Software 2.0 and, 325

S

Sabour, Sara, 192
Saturated neurons
  as flaw in calculating quadratic cost, 112–113
  minimizing impact using cross-entropy cost, 113–115
  reducing with cross-entropy cost and weight initialization, 131–135
  weight initialization, Glorot normal distribution, 136
Saving model parameters, 300
Schematic diagram
  activation values in feature map of convolutional layer, 164
  coding shallow network in Keras, 77–79
  of discriminator network, 268
  of generator network, 270
  of LSTM, 245
  of recurrent neural network, 241
  wide and deep modeling, 317
Schmidhuber, Jürgen, 244
Search, automating hyperparameter, 321
Search engines, NLP in, 23–24
Sedol, Lee, 62
See-in-the-Dark dataset, image processing, 47–48
Semantics, natural human language and, 34–35
`sentences` argument, word2vec, 210
Sentiment classifier
  bidirectional LSTM, 247–248
  convolutional, 236–239
  dense, 229–235
  LSTM architecture, 247
  LSTM hyperparameters, 246–247
  non-sequential architecture example, 251–255
  performance of model architectures, 256
seq2seq (sequence-to-sequence), and attention, 250
Sequential decision-making problems, 284
Sequential model, building for DQN agent, 297–298
sg argument, word2vec, 210
SG (skip-gram) architecture, 207, 208
SGD. *See* Stochastic gradient descent (SGD)
Shadow Dexterous Hand, OpenAI Gym, 70

Shallow network
 coding. *See* Coding shallow network in Keras
 for dense networks, 108–110
 intermediate-depth neural network in, 127–129
 vs. deep learning, 78–79
Shogi, AlphaZero and, 65–66
Short-term memory, LSTM, 245–246
***Sigmoid Function* Jupyter notebook, 105**
Sigmoid neuron(s)
 activation function of, 92–94
 for binary classification problems, 100–101, 105–106
 choosing, 96
 for shallow net in Keras, 79, 83
 softmax function with single neuron equivalent to using, 108
 weight initialization and, 133–137
Silver, David, 61–62, 65–66
Similarity score, running word2vec, 212–213
Simple neurons
 forming primary visual cortex, 6–7
 neocognition and, 8–9
`SimpleRNN()` layer, RNN sentiment classifier, 243
`size` argument, word2vec, 210
Skip-gram (SG) architecture, 207, 208
SLM Lab, 303–306, 316
Socially beneficial projects, deep learning projects, 318
Sodol, Lee, 62, 64
Softmax layer, fast food-classifying network, 106–108
Softmax probability output, Fast R-CNN, 184
Software dependencies, shallow net in Keras, 80
Sofware 2.0, deep learning models, 324–326
Speech recognition, NLP in, 24
Spell-checkers, 24
Squared error, as quadratic cost, 112
Stacked recurrent models, 248–250
StackGAN, photorealistic images from text, 45–46
State(s)
 deep Q-learning network theory and, 290–292
 DeepMind DQN and, 58
 DQN agent, remembering gameplay, 298
 Markov decision processes and, 286
 optimal policy in deep reinforcement learning and, 289–290
 reinforcement learning problems and, 56
 reinforcement learning via Cart-Pole game and, 286
 theory of reinforcement learning, 284
Static scatterplot, plotting word vectors, 214–216
Stemming, word
 forgoing removal of, 203–206
 overview of, 201
 preprocessing natural language via, 196
Stochastic gradient descent (SGD)
 escaping local minimum of cost via, 122–124
 fancy optimizers for, 145–147
 training deep networks using batch size and, 119–124
Stop words
 forgoing removal of, 203–206
 how to remove, 200
 removing in NLP, 195–196
Stride length
 as convolutional filter hyperparameter, 167
 pooling layers using, 169–170
 reducing computational complexity, 170
Style transfer, 44–45
Suleyman, Mustafa, 58
Supervised learning problems, machine learning, 53–54
Support vector machines, R-CNN, 183–184
Sutskever, Ilya, 14, 16
Sutton, Richard, 292
Syntax, natural human language and, 34–35

T

Tacotron, TTS engine, 36–37

Tanh neurons
- activation function of, 94
- choosing, 96
- with Glorot distributions, 136–137
- LSTM, 244–245

Target word
- converting natural words to word vectors, 27–28
- running word2vec, 207–209

Tensor processing units (TPUs), Google training neural networks, 64

TensorBoard dashboard, 152–154

TensorFlow, 321–323

TensorFlow Playground, 17–19, 100

Tensors, PyTorch
- automatic differentiation in, 342–343
- building deep neural network, 343–344
- compatibility with NumPy operations, 324
- features, 339–340

Terminal state, theory of reinforcement learning, 284

Text, creating photorealistic images from, 45–46

Text-to-speech (TTS) engine, Google Duplex, 36–37

Theano, deep learning library, 324

Theory, essential
- of deep Q-learning networks, 290–292
- of GANs, 259–262
- of reinforcement learning, 283–284
- of RNNs, 240–244
- of word2vec, 206–209

Threshold value, perceptron equation, 89–91

Tokenization
- examining IMDb data, 226–228
- natural human language and, 35–36
- preprocessing natural language, 195, 197–199

Torch, PyTorch as extension of, 323–324

`torch.nn.NLLLoss()` function, PyTorch, 344

TPUs (tensor processing units), Google training neural networks, 64

Traditional machine learning (ML) approach
- deep learning approach vs., 11–12
- entrants into ILSVRC using, 14–15
- natural human language in, 33–35
- one-hot encoding of words in, 25–26
- understanding, 12–13

`train()` method
- training DQN agent, 299
- training GAN, 275–281

Training
- AlexNet vs. LeNet-5, 16–17
- AlphaGo vs. AlphaGo Zero, 63–65
- TensorFlow Playground, 17–19

Training deep networks
- adversarial network, 272–274
- backpropagation, 124–125
- batch size and stochastic gradient descent, 119–122
- coding shallow network in Keras, 83–84
- convolutional sentiment classifier, 238
- cost functions, 111–115
- cross-entropy cost, 113–115
- data augmentation for, 145
- deep neural network in Keras, 147–149
- dense sentiment classifier, 232
- escaping local minimum, 122–124
- generative adversarial networks (GANs), 259–262, 275–281
- gradient descent, 115–117
- intermediate-depth neural network, 128–129
- intermediate net in Keras, 127–129
- key concepts, 130
- learning rate, 117–119
- minimizing cost via optimization, 115
- overview of, 111
- preventing overfitting with dropout, 142–145
- quadratic cost, 112–113
- recurrent neural networks (RNNs), 241

running word2vec, 208
saturated neurons, 112–113
summary, 129–130
transfer learning model of, 188–192
tuning hidden-layer and neuron counts, 125–126
via memory replay for DQN agent, 298–299

Transfer learning
machine vision and, 188–192
natural language and, 230
in NLP, 251
overview of, 188–192

Truncation, standardizing film review length, 228–229

TSNE ( ) method, plotting word vectors, 214–216

TTS (text-to-speech) engine, Google Duplex, 36–37

Two-dimensional images, flattening to one dimension, 82

Two-dimensional structure of visual imagery
overview of, 159–160
retaining in convolutional layers, 167
retaining using LeNet-5 in Keras, 172

U

U-Net, image segmentation, 187–188

ULMFiT (universal language model fine-tuning), transfer learning, 251

United States Postal Service, LeNet-5 reading ZIP codes, 11

Unity ML-Agents plug-in, 71, 304

Unstable gradients, improving deep networks, 137–139

Unsupervised learning problems, machine learning, 54

Upsampling layers, 187, 272

V

Validation data, 232–235, 239

Value functions, Q-learning, 291–292

Value networks, AlphaGo algorithm, 61

Value optimization
agents beyond DQN using, 306
RL agent using actor-critic algorithm and, 307–308

Vanishing gradient problem
in artificial neural networks, 137–138
performance degradation in deep CNNs, 179–180

Vector space
embeddings. *See* Word vectors
latent space similarities to, 42–43
word meaning represented by three dimensions, 27–29
word-vector arithmetic, 29–30

Venn diagram, 22, 50

VGGNet, 178–179, 188–192

Video games, 57–60

Viola, Paul, 12–13

Visual imagery, two-dimensional structure of, 159–160

Visual perception
cerebral cortex research on, 4–7
development of species on planet due to, 3–4

W

WaveNet, Google Duplex TTS engine, 36–37

Weight initialization, 131–137

Weighted sum, perceptron algorithm, 86–89

Weight(s)
backpropagation and, 125, 335–337
convolutional example of, 163–167
of kernels in convolutional layers, 160–162
minimizing cost via gradient descent, 115–116
notation for neural networks, 334

Wide and deep modeling approach, Google, 317

Wiesel, Torsten
  LeNet-5 model built on work of, 10–12
  machine vision using work of, 8–9
  research on visual cortex, 4–7
`window` argument, word2vec, 210
Wittgenstein, Ludwig, 21
Word embeddings. *See* Word vectors
Word vectors. *See also* word2vec
  arithmetic of, 29–30
  capturing word meaning, 195
  computational representations. *See* Language, computational representations of
  convolutional filters detecting triplets of, 239
  evaluating, 209
  localist vs. distributed representations, 32–33
  in NLP. *See* Natural language processing (NLP)
  online pretrained, 230
  plotting, 213–217
  training on natural language data, 229–230
  word2viz tool for exploring, 30–32
word2vec
  converting natural words to word vectors, 28
  essential theory behind, 206–209
  evaluating word vectors, 209
  FastText as leading alternative to, 209
  plotting word vectors, 213–217
  running, 209–213
  word embeddings, 206
Words
  creating embeddings with word2vec. *See* word2vec
  natural human language and, 33–35
  preprocessing natural language. *See* Preprocessing natural language data
`word_tokenize()` method, natural language, 199
`workers` argument, word2vec, 211

X

Xavier Glorot distributions, improving deep networks, 135–137

Y

Yelp review polarity, 316
YOLO (You Only Look Once), object detection, 185–186

Z

Zhang, Xiang, 315