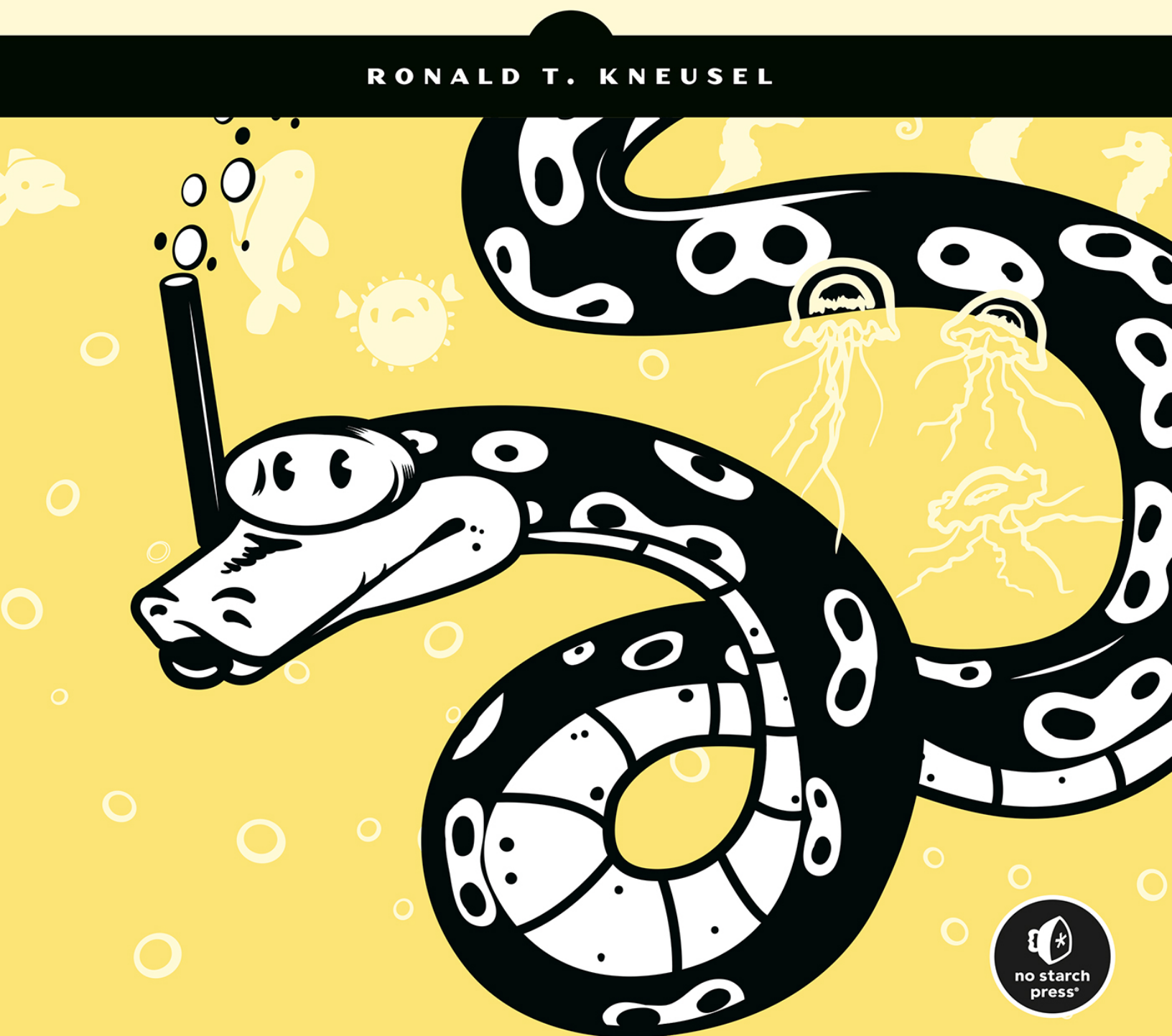


SECOND
EDITION

PRACTICAL DEEP LEARNING

A PYTHON-BASED INTRODUCTION

RONALD T. KNEUSEL



SECOND
EDITION

PRACTICAL DEEP LEARNING

A PYTHON-BASED INTRODUCTION

RONALD T. KNEUSEL



OceanofPDF.com

PRACTICAL DEEP LEARNING

2nd Edition

A Python-Based Introduction

by Ronald T. Kneusel



no starch press[®]

San Francisco

OceanofPDF.com

PRACTICAL DEEP LEARNING, 2ND EDITION. Copyright © 2025 by Ronald T. Kneusel.

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

First printing

29 28 27 26 25 1 2 3 4 5

ISBN-13: 978-1-7185-0420-2 (print)

ISBN-13: 978-1-7185-0421-9 (ebook)



Published by No Starch Press[®], Inc.
245 8th Street, San Francisco, CA 94103
phone: +1.415.863.9900
www.nostarch.com; info@nostarch.com

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Sabrina Plomitallo-González
Production Editor: Miles Bond
Developmental Editor: Eva Morrow
Cover Illustration: Gina Redman
Interior Design: Octopod Studios
Technical Reviewer: Scott Couwenhoven
Copyeditor: Sharon Wilkey
Proofreader: Daniel Wolff

The middle and right images in Figure 1-2 are via Wikimedia Commons and are reproduced with permission under CC-BY 2.0 (<https://creativecommons.org/licenses/by/2.0/deed.en>) and CC BY-SA 3.0 (<https://creativecommons.org/licenses/by-sa/3.0/deed.en>), respectively.

The Library of Congress has catalogued the first edition as follows:

Names: Kneusel, Ronald T., author.
Title: Practical deep learning : a python-based introduction / Ronald T. Kneusel.
Description: First edition. | San Francisco, CA : No Starch Press, Inc.,
[2021] | Includes index.
Identifiers: LCCN 2020035097 (print) | LCCN 2020035098 (ebook) | ISBN
9781718500747 (paperback) | ISBN 9781718500754 (ebook)
Subjects: LCSH: Machine learning. | Python (Computer program language)
Classification: LCC Q325.5 .K55 2021 (print) | LCC Q325.5 (ebook) | DDC 006.3/1--

dc23

LC record available at <https://lccn.loc.gov/2020035097>

LC ebook record available at <https://lccn.loc.gov/2020035098>

For customer service inquiries, please contact info@nostarch.com. For information on distribution, bulk sales, corporate sales, or translations: sales@nostarch.com. For permission to translate this work: rights@nostarch.com. To report counterfeit copies or piracy: counterfeit@nostarch.com. The authorized representative in the EU for product safety and compliance is EU Compliance Partner, Pärnu mnt. 139b-14, 11317 Tallinn, Estonia, hello@eucompliancepartner.com, +3375690241.

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

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

OceanofPDF.com

For my children: David, Peter, Paul, Monica, Joseph, and Francis

OceanofPDF.com

About the Author

Ronald T. Kneusel works in industry building deep learning (AI) systems. He also has extensive experience with medical imaging and the development of medical devices. His PhD in computer science is from the University of Colorado, Boulder, and he also holds a master's degree in physics from Michigan State University and an undergraduate degree in physics and mathematics from Valparaiso University.

His interests vary greatly but always orbit science, math, and technology. A child of the personal computer revolution in the late 1970s and early 1980s, Ron has been coding for longer than he's been able to drive. And, remarkably, it remains as fresh an enterprise now as it was all those decades ago.

OceanofPDF.com

About the Technical Reviewer

Scott Couwenhoven is an applied mathematician/data scientist in the remote sensing industry. His expertise is in computer vision, unsupervised representation learning, and geometric deep learning. He received a BS in mechanical engineering and an MS in applied and computational mathematics, both from the Rochester Institute of Technology.

OceanofPDF.com

BRIEF CONTENTS

Foreword to the First Edition

Acknowledgments

Introduction

Chapter 0: Environment and Mathematical Preliminaries

PART I: DATA IS EVERYTHING

Chapter 1: It's All About the Data

Chapter 2: Building the Datasets

PART II: CLASSICAL MACHINE LEARNING

Chapter 3: Introduction to Machine Learning

Chapter 4: Experiments with Classical Models

PART III: NEURAL NETWORKS

Chapter 5: Introduction to Neural Networks

Chapter 6: Training a Neural Network

Chapter 7: Experiments with Neural Networks

Chapter 8: Evaluating Models

PART IV: CONVOLUTIONAL NEURAL NETWORKS

Chapter 9: Introduction to Convolutional Neural Networks

Chapter 10: Experiments with Keras and MNIST

Chapter 11: Experiments with CIFAR-10

Chapter 12: A Case Study: Classifying Audio Samples

PART V: ADVANCED NETWORKS AND GENERATIVE AI

Chapter 13: Advanced CNN Architectures

Chapter 14: Fine-Tuning and Transfer Learning

Chapter 15: From Classification to Localization

Chapter 16: Self-Supervised Learning

Chapter 17: Generative Adversarial Networks

Chapter 18: Large Language Models

Afterword

Index

OceanofPDF.com

CONTENTS IN DETAIL

FOREWORD TO THE FIRST EDITION

ACKNOWLEDGMENTS

INTRODUCTION

- Who This Book Is For
- What You Can Expect to Learn about This Book
 - Terminology
 - What's New in the Second Edition
 - Synopsis

0

ENVIRONMENT AND MATHEMATICAL PRELIMINARIES

- The Operating Environment
 - NumPy
 - scikit-learn
 - TensorFlow with Keras
- Installing the Toolkits
- Basic Linear Algebra
 - Vectors
 - Matrices
 - Vector and Matrix Multiplication
- Statistics and Probability
 - Descriptive Statistics
 - Probability Distributions
 - Statistical Tests

Graphics Processing Units
Summary

PART I

DATA IS EVERYTHING

IT'S ALL ABOUT THE DATA

Classes and Labels

Features and Feature Vectors

- Types of Features

- Feature Selection and the Curse of Dimensionality

Qualities of a Good Dataset

- Interpolation and Extrapolation

- The Parent Distribution

- Prior Class Probabilities

- Confusers

- Dataset Size

Data Preparation

- Scaling Features

- Dealing with Missing Features

Training, Validation, and Test Data

- The Three Subsets

- Dataset Partitioning

- k-Fold Cross-Validation

Data Analysis

- How to Find Problems in the Data

- Cautionary Tales

Summary

BUILDING THE DATASETS

ses

east Cancer

NIST Digits

FAR-10

ita Augmentation

Reasoning

Methods

The Iris Dataset

The CIFAR-10 Dataset

mmary

PART II

CLASSICAL MACHINE LEARNING

INTRODUCTION TO MACHINE LEARNING

arest Centroid

Nearest Neighbors

ive Bayes

ee Classifiers

Recursion Primer

Decision Trees

Random Forests

pport Vector Machines

Margins

Support Vectors

Optimization

Kernels

mmary

EXPERIMENTS WITH CLASSICAL MODELS

Experiments with the Iris Dataset

- Testing the Classical Models

- Implementing a Nearest-Centroid Classifier

Experiments with the Breast Cancer Dataset

- Running Two Initial Tests

- Testing the Effect of Random Splits

- Adding k-Fold Validation

- Searching for Hyperparameters

Experiments with the MNIST Dataset

- Testing the Classical Models

- Analyzing Runtimes

- Experimenting with PCA Components

- Scrambling the Dataset

Classical Model Summary

- Nearest Centroid

- k-Nearest Neighbors

- Naive Bayes

- Decision Trees

- Random Forests

- Support Vector Machines

When to Use Classical Models

- Handling Small Datasets

- Dealing with Reduced Computational Requirements

- Having Explainable Models

- Working with Vector Inputs

Summary

PART III

NEURAL NETWORKS

INTRODUCTION TO NEURAL NETWORKS

atomy of a Neural Network

- The Neuron

- Activation Functions

- The Architecture of a Network

- Output Layers

- Weight and Bias Representation

Simple Neural Network Implementation

- Building the Dataset

- Implementing the Neural Network

- Training and Testing the Neural Network

Summary

TRAINING A NEURAL NETWORK

High-Level Overview

Gradient Descent

- Finding Minimums

- Updating the Weights

Stochastic Gradient Descent

- Batches and Minibatches

- Convex vs. Nonconvex Functions

- When to Stop Training

- The Learning Rate

- Momentum

Backpropagation

- A Simple Example

- An Abstract Example

Loss Functions

- Absolute and Mean Squared Error Loss

- Cross-Entropy Loss

- Weight Initialization
- Managing Model Complexity and Generalization
 - Overfitting
 - Regularization
 - L2 Regularization
 - Dropout
- Summary

EXPERIMENTS WITH NEURAL NETWORKS

- 1. Dataset
- 1. MLPClassifier Class
- Architecture and Activation Functions
 - The Code
 - The Results
- Batch Size
- Base Learning Rate
- Training-Set Size
 - Regularization
- Momentum
- Weight Initialization
- Feature Ordering
- Summary

VALUATING MODELS

- Definitions and Assumptions
- Why Accuracy Is Not Enough
- The 2x2 Confusion Matrix
- Metrics Derived from the 2x2 Confusion Matrix
 - Deriving Metrics from the 2x2 Table
 - Using Metrics to Interpret Models

More-Advanced Metrics

- Informedness and Markedness

- F1 Score

- Cohen's Kappa

- Matthews Correlation Coefficient

- Metric Implementation

The Receiver Operating Characteristics Curve

- Gathering the Models

- Plotting the Metrics

- Exploring the ROC Curve

- Comparing Models with ROC Analysis

- Generating an ROC Curve

Handling Multiple Classes

- Extending the Confusion Matrix

- Calculating Weighted Accuracy

- Considering the Multiclass Matthews Correlation Coefficient

Summary

PART IV

CONVOLUTIONAL NEURAL NETWORKS

INTRODUCTION TO CONVOLUTIONAL NEURAL NETWORKS

Why Convolutional Neural Networks?

Convolution

- Scanning with the Kernel

- Using Convolution for Image Processing

Anatomy of a Convolutional Neural Network

- Exploring the Types of Layers

- Passing Data Through the CNN

Convolutional Layers

- How They Work
 - In Action
 - Multiple Layers
 - Initialization
- Pooling Layers
- Fully Connected Layers
- Fully Convolutional Layers
- How the CNN Operates
- Summary

0

EXPERIMENTS WITH KERAS AND MNIST

- Building CNNs in Keras
 - Loading the MNIST Data
 - Building the Model
 - Training and Evaluating the Model
 - Plotting the Error
- Basic Experiments
 - Architecture Experiments
 - Training-Set Size, Minibatches, and Epochs
 - Optimizers
- Fully Convolutional Networks
 - Building and Training the Model
 - Making the Test Images
 - Testing the Model
- Scrambled MNIST Digits
- Summary

1

EXPERIMENTS WITH CIFAR-10

- CIFAR-10 Refresher
- Using the Full CIFAR-10 Dataset

- Building the Models
- Analyzing the Models
- Animal or Vehicle?
- Binary or Multiclass?
- Using the Augmented CIFAR-10 Dataset
- Summary

2

CASE STUDY: CLASSIFYING AUDIO SAMPLES

- Building the Dataset
 - Augmenting the Dataset
 - Preprocessing the Data
- Classifying the Audio Features
 - With Classical Models
 - With a Traditional Neural Network
 - With a Convolutional Neural Network
- Spectrograms
- Classifying Spectrograms
- Summary

PART V

ADVANCED NETWORKS AND GENERATIVE AI

3

ADVANCED CNN ARCHITECTURES

- Using the Keras Functional API
- VGG
 - Standardizing with Batch Normalization
 - Applying Dropout After Convolutional Layers
 - Building the VGG8 Model

- Testing the VGG8 Model
- ResNet
 - Mitigating the Vanishing Gradient Problem
 - Exploring ResNet Configurations
 - Building the ResNet-18 Model
 - Testing the ResNet-18 Model
- MobileNet
 - Implementing Depthwise Convolutions
 - Building the MobileNet Model
 - Testing the MobileNet Model
- Building an Ensemble
- Summary

4

FINE-TUNING AND TRANSFER LEARNING

- Fine-Tuning a Pretrained Model
 - VGG16 and MobileNet with CIFAR-10
 - Experiments
 - A Study in Fine-Tuning
- Learning New Features with Transfer Learning
 - Extracting CIFAR-10 Features with VGG16
 - Training Classical Models with CIFAR-10 Features
 - Detecting Anomalies with CIFAR-10 Features
 - Retrieving Images with CIFAR-10 Features
- Summary

5

FROM CLASSIFICATION TO LOCALIZATION

- Detection Experiments with MNIST
 - Building the Dataset
 - Building the Model
 - Running the Model

- Running an Advanced Detection Model
- Semantic Segmentation with U-Net
 - Implementing the U-Net Model
 - Running the Model and Interpreting Its Output
- Multilabel Classification
 - Dataset
 - Performance
- Summary

6 SELF-SUPERVISED LEARNING

- Building the Unlabeled Dataset
- Rotation Prediction
 - Building and Testing RotNet
 - Exploring Variations on the RotNet Theme
 - Examining RotNet Features
 - Fine-Tuning Experiments
- Siamese Networks
 - Building and Testing the Siamese Networks
 - Examining Siamese Network Features
 - Fine-Tuning Experiments
- For Further Exploration
- Summary

7 GENERATIVE ADVERSARIAL NETWORKS

- How GANs Work
- Conditional GANs
 - Building a GAN with Multilayer Perceptrons
 - Building a GAN with Convolutional Layers
 - Experimenting with Unconditional GANs
- Unconditional GANs

- Implementing a Conditional GAN
- Experimenting with Conditional GANs
- Exploring the Latent Space
 - Intrinsic Dimensionality
 - Latent Space Interpolation
- Summary

8

LARGE LANGUAGE MODELS

- Understanding Large Language Models
 - Evaluating the LLM Block Diagram
 - Tokenizing and Embedding
 - Exploring the Transformer Layer
 - Predicting the Next Token
- Context Learning
 - Configuring the Experiment
 - Running the Experiment
 - Testing Another LLM
- Running LLMs Locally
 - Representing Parameters with Quantization
 - Installing Ollama
 - Installing Open Source LLMs
 - Chatting with a Simple Chatbot
- LM Embeddings
 - Embeddings Encode Meaning
 - Semantic Search and Retrieval-Augmented Generation
- Sentiment Analysis
- LMs and Images
 - Describing Images
 - Classifying Images with an LLM?
- Are LLMs Creative?
 - The Divergent Association Task

DAT Scores as a Function of Temperature
Creative Writing as a Function of Temperature

Summary

AFTERWORD

INDEX

OceanofPDF.com

FOREWORD TO THE FIRST EDITION

Since the dawn of the modern digital era, scientists and engineers have taken inspiration from the human brain to imagine how massively parallel networks of simple neuron-like processors might learn and adapt from experience. There have been waves of excitement in this topic as new mathematical methods were developed.

In 1958, Frank Rosenblatt proposed a learning device called the Perceptron, which had the amazing property that it could learn any task that one could program it by hand to perform. The enthusiasm for this device vanished when Marvin Minsky and Seymour Papert performed a careful analysis showing limitations on what the device could be programmed to do, both in principle and in practice.

In the late 1980s, cognitive scientist David Rumelhart, along with Geoffrey Hinton and Ronald Williams, proposed a learning algorithm called backpropagation that had the potential to overcome the limitations identified by Minsky and Papert. Impressive demonstrations of the algorithm, such as a text-to-speech system called NETtalk, led to another surge of interest in neural networks. This time, enthusiasm in the field was dampened when the algorithm did not seem to scale up to handle larger problems.

Over the next 20 years, computers got faster, datasets got larger, and new software tools made it easier to build neural networks. With these developments, much larger models became feasible. The field was rechristened *deep learning*, and a new generation of practitioners were able to tackle problems on a scale that was previously unimaginable.

Although history suggests that we will once again hit a wall with modern deep learning, the field has proven that it can solve difficult, practical, high-impact problems. From voice-controlled assistants to human-expert-level diagnosis of medical images to autonomous vehicles to myriad other behind-the-scenes applications, our lives have been

transformed. The deep learning revolution is upon us, and the future holds untold promises of capabilities yet to come.

One might think that such advanced technology is beyond the understanding of most of us, but the underlying principles are quite understandable and accessible. Indeed, the academic grandparents of the deep learning revolution were psychologists by training. A standard desktop computer with open source software tools is adequate to explore the ideas and concepts in this textbook. With a modest investment in hardware upgrades (notably, a graphics processing unit, or GPU), the computer becomes what would have been considered a supercomputer a decade ago, enabling sophisticated research and implementation.

Dr. Kneusel is an expert in image processing and has over 15 years of industry experience with machine learning. He wrote *Practical Deep Learning* to make the field approachable to novices and hobbyists. With no assumption of background knowledge, it starts at the beginning. It shows how to build a dataset that will be useful for training a successful deep learning model. It then explores classical machine learning algorithms with the intent of grounding the methods that led to the deep learning revolution.

Practical Deep Learning provides not only a solid conceptual foundation but also the practical guidance readers will appreciate to design their own projects and solutions. It addresses how to tune and evaluate the performance of a machine learning model via the standards of current practice. Throughout the book, intuition is emphasized. Practical knowledge builds on intuition. *Practical Deep Learning* also serves as a springboard to help launch you on to more advanced treatments of the methods and algorithms. The last four chapters dive into convolutional neural networks, a workhorse of supervised deep learning. The experiments in these chapters use standard datasets familiar to all those who work in the field. These chapters culminate in a case study, an example of how to approach a problem, from a dataset to evaluating a predictive model.

No book is complete. *Practical Deep Learning* is an introduction. The final chapter of the book points you toward what you may want to

investigate next as you continue your journey into the deep learning revolution. Enjoy the exploration.

Michael C. Mozer, PhD
Professor, Department of Computer Science and
Institute of Cognitive Science
University of Colorado, Boulder

OceanofPDF.com

ACKNOWLEDGMENTS

I am not I in myself alone, but only in all others.
—David Bentley Hart

We are ourselves only in relation to all others. This is true in every area of our lives, including, even, in the writing of books.

First, I'd like to thank my family for their patience and encouragement to help me see this project through.

Next, I want to thank my editor, No Starch's very excellent and easy-to-work-with Eva Morrow, who turned a mass of rough text into a flowing and coherent book. The same is to be said of Paul Nord, who reviewed the first edition, and Scott Couwenhoven, who saved me from my foolish errors in the second. Any remaining errors are on me for not listening to their sage suggestions.

Finally, I want to thank all the good folks at No Starch Press for believing in the book and helping it become a reality.

OceanofPDF.com

INTRODUCTION



When I was in high school, I wanted to write a tic-tac-toe program that allowed the user to play against a computer. At the time, I was blissfully unaware of how real computer scientists approached such a problem. I had only my own thoughts, and those were to implement a lot of rules using the crude `if...then` statements and `gotos` supported by unstructured Applesoft BASIC. My program had a lot of rules—a few hundred lines' worth.

The game worked well enough until I found the sequence of moves that my rules didn't cover and was able to win every time. I felt certain that there must be a way to teach a computer how to do things by showing it examples instead of brute-force code and rules: a way to make the computer learn on its own.

As an undergraduate in the late 1980s, I was excited to sign up for a course in artificial intelligence (AI). The course finally answered my question about how to write a tic-tac-toe playing program, but the computer wasn't learning; it was still just using a clever algorithm. Incidentally, the same course assured us that while it was expected that someday a computer would beat the world's best chess player (it happened in 1997), it would be impossible for a computer to beat the best human at a game like Go. In March 2016, the AlphaGo deep learning program did just that.

In 2003, while working as a consultant for a scientific computing company, I was assigned to a project with a major medical device manufacturer. The goal was to classify, in real time, intravascular ultrasound images of coronary arteries by using *machine learning*: a subfield of AI that learns from data on its own, developing models that were not explicitly programmed by a human. This was what I was waiting for!

I was vaguely aware of machine learning and that there were strange beasts called neural networks that could do some interesting things, but for the most part, machine learning was a small research area and not something the average computer scientist paid much attention to. However, during the project, I fell in love with the idea of training a machine to do something useful without explicitly writing a lot of code. I kept learning on my own, even after the project ended.

Circa 2010, I was involved with another machine learning project, and the timing was perfect. People were just beginning to discuss a new approach to machine learning called *deep learning*, which revived the old neural networks. When 2012 rolled around, the floodgates opened. I was fortunate enough to be in the room at the 2012 International Conference on Machine Learning (ICML) in Edinburgh, Scotland, when Google presented its initial breakthrough deep learning results that responded to cats in YouTube videos. The room was crowded. After all, a whopping 800 people were at the conference. More recent conferences routinely exceed 10,000 attendees.

Machine learning and deep learning have already affected our lives. The recent advent of large language models (Chapter 18) guarantees that the effects will only grow with time. This book exists to help you learn the essentials of machine learning and deep learning.

Who This Book Is For

I wrote this book for readers who have no background in machine learning, but who are curious and willing to experiment. I've kept the math to a minimum. My goal is to help you understand core concepts and build intuition you can use going forward.

At the same time, I didn't want to write a book that simply instructed you on *how* to use existing toolkits but was devoid of any real substance as to *why*. While you can build models while caring only about the how, without the why, you'll be parroting rather than understanding, let alone moving the field forward with your own contributions.

As far as assumptions on my part, I assume you have some familiarity with computer programming, in any language. The language of choice for machine learning, whether you are a student or a major corporation, is Python, so that's the language we'll use in this book.

I'll also assume you're familiar with high school math, excluding calculus. A little calculus will creep in anyway, but you should be able to follow the ideas, even if the technique is unfamiliar. I'll also assume you know a bit of statistics and basic probability. If these concepts are new to you, or you are a bit rusty, Chapter 0 includes a brief summary. You can also dive deeper into these topics with my book *Math for Deep Learning* (No Starch Press, 2021).

Finally, deep learning in Python makes heavy use of the NumPy library, which extends Python by adding high-speed array-processing abilities ideally suited to scientific programming. If NumPy is new to you, you'll find a brief tutorial on the book's GitHub site, or you can dive deeper with *Python Tools for Scientists* by Lee Vaughan (No Starch Press, 2022).

What You Can Expect to Learn

By working through this book in its entirety, you'll gain the knowledge and skills to build and augment datasets, use classical machine learning techniques, work with neural networks, use convolutional neural networks (CNNs), and experiment with advanced deep learning methods.

You'll learn how classical machine learning techniques like k -nearest neighbors, random forests, and support vector machines paved the way for the advanced neural networks that you'll learn to develop later in the book.

Starting with the traditional feed-forward models that led directly to convolutional models, enabling computers to interpret images, you'll explore how to build neural networks from the ground up. Additionally, you'll learn how to evaluate models in order to properly judge their effectiveness.

You'll also explore advanced neural networks, including standard CNN architectures for image classification and detection; the what and why of fine-tuning and transfer learning used to leverage pretrained models; how to perform semantic segmentation to label every pixel of an image; and when to use multilabel classification to identify all the classes present in an input.

Sometimes there isn't enough data, and you'll discover that in those situations, self-supervised learning makes it possible to train models using often-plentiful unlabeled data.

Finally, you'll learn about generative AI in the form of generative adversarial networks, the precursor to text-to-image generators, and how to work with locally run large language models and the embeddings they provide.

About This Book

This book is about machine learning: building models that take input data and arrive at a conclusion from that data. That conclusion might be a label placing the object into a particular class of objects (like a certain kind of dog) or a continuous output value (say, the price one should ask

for a house with the given set of amenities). The key here is that the model learns from the data on its own. In effect, the model learns by example.

Think of the model as a mathematical function, $y = f(x)$, where y is the output, the class label, or the continuous value, and x is the set of features representing the unknown input. *Features* are measurements or information about the input that the model can use to learn what output to generate. For example, x might be a vector representing the length, width, and weight of a fish, where each of those measurements is a feature. Our goal is to find f , a mapping between x and y that we'll be able to use on new instances of x , for which we do not know y .

The standard way to learn f is to give our model (or algorithm) known data and have the model learn the parameters it needs to make f a useful mapping. This is why it's called *machine learning*: The machine is learning the parameters of the model. We're not thinking of the rules ourselves and cementing them in code. Indeed, for some model types like neural networks, it's not even clear *what* the model has learned, only that the model is now performing at a useful level.

Machine learning has three main branches: supervised learning, unsupervised learning, and reinforcement learning. The process I just described falls under *supervised learning*, in which we supervise the training of the model with a set of known x and y values, the *training set*. We call a dataset like this a *labeled* dataset because we know the y that goes with each x . *Unsupervised learning* attempts to learn the parameters used by the model, using only x . We will discuss unsupervised learning in the form of self-supervised and generative models.

Reinforcement learning trains models to perform tasks, like playing chess or Go. The model learns a set of actions to take given the current state of its world. This is an important area of machine learning, and it has recently achieved a high level of success on tasks once thought to belong solely to the domain of human capability. However, to keep the book manageable, we'll ignore reinforcement learning in favor of generative AI models like generative adversarial networks, diffusion models, and transformer-based large language models.

Terminology

One quick note on terminology. In the media, a lot of what I talk about in this book is referred to as *artificial intelligence (AI)*. While this is not wrong, it's somewhat misleading: Machine learning is one subfield of the broader field of AI. Another term you'll often hear is *deep learning*. This term is a bit nebulous, but for our purposes, we'll use it to mean machine learning with neural networks—in particular, neural networks with many layers (hence *deep*). Figure 1 shows the relationships among these terms.

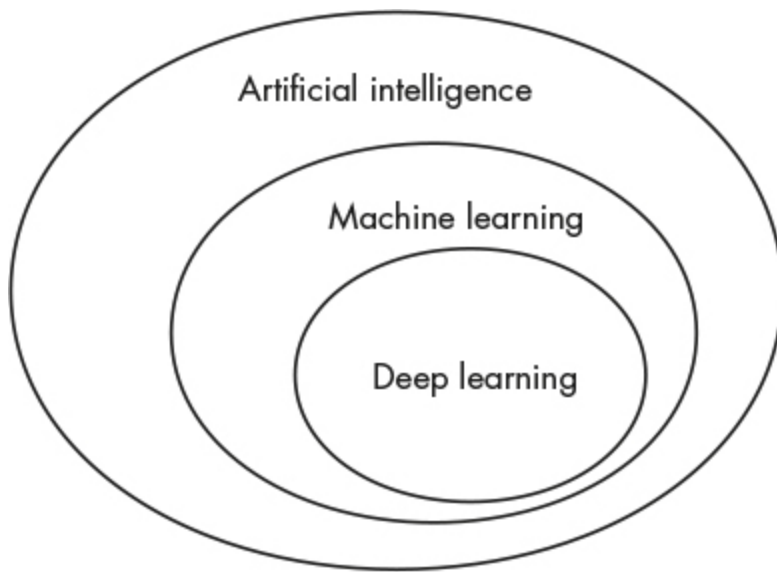


Figure 1: The relationships among artificial intelligence, machine learning, and deep learning

Of course, considerable variety exists within the fields of machine learning and deep learning. We'll encounter numerous models throughout this book. We could arrange them in what we'll call *the tree of machine learning*, pictured in Figure 2.

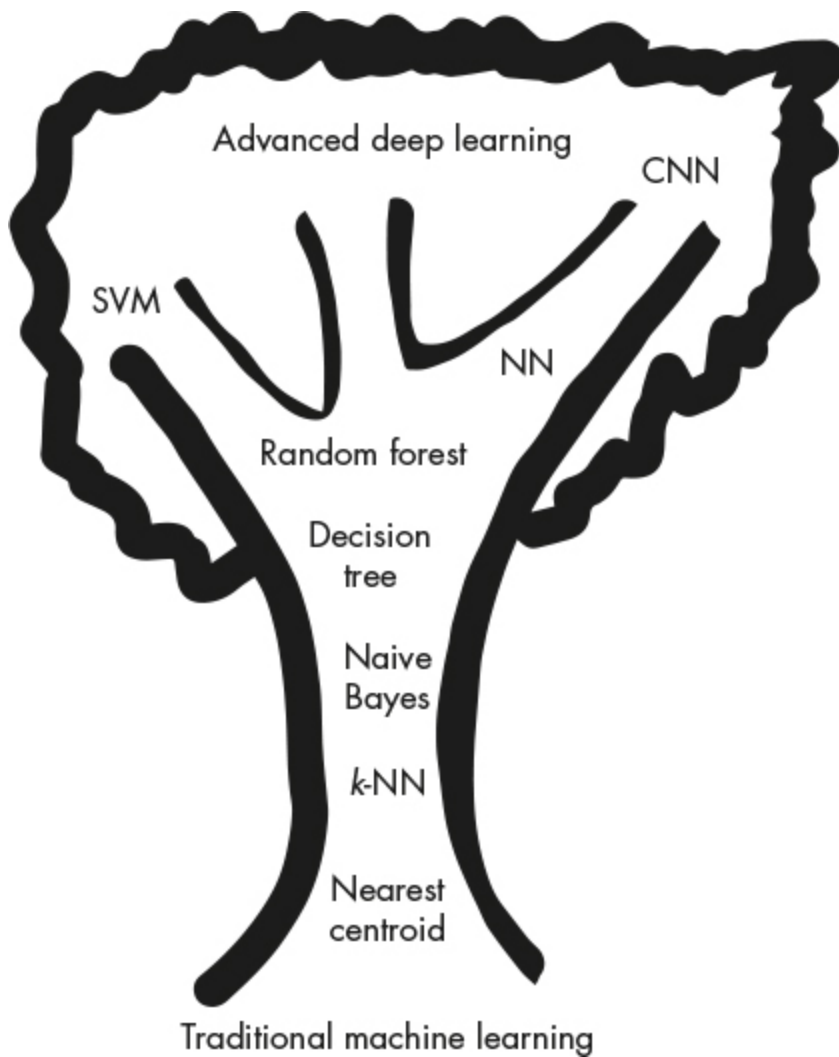


Figure 2: The tree of machine learning

The tree shows the growth from traditional machine learning at the root to modern deep learning at the top. Consider this a preview of what's to come; we'll look at each of these models in this book.

What's New in the Second Edition

Much has happened in deep learning since the first edition of this book was published in 2021. This edition includes six new chapters to reflect these advances in the field.

Chapter 13, the first new chapter, builds on the basic CNNs of the first edition by introducing advanced architectures for computer vision. Chapter 14 then explores two techniques that have become essential in

the toolkit of any deep learning practitioner: fine-tuning and transfer learning.

Classification is useful, but it's often necessary to go further by stating not only, for example, that there is a dog in an image, but also that the dog is located in a particular spot; this is called *detection*. Detection itself becomes insufficient when each pixel requires a label. In that case, semantic segmentation provides the necessary output. Finally, sometimes all we need to know is which classes are present in an image. For that, we need multilabel classification. All three approaches are covered in Chapter 15.

Next, we delve into self-supervised learning in Chapter 16. Labeling datasets is difficult, time-consuming, and often infeasible to the level necessary to build datasets large enough to train advanced models. Self-supervised learning attempts to overcome the tedium by pretraining models via pseudolabels generated from unlabeled inputs.

The last two chapters focus on generative AI, which has exploded on the scene and promises to radically transform our society. Chapter 17 explores generative adversarial networks, the precursor to image generators like DALL-E and Stable Diffusion. Chapter 18 experiments with large language models like GPT-4 to understand their operation at a high level and to put them to work for practical applications.

Synopsis

The book begins with Chapter 0, detailing our operating environment and relevant mathematical preliminaries. Specifically, this chapter tells you how to set up your assumed working environment. It also includes sections about vectors, matrices, probability, and statistics that you can use as refreshers or for background.

The remainder of the book is divided into five parts. Part I lays the foundation for all that follows by driving home the importance of data. For deep learning, data is everything.

Chapter 1: It's All About the Data Bad datasets lead to bad models; I'll teach you what makes a good one.

Chapter 2: Building the Datasets You'll build the datasets used throughout the book. You'll also learn how to augment datasets.

Part II familiarizes you with classical machine learning techniques. These techniques, still useful on their own, will prepare you for neural networks and deep learning.

Chapter 3: Introduction to Machine Learning To understand where you're going, it can help to know where you came from. Here we'll cover some of the original machine learning models.

Chapter 4: Experiments with Classical Models This chapter discusses the strengths and weaknesses of the old-school approach to machine learning. We'll refer to these results for comparison purposes throughout the book.

Part III introduces neural networks, the heart of modern deep learning and AI. The foundation laid in this part transfers directly to the remainder of the book. The advanced neural network models presented in Parts IV and V are extensions to the core neural network presented in these chapters.

Chapter 5: Introduction to Neural Networks Modern deep learning is all about neural networks; I'll introduce them here.

Chapter 6: Training a Neural Network This challenging chapter provides the necessary knowledge to understand how neural networks are trained. Some basic calculus slipped into this chapter, but don't panic; it's discussed at a high level to give you intuition, and the notation isn't as frightening as it might seem at first.

Chapter 7: Experiments with Neural Networks Here we run experiments to get a feel for actually working with data.

Chapter 8: Evaluating Models To understand the results presented in machine learning papers, talks, and lectures, you need to understand how to evaluate models. This chapter will take you through the process.

Part IV teaches about convolutional neural networks, the kind of model that triggered the deep learning revolution in 2012.

Chapter 9: Introduction to Convolutional Neural Networks

The deep learning we'll focus on in this book is embodied in the idea of a convolutional neural network (CNN). This chapter discusses the basic building blocks of these networks.

Chapter 10: Experiments with Keras and MNIST Here we explore how CNNs work by experimenting with the MNIST dataset, the workhorse of deep learning.

Chapter 11: Experiments with CIFAR-10 The MNIST dataset, useful as it is, is a simple one for CNNs to master. Here we explore another workhorse dataset, CIFAR-10, which consists of actual images and will challenge our models.

Chapter 12: A Case Study: Classifying Audio Samples We conclude our introduction to CNNs with a case study. We start with a new dataset, one not in widespread use, and work through the process of building a good model for it. This chapter uses everything presented in previous chapters, from building and augmenting data to classical models, traditional neural networks, CNNs, and ensembles of models.

Part V presents new chapters detailing the advanced deep learning techniques, from advanced CNN architectures to fine-tuning and transfer learning, that allow us to leverage the learning encapsulated in pretrained networks. Additional chapters experiment with localization techniques, self-supervised learning, generative adversarial models for image creation, and the large language models that have proven themselves a quantum leap in AI in recent years.

Chapter 13: Advanced CNN Architectures The CNNs of Part IV form the foundation of the more advanced CNN architectures currently in use for computer vision tasks. In this chapter, you'll learn about some of these advanced networks.

Chapter 14: Fine-Tuning and Transfer Learning We need not always train models from scratch. This chapter explores fine-tuning and transfer learning, including the difference between the two,

and how they allow the building of models by using models pretrained on other datasets as a foundation.

Chapter 15: From Classification to Localization While it's nice to know there is a cat in an image, it's even nicer to know *where* it is in the image (detection) and nicer still to assign a label to every image pixel (semantic segmentation). These are the goals of this chapter, with multilabel classification thrown in for good measure.

Chapter 16: Self-Supervised Learning Labeling datasets is tedious at best and sometimes virtually impossible. This chapter explores an emerging use case where models learn from unlabeled data via automatic proxy labels.

Chapter 17: Generative Adversarial Networks Many of the most impressive of deep learning's achievements are due, at least in part, to generative adversarial networks (GANs). In this chapter, we explore GANs to understand how they work.

Chapter 18: Large Language Models The dawn of powerful large language models (LLMs) in fall 2022, like ChatGPT, has ushered in a new era in AI. This chapter explores LLMs and the transformer architecture they are based on before experimenting with LLMs of all sizes, from those that fit on your CPU-only desktop to state-of-the-art models.

You can find all the code files and other resources on the book's GitHub site at <https://github.com/rkneusel9/PracticalDeepLearning2E>. If you have questions or comments, feel free to contact me via email at rkneuselbooks@gmail.com or visit my website at <https://www.rkneusel.com>.

Let's now get started by configuring our operating environment.

OceanofPDF.com

0

ENVIRONMENT AND MATHEMATICAL PRELIMINARIES



This chapter introduces our operating environment and details how to set it up. It also includes a primer on some of the math we will encounter. We'll end with a brief note about graphics processors, or GPUs, which you may have heard are essential for deep learning. For our purposes, they're not, so don't worry—this book won't suddenly cost you a lot of money.

The Operating Environment

This section details the environment we'll assume you're using as you follow along with the experiments throughout the book. Our underlying assumption is that we're using a 64-bit Linux system. The exact distribution is not critical, but to make explanations simpler in the presentation, we'll also assume that we're using Ubuntu. Given the excellent support behind the Ubuntu distribution, we trust that any newer distributions will work similarly. The Python language is our lingua franca, the common language of machine learning.

Let's look at a quick overview of the Python toolkits that we'll be using.

NumPy

NumPy is a Python library that adds array-processing abilities to Python. While we can use Python lists like one-dimensional arrays, in practice they are too slow and inflexible. The NumPy library adds the array features missing from Python that are necessary for many scientific applications. The other libraries we'll use require NumPy as a base library.

scikit-learn

The traditional machine learning models we'll explore in this book are within scikit-learn (sklearn). This library uses NumPy arrays. It implements a standardized interface to many machine learning models as well as an entire host of other functionality that we won't even have time to touch. I strongly encourage you to review the official documentation (<https://scikit-learn.org/stable/documentation.xbhtml>) as you become more familiar with machine learning and the tools behind it.

TensorFlow with Keras

Deep learning is hard enough to understand, let alone implement efficiently and correctly, so instead of attempting to write CNNs from scratch, we'll use one of the popular toolkits already in active development. From its inception, the deep learning community has supported the development of toolkits to make deep networks easier to use and has made the toolkits open source with very generous licenses. The toolkit we'll use is TensorFlow, which includes Keras as its Python interface.

TensorFlow is an open source Google product that implements the core functionality of deep neural networks for many platforms. We selected Keras not only because it's popular and in active development, but also because it's straightforward to use. Our goal is to become

familiar with deep learning to the point where we can implement models and use them with a minimum of programming overhead.

Installing the Toolkits

We can't reasonably give an exhaustive guide for installing the toolkits on all systems and hardware. Instead, we'll provide step-by-step instructions for the specific operating system we'll use as a reference. These steps, along with the minimum version numbers of the libraries, should be enough for most readers to get a working system in place.

Remember, we're assuming that we're working in a Linux environment, specifically Ubuntu. Ubuntu is a widely used Linux distribution, and it runs on almost any modern computer system. Other Linux distributions will work, as will macOS and Windows, but the instructions here are specific to Ubuntu.

A freshly installed Ubuntu base desktop system gives us Python for free. To install the remaining packages, we need to go into a shell and execute the following sequence of commands in order:

```
$ sudo apt-get update
$ sudo apt-get install python3-pip
$ sudo apt-get install build-essential python3-dev
$ pip3 install numpy
$ pip3 install scipy
$ pip3 install matplotlib
$ pip3 install scikit-learn
$ pip3 install tensorflow-cpu==2.10.0
$ pip3 install pillow
```

Once the installation is complete, we'll have installed the latest versions of the libraries and toolkits.

Any later version of these libraries will, with high probability, also suffice, with the exception of TensorFlow. For that, we specify a version 2 installation to ensure that the proper version of the Keras library is included.

The Pillow library is an image-processing library, and Matplotlib is for plotting.

INSTALLING ON UBUNTU 24.04 OR LATER

Ubuntu 24.04 and later versions require `pip3` to run inside an environment. Fortunately, setting up an environment is straightforward. First, install `venv`:

```
$ sudo apt-get install python3-venv
```

Then, from your home directory, create the `pd1` environment and activate it:

```
$ python3 -m venv pd1
$ source pd1/bin/activate
```

You'll know the environment is active by the `(pd1)` prefix before the command prompt. I chose `pd1` to stand for *Practical Deep Learning*, but you can change it to whatever you want.

Now install the packages with the `pip3` instructions from earlier. For example:

```
(pd1) $ pip3 install numpy
```

To deactivate the environment, enter `deactivate` at the command prompt. Use the `source` command to reactivate the environment. You must activate the environment before running the experiments. Environments encapsulate different Python library configurations and prevent conflicts.

The following two sections are light introductions to some of the math that will creep into the book. For a more complete introduction, see my book *Math for Deep Learning* (2021).

Basic Linear Algebra

We're about to look at vectors and matrices. The math that deals with these concepts falls under the general heading of linear algebra, or matrix theory. As you might imagine, linear algebra is a complex field. All you need to know for this book is what a vector is, what a matrix is, and how we can multiply two vectors, or two matrices, or vectors and matrices together. You'll see later that this gives us a powerful way to implement specific models, particularly neural networks.

Vectors

A *vector* is a one-dimensional (1D) list of numbers. Mathematically, a vector might appear as follows, with the third element given as $a_2 = 2$:

$$a = [0, 1, 2, 3, 4]$$

We're following the programming convention of indexing from 0, so a_2 gives us the third element in the vector.

The previous horizontal vector is a *row vector*. In mathematical expressions, however, we usually assume vectors are vertical:

$$a = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

A vertical vector is a *column vector*. This vector has five elements, and we denote it as a five-element column vector. In this book, we'll typically use vectors to represent one *sample*: one set of features that we'll input to a model.

Mathematically, we use vectors to represent points in space. If we're talking about the two-dimensional (2D) Cartesian plane, we locate a point with a vector of two numbers, (x, y) , where x is the distance along the x-axis and y is the distance along the y-axis. That vector represents a

point in two dimensions, even though the vector itself has only one dimension. If we have three dimensions, we need a vector with three elements, (x, y, z) .

In machine learning, since we often use vectors to represent the inputs to our models, we'll be working with dozens to hundreds of dimensions. While we can't plot them as points in space, mathematically, that's what they are. As you'll see, some models, such as k -nearest neighbors (k -NN), use the feature vectors as just that—points in a high-dimensional space.

Matrices

A *matrix* is a 2D array of numbers, and we index a particular entry by its row number and column number. For example, this is a matrix:

$$a = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

If we want to refer to the 7, we write $a_{1,2} = 7$. Again, we're indexing from 0. Because this matrix a has three rows and four columns, we call it a 3×4 matrix.

Vector and Matrix Multiplication

The simplest way to think of multiplying two vectors together is to multiply their corresponding elements. For example:

$$[1, 2, 3] \times [4, 5, 6] = [4, 10, 18]$$

This is the most common way to multiply an array when using a toolkit like NumPy, and we'll make heavy use of this in the chapters that follow. However, in mathematics, this is seldom actually done.

When multiplying vectors together mathematically, we need to know whether they are row or column vectors. We'll work with two vectors, $A = (a, b, c)$ and $B = (d, e, f)$, which, following mathematical convention, we assume to be column vectors. Adding a superscript T

turns a column vector into a row vector. One of the mathematically allowed ways to multiply A and B is:

$$AB^T = \begin{bmatrix} a \\ b \\ c \end{bmatrix} [d \ e \ f] = \begin{bmatrix} ad & ae & af \\ bd & be & bf \\ cd & ce & cf \end{bmatrix}$$

This is called the *outer product*. The other mathematically allowed way to multiply A and B is called the *inner product*, or *dot product*:

$$A^T B = [a \ b \ c] \begin{bmatrix} d \\ e \\ f \end{bmatrix} = ad + be + cf$$

Notice that the outer product becomes a matrix, and the inner product becomes a single number, a *scalar*.

When multiplying a matrix and a vector, the vector is typically on the right side of the matrix. The multiplication can proceed if the number of columns in the matrix matches the number of elements in the vector, which we again assume to be a column vector. The result is also a vector with as many elements as there are rows in the matrix (read $ax + by + cz$ as a single element). For example:

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \end{bmatrix}$$

Here we've multiplied a 2×3 matrix by a 3×1 column vector to get a 2×1 output vector. Notice that the number of columns of the matrix and the number of rows of the vector match. If they don't, the multiplication is not defined. Also, notice that the values in the output vector are sums of products of the matrix and vector. This same rule applies when multiplying two matrices:

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \begin{bmatrix} A & B \\ C & D \\ E & F \end{bmatrix} = \begin{bmatrix} aA + bC + cE & aB + bD + cF \\ dA + eC + fE & dB + eD + fF \end{bmatrix}$$

Multiplying a 2×3 matrix by a 3×2 matrix gives us a 2×2 answer.

When we get to CNNs, we'll work with arrays that have three and even four dimensions. We generally refer to such arrays as *tensors*. If we imagine a stack of matrices, all the same size, we get a three-dimensional (3D) tensor. We can use the first index to refer to any one of the matrices, and the remaining two indices to refer to a particular element of that matrix. Similarly, if we have a stack of 3D tensors, we have a four-dimensional (4D) tensor, and we can use the first index of that to refer to any one of the 3D tensors.

The main points of this section are that vectors have one dimension, matrices have two dimensions, there are rules for multiplying these objects together, and our toolkits will work with 4D tensors in the end. We'll review some of these points as we encounter them later in the book.

Statistics and Probability

The topics of statistics and probability are so broad that often it's better either to say almost nothing or to write a book or two. Therefore, I'll mention only key ideas that we'll use throughout the book and leave the rest to you to pick up as you see fit. I'll assume you know some probability basics from flipping coins and rolling dice.

Descriptive Statistics

When conducting experiments, we need to report the results in a meaningful way. Typically, for us, we'll report results as the *mean* (arithmetic average), plus or minus a quantity we call the *standard error of the mean* (SE). Let's define the SE through an example.

If we have many measurements x (say, the length of a part of a flower), we calculate the mean ($\bar{x}$) by adding all the values together and

dividing by the number of added values. Once we have the mean, we calculate the average spread of the individual values around the mean by subtracting each value from the mean, squaring the result, and adding all these squared values together before dividing by the number of added values minus one. This number is the *variance*. If we take the square root of this value, we get the *standard deviation* (σ), which we'll see again shortly. Given the standard deviation, we calculate the standard error of the mean as $SE = \sigma / \sqrt{n}$, where n is the number of values that we used to calculate the mean. The smaller the SE is, the more tightly the values cluster around the mean. We can interpret this value as the uncertainty we have about the mean value. This means we expect the actual mean, which we don't really know, to be between $\bar{x} - SE$ and $\bar{x} + SE$.

Sometimes we'll talk about the median instead of the mean. The *median* is the middle value, the value that half of our samples are below and half are above. To find the median for a set of values, we first sort the values numerically and then find the middle value. This is the exact middle value if we have an odd number of samples, or the mean of the two middle values if we have an even number of samples. The median is sometimes more useful than the mean if the samples do not have an even spread around the mean. The classic example is income. A few very rich people move the mean income up to the point where it does not have much meaning. Instead, the median, the value where half the people make less and half make more, is more representative.

Descriptive statistics is the term we use to describe values we derive from a dataset to understand it. We just reviewed three of them: the mean, the median, and the standard deviation. In later chapters, we'll see how to use these and how we can plot them to further understand datasets.

Probability Distributions

In this book, we'll talk about probability distributions. You can think of a *probability distribution* as an oracle of sorts: When asked, it will give us a number or set of numbers. For example, when we train a model, we use numbers, or sets of numbers, that we measure; we can think of those

numbers as coming from a probability distribution. We'll refer to that distribution as the parent distribution. The *parent distribution* generates the data we'll feed our model. Another more Platonic way to think about it is as the ideal set of data that our data is approximating.

Probability distributions come in many forms; some even have names. The two that we'll encounter are the most common: uniform and normal distributions.

A uniform distribution is what we get if we roll a fair die. If the die has six sides, the likelihood of getting any value, 1 through 6, is the same. If we roll the die 100 times and tally the numbers that come up, the tally will be roughly equal for each number and, in the long run, we can easily convince ourselves that the numbers will even out.

Therefore, a *uniform distribution* is an oracle that is equally likely to give us any of its allowed responses. Mathematically, we write uniform distributions as $U(a, b)$, where U means *uniform*, and a and b are the range of values bracketing the response. Unless we specify that the distribution gives only integers, the response can be any real number. Notationally, we write $x \sim U(0, 1)$ to mean that x is a value returned by the oracle that gives real numbers in the range $(0, 1)$ with equal likelihood. Also, note that using parentheses to bracket a range excludes the associated bound, while using square brackets includes it. Thus, $U[0, 1)$ returns values from 0 to 1, including 0 but excluding 1.

A *normal distribution*, also called a *Gaussian* distribution, looks like a bell curve. In this shape, one value is most likely, and the likelihood of the other values decreases as we get farther from the most likely value. The most likely value is the mean, $\bar{x}$, and the parameter that controls how quickly the likelihood drops to 0 (without ever really reaching it) is the standard deviation, σ (sigma). For our purposes, if we want a sample from a normal distribution, we write $x \sim N(\bar{x}, \sigma)$ to mean x is drawn from a normal distribution with a mean of $\bar{x}$ and a standard deviation of σ .

Statistical Tests

Another topic that pops up from time to time is the *statistical test*, a measurement we use to decide whether a particular hypothesis is likely

true. Typically, the hypothesis relates to two sets of measurements, and the hypothesis is that the two sets of measurements came from the same parent distribution. If the statistic we calculate with the test is outside a certain range, we reject the hypothesis and claim we have evidence that the two sets of measurements are *not* from the same parent distribution.

We'll usually use the *t-test*, a common statistical test that assumes normally distributed data. Because we assume that our data is normally distributed, which may be true, the t-test is known as a *parametric test*.

Sometimes we'll use the *Mann-Whitney U test*, which is like a t-test in that it helps us decide whether two samples are from the same parent distribution, but it makes no assumption about the distribution of the data values in the sample. We call these tests *nonparametric tests*.

Whether the test is parametric or nonparametric, the value we ultimately get from the test is a *p-value*. It represents the probability that we would see the calculated test statistic value, or one more extreme, if the hypothesis that the samples come from the same parent distribution is true. If the p-value is low, we have evidence that the hypothesis is likely not true.

The usual p-value cutoff is 0.05, indicating a 1 in 20 chance that we'd measure the test statistic value (t-test or Mann-Whitney U) even if the samples came from the same parent distribution. However, in recent years, it's become clear that this threshold is too generous. When p-values are near 0.05 but not above, we begin to think there is some evidence against the hypothesis. If the p-value is, say, 0.001 or less, then we have strong evidence that the samples are not from the same parent distribution. In this case, we say that the difference is *statistically significant*.

Graphics Processing Units

One of the enabling technologies for modern deep learning was the development of powerful *graphics processing units (GPUs)*. These are co-computers implemented on graphics cards. Originally designed for video gaming, the highly parallel nature of GPUs has been adapted to the extreme computational demands of deep neural network models.

Many advances of recent years wouldn't have been possible without the supercomputer-like abilities GPUs provide to even basic desktop computers. NVIDIA is the leader in the creation of GPUs for deep learning, and via its Compute Unified Device Architecture (CUDA), NVIDIA has been foundational to the success of deep learning. It's not an understatement to say that without GPUs, deep learning would not have happened, or at least would not have been so widely used.

That said, we're not expecting GPUs to be present for the models we'll work with in this book. We'll use small enough datasets and models so that we can train in a reasonable amount of time by using just a CPU. We've already enforced this decision in the installed packages, since our version of TensorFlow is a CPU-only version.

If you have a CUDA-capable GPU and want to use it for the deep learning portion of this book, please do so, but you don't need to purchase one to run the examples. If you're using a GPU, be sure to install CUDA properly before installing the packages indicated previously and be sure to install a GPU-enabled version of TensorFlow. The sklearn toolkit is CPU only.

Summary

In this chapter, we reviewed our operating environment. Next, we described the essential Python toolkits we'll use throughout the book and gave detailed instructions for installing the toolkits, assuming an Ubuntu 24.04 or later Linux distribution. As mentioned, the toolkits will work just as nicely on many other Linux distributions as well as macOS. We then briefly reviewed some of the math we'll encounter later and ended with an explanation of why we do not need GPUs for our models.

In the next chapter, we begin our exploration of data and datasets.

PART I

DATA IS EVERYTHING

Deep learning is an empirical science. It seeks to build models representing processes that generate the kind of data we're interested in, and to do that effectively, we need many examples of that kind of data. The chapters in this part demonstrate how to work with data and how to build the datasets used throughout the book. In the end, data *is* everything.

OceanofPDF.com

1

IT'S ALL ABOUT THE DATA



Developing a proper dataset is the single most important part of building a successful machine learning model. Machine learning models live and die by the adage *garbage in, garbage out*. As you saw in Chapter 0, a model uses training data to configure itself to the problem. If the training data isn't representative of the data that the model will receive when we use it, we can't expect our model to perform well. In this chapter, you'll learn how to create a good dataset that represents the data the model will encounter in the wild.

Classes and Labels

In this book, we're exploring *classification*: We're building models that put data into discrete categories, or *classes*, like dog breed, flower type, and digit. To represent classes, we give each input in our training set an identifier called a *label*. A label could be the string "Border Collie" or a number like 0 or 1.

Models don't know what their inputs represent. They don't care whether the input is a picture of a border collie or the value of Google stock. To the model, it's all numbers. The same is true of labels. Because

the label for the input has no intrinsic meaning to the model, we can represent classes however we choose.

In practice, class labels are usually integers starting with 0. So, if we have 10 classes, the class labels are 0, 1, 2,..., 9. In Chapter 2, we'll work with a dataset that has 10 classes representing images of different real-world things. We'll simply map them to integers as in Table 1-1.

Table 1-1: Label Classes with Integers

Label	Actual class
0	airplanes
1	automobiles
2	birds
3	cats
4	deer
5	dogs
6	frogs
7	horses
8	ships
9	trucks

With this, we label every training input that's a dog as 5, and every input that's a truck as 9. But what exactly are we labeling? In the next section, we'll cover features and feature vectors, the very lifeblood of machine learning.

Features and Feature Vectors

Machine learning models take features as inputs and deliver, in the case of a classifier, a label as output. So what are these features, and where do

they come from?

For most models, *features* are numbers. What the numbers represent depends on the task at hand. If we're identifying flowers based on measurements of their physical properties, our features are those measurements. If we're using the dimensions of cells in a medical sample to predict whether a tumor is breast cancer, the features are those dimensions. With modern techniques, the features might be the pixels of an image (numbers), a sound's frequency (numbers), or even how many foxes a camera trap counts over a two-week period (numbers).

Features, then, are whatever numbers we want to use as inputs. The goal of training a model is to get it to learn a relationship between the input features and the output label. We assume that a relationship exists between the input features and output label before training the model. If the model fails to train, there might be no relationship to learn.

Features are numbers representing a measured or known thing. *Feature vectors* are sets of these numbers we use as inputs to the model. After training, we give feature vectors with unknown class labels to the model, and the model's output predicts the class label based on the relationships it discovered during training. If the model is repeatedly making poor predictions, the selected features might not be sufficient to capture the relationship. Before we discuss what makes a good feature, let's take a closer look at the features themselves.

Types of Features

We can use different kinds of numbers as features, and as you'll see, not all are created equal. Sometimes we have to manipulate them before we use them with a model.

Floating-Point Numbers

In Chapter 2, we'll build a historic flower dataset. The features of that dataset are actual measurements of details like a flower's sepal width and height (in centimeters). A typical measurement might be 2.33 cm. This

is a *floating-point* number: a number with a decimal point, or a *real* number.

Most models want to work with floating-point numbers, so you can use the measurements as they are. Floating-point numbers are *continuous*, meaning there are an infinite number of values between any two numbers, so we have a smooth transition between them. As you'll see later, some models expect continuous values.

Interval Values

Floating-point numbers don't work for everything, however; for example, flowers can't have 10.14 petals, though they might have 9, 10, or 11. These numbers are *integers*: whole numbers without a fractional part or a decimal point. Unlike floating-point numbers, they are *discrete*, which means they pick out only certain values, leaving gaps in between. Fortunately, integers are just special real numbers, so models can use them as they are.

In the petal example, the difference between 9, 10, and 11 is meaningful in that 11 is bigger than 10, and 10 is bigger than 9. Not only that, but 11 is bigger than 10 in exactly the same way that 10 is bigger than 9. The difference, or *interval* value, between the values is the same: 1.

The pixels in an image are interval values because they represent the (assumed linear) response of a measurement device, like a camera or an MRI machine, to a physical process like intensity and color of visible light or the number of hydrogen protons in free water in tissue. The key point is that if value z is the next number in the sequence after value y , and value x is the number before value y , then the difference between x and y is the same difference as between y and z .

Ordinal Values

Sometimes the interval between the values is not the same. For example, models may include someone's educational level to predict whether they'll default on a loan (whether such a model should exist is an entirely different topic).

Encoding someone's educational level by counting their years of schooling works because the difference between 10 years of schooling and 8 is the same as the difference between 8 years of schooling and 6. However, if we simply assign 1 for "completed high school," 2 for "has an undergraduate degree," and 3 for "has a doctorate or other professional degree," we'd probably be in trouble; while $3 > 2 > 1$ is true, the difference between the values represented by 3 and 2 and by 2 and 1 is not the same. We call these features *ordinal* because they express an ordering, but the differences between the values are not necessarily the same.

Categorical Values

Sometimes we use numbers as codes. We might encode sex as 0 for male and 1 for female, for example. In this case, 1 isn't understood to be greater than or less than 0, so these aren't interval or ordinal values. Instead, they're *categorical*. They express a category but say nothing about any relationship between the categories.

Another common example is color. We might use 0 for red, 1 for green, and 2 for blue. Again, no relationship exists between 0, 1, or 2 in this case. Typically, we can't use categorical features with our models as they are, since most types of machine learning models expect at least ordinal, if not interval, numbers.

We can make categorical values ordinal by using the following trick. If we're using a person's sex as an input, for example, instead of saying 0 for male and 1 for female, we create a two-element vector, one element for each possibility. The first digit in the vector indicates whether the input is male by signaling either 0 (meaning they're not male) or 1 (meaning they are). The second digit indicates whether they're female. We map the categorical values to a binary vector, as Table 1-2 shows.

Table 1-2: Representing Categories as Vectors

Categorical value	Vector representation
0	1 0

Categorical value		Vector representation
1	→	0 1

Here a 0 in the “is male” feature is meaningfully less than a 1 in that feature, which fits the definition of an ordinal value. Changing categorical values into ordinal requires us to expand the number of features in our feature vector, as we need one feature for each of the possible categorical values. With 5 colors, for example, we need a 5-element vector; with 5,000, a 5,000-element vector.

To use this scheme, the categories must be mutually exclusive, meaning there will be only one 1 in each row. Because there’s always only one nonzero value per row, we sometimes call this approach *one-hot encoding*.

Feature Selection and the Curse of Dimensionality

Feature selection is the process of choosing which features to use in your feature vectors. During selection, you shouldn’t include features you don’t need. Here’s a good rule of thumb: The feature vector should contain only features that capture aspects of the data that allow the model to generalize to new data.

Features should capture aspects of the data that help the model separate the classes. Providing a more explicit explanation is impossible, since the set of best features is always specific to the dataset. But we can apply certain concepts to help guide us toward a useful set of features for whatever dataset we’re working with.

Like many aspects of machine learning, selecting features comes with trade-offs. We need enough features to capture all the relevant parts of the data so that the model has something to learn from, but if we have too many features, we fall victim to the curse of dimensionality.

To explain what this means, let’s look at an example. Suppose we restrict our features to the range $[0, 1)$. That’s not a typo; in this interval notation, a square bracket means we include the bound in the range, and

a parenthesis means we exclude the bound. So here we allow 0 but not 1. We'll also assume that our feature vectors are either two-dimensional or three-dimensional. That way, we can plot each feature vector as a point in a 2D or 3D space. Finally, we'll simulate datasets by selecting feature vectors, 2D or 3D, uniformly at random so that each element of the vector is in $[0,1)$.

Let's fix the number of samples at 100. If we have two features, or a 2D space, we can represent 100 randomly selected 2D vectors as in the top of Figure 1-1. If we have three features, or a 3D space, those same 100 features look like the bottom of Figure 1-1.

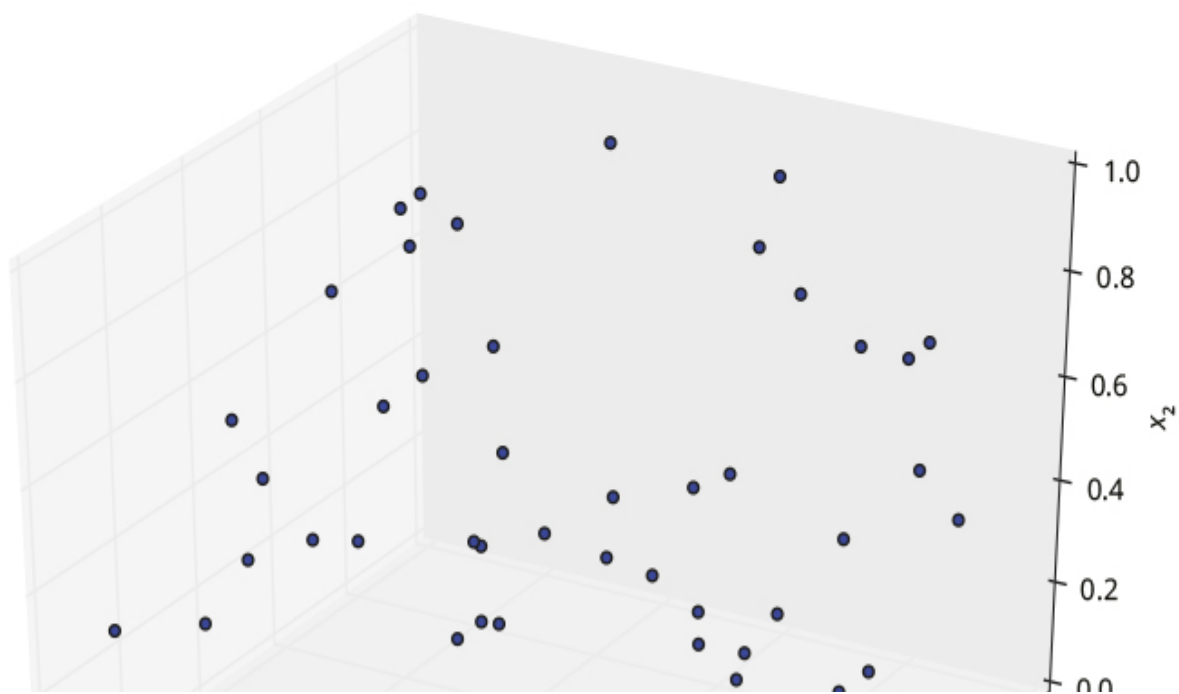
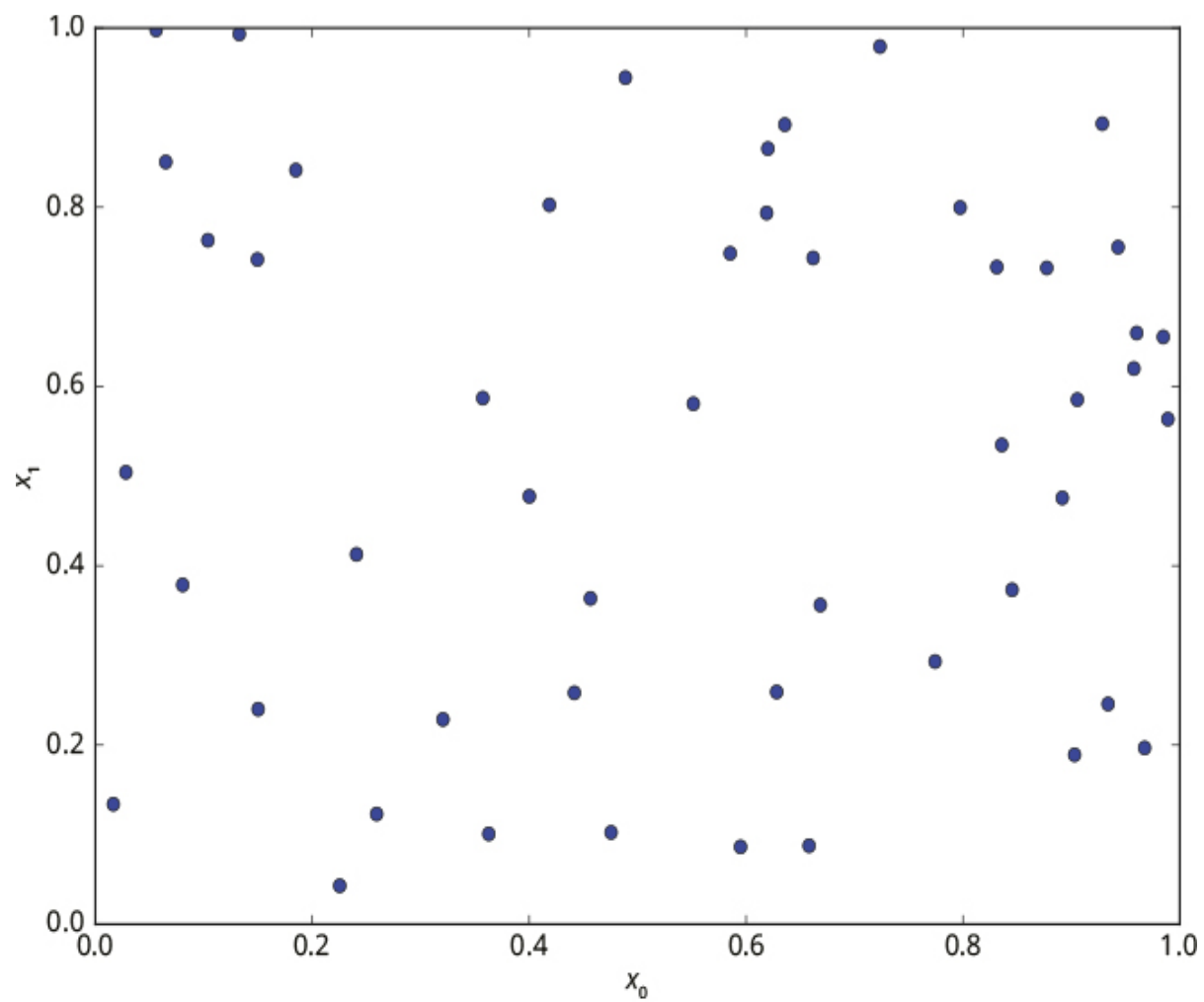


Figure 1-1: One hundred random samples in 2D space (top) and in 3D space (bottom)

Since we're assuming our feature vectors can come from anywhere in the 2D or 3D space, we want our dataset to sample as much of that space as possible so that it represents the space well.

We can measure how well the 100 points are filling the space by splitting each axis into 10 equal sections. Let's call these sections *bins*. We'll end up with 100 bins in the 2D space because it has two axes (10×10), and 1,000 in the 3D space because it has three axes ($10 \times 10 \times 10$). If we count the number of bins occupied by at least one point and divide that number by the total number of bins, we'll get the fraction of occupied bins.

Doing this gives us 0.410 (out of a maximum of 1.0) for the 2D space and 0.048 for the 3D space. This means that 100 samples were able to cover about half of the 2D feature space. Not bad! But 100 samples in the 3D feature space covered only about 5 percent of the space. To fill the 3D space to the same fraction as the 2D space, we need about 1,000 samples—or 10 times as many as we have. This general rule applies as the dimensionality increases: A 4D feature space requires 10,000 samples, while a 10D feature space requires 10,000,000,000!

As the number of features increases, the amount of required training data to get a representative sampling of the possible feature space increases dramatically, approximately as 10^d , where d is the number of dimensions. This is the *curse of dimensionality*, and it was the bane of machine learning for decades. Fortunately for us, modern deep learning has overcome this curse, but it's still relevant when working with traditional models like the ones we'll explore in Chapter 3.

For example, a typical color image on a computer might have 1,024 pixels on a side, with each pixel requiring 3 bytes to specify the color as a mix of red, green, and blue. To use this image as input to a model, we'd need a feature vector with $d = 1,024 \times 1,024 \times 3 = 3,145,728$ elements. This means we'd need some $10^{3,145,728}$ samples to populate our feature space. Clearly, this is not possible. We'll overcome this by using a convolutional neural network (CNN) in Chapter 9.

Now that you know about classes, features, and feature vectors, let's describe what it means to have a good dataset.

Qualities of a Good Dataset

The dataset is everything. This is no exaggeration, since we build the model from the dataset. The model has parameters—whether they're the weights and biases of a neural network, the probabilities of each feature occurring in a naive Bayes model, or the training data itself in the case of nearest neighbors. We use the training data to learn the parameters. The parameters encode the model's knowledge.

Let's back up a little and define the term *dataset* as we'll use it in this book: A dataset is a collection of pairs of values, $\{X, Y\}$, where X is an *input* to the model and Y is a *label*. Here X is a set of values that we've measured and grouped together, like the length and width of flower parts, and Y is the output we want the model to tell us, such as which flower or which animal the data best represents.

For *supervised learning*, we act as the teacher, and the model acts as the student. We're teaching the student by presenting example after example, saying things like “this is a cat” and “this is a dog,” much as we'd teach a small child with a picture book. In this case, the dataset is a collection of examples, and *training* consists of showing the examples to the model repeatedly, until the model “gets it.” We're conditioning and adjusting the parameters of the model to minimize the error made by the model for this particular dataset. This is the learning part of machine learning.

Interpolation and Extrapolation

Interpolation is the process of estimating within a certain known range, and *extrapolation* occurs when we use existing data to estimate outside the known range. Generally speaking, our models are more accurate when they interpolate, which means we need a dataset that's a comprehensive representation of the range of values that we might encounter as inputs to the model.

As an example, let's look at Table 1-3, which shows world population, in billions, from 1910 to 1960. We have data for every 10 years in our known range, 1910 to 1960.

Table 1-3: The World Population by Decade

Year	Population (billions)
1910	1.750
1920	1.860
1930	2.070
1940	2.300
1950	2.557
1960	3.042

If we find the best-fitting line to plot through this data, we can use it as a model to predict values. This *linear regression* allows us to estimate the population for any year we choose. We'll skip the actual fitting process, which you can do simply with online tools, and jump to the model:

$$p = 0.02509y - 46.28$$

For any year, y , we can get an estimate of the population, p . What was the world population in 1952? While we don't have data for 1952 in our table, we can estimate it with the model like so:

$$p = 0.02509(1952) - 46.28 = 2.696 \text{ billion}$$

Checking the actual world population data for 1952 shows that it was 2.637 billion, so our estimate of 2.696 billion is only some 60 million off. The model seems to be pretty good!

In using the model to estimate the world population in 1952, we performed interpolation. We made an estimate for a value that was between our existing data points, and the model gave us a good result.

Extrapolation, on the other hand, is measuring beyond what is known, outside the range of our data.

Let's use our model to estimate the world population in 2000, 40 years after the data we built our model on ends:

$$p = 0.02509(2000) - 46.28 = 3.900 \text{ billion}$$

According to the model, the value should be close to 3.9 billion, but we know from actual data that the world population in 2000 was 6.089 billion. Our model is off by over 2 billion people because we applied the model to input that it isn't suited for. If we remain in the range of inputs that the model is "trained" to know about (namely, dates from 1910 through 1960), the model performs well enough. Once we go beyond the model's training, however, it falls apart because it assumes knowledge we don't possess.

When we interpolate, the model uses examples that are similar to the set of examples it saw during training. Perhaps unsurprisingly, it performs better on these examples than when we extrapolate and ask the model to go beyond its training.

When it comes to classification, we must have comprehensive training data. Let's say we're training a model to identify dog breeds. Our dataset contains hundreds of images of classic black-and-white border collies like the one on the left in Figure 1-2. If we then give the model a new image of a classic border collie, we'll hopefully get back a correct label: "Border Collie." This is akin to asking the model to interpolate; it's working with something it has already seen because the "Border Collie" label in the training data included many examples of classic border collies.



Figure 1-2: A border collie with classic markings (left), a border collie with nonstandard markings (middle), and an Australian shepherd (right)

However, not every border collie has the classic markings, and some might be more similar to the collie in the middle of Figure 1-2. Since we didn't include images like this in the training set, the model must now try to go beyond what we trained it to do and give a correct output label for an instance of a class we trained it on but of a type we did not train it with.

The model will likely fail, giving a false output like "Australian Shepherd," a breed similar to a border collie, shown on the right in Figure 1-2. The key concept to remember is that the dataset must cover the full range of variation *within* the classes the model will see when predicting labels for unknown inputs.

The Parent Distribution

The dataset must be representative of the classes it's modeling. Buried in this idea is the assumption that our data has a *parent distribution*, an unknown data generator that created the particular dataset we're using.

Consider this parallel from philosophy. The ancient Greek philosopher Plato uses the concept of ideals. In his view, an ideal chair exists somewhere "out there," and all existing chairs are imperfect copies of that ideal chair. This is what we mean by the relationship between the dataset we're using, the copy, and the parent distribution, the ideal generator. We want the dataset to be a first-rate representation of the ideal.

Think of a dataset as a sample from an unknown process that produces data according to the parent distribution. The type of data it produces—the values and ranges of the features—follow an unknown, statistical rule. For example, when rolling a die, each of the six values is equally likely in the long run. We call this a *uniform parent distribution*. If we make a bar graph of the number of times each value appears as we roll the die many times, we'll get a (more or less) horizontal line since each value is equally likely to happen. As another example, when measuring the height of adults, the distribution has a form with two humps, one around mean male height and the other around mean female height.

The parent distribution generates this overall shape. The training data, the test data, and the data we give the model to make decisions must all come from the same parent distribution. This is a fundamental assumption models make and shouldn't seem too surprising. Still, sometimes we can easily get mixed up and train with data from one parent distribution while testing or using the model with data from a different parent distribution. (Training with one parent distribution and using that model with data from a different distribution is an active research area at the moment. Search for *domain adaptation*.)

Prior Class Probabilities

The *prior class probability* is the probability with which each class in the dataset appears in the wild. In general, our dataset should match the prior probabilities of the classes. If class A appears 85 percent of the time and class B only 15 percent of the time, then we want class A to appear 85 percent of the time and class B to appear 15 percent of the time in our training set.

Exceptions occur, however. Say one of the classes we want the model to learn is rare, showing up only once for every 10,000 inputs. If we make the dataset strictly follow the actual prior probabilities, the model might not see enough examples of the rare class to learn anything helpful about it. And what if the rare class is the one we're most interested in?

For example, say we're building a robot that locates four-leaf clovers. We'll assume that we already know that the input to the model is a clover; we just want to know whether it has three or four leaves. We know that an estimated 1 in every 5,000 clovers has four leaves. Building a dataset with 5,000 three-leaf clovers for every instance of a four-leaf clover seems reasonable until we realize that a model that simply says every input is a three-leaf clover will be right, on average, 4,999 times out of 5,000! It will be an extremely accurate but completely useless model because it never finds the class we're interested in.

Instead, we might use a 10:1 ratio of three-leaf to four-leaf clovers. Or, when training the model, we might start with an even number of three- and four-leaf clovers, and then, after training for a time, change to a mix that is increasingly closer to the actual prior probability. This trick doesn't work for all model types, but it does work for neural networks. Intuitively, we can imagine the network learning first about the visual difference between a three-leaf and four-leaf clover and then learning something about the actual likelihood of encountering a four-leaf clover as the mix changes to be closer to the actual prior probabilities.

In reality, the trick is used because it often results in better-performing models. For much of machine learning, especially deep learning, empirical tricks and techniques are well in advance of any theory to back them up. "It just works better; that's why" is still a valid, though ultimately unsatisfying, answer to many questions about why a particular approach works well.

The research community is still actively investigating how to work with imbalanced data. Some choose to start with a more balanced ratio of classes; others use data augmentation (see Chapter 2) to boost the number of samples from the underrepresented class.

Confusers

While we need to include examples in our dataset reflecting all the natural variation in the classes we want to learn, at times it's particularly important to include training samples that are similar to one or more classes but aren't examples of that class.

Consider two models. The first learns the difference between images of dogs and images of cats. The second learns the difference between images of dogs and those that are not dogs. The first model has it easy. The input is either a dog or a cat, and the model is trained using images of dogs and images of cats. However, for the second model, what should the “not dog” images be? Given the preceding discussion, we’ll need images covering the space of images the model will see in the wild.

We can take this one step further. If we want to tell the difference between dogs and not dogs, we should include wolves in the “not a dog” class. If we don’t, the model will return a “dog” classification when it encounters a wolf because it hasn’t learned enough to tell the difference. If we build the dataset by using hundreds of “not dog” images of penguins and parrots, should we be surprised if the model decides to call a wolf a dog?

In general, the dataset should include *confusers*, or hard negatives—examples that are similar enough to other classes to be mistaken for them but that don’t belong in the class. Confusers give the model a chance to learn the more precise features of a class. Hard negatives are particularly useful when distinguishing between something and everything else, as in “dog” versus “not dog.”

Dataset Size

We’ve talked about the kind of data to include in a dataset, but how much of it do we need? “All of it” is a temptingly cheeky answer. For our model to be as precise as possible, we should use as many examples as possible. But getting all the data is rarely feasible.

Choosing the size of your dataset means considering a trade-off between accuracy and the time and energy it takes to acquire the data. As in the clover example, sometimes the key class of the dataset is rare or seldom encountered. Because labeled data is generally expensive and slow to acquire, we should have some idea of how much we need before we get started.

Unfortunately, no formula answers the question of how much data is enough. But after a certain point, the benefit of additional data has

diminishing returns. Moving from 100 examples to 1,000 might dramatically boost the model's accuracy, but moving from 1,000 to 10,000 examples might offer only a small increase in accuracy. We must balance the increased accuracy against the effort and expense of acquiring an additional 9,000 training examples.

Another factor to consider is the model itself. Models have a *capacity*, which determines the complexity they can support relative to the amount of training data available. The capacity of a model directly relates to its number of parameters.

A larger model with more parameters requires a lot of training data to find the proper parameter settings. And though having more training examples than model parameters is often a good idea, deep learning can work well when we have less training data than parameters. For example, if the classes are very different from each other (think buildings versus oranges) and we can easily tell the difference, the model will likely also learn the difference quickly, so we can get away with fewer training examples. On the other hand, if we're trying to separate wolves from huskies, we might need a lot more data. We'll discuss what to do when you don't have enough training data in Chapter 2, but none of those tricks are a good substitute for simply getting more data.

The only correct answer to the question of how much data is needed is "all of it." Get as much as is *practical*, given the constraints of the problem: expense, time, rarity, and so forth.

Data Preparation

Before we move on to building actual datasets, let's cover two situations you'll likely encounter before you can feed the dataset to a model: how to scale features and what to do if a feature value is missing.

Scaling Features

A feature vector built from a set of different features might have a variety of ranges. One feature might take on a wide range of values, say, -1,000 to 1,000, while another might be restricted to a range of 0 to 1.

Some models won't work well when this happens, as one feature dominates the others because of its range. Also, some model types are happiest when features have a mean value close to 0.

The solution to these issues is scaling. We'll assume for the time being that every feature in the feature vector is continuous. We'll work with a fake dataset consisting of 5 features and 15 samples. This means that our dataset has 15 samples (feature vectors and their labels), and each of the feature vectors has 5 elements. We'll assume that the dataset has 3 classes. The dataset looks like Table 1-4.

Table 1-4: A Hypothetical Dataset

Sample	x_0	x_1	x_2	x_3	x_4	Label
0	6,998	0.1361	0.3408	0.00007350	78,596,048	0
1	6,580	0.4908	3.0150	0.00004484	38,462,706	1
2	7,563	0.9349	4.3465	0.00001003	6,700,340	2
3	8,355	0.6529	2.1271	0.00002966	51,430,391	0
4	2,393	0.4605	2.7561	0.00003395	27,284,192	0
5	9,498	0.0244	2.7887	0.00008880	78,543,394	2
6	4,030	0.6467	4.8231	0.00000403	19,101,443	2
7	5,275	0.3560	0.0705	0.00000899	96,029,352	0
8	8,094	0.7979	3.9897	0.00006691	7,307,156	1
9	843	0.7892	0.9804	0.00005798	10,179,751	1
10	1,221	0.9564	2.3944	0.00007815	14,241,835	0
11	5,879	0.0329	2.0085	0.00009564	34,243,070	2
12	923	0.4159	1.7821	0.00002467	52,404,615	1
13	5,882	0.0002	1.5362	0.00005066	18,728,752	2
14	1,796	0.7247	2.3190	0.00001332	96,703,562	1

As this is the first dataset covered in the book, we'll discuss it thoroughly to introduce some notation. The first column in Table 1-4 is the *sample number*. The sample is an input—in this case, a collection of five features representing a feature vector. Notice that the numbering starts at 0. As we'll be using Python arrays (NumPy arrays) for data, we'll start counting at 0 in all cases.

The next five columns are the features in each sample, labeled x_0 to x_4 , again starting indices at 0. The final column is the class label. Since we have three classes, the labels run from 0 through 2. Five samples are from class 0, five are from class 1, and five are from class 2. Therefore, this is a small but balanced dataset; the prior probability of each class is 33 percent, which should, ideally, be close to the actual prior probability of the classes appearing in the wild.

If we had a model, each row would be its own input. Writing $\{x_0, x_1, x_2, x_3, x_4\}$ to refer to these is tedious, so instead we'll use an uppercase letter when referring to a full feature vector. For example, we'd refer to sample 2 as X_2 for dataset X . We'll also sometimes use matrices (2D arrays of numbers) labeled with uppercase letters for clarity. To refer to a single feature, we'll use a lowercase letter with subscript—for example, x_3 .

Let's look at the ranges of the features. Table 1-5 shows the minimum, maximum, and range (the difference between the maximum and minimum) of each feature.

Table 1-5: The Minimum, Maximum, and Range of the Features in Table 1-4

Feature	Minimum	Maximum	Range
x_0	843.0	9,498.0	8,655.0
x_1	0.0002	0.9564	0.9562
x_2	0.0705	4.8231	4.7526
x_3	4.03e-06	9.564e-05	9.161e-05

Feature	Minimum	Maximum	Range
x_4	6,700,340.0	96,703,562.0	90,003,222.0

Note the use of computer notation like $9.161\text{e-}05$. This is how computers represent scientific notation: $9.161 \times 10^{-5} = 0.00009161$. Notice, also, that each feature covers a very different range. Because of this, we'll want to scale the features so their ranges are more similar. Scaling is a valid approach prior to training a model as long as you scale all new inputs the same way.

Mean Centering

The simplest form of scaling is *mean centering*. This is easy to do: From each feature, simply subtract the mean (average) value of the feature over the entire dataset. The mean over a set of values, x_i $i = 0, 1, 2, \dots$ is the sum of each value divided by the number of values:

$$\bar{x} = \frac{1}{N} \sum_{i=0}^N x_i$$

The mean value for feature x_0 is 5,022, so to center x_0 , we replace each value like so:

$$x_i \leftarrow x_i - 5,022, \quad i = 0, 1, 2, \dots$$

In this case, the i index is across the samples, not the other elements of the feature vector.

Repeating the preceding steps for the mean value of all the other features will center the entire dataset. The result is that the mean value of each feature over the dataset is now 0, meaning the feature values themselves are all above or below 0. For deep learning, mean centering is often done by subtracting a mean image from each input image.

Changing the Standard Deviation to 1

Mean centering helps, but the distribution of values around 0 remains the same as before we subtracted the mean. All we did was shift the data down toward 0. We call the spread of values around the mean the *standard deviation*, and we compute it as the square root of the average difference of the data values and the mean:

$$\sigma = \sqrt{\frac{\sum_{i=0}^N (x_i - \bar{x})^2}{N-1}}$$

The letter σ (sigma) is the usual name for the standard deviation in mathematics. You don't need to memorize this formula. It's there to show you how to calculate a measure of the spread, or range, of the data relative to the mean value of the data.

Mean centering changes $\bar{x}$ to 0, but it doesn't change σ . Sometimes we want to go further and, along with mean centering, change the spread of the data so that the ranges are the same, meaning the standard deviation for each feature is 1. Fortunately, doing this is straightforward. We replace each feature value, x , with the following, where $\bar{x}$ and σ are the mean and standard deviation, respectively, of that feature across the dataset:

$$x \leftarrow \frac{x - \bar{x}}{\sigma}$$

For example, we can store the preceding toy dataset as a 2D NumPy array:

```
x = np.array([
    [6998, 0.1361, 0.3408, 0.00007350, 78596048],
    [6580, 0.4908, 3.0150, 0.00004484, 38462706],
    [7563, 0.9349, 4.3465, 0.00001003, 6700340],
    [8355, 0.6529, 2.1271, 0.00002966, 51430391],
    [2393, 0.4605, 2.7561, 0.00003395, 27284192],
    [9498, 0.0244, 2.7887, 0.00008880, 78543394],
    [4030, 0.6467, 4.8231, 0.00000403, 19101443],
    [5275, 0.3560, 0.0705, 0.00000899, 96029352],
    [8094, 0.7979, 3.9897, 0.00006691, 7307156],
    [ 843, 0.7892, 0.9804, 0.00005798, 10179751],
```

```
[1221, 0.9564, 2.3944, 0.00007815, 14241835],  
[5879, 0.0329, 2.0085, 0.00009564, 34243070],  
[ 923, 0.4159, 1.7821, 0.00002467, 52404615],  
[5882, 0.0002, 1.5362, 0.00005066, 18728752],  
[1796, 0.7247, 2.3190, 0.00001332, 96703562],  
])
```

This allows us to process the entire dataset in one line of code:

```
x = (x - x.mean(axis=0)) / x.std(axis=0)
```

We call this approach *standardization*, or *normalizing*, and you should do it to most datasets, especially when using one of the traditional models we'll discuss in Chapter 3. Whenever possible, standardize your dataset so that the features have a mean of 0 and a standard deviation of 1.

If we standardize the preceding dataset, what will it look like? Subtracting, per feature, the mean value of that feature and dividing by the standard deviation gives us a new dataset (Table 1-6). Here we've shortened the numbers to four decimal digits for display and have dropped the label.

Table 1-6: The Data in Table 1-4 Standardized

Sample	x_0	x_1	x_2	x_3	x_4
0	0.6930	-1.1259	-1.5318	0.9525	1.1824
1	0.5464	-0.0120	0.5051	-0.0192	-0.1141
2	0.8912	1.3826	1.5193	-1.1996	-1.1403
3	1.1690	0.4970	-0.1712	-0.5340	0.3047
4	-0.9221	-0.1071	0.3079	-0.3885	-0.4753
5	1.5699	-1.4767	0.3327	1.4714	1.1807
6	-0.3479	0.4775	1.8823	-1.4031	-0.7396
7	0.0887	-0.4353	-1.7377	-1.2349	1.7456

Sample	x_0	x_1	x_2	x_3	x_4
8	1.0775	0.9524	1.2475	0.7291	-1.1207
9	-1.4657	0.9250	-1.0446	0.4262	-1.0279
10	-1.3332	1.4501	0.0323	1.1102	-0.8966
11	0.3005	-1.4500	-0.2615	1.7033	-0.2505
12	-1.4377	-0.2472	-0.4340	-0.7032	0.3362
13	0.3016	-1.5527	-0.6213	0.1780	-0.7517
14	-1.1315	0.7225	-0.0250	-1.0881	1.7674

If you compare the two tables, you'll see that the features are more similar after our manipulations than in the original set. Looking at x_3 , the mean of the values is $-1.33\text{e-}16 = -1.33 \times 10^{-16} = -0.000000000000000133$, which is virtually 0. Good! This is what we want. If you perform the calculations, you'll find that the means of the other features are similarly close to 0. What about the standard deviation? For x_3 it's 0.99999999, which is virtually 1; again, this is what we'd like. We'll use this new, transformed dataset to train the model.

Therefore, we must apply the per-feature means and standard deviations, as measured on the training set, to any new inputs we're giving to the model:

$$x_{\text{new}} \leftarrow \frac{x_{\text{new}} - \bar{x}_{\text{train}}}{\sigma_{\text{train}}}$$

Here x_{new} is the new feature vector we want to apply to the model, and $\bar{x}_{\text{train}}$ and σ_{train} are the mean and standard deviation, per feature, from the training set.

Dealing with Missing Features

Sometimes we don't have all the features we need for a sample; for example, we might have forgotten to make a measurement. These are

missing features, and we need to find a way to correct them, since most models don't have the ability to accept missing data.

One solution is to fill in the missing values with those outside the feature's range, in the hopes that the model learns to ignore those values or make more use of other features. Indeed, some more advanced deep learning models intentionally zero some of the input as a form of regularization (you'll see what that means in later chapters).

For now, you'll learn the second most obvious solution: replacing missing features with the mean value of features over the dataset. Let's look again at our practice dataset from earlier. This time, we'll have missing data to deal with (Table 1-7).

Table 1-7: The Sample Dataset (Table 1-4) with Some Holes

Sample	x_0	x_1	x_2	x_3	x_4	Label
0	6,998	0.1361	0.3408	0.00007350	78,596,048	0
1	...	0.4908	...	0.00004484	38,462,706	1
2	7,563	0.9349	4.3465	...	6,700,340	2
3	8,355	0.6529	2.1271	0.00002966	51,430,391	0
4	2,393	0.4605	2.7561	0.00003395	27,284,192	0
5	9,498	...	2.7887	0.00008880	78,543,394	2
6	4,030	0.6467	4.8231	0.00000403	...	2
7	5,275	0.3560	0.0705	0.00000899	96,029,352	0
8	8,094	0.7979	3.9897	0.00006691	7,307,156	1
9	...	...	0.9804	...	10179751	1
10	1,221	0.9564	2.3944	0.00007815	14,241,835	0
11	5,879	0.0329	2.0085	0.00009564	34,243,070	2
12	923	...	...	0.00002467	...	1
13	5,882	0.0002	1.5362	0.00005066	18,728,752	2

Sample	x_0	x_1	x_2	x_3	x_4	Label
14	1,796	0.7247	2.3190	0.00001332	96,703,562	1

The ellipses indicate missing values. Table 1-8 shows the means of each feature, ignoring missing values.

Table 1-8: The Means for Features in Table 1-7

x_0	x_1	x_2	x_3	x_4
5,223.6	0.5158	2.345	4.71e-05	42,957,735.0

If we replace each missing value with the mean, we'll get a dataset we can standardize and use to train a model.

Of course, real data is better, but the mean is the simplest substitute we can reasonably use. If the dataset is large enough, we might instead generate a histogram of each feature's values and select the mode (the most common value). However, using the mean should work out just fine, especially if your dataset has a lot of samples and the number of missing features is fairly small.

Training, Validation, and Test Data

Now that we have a dataset (a collection of feature vectors), we're ready to start training a model, right? Well, actually, no. That's because we don't want to use the entire dataset for training. We should use some of the data for other purposes, so we'll need to split it into at least two subsets, although ideally we'd have three. We call these subsets the training data, test data, and validation data.

The Three Subsets

The *training data* is the subset we use to train the model. The important point here is selecting feature vectors that well represent the parent distribution of the data.

The *test data* is the subset we use to evaluate how well the trained model is doing. We *never* use the test data when training the model; that would be cheating, because we'd be testing the model on data it's seen before. Put the test dataset aside until the model is complete, and then use it to evaluate the model.

The third dataset is the *validation data*. Not every model needs a validation dataset, but it's helpful for deep learning models. We use the validation dataset during training as though it's test data to get an idea of how well the training is working. Validation data can help us make decisions about when to stop training or whether we're using the proper model.

For example, a neural network has a certain number of layers, each with a number of nodes. We call this the *architecture* of the model. During training, we can test the performance of the neural network with the validation data to figure out whether we should continue training or stop and try a different architecture. We don't train the model with the validation set, and we don't use the validation set to modify model parameters. We also can't use validation data when reporting actual model performance, since we used results based on the validation data to select the model in the first place. This would make it seem like the model is doing better than it is.

Figure 1-3 illustrates the three subsets and their relationships to one another. On the left is the whole dataset. This is the entire collection of feature vectors and associated labels. On the right are the three subsets. The training data and the validation data work together to train and develop the model, while the test data is held back until the model is ready for it. The size of the cylinders reflects the relative amount of data that should fall into each subset, though in practice the validation and test subsets might be even smaller.

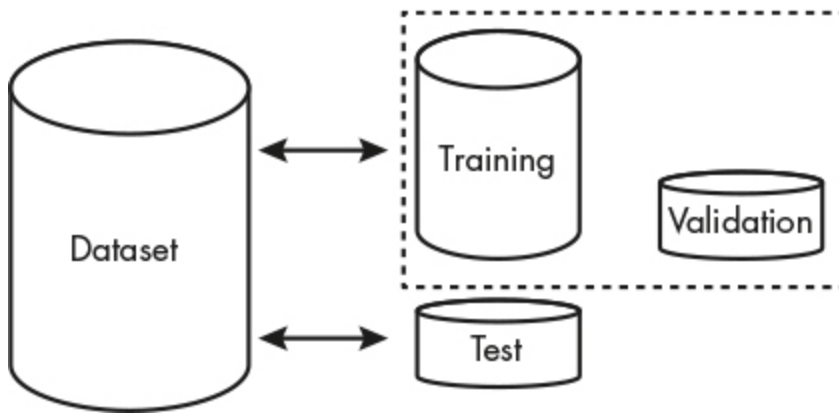


Figure 1-3: The relationships among training, validation, and test subsets

To recap: Use the training and validation sets to build the model and use the test set to evaluate it.

Dataset Partitioning

How much data should go into each dataset? A typical split is 90 percent for training, 5 percent for validation, and 5 percent for testing. For deep learning models, this is fairly standard.

If we're working with a very large dataset, we could go as low as 1 percent each for validation and testing. For classic models, which might not learn as well, we might want to make the test dataset larger to ensure we're able to generalize to a wide variety of possible inputs. In those cases, we might try 80 percent for training and 10 percent each for validation and test. If we aren't using validation data, the full 20 percent might go to testing. Larger test sets can be appropriate for multiclass models that have classes with low prior probabilities. Or we might increase the number of rare classes in the test set since we won't use it to define the model. This might be especially important should missing the rare class be a costly event (for example, missing a tumor in a medical image).

Now that we've determined how much data to put into each set, let's use scikit-learn (sklearn) to generate a dummy dataset that we can partition:

```
>>> import numpy as np
>>> from sklearn.datasets import make_classification
```

```
>>> x, y = make_classification(n_samples=10000, weights=(0.9, 0.1))
>>> x.shape
(10000, 20)
>>> len(np.where(y == 0)[0])
8969
>>> len(np.where(y == 1)[0])
1031
```

Here we're using 2 classes and 20 features to generate 10,000 samples. The dataset is imbalanced, with 90 percent of the samples in class 0 and 10 percent in class 1. The output is a 2D array of samples (x) and associated 0 or 1 labels (y). The dataset is generated from multidimensional Gaussians that are the analogs of the normal bell curve in more than one dimension, but that doesn't matter to us right now. The useful part for us is the collection of feature vectors and labels, so we can look at ways to split the dataset into subsets.

The key to the preceding code is the call to `make_classification`, which accepts the number of samples requested and the fraction for each class. The `np.where` calls find all the class 0 and class 1 instances so that `len` can count them.

Earlier, we talked about the importance of preserving (or at least approaching) the actual prior probabilities of the classes in our dataset. If one class makes up 10 percent of real-world cases, it should ideally make up 10 percent of our dataset. Now we need to find a way to preserve this prior class probability in the subsets for training, validation, and test. We can do this in two main ways: partitioning by class and random sampling.

Partitioning by Class

The exact approach of partitioning by class, which is suitable when the dataset is small or one class is rare, is to determine the number of samples representing each class and then set aside selected percentages of each, by class, before merging them. So, if we have 9,000 samples from class 0 and 1,000 samples from class 1, and we want to put 90 percent of the data into training and 5 percent each into validation and test, we would select 8,100 samples, at random, from the class 0

collection and 900 samples, at random, from the class 1 collection to make up the training set. Similarly, we'd randomly select 450 of the remaining 900 unused class 0 samples for the validation set along with 50 of the remaining unused class 1 data. The remaining class 0 and class 1 samples become the test set.

Listing 1-1 shows the code to construct the subsets using a 90/5/5 split of the original data.

```
import numpy as np
from sklearn.datasets import make_classification

❶ a, b = make_classification(n_samples=10000, weights=(0.9, 0.1))
   idx = np.where(b == 0)[0]
   x0 = a[idx,:]
   y0 = b[idx]
   idx = np.where(b == 1)[0]
   x1 = a[idx,:]
   y1 = b[idx]

❷ idx = np.argsort(np.random.random(y0.shape))
   y0 = y0[idx]
   x0 = x0[idx]
   idx = np.argsort(np.random.random(y1.shape))
   y1 = y1[idx]
   x1 = x1[idx]

❸ ntrn0 = int(0.9*x0.shape[0])
   ntrn1 = int(0.9*x1.shape[0])

   xtrn = np.zeros((int(ntrn0+ntrn1), 20))
   ytrn = np.zeros(int(ntrn0+ntrn1))
   xtrn[:ntrn0] = x0[:ntrn0]
   xtrn[ntrn0:] = x1[:ntrn1]
   ytrn[:ntrn0] = y0[:ntrn0]
   ytrn[ntrn0:] = y1[:ntrn1]

❹ n0 = int(x0.shape[0]-ntrn0)
```

```

n1 = int(x1.shape[0]-ntrn1)
xval = np.zeros((int(n0/2+n1/2), 20))
yval = np.zeros(int(n0/2+n1/2))
xval[: (n0//2)] = x0[ntrn0:(ntrn0+n0//2)]
xval[(n0//2):] = x1[ntrn1:(ntrn1+n1//2)]
yval[: (n0//2)] = y0[ntrn0:(ntrn0+n0//2)]
yval[(n0//2):] = y1[ntrn1:(ntrn1+n1//2)]

❸ xtst = np.concatenate((x0[(ntrn0+n0//2):], x1[(ntrn1+n1//2):]))
ytst = np.concatenate((y0[(ntrn0+n0//2):], y1[(ntrn1+n1//2):]))

```

Listing 1-1: Exact construction of training, validation, and test datasets

This code includes a lot of bookkeeping. First, we create the dummy dataset ❶ and split it into class 0 and class 1 collections, stored in `x0`, `y0` and `x1`, `y1`, respectively. We then randomize the ordering ❷. This lets us pull off the first n samples for the subsets without worrying that we might be introducing a bias because of ordering in the data. Because of the way sklearn generates the dummy dataset, this step isn't required, but it's always a good idea to ensure randomness in the ordering of samples.

Because we store the feature vectors in one array and the labels in another, the NumPy shuffle methods won't work to reorder the samples. Instead, we generate a random vector of the same length as our number of samples and then use `argsort` to return the indices of the vector that would put it in sorted order. Since the values in the vector are random, the ordering of the indices used to sort it will also be random. These indices then reorder the samples and labels so that each label is still associated with the correct feature vector.

Next, we extract the first 90 percent of samples for the two classes and build the training subset with samples in `xtrn` and labels in `ytrn` ❸. We do the same for the 5 percent validation set ❹ and the remaining 5 percent for the test set ❺.

Partitioning by class is tedious, to say the least. We do know, however, that the ratio of class 0 to class 1 in each of the subsets is exactly the same.

Random Sampling

In general, we don't need to be so precise to require partitioning by class. The second common method for partitioning the full dataset is via random sampling. If we have enough data, we can build our subsets by randomizing the full dataset and then extracting the first 90 percent of samples as the training set, the next 5 percent as the validation set, and the last 5 percent as the test set. Listing 1-2 shows this method.

```
❶ x, y = make_classification(n_samples=10000, weights=(0.9, 0.1))
   idx = np.argsort(np.random.random(y.shape[0]))
   x = x[idx]
   y = y[idx]

❷ ntrn = int(0.9*y.shape[0])
   nval = int(0.05*y.shape[0])

❸ xtrn = x[:ntrn]
   ytrn = y[:ntrn]
   xval = x[ntrn:(ntrn+nval)]
   yval = y[ntrn:(ntrn+nval)]
   xtst = x[(ntrn+nval):]
   ytst = y[(ntrn+nval):]
```

Listing 1-2: Random sampling of training, validation, and test datasets

We randomize the dummy dataset stored in `x` and `y` ❶. We need to know how many samples to include in each subset. The number of samples for the training set is 90 percent of the total in the dataset ❷, while the number in the validation set is 5 percent of the total. The remainder, also 5 percent, is the test set ❸.

This method is much simpler than the one in Listing 1-1. However, the possible downside is that the mix of classes in each subset might not be quite the fractions we want. For example, imagine we want a training set of 9,000 samples, or 90 percent of the original 10,000 samples, with 8,100 of them from class 0 and 900 of them from class 1. Running the

Listing 1-2 code 10 times gives the splits between class 0 and class 1 in the training set shown in Table 1-9.

Table 1-9: Ten Training Splits Generated by Random Sampling

Run	Class 0	Class 1
1	8,058 (89.5)	942 (10.5)
2	8,093 (89.9)	907 (10.1)
3	8,065 (89.6)	935 (10.4)
4	8,081 (89.8)	919 (10.2)
5	8,045 (89.4)	955 (10.6)
6	8,045 (89.4)	955 (10.6)
7	8,066 (89.6)	934 (10.4)
8	8,064 (89.6)	936 (10.4)
9	8,071 (89.7)	929 (10.3)
10	8,063 (89.6)	937 (10.4)

The number of samples in class 1 ranges from as few as 907 samples to as many as 955. As the number of samples of a particular class in the full dataset decreases, the number in the subsets starts to vary more. This is especially true of smaller subsets, like the validation and test sets. Table 1-10 shows a separate run, this time looking at the number of samples from each class in the test set.

Table 1-10: Ten Test Splits Generated by Random Sampling

Run	Class 0	Class 1
1	446 (89.2)	54 (10.8)
2	450 (90.0)	50 (10.0)

Run	Class 0	Class 1
3	444 (88.8)	56 (11.2)
4	450 (90.0)	50 (10.0)
5	451 (90.2)	49 (9.8)
6	462 (92.4)	38 (7.6)
7	441 (88.2)	59 (11.8)
8	449 (89.8)	51 (10.2)
9	449 (89.8)	51 (10.2)
10	438 (87.6)	62 (12.4)

In the test set, the number of samples from class 1 ranges from 38 to 62.

Will these differences influence how well the model learns? Probably not, but they might make the test results look better than they are, as most models struggle to identify the classes that are least common in the training set. A pathological split could result in having no examples from a particular class, but in practice, that result isn't likely unless your pseudorandom number generator is particularly poor. Still, keeping the possibility in mind is worthwhile. If concerned, use the exact split approach in Listing 1-1. The better solution is, as always, to get more data.

Algorithmically, the steps to produce the training, validation, and test splits are as follows:

1. Randomize the order of the full dataset so that classes are evenly mixed.
2. Calculate the number of samples in the training (n_{trn}) and validation (n_{val}) sets by multiplying the number of samples in the full dataset by the desired fraction. The remaining samples will fall into the test set.
3. Assign the first n_{trn} samples to the training set.

4. Assign the next n_{val} samples to the validation set.
5. Finally, assign the remaining samples to the test set.

At all times, ensure that the order of the samples is truly random, and reorder the labels in the exact same sequence as the feature vectors. If this is done, this simple splitting process will give a good split unless the dataset is tiny or some classes are rare.

We neglected to discuss one consequence of this approach: If the full dataset is small to begin with, partitioning it makes the training set even smaller. Chapter 2 presents a powerful approach to dealing with a small dataset that's used heavily in deep learning. But first, let's look at a principled way to work with a small dataset to get an idea of how well it will perform on new data.

k-Fold Cross-Validation

Whereas modern deep learning models typically need very large datasets (meaning we're able to use a single training/validation/test split), more traditional machine learning models, like those in Chapter 3, often work with datasets that are generally too small for deep learning models. If we use a single training/validation/test split on those datasets, we might be holding back too much data for testing, or else have too few samples in the test set to get a meaningful measurement of how well the model is working.

One way to address this issue is to use *k-fold cross-validation*, a technique ensuring that the model uses each sample in the dataset at some point for training and testing. Use this technique for small datasets intended for traditional machine learning models. It can also be helpful as a way to decide between different models.

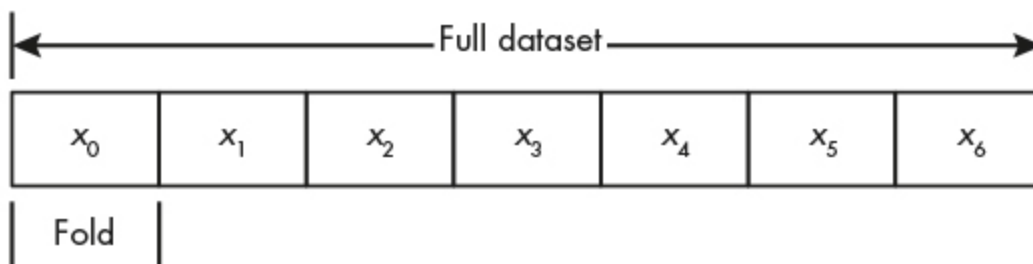
To perform *k-fold cross-validation*, first partition the full, randomized dataset into k non-overlapping groups, $x_0, x_1, x_2, \dots, x_{k-1}$. Your k value is arbitrary, though it typically ranges from 5 to 10. Figure 1-4(a) shows this split, imagining the entire dataset laid out horizontally.

We can train a model by holding x_0 back as test data and using the other groups, $x_1, x_2, \dots, x_{k-1}$ as training data. We'll ignore validation data

for the time being; after building the current training data, we might choose to hold back some of it as validation data. Call this trained model m_0 . Next, we start over from scratch, this time holding back x_1 as test data and then training with all the other groups, including x_0 . We'll get a new trained model, m_1 . By design, m_0 and m_1 are the same type of model. We're interested in multiple instances of the same type of model trained with different subsets of the full dataset.

Repeat this process for each of the groups, as in Figure 1-4(b), and we'll have k models, each trained with $(k - 1)/k$ of the data, while holding back $1/k$ of the data for testing. What k should be depends on how much data is in the full dataset. A larger k means more training data but less test data. If the per-model training time is low, tend toward a larger k , as this increases the per-model training-set size.

(a)



(b)

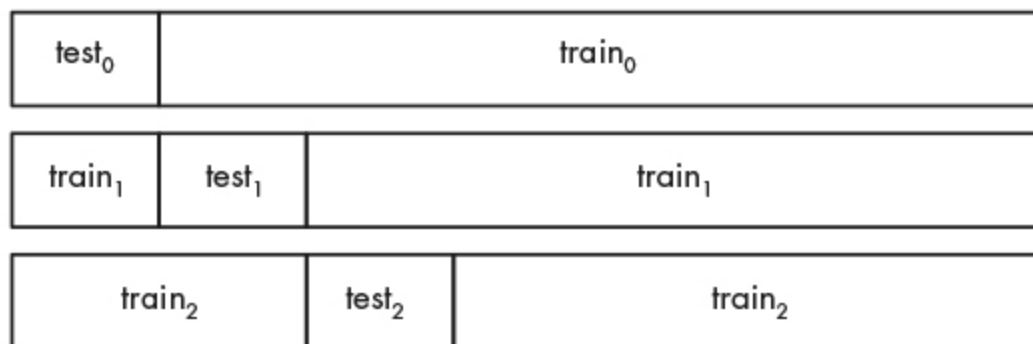


Figure 1-4: k -fold cross-validation. Partitioning the dataset (a). The first three train/test splits (b).

After we've trained the k models, we evaluate them individually and average their metrics to get an idea of how a model trained on the full dataset would behave. In Chapter 8, you'll learn ways to evaluate a

model. If using k -fold cross-validation to select from two or more models (say, between using k -NN or a support vector machine, two examples of classical machine learning models that you'll learn about later in the book), repeat the full training and evaluation process for each type of model and compare their results.

Once we have an idea of how well the model is performing on the averaged evaluation metrics, we can start over again and train the selected model type by using *all* of the dataset for training. This is the advantage of k -fold cross-validation: We can have our cake and eat it too.

Data Analysis

It's quite easy to assemble features and feature vectors and then put the training, validation, and test sets together without pausing to *look* at the data and see if it makes sense. This is especially true with deep learning models that use huge collections of images or other multidimensional data. Here are a few problems we should look out for:

Mislabeled data Assume we're building a large dataset with hundreds of thousands of labeled samples. Further, assume that we're going to use the dataset to build a model that can tell the difference between dogs and cats. Naturally, we need to feed the model many dog images and many cat images. No problem; we'll collect images using something like Google Images. But if we simply set up a script to download image search results matching "dog" and "cat," we'll also get a lot of other images that are not of dogs or cats, or images that contain dogs and cats along with other things. The labels won't be perfect. While deep learning models can be resistant to such label noise, we should avoid it whenever possible.

Missing or outlier data Imagine we have a collection of feature vectors, and we have no idea how common missing features are. If a large percentage of a particular feature is missing, that feature becomes a hindrance to the model, and we should eliminate it. Or

if the data includes extreme outliers, we might remove those samples, especially if we're going to standardize, since outliers strongly affect the mean subtracted from the feature values.

How to Find Problems in the Data

How can we look for these problems in the data? For feature vectors, we can load the dataset into a spreadsheet if the dataset isn't too large; write a Python script to summarize the data, feature by feature; or bring the data into a statistics program and examine it that way.

Typically, when summarizing values statistically, we look at the mean and standard deviation, as well as the largest value and the smallest value. We can also look at the median, which we get by sorting the values from smallest to largest and picking the one in the middle. (If the number of values is even, average the two middle values.) Let's look at one of the features from our earlier example. After sorting the values from smallest to largest, we summarize the data in the following way:

x_2			
0.0705			
0.3408			
0.9804			
1.5362			
1.7821	Mean ($\bar{x}$)	=	2.3519
2.0085	Standard deviation (σ)	=	1.3128
2.1271	Standard error (SE)	=	0.3390
2.3190	Median	=	2.3190
2.3944	Minimum	=	0.0705
2.7561	Maximum	=	4.8231
2.7887			
3.0150			

x_2
3.9897
4.3465
4.8231

We've already explored the concepts of mean, minimum, maximum, and standard deviation. I've highlighted the median in the list of features on the left. Notice that after sorting, the median appears in the exact middle of the list. We often refer to the median as the 50th percentile because the same amount of data is above it as below.

A new value is also listed, the *standard error*, also called the *standard error of the mean*. This is the standard deviation divided by the square root of the number of values in the dataset:

$$SE = \frac{\sigma}{\sqrt{n}}$$

The standard error is a measure of the difference between our mean value, $\bar{x}$, and the mean value of the parent distribution. If we have more measurements, we'll have a better idea of the parent distribution that's generating the data, and so the mean value of the measurements will be closer to the mean value of the parent distribution.

Notice also that the mean and the median are relatively close to each other. The phrase *relatively close* has no rigorous mathematical meaning, but we can use it as an ad hoc indicator that the data might be normally distributed, meaning we could reasonably replace the missing values by the mean (or median), as you saw previously.

Listing 1-3 shows the preceding values computed using NumPy.

feature_stats.py

```
import numpy as np
```

```
❶ f = [0.3408, 3.0150, 4.3465, 2.1271, 2.7561,
        2.7887, 4.8231, 0.0705, 3.9897, 0.9804,
        2.3944, 2.0085, 1.7821, 1.5362, 2.3190]
```

```

f = np.array(f)

print("mean    = %0.4f" % f.mean())
print("std     = %0.4f" % f.std())
❷ print("SE      = %0.4f" % (f.std()/np.sqrt(f.shape[0])))
print("median = %0.4f" % np.median(f))
print("min     = %0.4f" % f.min())
print("max     = %0.4f" % f.max())

```

Listing 1-3: Calculating basic statistics

After loading NumPy, we manually define the x_2 features (`f`) and turn them into a NumPy array ❶. Calculating the desired values is then straightforward, as all of them, except the standard error, are simple method or function calls. We calculate the standard error via the preceding formula ❷, where the first element of the tuple that NumPy returns for the shape is the number of elements in a vector.

Numbers are nice, but pictures are often better. Let's view the standardized values of our dataset by generating a box plot in Python. Then we'll discuss what the plot is showing us. The code to create the plot is in Listing 1-4.

box_plot.py

```

import numpy as np
import matplotlib.pyplot as plt

❶ d = [[ 0.6930, -1.1259, -1.5318,  0.9525,  1.1824],
        [ 0.5464, -0.0120,  0.5051, -0.0192, -0.1141],
        [ 0.8912,  1.3826,  1.5193, -1.1996, -1.1403],

        [ 1.1690,  0.4970, -0.1712, -0.5340,  0.3047],
        [-0.9221, -0.1071,  0.3079, -0.3885, -0.4753],
        [ 1.5699, -1.4767,  0.3327,  1.4714,  1.1807],
        [-0.3479,  0.4775,  1.8823, -1.4031, -0.7396],
        [ 0.0887, -0.4353, -1.7377, -1.2349,  1.7456],
        [ 1.0775,  0.9524,  1.2475,  0.7291, -1.1207],
        [-1.4657,  0.9250, -1.0446,  0.4262, -1.0279],
        [-1.3332,  1.4501,  0.0323,  1.1102, -0.8966],

```



```
[ 0.3005, -1.4500, -0.2615,  1.7033, -0.2505],  
[-1.4377, -0.2472, -0.4340, -0.7032,  0.3362],  
[ 0.3016, -1.5527, -0.6213,  0.1780, -0.7517],  
[-1.1315,  0.7225, -0.0250, -1.0881,  1.7674]]
```

```
❷ d = np.array(d)  
plt.boxplot(d)  
plt.show()
```

Listing 1-4: A box plot of the standardized toy dataset

The values are in Table 1-6. We can store the data as a 2D array and make the box plot by using Listing 1-4. We manually define the array ❶ and then plot it ❷. The plot is interactive, so experiment with the environment provided until you feel comfortable. The old-school floppy disk icon stores the plot to your disk.

Figure 1-5 shows the box plot the program generates.

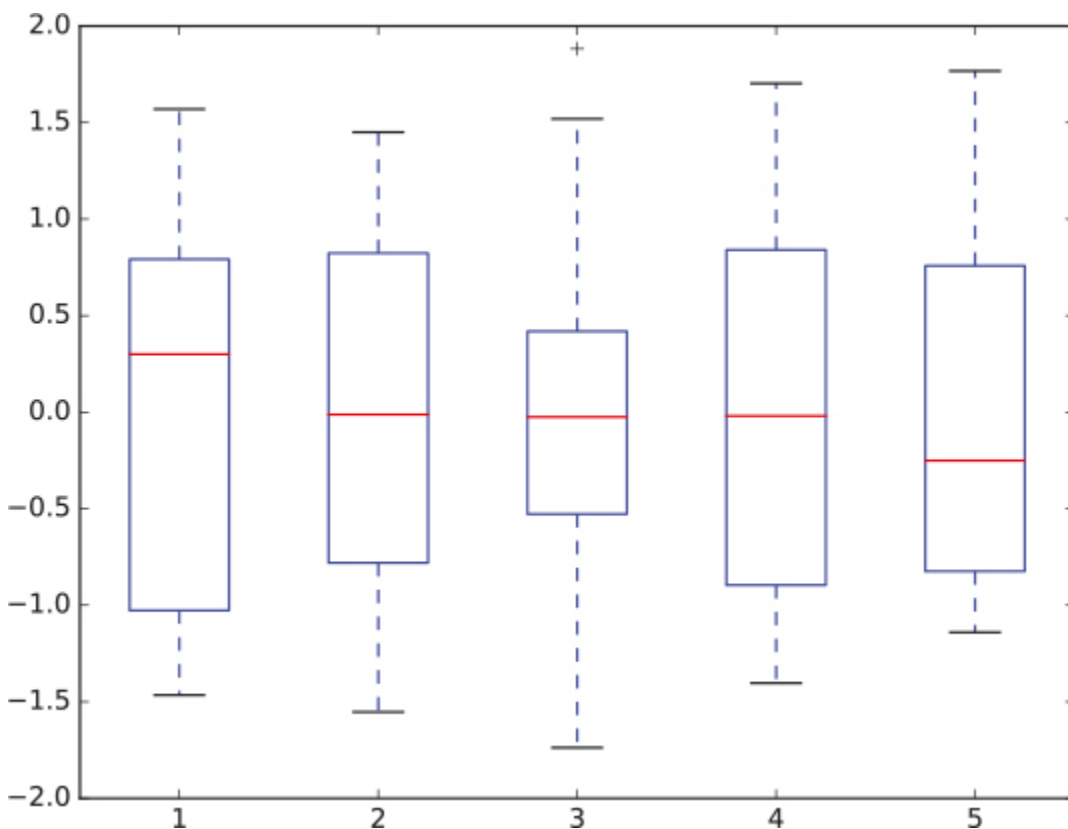


Figure 1-5: The box plot produced by Listing 1-4

You'll learn how to interpret the box plot by examining the box representing the standardized feature x_2 , which Figure 1-6 shows.

The lower box line, Q1, marks the end of the first quartile. This means that 25 percent of the data values for a feature are less than this value. The median, Q2, is the 50 percent mark and the end of the second quartile.

Half the data values are less than this value. The upper box line, Q3, is the 75 percent mark. The remaining 25 percent of the data values are above Q3.

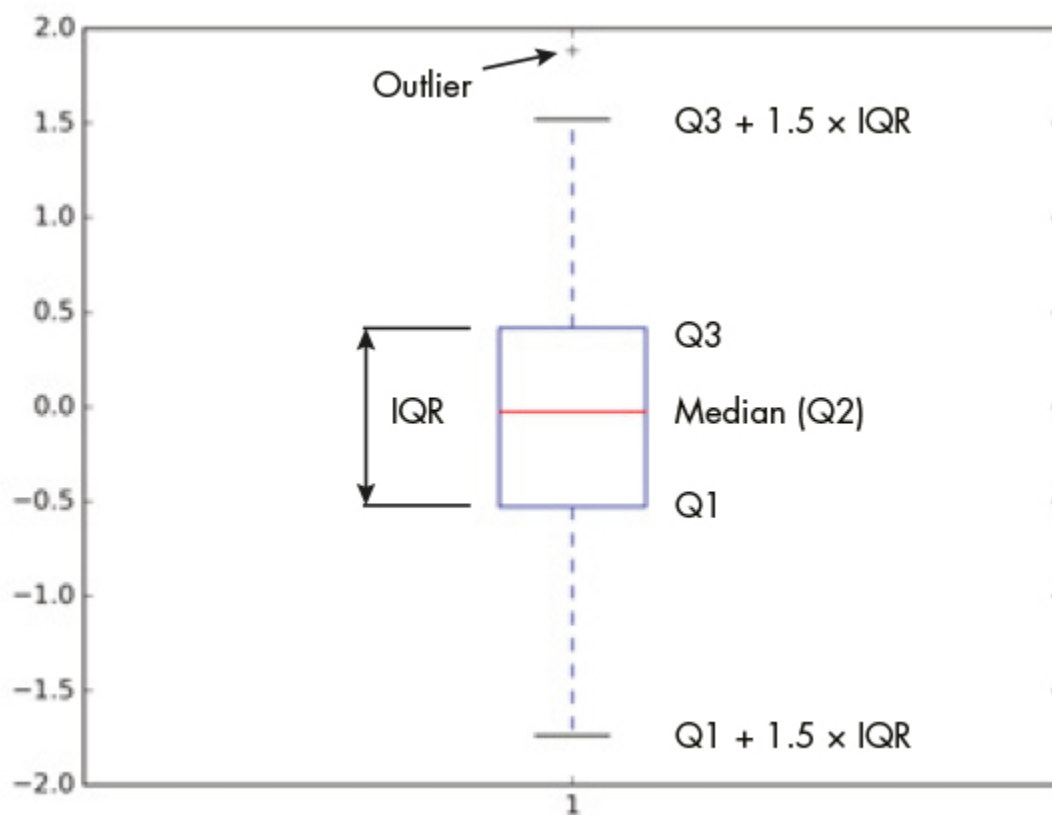


Figure 1-6: The standardized feature x_2 from the dataset

The two lines above and below the box are the *whiskers*. (Matplotlib calls them *fliers*, but this is an unconventional term.) The whiskers are the values at $Q1 - 1.5 \times IQR$ (interquartile range) and $Q3 + 1.5 \times IQR$. By convention, we consider values outside this range *outliers*.

Looking at outliers can help us realize they're mistakes in data entry and drop them from the dataset. However, be prepared to justify your

actions should you ever plan on publishing or otherwise presenting results based on the dataset. Similarly, you might be able to drop samples with missing values, but make sure there's no systematic error causing the missing data, and check that you're not introducing bias into the data by dropping those samples. In the end, common sense should override slavish adherence to convention.

Cautionary Tales

At the risk of being repetitive, *look at your data*. The more you work with it, the more you will understand it, and the more effectively you'll be able to make reasonable decisions about what goes in and what comes out, and *why*. Recall that the goal of the dataset is to faithfully represent the parent distribution, or what the data looks like in the wild when using the model.

Let's discuss two quick anecdotes that illustrate ways models may learn what we didn't intend or even consider.

The first was told to me as an undergraduate in the 1980s. In this story, an early form of neural network was tasked with detecting tank and non-tank images. The neural network seemed to work well in testing, but when used in the field, the detection rate dropped rapidly. The researchers realized that the tank images were taken on a cloudy day, and the non-tank ones were taken on a sunny day. The recognition system had not learned the difference between tanks and non-tanks at all; instead, it had learned the difference between cloudy and sunny days. The moral of this story is that the training set needs to include *all* the conditions the model will see in the wild.

I heard the second anecdote in a talk at the Neural Information Processing Systems (NIPS) 2016 conference in Barcelona, Spain, and later found it repeated in the researchers' paper ("Why Should I Trust You?": Explaining the Predictions of Any Classifier" by Marco Tulio Ribeiro et al., 2016). To demonstrate their technique for getting a model to explain its decisions, the authors had trained a model that claimed to tell the difference between images of huskies and images of wolves. The model appeared to work rather well, and during the talk, the authors polled the audience composed of machine learning

researchers about the believability of the model. Most thought it was a good model. Then, using their technique, the speakers revealed that the network had not learned much, if anything, about the difference between huskies and wolves. Instead, it had learned that the wolf pictures had snow in the background and the husky pictures did not.

Think about your data and be on the lookout for unintended consequences. Models are not human. We bring a lot of preconceived notions and unintended biases to the dataset.

Summary

In this chapter, we described the components of a dataset (classes, labels, features, and feature vectors) and then characterized a good dataset, emphasizing the importance of ensuring that the dataset well represents the parent distribution. We then described basic data-preparation techniques including how to scale data and one approach for dealing with missing features. After that, we learned how to separate the full dataset into training, validation, and test subsets and how to apply k -fold cross-validation, which is especially useful with small datasets. We ended the chapter with tips on examining data to confirm it makes sense.

In the next chapter, we'll directly apply the content in this chapter to construct the datasets we'll use throughout the remainder of this book.

2

BUILDING THE DATASETS



In this chapter, we'll put Chapter 1's advice into practice to build the datasets we'll use throughout the remainder of the book. Some of these datasets are well suited to traditional models because they consist of feature vectors. Others are better for deep learning models that work with multidimensional inputs—in particular, images, or things that we can visualize as images.

We'll work through acquiring raw data and preprocessing it to make it suitable for our tools. We won't make actual training/validation/test splits until we use these datasets for specific models. Preprocessing data is often one of the most labor-intensive of machine learning tasks. If it's not done, or not done well, your model may end up being far less useful than you want it to be.

Irises

Perhaps the most classic of all machine learning datasets is the iris flower dataset, developed in 1936 by R.A. Fisher and described in his paper, “The Use of Multiple Measurements in Taxonomic Problems.” It's a small dataset of 3 classes that each have 50 samples. The dataset

includes 4 features: sepal width, sepal length, petal width, and petal length, all in centimeters. The three classes are *I. setosa*, *I. versicolor*, and *I. virginica*.

This dataset is built into sklearn, but we'll download it from the University of California, Irvine, Machine Learning Repository to practice working with externally sourced data and introduce a rich collection of datasets suitable for many traditional machine learning models. The main repository is located at <https://archive.ics.uci.edu/ml/index.php>, but you can download the iris dataset directly from <https://archive.ics.uci.edu/dataset/53/iris>.

To download the dataset, click **Download**, extract the ZIP file contents, and save the *iris.data* file, ideally to a new directory called *iris*. Let's take a look at the start of this file:

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
```

Because the class names at the end of each line are all the same, we immediately suspect that the samples are sorted by class. The rest of the file confirms this. As emphasized in Chapter 1, we must randomize the data before training a model. We also need to replace the class names with integer labels. We can load the dataset into Python with the script in Listing 2-1.

```
import numpy as np

❶ with open("iris.data") as f:
    lines = [i[:-1] for i in f.readlines()]

❷ n = ["Iris-setosa", "Iris-versicolor", "Iris-virginica"]
    x = [n.index(i.split(",")[-1]) for i in lines if i != ""]
    x = np.array(x, dtype="uint8")
```

```
❸ y = [[float(j) for j in i.split(",")[:-1]] for i in lines if i != ""]
    y = np.array(y)

❹ i = np.argsort(np.random.random(x.shape[0]))
    x = x[i]
    y = y[i]

❺ np.save("iris_features.npy", y)
    np.save("iris_labels.npy", x)
```

Listing 2-1: Loading the raw iris dataset and mapping to the standard format

First, we load the text file containing the data. The list comprehension removes the extraneous newline character ❶. Next, we create the vector of labels by converting each text label into an integer: 0, 1, or 2. The last element in the list, created by splitting a line along commas, is the text label. We want NumPy arrays, so we turn the list into one. Since the labels are never negative or larger than 2, we save a bit of space by making the data type an unsigned 8-bit integer ❷.

Creating the feature vectors as a 150-row by 4-column matrix comes next via a double list comprehension. The outer comprehension (i) moves over lines from the file, and the inner one (j) takes the list of measurements for each sample and turns them into floating-point numbers. We then convert the list of lists into a 2D NumPy array ❸. We finish by randomizing the dataset as we did previously ❹, and finally, we write the NumPy arrays to disk so we can use them later ❺.

Figure 2-1 shows a box plot of the features.

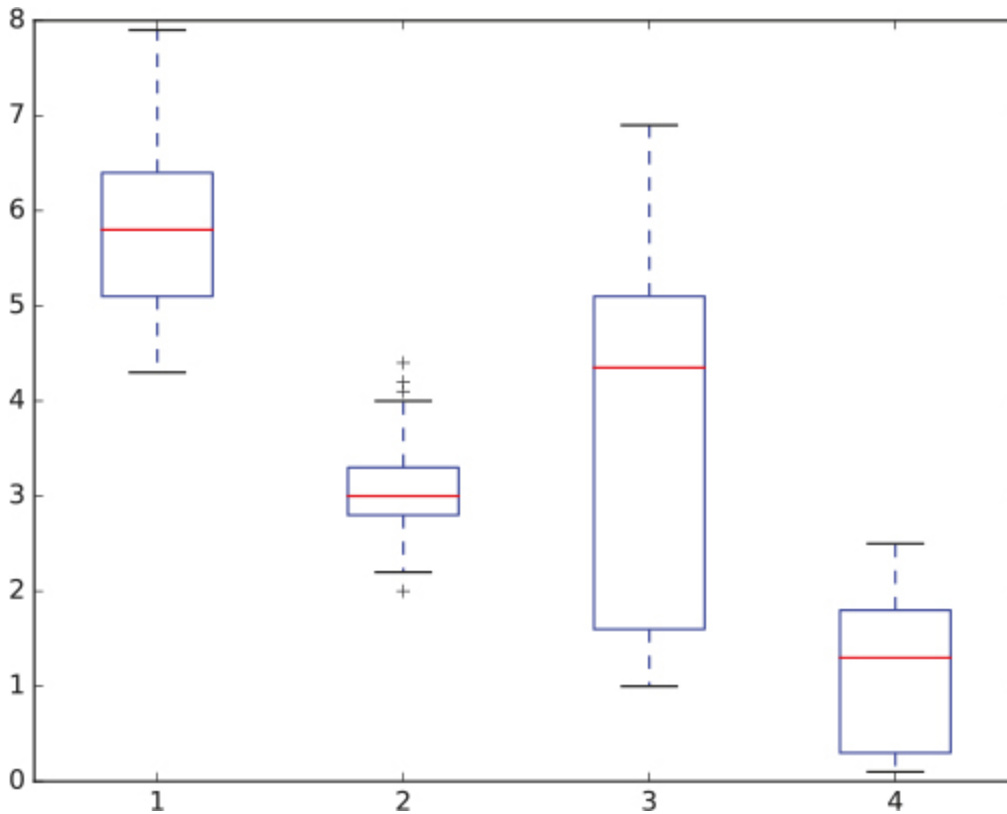


Figure 2-1: A box plot of the four iris dataset features

This is a well-behaved dataset, but the second feature has possible outliers. Because all the features have similar scales, we'll use the features as they are.

Breast Cancer

Our second dataset, the Wisconsin Diagnostic Breast Cancer dataset, is also in sklearn. We'll follow the previous procedure and download the dataset from the UCI Machine Learning Repository to see how to process it. Just as it's crucial to build a good dataset to have any hope of training a good model, it's equally important to learn how to work with data sources that aren't in the format we want. Should you decide to make machine learning and data science a career, you'll be faced with this issue almost daily.

Download the dataset at

<https://archive.ics.uci.edu/dataset/15/breast+cancer+wisconsin+original>, again

by clicking **Download**, creating a data directory, and unzipping to extract the *wdbc.data* file.

This dataset contains cell measurements taken from slides of fine-needle biopsies of breast masses. There are 30 continuous features and 2 classes: malignant (cancer, 212 samples) and benign (no cancer, 357 samples). This is also a popular dataset, with over 670,000 downloads.

Let's look at the first line of the file:

```
842302,M,17.99,10.38,122.8,1001,0.1184, ...
```

The first element is a patient ID number that we don't need to worry about. The second element is the label: *M* for malignant or *B* for benign. The remaining numbers in this line are 30 measurements related to cell size.

As the features themselves are of different scales, we'll create a standardized version alongside creating the raw dataset. This is the entirety of the dataset, and we'll have to hold some of it back for testing, so we don't need to record the per-feature means and standard deviations in this case. If we were able to acquire more data generated in the same way, perhaps from an old forgotten file, we'd need to keep these values so that we could standardize the new inputs. Listing 2-2 contains the script to build this dataset and to generate a summary box plot.

```
import numpy as np
import matplotlib.pyplot as plt

❶ with open("wdbc.data") as f:
    lines = [i[:-1] for i in f.readlines() if i != ""]

❷ n = ["B", "M"]
    x = np.array([n.index(i.split(",")[1]) for i in lines], dtype="uint8")
    y = np.array([[float(j) for j in i.split(",")[2:]] for i in lines])
    i = np.argsort(np.random.random(x.shape[0]))
    x = x[i]
    y = y[i]
    z = (y - y.mean(axis=0)) / y.std(axis=0)
```

```
❸ np.save("bc_features.npy", y)
   np.save("bc_features_standard.npy", z)
   np.save("bc_labels.npy", x)
   plt.boxplot(z)
   plt.show()
```

Listing 2-2: Loading the raw breast cancer dataset

First, read in the raw text data ❶. Then extract each label and map it to 0 for benign or 1 for malignant. Note that we use 1 for the natural target case so that a model that outputs a probability value indicates likelihood of finding cancer ❷. We extract the 30 features per sample as floats, using a nested list comprehension to first pull out the text of the features (i) and then map them to floats (j). This produces a nested list, which NumPy conveniently converts into a matrix of 569 rows and 30 columns.

Next, we randomize the dataset and calculate the standardized version by subtracting, per feature, the mean value of that feature and dividing it by the standard deviation. We'll work with this version and examine it in the box plot of Figure 2-2 ❸, which shows all 30 features after standardization.

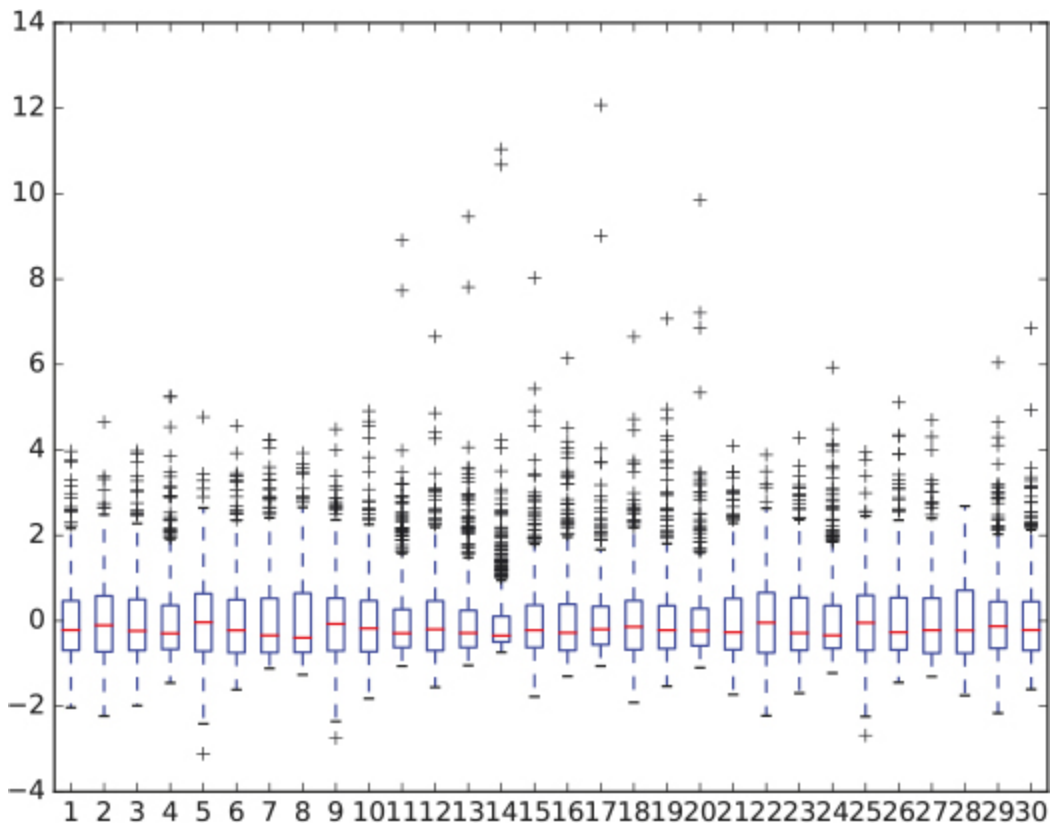


Figure 2-2: A box plot of the 30 breast cancer dataset features

In this case, we don't need to know what the features represent. We'll work with the dataset under the assumption that the selected features are sufficient to determine malignancy. Our models will indicate whether this is true. All features are now of the same scale, as we can see by the location of the boxes on the y-axis; they cover basically the same range.

One characteristic of the data is immediately evident: Many apparent outliers exist, as the interquartile range calls out (see Figure 1-6 on page 41). These aren't necessarily bad values, but they indicate that the data isn't normally distributed; it doesn't, per feature, follow a bell curve-type distribution.

MNIST Digits

Our next dataset isn't typically composed of feature vectors but is instead made up of thousands of small images of handwritten digits.

This dataset is the workhorse of modern machine learning and one of the first datasets deep learning researchers go to when looking to test new ideas. It's overused, but that's because it's so well understood and simple to work with.

The dataset has a long history, but the version we'll use, the most common version, is the MNIST dataset. We'll use Keras to download and format the dataset.

Keras returns the dataset as 3D NumPy arrays. The first dimension is the number of images: 60,000 for training and 10,000 for test. The second and third dimensions are the pixels of the images. The images are 28×28 pixels in size. Each pixel is an unsigned 8-bit integer, [0, 255].

Because we want to work with models that expect vectors as inputs, and because we want to use this dataset to illustrate certain properties of models later in the book, we'll create additional datasets from this initial one. To do so, we'll first unravel the images to form feature vectors so that we can use this dataset with traditional models that expect vector inputs. Second, we'll use images, but we'll permute their order in the dataset. We'll permute the order of the pixels of each image in the same way, so while the pixels will no longer be in the order that produces the digit image, the reordering will be deterministic and applied consistently across all images. Third, we'll create an unraveled feature vector version of these permuted images. We'll use these additional datasets to explore differences between traditional neural networks and CNN models.

Use Listing 2-3 to build the dataset files.

```
import numpy as np
import keras
from keras.datasets import mnist

❶ (xtrn, ytrn), (xtst, ytst) = mnist.load_data()
   idx = np.argsort(np.random.random(ytrn.shape[0]))
   xtrn = xtrn[idx]
   ytrn = ytrn[idx]
   idx = np.argsort(np.random.random(ytst.shape[0]))
   xtst = xtst[idx]
```

```

ytst = ytst[idx]

np.save("mnist_train_images.npy", xtrn)
np.save("mnist_train_labels.npy", ytrn)
np.save("mnist_test_images.npy", xtst)
np.save("mnist_test_labels.npy", ytst)

❷ xtrnv = xtrn.reshape((60000, 28*28))
   xtstv = xtst.reshape((10000, 28*28))
   np.save("mnist_train_vectors.npy", xtrnv)
   np.save("mnist_test_vectors.npy", xtstv)

❸ idx = np.argsort(np.random.random(28*28))
   for i in range(60000):
       xtrnv[i,:] = xtrnv[i, idx]
   for i in range(10000):
       xtstv[i,:] = xtstv[i, idx]
   np.save("mnist_train_scrambled_vectors.npy", xtrnv)
   np.save("mnist_test_scrambled_vectors.npy", xtstv)

❹ t = np.zeros((60000, 28, 28))

   for i in range(60000):
       t[i,:,:] = xtrnv[i,:].reshape((28, 28))
   np.save("mnist_train_scrambled_images.npy", t)
   t = np.zeros((10000, 28, 28))
   for i in range(10000):
       t[i,:,:] = xtstv[i,:].reshape((28, 28))
   np.save("mnist_test_scrambled_images.npy", t)

```

Listing 2-3: Loading and building the various MNIST datasets

We start by telling Keras to load the MNIST dataset ❶. When run for the first time, Keras shows a message about downloading the dataset.

The dataset itself is stored in four NumPy arrays. The first, `xtrn`, has a shape of (60,000, 28, 28) for the 60,000 training images, each 28×28 pixels. The associated labels are in `ytrn` as integers, [0, 9]. The 10,000

test images are in `xtst` with labels in `ytst`. We also randomize the order of the samples and write the arrays to disk for future use.

Next, we unravel the training and test images and turn them into vectors of 784 elements ❷. Unraveling takes the first row of pixels followed by the second row and so on until all rows are laid end to end. We get 784 elements because $28 \times 28 = 784$.

Following this, we generate a permutation of the 784 elements in the unraveled vectors (`idx`) ❸. We use the permuted vectors to form new, scrambled digit images and store them on disk ❹. The scrambled images are made from the scrambled vectors by undoing the unravel operation. In NumPy, this is just a call to the `reshape` method of the vector arrays.

At no time do we alter the relative ordering of the images, so we need to store only one file each for the train and test labels. Figure 2-3 shows representative digits from the MNIST dataset.



Figure 2-3: Representative MNIST digit images

We don't need to standardize the images, as we know they're all on the same scale since they're pixels. We'll sometimes scale them as we use them, but for now we can leave them on disk as byte grayscale images. Table 2-1 shows the training distribution.

Table 2-1: Digit Frequencies for the MNIST Training Set

Digit	Count
0	5,923
1	6,742
2	5,958
3	6,131
4	5,842
5	5,421
6	5,918
7	6,265
8	5,851
9	5,949

The dataset is reasonably balanced, so we don't need to worry about imbalanced data.

CIFAR-10

CIFAR-10 is another standard deep learning dataset that's small enough for us to use without requiring a lot of training time or a GPU. As with MNIST, we can extract the dataset with Keras, which downloads it the first time it's requested. The source page for CIFAR-10 is at <https://www.cs.toronto.edu/%7Ekriz/cifar.xhtml>.

Consider perusing the page to learn more about where the dataset came from. It consists of 60,000 32×32-pixel RGB images from 10 classes, with 6,000 samples in each class. The training set contains

50,000 images, and the test set contains 10,000 images. Table 2-2 shows the 10 classes.

Table 2-2: CIFAR-10 Class Labels and Names

Label	Class
0	airplanes
1	automobiles
2	birds
3	cats
4	deer
5	dogs
6	frogs
7	horses
8	ships
9	trucks

Figure 2-4 shows, row by row, a collection of representative images from each class.



Figure 2-4: Representative CIFAR-10 images

Let's extract the dataset, store it for future use, and create vector representations, much as we did for MNIST. The script to do all this is in Listing 2-4.

```
import numpy as np
import keras
from keras.datasets import cifar10

❶ (xtrn, ytrn), (xtst, ytst) = cifar10.load_data()
   idx = np.argsort(np.random.random(ytrn.shape[0]))
   xtrn = xtrn[idx]
   ytrn = ytrn[idx]
```

```

idx = np.argsort(np.random.random(ytst.shape[0]))
xtst = xtst[idx]
ytst = ytst[idx]

np.save("cifar10_train_images.npy", xtrn)
np.save("cifar10_train_labels.npy", ytrn)
np.save("cifar10_test_images.npy", xtst)
np.save("cifar10_test_labels.npy", ytst)

❷ xtrnv = xtrn.reshape((50000, 32*32*3))
xtstv = xtst.reshape((10000, 32*32*3))
np.save("cifar10_train_vectors.npy", xtrnv)
np.save("cifar10_test_vectors.npy", xtstv)

```

Listing 2-4: Loading and building the various CIFAR-10 datasets

We first load CIFAR-10 from Keras ❶. As with MNIST, the dataset downloads automatically the first time that the code is run. And as with MNIST, we randomize the train and test splits. The training data is in `xtrn` as a (50,000, 32, 32, 3) array. The last dimension is for the three color components for each pixel: red, green, and blue. The test data is similar, and is in `xtst` as a (10,000, 32, 32, 3) array.

We write the randomized train and test images to disk. Next, we unravel the images to produce $32 \times 32 \times 3 = 3,072$ -element feature vectors representing the images ❷ and write them to disk.

Data Augmentation

As we saw in Chapter 1, the dataset is everything, so it needs to be as complete as possible. We'll normally achieve this by carefully selecting samples that fit within the range of inputs the model encounters when we use it. We need the model to interpolate, not extrapolate. But sometimes, even though we have a wide range of possible samples, we don't have a lot of actual samples. This is where data augmentation can help.

Data augmentation uses the data in the existing dataset to generate new possible samples to add to the set. These samples are always based,

in some way, on the existing data. Data augmentation is a powerful technique and is particularly helpful when our dataset is small. In a practical sense, data augmentation should probably be used whenever it's feasible.

Data augmentation takes the data we already have and modifies it to create new samples that could have plausibly come from the same parent distribution as our actual data. If we were patient enough to keep collecting real data, we could, in theory, measure those new samples. Sometimes data augmentation can go beyond what we'd measure, yet still help the model learn to generalize to the actual data. For example, a model using images as input might benefit from unrealistic colors or backgrounds when the actual inputs to the model would never use those colors or backgrounds.

While data augmentation works in many situations and is a mainstay of deep learning, you won't always be able to use it because not all data can be realistically enhanced. In this section, we'll take a look at why we might consider using data augmentation and how to go about doing it.

We'll then augment two of the datasets we developed previously so that when we build models we can see how augmentation affects the models' learning. As far as augmentation is concerned, here's a rule of thumb: In general, you should perform data augmentation whenever possible, especially if the dataset is small.

Reasoning

For many models, the solution to the curse of dimensionality is to fill in the space of possible inputs with more training data (see Chapter 1). Data augmentation is one way we can fill in this space.

We'll need to do this in the future; in Chapter 3, for example, we'll meet k -nearest neighbors, perhaps the simplest of all classifiers. This classifier critically depends on having enough training data to adequately fill in the input feature space. If the data has three features, the space is 3D and the training data will fit into a cube in that space. The more training data we have, the more samples we'll have in the cube, and the better the classifier will do. That's because the classifier

measures the distance between points in the training data and that of a new, unknown feature vector and then votes on which label to assign. The denser the space is with training points, the more often the voting process succeeds.

Loosely speaking, data augmentation fills in this space. For most datasets, acquiring more samples from the parent distribution won't fill in every part of the feature space but will create a more complete picture of what the parent distribution looks like in the feature space.

When we work with modern deep learning models (Chapter 9), we'll see that data augmentation has additional benefits. During training, a neural network becomes conditioned to learn features of the training data. If the features the network learns to pay attention to are useful for distinguishing the classes, all is well. But, as we saw with the wolf and husky example in Chapter 1, sometimes the network learns the wrong details, which we can't use to generalize to new inputs.

Regularization is the process of taking steps to avoid this tendency. It helps the network learn important features of the training data that generalize as we want them to. Short of acquiring more actual data, data augmentation is perhaps the simplest way to regularize the network as it learns. It conditions the learning process to not pay attention to quirks of the particular samples selected for the training set but to instead focus on more general features of the data. At least, that is the hope.

Another benefit of data augmentation is that it lessens the likelihood of overfitting when training. We'll discuss overfitting more in Chapter 6, but it happens when the model learns the training data nearly perfectly without learning to generalize to new inputs.

Using a small dataset can lead to overfitting if the model is able to memorize the training data. Data augmentation increases the dataset size, reducing the probability of overfitting and possibly allowing use of a model with a larger capacity. (Capacity is a nebulous concept. Think "larger" in that the model can learn more of what is important in the training data, while still generalizing to new data.)

In data augmentation as it relates to the training/validation/test split of the dataset, you *must* be sure that every augmented sample belongs to

the same set. For example, if we augment sample X_{12345} , and this sample is assigned to the training set, then we must ensure that *all* the augmented samples based on X_{12345} are also members of the training set. This is so important that it's worth reiterating: *Never mix an augmented sample based on an original sample among the training, validation, and test sets.*

Not following this rule means our beliefs about the quality of the model are unfounded, or at least partially unwarranted, because some samples in the validation and test sets are, essentially, also in the training set, since they're based on the training data. It's easy to make this mistake, especially when working with others or with a database of some kind.

The correct way to augment data is *after* making the training, validation, and test splits. Then, augment at least the training data and label all the new samples as training data.

It might make sense to also augment the validation and test splits if you don't have a lot of either. I haven't run across any studies that tried to be rigorous about the effects of augmenting the validation and test data, but conceptually, it might help.

Methods

To augment a dataset, we need to generate new samples from it that are plausible, meaning they could really occur in the dataset. For images, this is straightforward; you can often rotate the image, or flip it horizontally or vertically. Other times, you can manipulate the pixels themselves to change the contrast or alter the colors. Some have even gone so far as to swap entire color bands—swapping the red channel with the blue channel, for example.

The manipulations must make sense. A subtle rotation might mimic a change in the camera's orientation, and a left-to-right flip might mimic the experience of looking in a mirror. But a top-to-bottom flip likely wouldn't be as realistic. True, a monkey might hang upside down in a picture, but flipping the picture would flip the tree and the ground

as well. On the other hand, you might be able to do a top-to-bottom flip in an aerial image, which shows objects in any orientation.

While images are generally straightforward to augment, augmentation of a feature vector is more subtle. It's not always clear how to do it or if it's even possible. What can we do in that case?

If we encoded color as a one-hot vector of, say, red, green, or blue, and an instance of a class can be red or green or blue, then we may augment by shifting the color between red, green, and blue. If a sample can represent male or female, we could change those values to get a new sample of the same class with a different sex.

These are unusual steps to take, however, as you typically try to augment continuous values, creating a new feature vector that still represents the original class. We'll examine one way to do this next by augmenting the iris dataset. After that, we'll augment the CIFAR-10 dataset to see how to work with images.

The Iris Dataset

The iris dataset has 150 samples from three classes, each with four continuous features. We'll augment it by using *principal component analysis (PCA)*. This is an old technique, in use for over a century. It was common in machine learning before the advent of deep learning to combat the curse of dimensionality, because it can reduce the number of features in a dataset. It also has a variety of uses outside of machine learning.

Imagine that we have a dataset with only two features—for example, the first two features of the iris dataset. The scatterplot in Figure 2-5 shows us where the first two features of the iris dataset for classes 1 and 2 fall in 2D space.

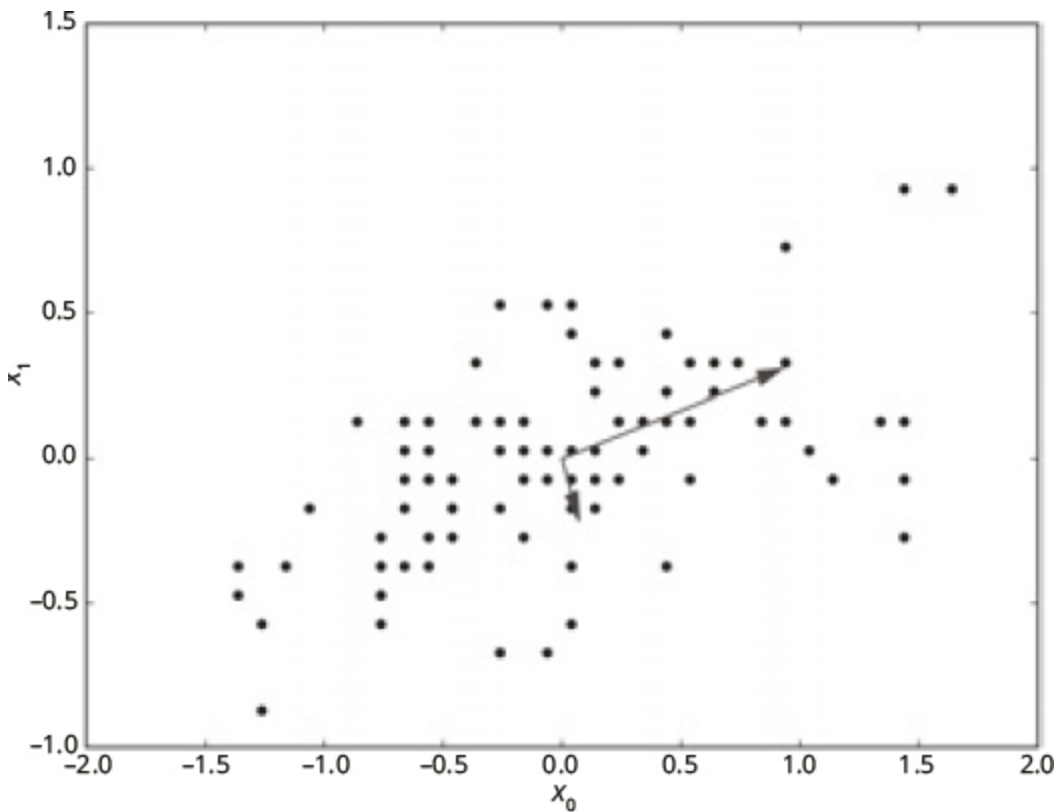


Figure 2-5: The first two iris features for classes 1 and 2, with their principal components

The plot has shifted the origin to (0,0) by subtracting the mean value of each feature. This changes only its origin, not the variance or scatter of the data.

PCA tells us the directions, or *principal components*, of the variance of the data. Since the data in this plot is 2D, we have two components: the two arrows. If we had 100 features, we would have up to 100 principal components.

The principal components also tell us how much of the variance of the data each of these directions explains. In the plot, the length of the arrow corresponds to the fraction of the total variance explained by each component. The largest component is along the diagonal that matches the greatest scatter of the points. Traditional machine learning uses PCA to reduce the number of features while still (hopefully) representing the dataset well. PCA can help fight the curse of dimensionality by finding the principal components and then throwing

the less influential ones away. However, for data augmentation, we want to keep all the components.

Listing 2-5 contains the code that produces Figure 2-5.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import decomposition

❶ x = np.load("../data/iris/iris_features.npy")[:, :2]
   y = np.load("../data/iris/iris_labels.npy")
   idx = np.where(y != 0)
   x = x[idx]
   x[:, 0] -= x[:, 0].mean()
   x[:, 1] -= x[:, 1].mean()

❷ pca = decomposition.PCA(n_components=2)
   pca.fit(x)
   v = pca.explained_variance_ratio_

❸ plt.scatter(x[:, 0], x[:, 1], marker='o', color='b')
   ax = plt.axes()
   x0 = v[0]*pca.components_[0, 0]
   y0 = v[0]*pca.components_[0, 1]
   ax.arrow(0, 0, x0, y0, head_width=0.05, head_length=0.1, fc='r', ec='r')
   x1 = v[1]*pca.components_[1, 0]
   y1 = v[1]*pca.components_[1, 1]
   ax.arrow(0, 0, x1, y1, head_width=0.05, head_length=0.1, fc='r', ec='r')
   plt.xlabel("$x_0$", fontsize=16)
   plt.ylabel("$x_1$", fontsize=16)
   plt.show()
```

Listing 2-5: The iris PCA plot

Much of the preceding code makes the plot ❸. The imports are standard except for a new one from sklearn, the `decomposition` module. We load the previously saved iris dataset, keeping only the first two features in `x` and the labels in `y`. We then keep only class 1 and class 2 features by

excluding class 0. Next, we subtract the per-feature means to center the data around the point (0,0) ❶.

Then we create the PCA object and fit the iris data to it ❷. There are two features, so the number of components in this case is also two. The PCA Python class mimics the standard approach of sklearn: It defines the model, then fits data to it. Once this is done, we have the principal components stored in `pca` and accessible via the `components_` member variable. We set `v` to a vector representing the fraction of the variance in the data explained by each of the principal component directions. Since there are two components, this vector also has two components.

The components are always listed in decreasing order so that the first component is the direction describing the majority of the variance, the second component is the next most important, and so on. In this case, the first component describes some 84 percent of the variance, and the second describes the remaining 16 percent. We'll use this ordering when generating new augmented samples. Here we use the fraction to scale the length of the arrows in the plot, showing the principal component directions and relative importance.

Once you know the principal components, you can use PCA to create derived variables by rotating the data to align it with the principal components. The `transform` method of the PCA class does this by mapping an input (in our case, the original data) to a new representation where the variance aligns with the principal components. This mapping is exact, and you can reverse it by using the `inverse_transform` method.

Doing this alone doesn't generate new samples for us. If we take the original data, x , transform it to the new representation, and then inverse transform it, we end up where we started, with x . But, if we transform x and then, before calling the inverse transform, modify some of the principal components, we return a new set of samples that are based on x . This is precisely what we want for data augmentation. Next, we'll see which components to modify, and how.

The `pca` class orders components by their importance. We want to keep the most important components as they are so that the inverse transform produces data that looks much like the original data. We

don't want to transform values too much, or the new samples won't be plausible instances of the class we claim they represent. We'll arbitrarily say that we want to keep the components that, cumulatively, represent 90 to 95 percent of the variance in the data. We won't modify these at all.

We modify the remaining components by adding normally distributed noise, which follows the bell curve so that most of the time the value is near the middle, which we'll set to 0 (meaning no change to the component), and increasingly rarely to larger values. We'll add the noise to the existing component and call the inverse transform to produce new samples that are similar but not identical to the originals.

Let's make sense of the previous paragraph with code. Listing 2-6 shows our approach to generating augmented data.

iris_data_augmentation.py

```
import numpy as np
from sklearn import decomposition
```

```
❶ def generateData(pca, x, start):
    original = pca.components_.copy()
    ncomp = pca.components_.shape[0]
    a = pca.transform(x)
    for i in range(start, ncomp):
        pca.components_[i,:] += np.random.normal(scale=0.1, size=ncomp)
    b = pca.inverse_transform(a)

    pca.components_ = original.copy()
    return b
```

```
def main():
```

```
❷ x = np.load("../data/iris/iris_features.npy")
    y = np.load("../data/iris/iris_labels.npy")
```

```
    N = 120
    x_train = x[:N]
    y_train = y[:N]
    x_test = x[N:]
```

```

y_test = y[N:]

pca = decomposition.PCA(n_components=4)
pca.fit(x)
print(pca.explained_variance_ratio_)
start = 2
❸ nsets = 10
nsamp = x_train.shape[0]
newx = np.zeros((nsets*nsamp, x_train.shape[1]))
newy = np.zeros(nsets*nsamp, dtype="uint8")

❹ for i in range(nsets):
    if (i == 0):
        newx[0:nsamp,:] = x_train
        newy[0:nsamp] = y_train
    else:
        newx[(i*nsamp):(i*nsamp+nsamp),:] =
            generateData(pca, x_train, start)
        newy[(i*nsamp):(i*nsamp+nsamp)] = y_train

❺ idx = np.argsort(np.random.random(nsets*nsamp))
newx = newx[idx]
newy = newy[idx]
np.save("../data/iris/iris_train_features_augmented.npy", newx)
np.save("../data/iris/iris_train_labels_augmented.npy", newy)
np.save("../data/iris/iris_test_features_augmented.npy", x_test)
np.save("../data/iris/iris_test_labels_augmented.npy", y_test)

main()

```

Listing 2-6: Augmenting the iris data with PCA

The `main` function ❷ loads the existing iris data, `x`, and the corresponding labels, `y`, and then calls PCA, this time using all four features of the dataset. This gives us the four principal components telling us how much of the variance is explained by each component:

```
0.92461621 0.05301557 0.01718514 0.00518309
```

The first two principal components describe over 97 percent of the variance. Therefore, we'll leave the first two components, indices 0 and 1, alone and start with index 2 when we want to generate new samples.

We declare the number of sets we'll define ❸. Here a *set* means a new collection of samples. Since the samples are based on the original data that's set aside for training (`x_train`, with 120 samples), each new set will contain 120 samples as well. In fact, they'll be in the same order as the original samples so that the class label that should go with each of these new samples is in the same order as the class labels in `y_train`.

We don't want to lose our original data, either, so `nsets = 10` puts the original data and nine new sets of samples based on that original data—for a total of 1,200 samples—in the new dataset. We grab the number of samples in `x_train`, 120, and define the arrays to hold our new features (`newx`) and associated labels (`newy`).

Next, we loop to generate the new samples, one set of 120 at a time ❹. The first pass simply copies the original data into the output arrays. The remaining passes are similar, updating the source and destination indices of the output arrays appropriately, but instead of assigning `x`, we assign the output of `generateData`. When the loop is done, we scramble the order of the entire dataset and write it to disk ❺.

All the magic is in `generateData` ❶. We pass in the PCA object (`pca`), the original data (`x_train`), and the starting principal component index (`start`). We set the last argument to 2 to leave the two most important components alone. We keep a copy of the actual components so we can reset the `pca` object before we return. Then we define `ncomp`, the number of principal components, for convenience and call the forward transformation mapping the original data along the principal components.

The loop updates the two least important components by adding a random value drawn from a normal curve with mean value 0 and a standard deviation of 0.1. Choosing 0.1 is arbitrary; if the standard deviation is small, the new samples will be near the old samples, while if it's larger, they'll be farther away and possibly not representative of the class anymore.

Next, we call the inverse transformation by using the modified principal components and restore the actual components. Finally, we return the new set of samples. Figure 2-6 shows the new dataset.

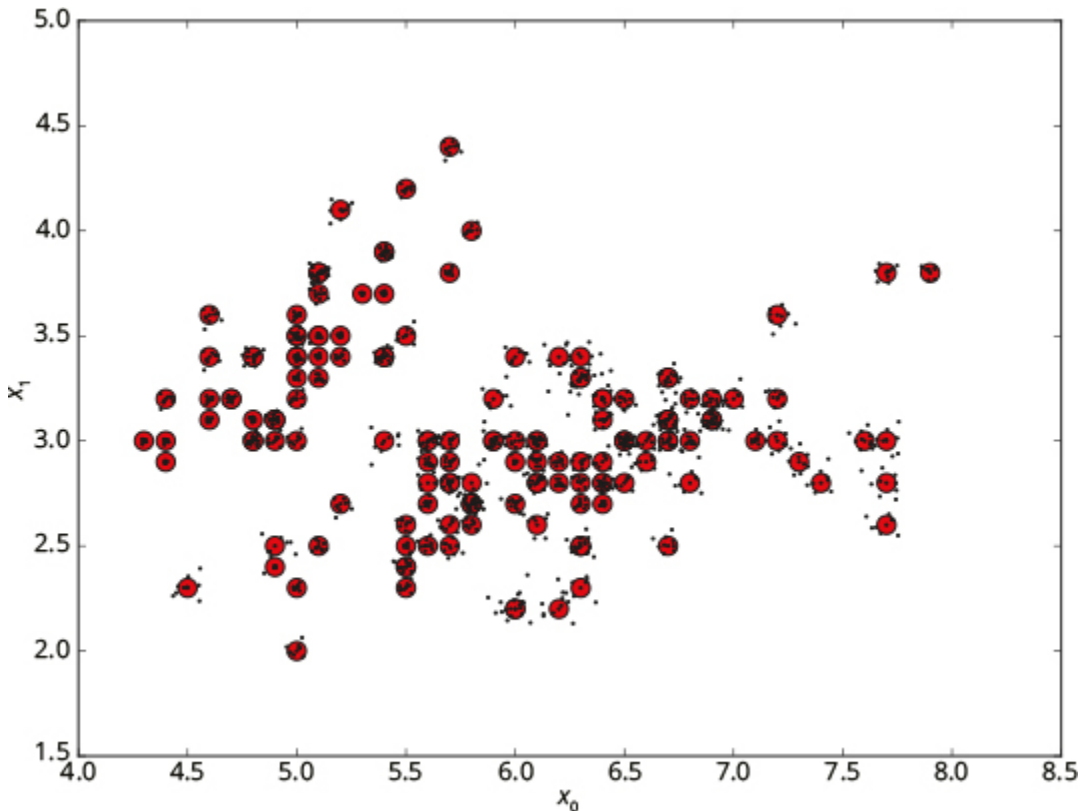


Figure 2-6: The first two features of the original iris dataset (larger circles) and the augmented features generated by Listing 2-6 (smaller dots)

The larger circles are from our original dataset, and the small black dots are the augmented samples. They all fall near an existing sample, which we expect from modifying only the weakest of the principal components. Since we copied the original data into the augmented dataset, each big dot has a small dot at the center.

This approach is appropriate for continuous features only, and you should be careful to modify only the weakest of the principal components by a small amount. Experimentation is important here. As an exercise, try applying the same technique to augment the breast cancer dataset, which also consists of continuous features.

The CIFAR-10 Dataset

While augmenting the iris dataset involved a lot of discussion and some less-than-obvious math, augmenting images is generally a lot simpler but still just as effective when training modern models. Let's build a new dataset with additional versions of the existing images.

Figure 2-4 shows representative images from each class in the CIFAR-10 dataset. These are color images stored as RGB data for the red, green, and blue channels. They were taken from ground level, so left and right flips make sense here, while top and bottom flips don't. Small rotations and translations (shifting the image in the x or y direction, or both) are common techniques.

What do we do with pixels that have no data after the shift or rotate? If we shift an image 3 pixels to the left, we need to fill in the three columns on the right. Or if we rotate to the right, pixels at the upper right and lower left will need to be filled in. We can handle this in several ways. One is to simply leave the pixels black, or all 0 values, and let the model learn that no helpful information is there. Another is to replace the pixels with the mean value of the image, which also provides no information and which the model will, hopefully, ignore. However, the most popular solution is to crop the image.

The image is 32×32 pixels. Pulling a random patch from the image of, say, 28×28 pixels is the equivalent of shifting the image by a random x or y position of up to 4 pixels without needing to fill in anything. If we rotate the image first, which requires interpolation of the pixels, and then crop to remove the edge regions, we'll again have no empty pixels to worry about.

We need to consider one point here. So far, we've talked about building a dataset for training a model. What should we do when we want to use the model? Rather than handing the model random croppings of the test inputs as well, we hand the model a cropping centered on the image. So, for CIFAR-10, we take each 32×32 test input and crop it to 28×28 by dropping the outer 6 pixels, then present that to the model. We do this because the center crop still represents the actual test image and not an augmented version of it.

Figure 2-7 illustrates what we mean by rotations, flips, random croppings for training, and center cropping for testing.

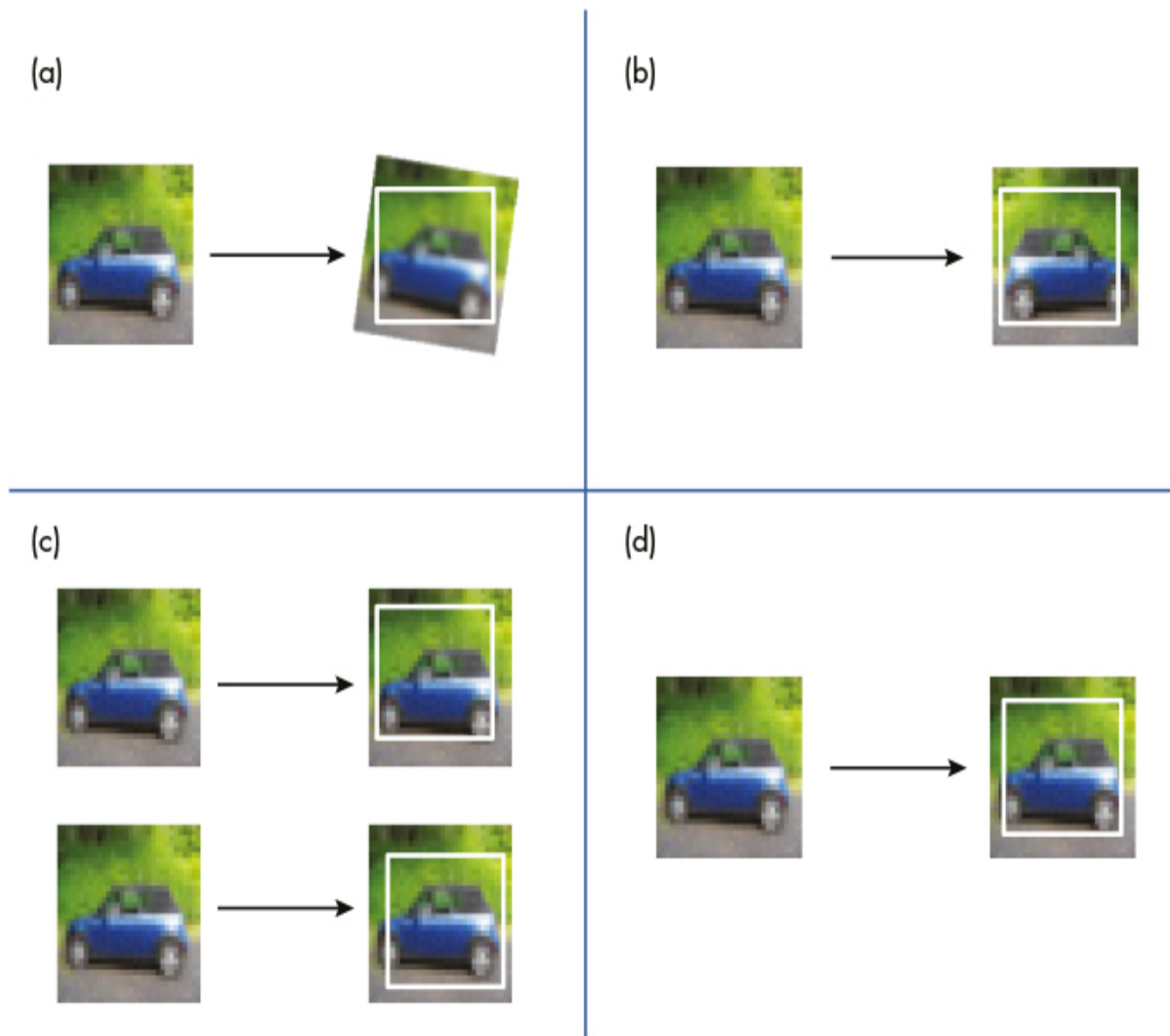


Figure 2-7: Examples of rotations, flips, and croppings

In (a), we rotate the image and take a center crop. The output image is in the white square. In (b), we flip left to right and crop randomly. In (c), we take two random crops without flipping, and in (d) we take a center crop for testing, without any rotation or flip. Some people augment test images, but we won't do so here.

Listing 2-7 shows how to augment the CIFAR-10 training set with random crops, rotations, and flips.

```
cifar10_augment.py
import numpy as np
from PIL import Image
```

```

❶ def augment(im, dim):
    img = Image.fromarray(im)
    if (np.random.random() < 0.5):
        img = img.transpose(Image.FLIP_LEFT_RIGHT)
    if (np.random.random() < 0.3333):
        z = (32-dim)/2
        r = 10*np.random.random()-5
        img = img.rotate(r, resample=Image.BILINEAR)
        img = img.crop((z, z, 32-z, 32-z))
    else:
        x = int((32-dim-1)*np.random.random())
        y = int((32-dim-1)*np.random.random())
        img = img.crop((x, y, x+dim, y+dim))
    return np.array(img)

def main():
    ❷ x = np.load("../data/cifar10/cifar10_train_images.npy")
        y = np.load("../data/cifar10/cifar10_train_labels.npy")
        factor = 10
        dim = 28
        z = (32-dim)/2
        newx = np.zeros((x.shape[0]*factor, dim, dim, 3), dtype="uint8")
        newy = np.zeros(y.shape[0]*factor, dtype="uint8")
        k = 0
        ❸ for i in range(x.shape[0]):
            im = Image.fromarray(x[i,:])
            im = im.crop((z, z, 32-z, 32-z))
            newx[k,...] = np.array(im)
            newy[k] = y[i]
            k += 1
            for j in range(factor-1):
                newx[k,...] = augment(x[i,:], dim)
                newy[k] = y[i]
                k += 1
        idx = np.argsort(np.random.random(newx.shape[0]))
        newx = newx[idx]
        newy = newy[idx]
        np.save("../data/cifar10/cifar10_aug_train_images.npy", newx)

```



```
np.save("../data/cifar10/cifar10_aug_train_labels.npy", newy)
```

```
④ x = np.load("../data/cifar10/cifar10_test_images.npy")
newx = np.zeros((x.shape[0], dim, dim, 3), dtype="uint8")

for i in range(x.shape[0]):
    im = Image.fromarray(x[i,:])
    im = im.crop((z, z, 32-z, 32-z))
    newx[i,...] = np.array(im)
np.save("../data/cifar10/cifar10_aug_test_images.npy", newx)
```

Listing 2-7: Augmenting the CIFAR-10 dataset

The `main` function loads the existing dataset and defines the augmentation factor, crop size, and a constant for defining a center crop ②.

The new image will be put in `newx`, which has the dimensions (500,000, 28, 28, 3); there are 50,000 training images, each with 32×32 pixels and 3 color bands. We set the augmentation factor to 10. Similarly, there will be 500,000 labels. The counter, `k`, indexes into this new dataset. For every image in the old dataset, we'll create nine new versions and center crop the original ③.

As the dataset consists of images, it's easiest to work with the data in image form, so we make the current sample a PIL image in order to easily crop it. This is the center crop of the original image. We store it in the new output array.

The code uses two Python idioms here that we'll see more than once. The first is to turn a NumPy array representing an image into a PIL image:

```
im = Image.fromarray(arr)
```

The second is to go the other way and turn a PIL image into a NumPy array:

```
arr = np.array(im)
```

We must be sure that the NumPy array is a valid image data type like unsigned byte (`uint8`). Use the `astype` NumPy array method to cast between types, remembering that you bear all responsibility for understanding what that casting entails.

Referring back to Listing 2-7, we're creating the nine versions of the current image. For each of these, we copy the label and assign the output array an augmented version. We'll describe the `augment` function shortly. Once the new dataset is constructed, we scramble the order and write the augmented training dataset to disk ❸.

We're not quite done. We created an augmented training set that cropped the original 32×32 images to 28×28 . We must, therefore, at least crop the original test set ❹. As stated previously, we use a center crop and no augmentation of the test data. Therefore, we simply load the test dataset, define the new output test dataset, and run a loop that crops the 32×32 images to 28×28 . When done, we write the cropped test data to disk. Note that we didn't modify the order of the images in the test set; we simply cropped them, so we do not need to write a new file for the test labels.

The `augment` function ❶ is where all the action is. We immediately change the input NumPy array into a PIL image object. We next decide, with a 50-50 chance, whether we'll flip the image left to right. Note that we don't crop the image just yet.

Next, we ask whether we should rotate the image. We select rotation with a probability of 33 percent (1 in 3 chance). I chose this number since it seems that we might want to crop randomly more often than we rotate. We could even drop this probability down to 20 percent (1 in 5 chance). If we do rotate, we select the rotation angle, $[-5, 5]$, and then call the `rotate` method using bilinear interpolation to make the rotated image look a bit nicer than using the nearest neighbor, which is the PIL default. Next, we center crop the rotated image. This way, we don't get any black pixels on the edges where the rotation had no image information to work with.

If we don't rotate, we're free to select a random crop. We choose the upper-left corner of this random crop, ensuring that the cropped square

won't exceed the dimensions of the original image. Finally, we convert the data back to a NumPy array and return.

Summary

In this chapter, we built four datasets that we'll use as examples throughout the rest of the book. The first two, irises and breast cancer histology, are based on feature vectors. The last two, MNIST and CIFAR-10, are represented as images. We then explored two data augmentation methods: augmenting a feature vector of continuous values by using PCA and, more critical for deep learning, augmenting images via basic transformations.

In the next chapter, we'll transition to our discussion of classical machine learning models. In Chapter 4, we'll use these datasets with those models.

OceanofPDF.com

PART II

CLASSICAL MACHINE LEARNING

Before deep learning, there was machine learning; or, to be more precise, deep learning is a subfield of machine learning. Classical machine learning uses many of the framework pieces found in deep learning. Indeed, a complete understanding of deep learning requires understanding what came before. Classical machine learning prepares us for neural networks and deep learning, while teaching us that, at times, oldies are still goodies.

OceanofPDF.com

3

INTRODUCTION TO MACHINE LEARNING



The term *classical machine learning* implies that there's something newer that makes older techniques classical. We'll get to this newer technique, deep learning, in the chapters that follow. But first, we need to build our intuition by examining older techniques that will help cement concepts for us and that are still useful when the situation warrants.

Including some sort of history here is tempting. To keep to the practical nature of this book, we won't, but a full history of machine learning is needed, and as of this writing, I have not found one. Historians reading this, please take note. Machine learning is not new; the techniques of this chapter go back decades and have had considerable success on their own. However, the successes were always limited in a way that deep learning has now largely overcome. Still, owning a hammer doesn't make everything a nail.

You will encounter problems that are well suited to these older techniques. This might be because too little data is available to train a deep model, because the problem is simple and easily solved by a classical technique, or because the operating environment is not conducive to a large, deep model (think microcontroller). Many of these techniques are easier to understand conceptually than a deep model is,

and all the comments in earlier chapters about building datasets, as well as the comments in Chapter 8 about evaluating models, still apply.

The following sections introduce several popular classical models, not in great detail, but in essence. All these models are supported by `sklearn`. In Chapter 4, we'll apply the models to some of the datasets we developed in Chapter 2. This will give us an idea of the models' relative performance when compared to one another and provide a baseline for comparing the performance of deep models in subsequent chapters.

We'll examine six classical models. The order in which we discuss them roughly tracks with the complexity of the model type. The first three—nearest centroid, k -nearest neighbors, and naive Bayes—are relatively easy to understand and implement. The last three—decision trees, random forests, and support vector machines—are harder, but we'll do our best to understand what's going on.

Nearest Centroid

Assume we want to build a classifier and that we have a properly designed dataset of n classes (see Chapter 1). For simplicity, we'll assume that we have m samples of each of the n classes. While this isn't necessary, it simplifies the notation. Since our dataset is properly designed, we have training samples and test samples. We don't need validation samples in this case, so we can throw them into the training set. Here the sample is a feature vector of floating-point values.

The goal of selecting components for a feature vector is to end up with a feature vector that makes the various classes distinct in the feature space. Let's say the feature vector has w features; we can think of the feature vector as the coordinates of a point in a w -dimensional space. If $w = 2$ or $w = 3$, we can graph the feature vectors. Mathematically, there's no reason to restrict w to 2 or 3; everything we describe here works in 100, 500, or 1,000 dimensions. However, it won't work equally well: The dreaded curse of dimensionality will creep in and eventually require an exponentially large training dataset, but we'll ignore this elephant in the room for now.

If the features are well chosen, we might expect a plot of the points in the w -dimensional space to group the classes so that all of the samples from class 0 are near one another, and all the samples from class 1 are near one another but distinct from class 0, and so forth. If this is our expectation, how might we use this knowledge to assign a new, unknown sample to a particular class? This is the goal of classification, but in this case, given our assumption that the classes are well separated in the feature space, what is something simple we could do?

Figure 3-1 shows a hypothetical 2D feature space with four distinct classes.

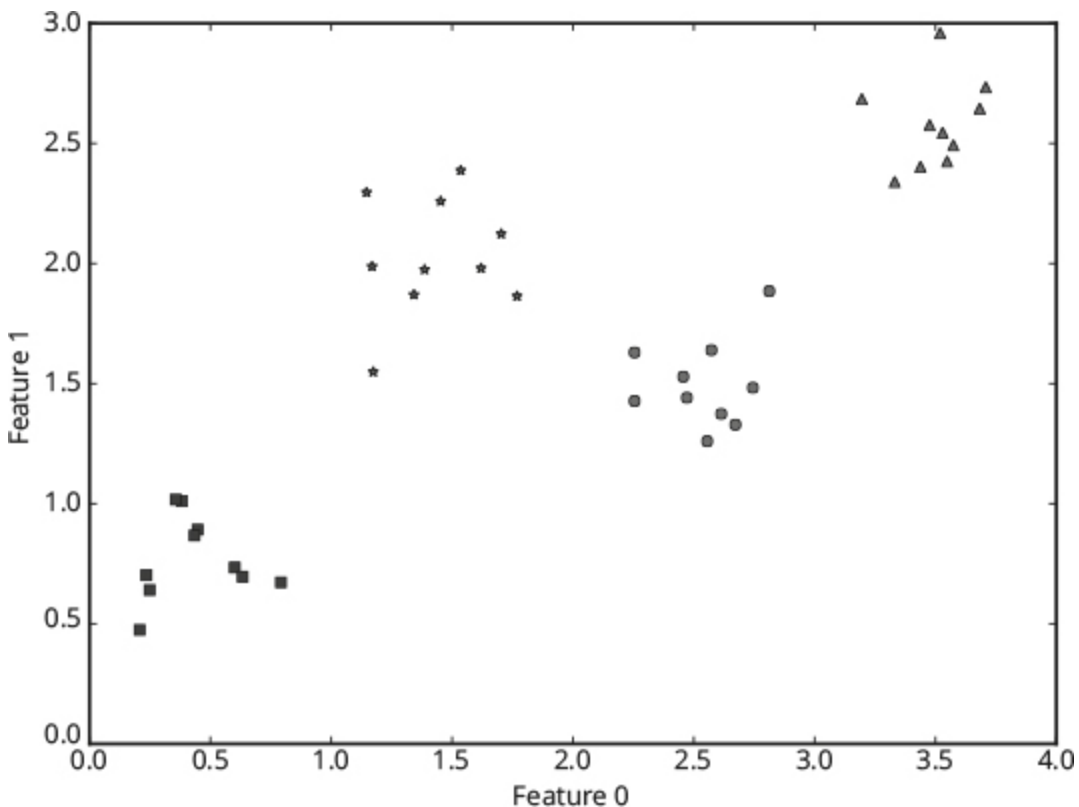


Figure 3-1: A hypothetical 2D feature space with four distinct classes

The classes are clearly separated in this toy example. A new, unknown feature vector will fall into this space as a point. The goal is to assign a class label to the new point—either square, star, circle, or triangle.

Since the points of Figure 3-1 are so well grouped, we might think that we could represent each group by an average position in the feature

space. Instead of the 10 square points, we'd use a single point to represent the squares. This seems an entirely reasonable step.

The average point of a group of points has a name: the *centroid*, the center point. We know that we can compute the average of a set of numbers by adding them up and dividing by how many we added. To find the centroid of a set of points in 2D space, we first find the average of all the x-axis coordinates and then the average of all the y-axis coordinates. If we have three dimensions, we'll do this for the x-, y-, and z-axes. If we have w dimensions, we'll do it for each of the dimensions. In the end, we'll have a single point that we can use to represent the entire group. If we do this for our toy example, we get Figure 3-2, where the centroid is the large marker in each distinct class.

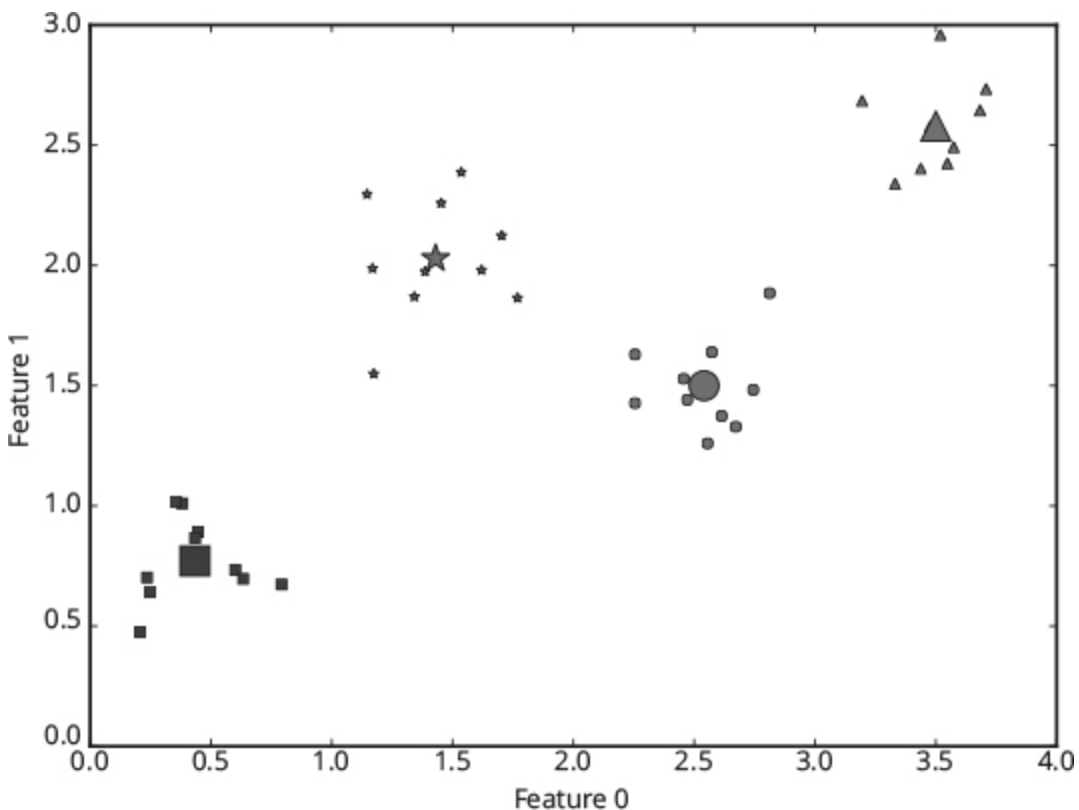


Figure 3-2: A hypothetical 2D feature space with four distinct classes and their centroids

The centroid can be helpful to us if we're given a new, unknown sample, as it is a point in the feature space (as mentioned previously). We can then measure the distance between this point and each of the centroids and assign the class label of the closest centroid. The idea of

distance is somewhat ambiguous; there are many ways to define distance. For example, drawing a straight line between the two points is the *Euclidean distance*. If we have two points, (x_0, y_0) and (x_1, y_1) , then we calculate the Euclidean distance between them with the following:

$$d = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$$

If we have three dimensions, the distance between two points becomes this:

$$d = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2 + (z_0 - z_1)^2}$$

We can generalize this to w dimensions for two points, x_0 and x_1 :

$$d = \sqrt{\sum_{i=0}^{w-1} (x_0^i - x_1^i)^2}$$

Here x_0^i is the i th component of the point x_0 . This means, component by component, find the difference between the two points, square it, and add it to the squared difference of all the other components. Then, take the square root.

Figure 3-3 shows a sample point in the feature space as well as the distances to the centroids.

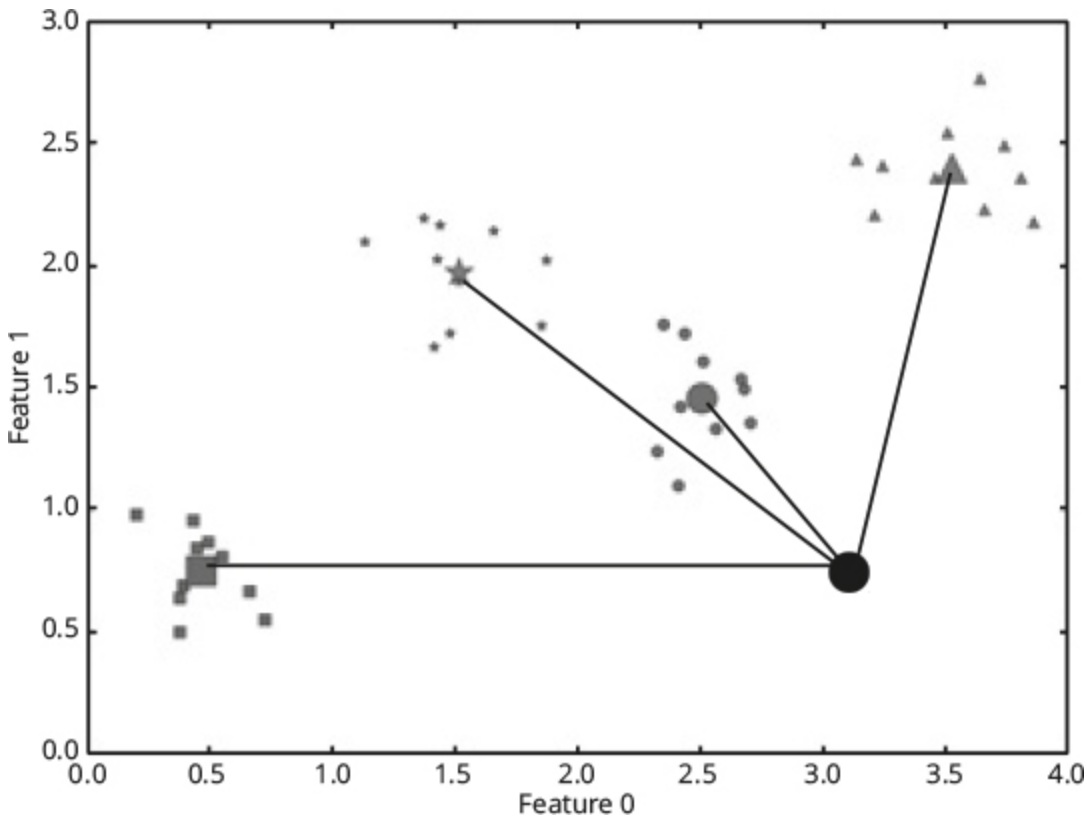


Figure 3-3: A hypothetical 2D feature space with four distinct classes, their centroids, and a new, unknown sample

The shortest distance is to the circle class, so we'd assign the new sample to that class.

The process we just implemented is a *nearest-centroid* classifier, also called *template matching*. We use the class centroid as a proxy for the class as a whole. Then, new samples use those centroids to decide on a label.

Why isn't this classifier used more, since it seems so simple? There are several reasons, one being the curse of dimensionality. As the number of features increases, the space gets larger and larger, and we need exponentially more training data to get an idea of where the centroids should be. A large feature space implies that this might not be the right approach.

A more severe problem exists, however. Our toy example had very tight groups. If the groups are more diffuse, even overlapping, then the

selection of the nearest centroid becomes problematic. How would we know whether the closest centroid represents class A or class B?

Still more severe is that a particular class might fall into *two* or more distinct groups. If we calculate the centroid of only the class as a whole, the centroid will be between the groups for the class and not represent either cluster well. We'd need to know that the class is split between groups and use multiple centroids for the class. If the feature space is small, we can plot it and see that the class is divided between groups. However, if the feature space is larger, there's no easy way for us to decide that the class is divided between multiple groups and that multiple centroids are required.

For elementary problems, nearest centroid might be ideal. Not every application deals with difficult data. We might be building an automated system that needs to make easy decisions on new inputs. In that case, this simple classifier might be a perfect fit.

k-Nearest Neighbors

As we saw earlier, one problem with a centroid approach is that the classes might be divided among multiple groups in the feature space. As the number of groups increases, so would the number of centroids necessary to specify the class. This implies another approach. Instead of computing centroids per class, what if we use the training data as is and select the class label for a new input sample by finding the closest member of the training set and using its label?

This type of classifier is a *nearest-neighbor* classifier. If we look at only the closest sample in the training set, we are using one neighbor, so we call the classifier a 1-nearest neighbor, or 1-NN, classifier. But we don't need to look at only the nearest training point. We might want to look at several and then vote to assign a new sample to the most common class label. In the event of a tie, we can select one of the class labels at random. If we use three nearest neighbors, we have a 3-NN classifier, and if we use k neighbors, we have a k -NN classifier.

Let's revisit the hypothetical dataset of Figure 3-1, but generate a new version where the tight clusters are more spread out. We still have

2 features and 4 classes with 10 examples each. Let's set $k = 3$, a typical value. To assign a label to a new sample, we plot the sample in the feature space and then find the three closest training data points to it. Figure 3-4 shows the three nearest neighbors for three unknown samples.

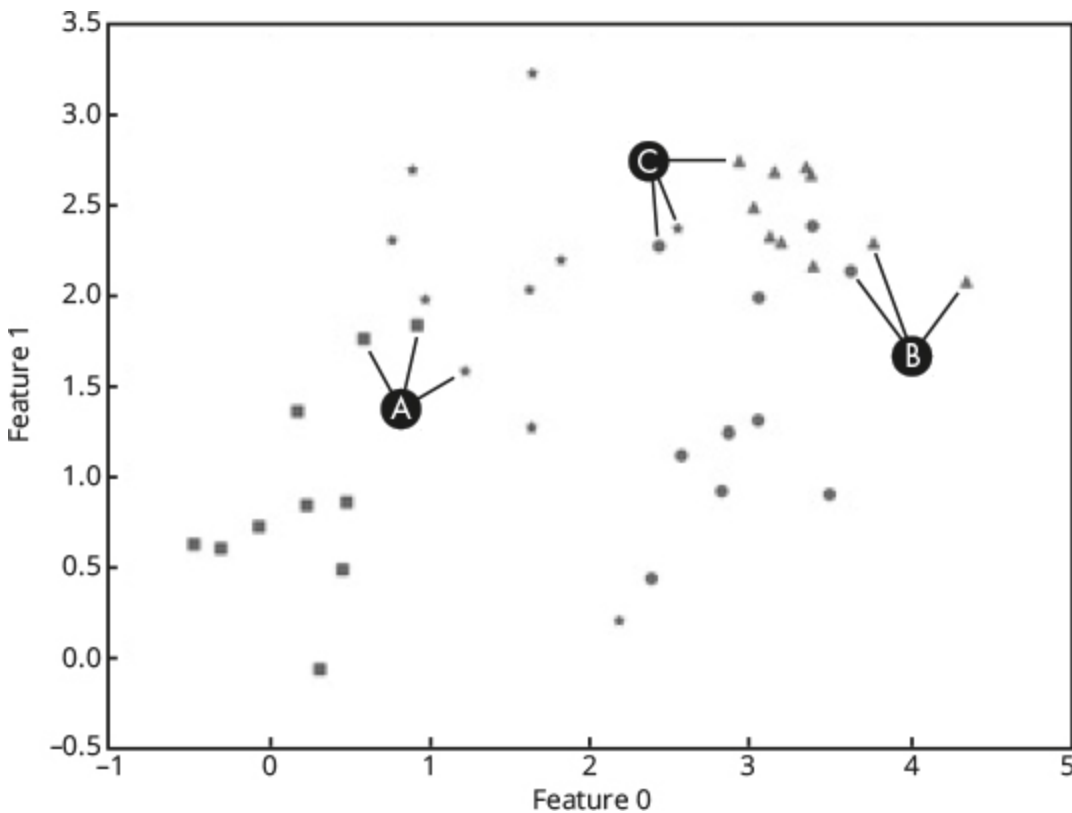


Figure 3-4: Applying k -NN for $k = 3$ to three unknown samples A, B, and C

The three training data points closest to sample A are square, square, and star. Therefore, by majority vote, we assign sample A to the class square. Similarly, the three closest training data points for sample B are circle, triangle, and triangle. Therefore, we declare sample B to be of class triangle.

With sample C, each of the three closest training samples are from a different class: circle, star, and triangle. So, voting is a tie. When this happens, the k -NN implementation has to make a choice. The simplest option is to select the class label at random, since we might argue that any of the three is equally as likely. Alternatively, we might believe a little more strongly in the value of the distance between the unknown

sample and the training data and select the one with the shortest distance. In this case, we'd label sample C with class star.

The beauty of a k -NN classifier is that the training data *is* the model; no training step is necessary. Of course, the training data must be carried around with the model and, depending on the size of the training set, finding the k -nearest neighbors for a new input sample might be computationally expensive. People have worked for decades to try to speed up the neighbor search or store the training data more efficiently, but in the end, the curse of dimensionality remains an issue.

Some k -NN classifiers have performed very well: If the dimensionality of the feature space is small enough, k -NN might be attractive. We need to have a balance between training data size, which leads to better performance but more storage and more laborious searching for neighbors, and the dimensionality of the feature space. The same sort of scenario that might make nearest centroid a good fit will also make k -NN a good fit. However, k -NN is perhaps more robust with diffuse or overlapping class groups than with nearest centroid. If the samples for a class are split among several groups, k -NN will be superior to nearest centroid.

Naive Bayes

Widely used in natural language processing research, the *naive Bayes* classifier is simple to implement and straightforward to understand, though it requires some math. A description of what's happening will make the math understandable even if the notation is unfamiliar.

The technique uses Bayes' theorem (see "An Essay Towards Solving a Problem in the Doctrine of Chances" by Thomas Bayes, published in 1763). The theorem relates probabilities, and its modern formulation is as follows:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

This uses mathematical notation from probability theory that we need to review in order for you to understand how we'll use this theorem to

implement a classifier.

The expression $P(A|B)$ represents the probability that event A has occurred, given event B has already occurred. In this context, it's called the *posterior probability*. Similarly, $P(B|A)$ represents the probability that event B has occurred, given event A has occurred. We call $P(B|A)$ the *likelihood* of B, given A. Finally, $P(A)$ and $P(B)$ represent, respectively, the probability that event A has occurred, regardless of event B, and the probability that event B has occurred, regardless of event A. We call $P(A)$ the *prior probability* of A. $P(B)$ is the *evidence*, the probability of B happening regardless of A.

Bayes' theorem gives us the probability of something happening (event A) given that we already know something else has happened (event B). So how does this help us classify? We want to know whether a feature vector belongs to a given class. We know the feature vector, but we don't know the class. So if we have a dataset of m feature vectors, where each feature vector has n features, $x = \{x_1, x_2, x_3, \dots, x_n\}$, then we can replace the B in Bayes' theorem with each of the features in the feature vector. We can also replace A with y , the class label we want to assign to a new, unknown feature vector x . The theorem now looks like this:

$$P\left(y|x_1, x_2, x_3, \dots, x_n\right) = \frac{P(x_1, x_2, x_3, \dots, x_n|y)P(y)}{P(x_1, x_2, x_3, \dots, x_n)}$$

Let's explain this a bit. Bayes' theorem states that if we know the likelihood of having x be our feature vector given that y is the class, and we know how often class y shows up (this is $P(y)$, the prior probability of y), then we can calculate the probability that the class of the feature vector x is y . If we are able to do this for all the possible classes (that is, all the y values), we can select the highest probability and label the input feature vector x as belonging to that class, y .

Recall that a training dataset is a set of pairs, (x^i, y^i) , for a known feature vector, x^i , and a known class it belongs to, y^i . Here the i superscript is counting the feature vector and label pairs in the training dataset. Given a dataset like this, we can calculate $P(y)$ by making a

histogram of how often each class label shows up in the training set. We believe that the training set fairly represents the parent distribution of possible feature vectors so that we can use the training data to calculate the values we need to make use of Bayes' theorem. (See Chapter 1 for techniques to ensure that the dataset is a good one.)

Once we have $P(y)$, we need to know the likelihood, $P(x_1, x_2, x_3, \dots, x_n | y)$. Unfortunately, we can't calculate this directly. But all is not lost: We'll make an assumption that lets us move ahead.

We'll assume that each of the features in x is statistically independent; the fact that we measure a particular x_1 has nothing whatsoever to do with the values of any of the other $n - 1$ features. This isn't true always, or even most of the time, but in practice, it turns out that this assumption is often close enough to true that we can get by. This is why it's called *naive* Bayes, as it's naive to assume that the features are independent of one another. That assumption is most definitely not true, for example, when our input is an image, as its pixels are highly dependent on one another. Pick one at random, and the pixels next to it are almost certainly within a few values of it.

When two events are independent, their *joint probability*, the probability that they both happen, is simply the product of their individual probabilities. The independence assumption lets us change the likelihood portion of Bayes' theorem like so:

$$P(x_1, x_2, x_3, \dots, x_n | y) \approx \prod_{i=1}^n P(x_i | y)$$

The Π symbol means *multiplied together*, much like the Σ symbol means *added together*. The right side of the equation is saying that if we know the probability of measuring a particular value of a feature (say, feature x_i), given that the class label is y , we can get the likelihood of the entire feature vector x , given class label y , by multiplying each of the per-feature probabilities together.

If our dataset consists of categorical values, or discrete values like integers (for example, age), then we can use the dataset to calculate the

$P(x_i|y)$ values by building a histogram for each feature for each class. For example, say feature x_2 for class 1 has the following values:

7, 4, 3, 1, 6, 5, 2, 8, 5, 4, 4, 2, 7, 1, 3, 1, 1, 3, 3, 3, 0, 3,
4, 4, 2, 3, 4, 5, 2, 4, 2, 3, 2, 4, 4, 1, 3, 3, 3, 2, 2, 4, 6, 5,
2, 6, 5, 2, 6, 6, 3, 5, 2, 4, 2, 4, 5, 4, 5, 5, 2, 5, 3, 4, 3, 1,
6, 6, 5, 3, 4, 3, 3, 4, 1, 1, 3, 5, 4, 4, 7, 0, 6, 2, 4, 7, 4, 3,
4, 3, 5, 4, 6, 2, 5, 4, 4, 5, 6, 5

If this is the case, each value occurs with the following probability, which comes from dividing the number of times each value occurs by 100, the total number of values in the dataset:

0: 0.02
1: 0.08
2: 0.15
3: 0.20
4: 0.24
5: 0.16
6: 0.10
7: 0.04
8: 0.01

This histogram is exactly what we need to find $P(x_2|y = 1)$, the probability for feature 2 when the class label is 1. For example, we can expect a new feature vector of class 1 to have $x_2 = 4$ about 24 percent of the time, and $x_2 = 1$ about 8 percent of the time.

By building tables like this for each feature and each class label, we can complete our classifier for the categorical and discrete cases. For a new feature vector, we use the tables to find the probability that each feature would have that value. We multiply each of those probabilities together and then multiply by the prior probability of that class. This, repeated for each of the m classes in the dataset, will give us a set of m posterior probabilities. To classify the new feature vector, select the largest of these m values and assign the corresponding class label.

How do we calculate $P(x_i|y)$ if the feature values are continuous? We might bin the continuous values and then make tables, as in the discrete case, or we can make one more assumption. We need to make an assumption about the distribution of possible x_i feature values that we could measure. As most natural phenomena seem to follow a normal distribution (see Chapter 0), let's assume that the features all follow normal distributions. We define a normal distribution by its mean value (μ , mu) and a standard deviation (σ , sigma). The *mean value* is the average value we'd expect if we drew samples from the distribution repeatedly. The *standard deviation* is a measure of how wide the distribution is—how spread out it is around the mean value.

Mathematically, we want to replace each $P(x_i|y)$ like so for each feature in our feature vector:

$$P(x_i|y) \approx N(\mu_i, \sigma_i)$$

Here $N(\mu_i, \sigma_i)$ indicates a normal distribution centered around a mean value (μ) and defined by a spread (σ).

We don't really know the exact μ and σ values, but we can approximate them from the training data. For example, assume that the training data consists of 25 samples, where the class label is 0. Further, assume that the following are the values of feature 3 (that is, x_3) in those cases:

0.21457111,	4.3311102,	5.50481251,	0.80293956,	2.5051598,
2.37655204,	2.4296739,	2.84224169,	-0.11890662,	3.18819152,
1.6843311,	4.05982237,	4.14488722,	4.29148855,	3.22658406,
6.45507675,	0.40046778,	1.81796124,	0.2732696,	2.91498336,
1.42561983,	2.73483704,	1.68382843,	3.80387653,	1.53431146

We'd use $\mu_3 = 2.58$ and $\sigma_3 = 1.64$ when setting up the normal distribution for feature 3 for class 0, since the average of these values is 2.58 and the standard deviation is 1.64.

When a new unknown sample is given to the classifier, we compute the probability of the given x_3 happening if the actual class was class 0

by using the following equation:

$$P\left(x_3 \middle| y = 0\right) = \frac{1}{\sigma_3 \sqrt{2\pi}} e^{-\frac{1}{2} \left(\frac{x_3 - \mu_3}{\sigma_3}\right)^2}$$

This equation comes from the definition of a normal distribution with mean μ and standard deviation σ . It says that the likelihood of a particular feature value, given the class is y , is distributed around the mean value we measured from the training data according to the normal distribution. This is an assumption we are making on top of the independence assumption between features.

We use this equation for each of the features in the unknown feature vector. We then multiply the resulting probabilities together and multiply that value by the prior probability of class 0 happening. We repeat this process for each of the classes. In the end, we'll have m numbers, the probabilities of the feature vector belonging to each of the m classes. To make a final decision, we'd do what we did before: select the largest of these probabilities and label the input as being of the corresponding class.

Note that we ignored the denominator of Bayes' theorem because it's a constant across all the calculations, and since we always select the largest posterior probability, we really don't care whether we divide each value by a constant. We'll select the same class label regardless.

For the discrete case, it's possible that our training set does not have any instances of a value that rarely shows up. We ignored that too, but it is a problem; if the value never shows up, the $P(x_i|y)$ we use would be 0, making the entire posterior probability 0. This often happens in natural language processing when a particular word is rarely used. The Laplace smoothing technique gets around this, but for our purposes, we claim that a "good" training set represents *all* possible values for the features and simply press ahead. The `sklearn.MultinomialNB` naive Bayes classifier for discrete data uses Laplace smoothing by default.

Tree Classifiers

Tree classifiers work by asking a series of yes-or-no questions about the unknown feature vector. The answer to each question moves from the root of the tree to its leaves, which contain the class label to assign based on the series of questions leading from the root to the leaf. Let's work through a basic example.

The left side of Figure 3-5 shows an X-ray image of a puppy with a malformed right hip socket. Since the puppy is on its back in the X-ray, the right hip socket is on the left side of the image. The right side of the figure shows the corresponding histogram of the pixel intensities (8-bit values, $[0, 255]$).

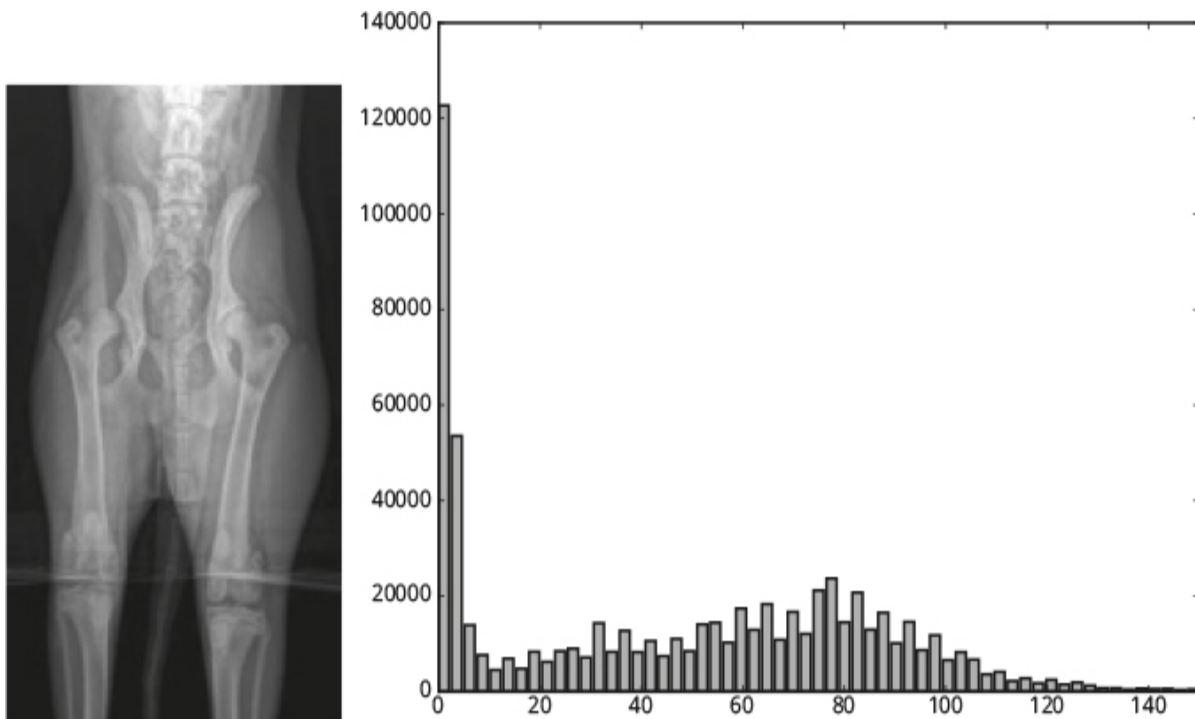


Figure 3-5: An X-ray image of a puppy (left) and the corresponding histogram of 8-bit pixel values $[0, 255]$ (right)

This histogram has two modes, corresponding to the dark background and the lighter-intensity X-ray data.

If we want to classify each pixel of the image into either background or X-ray, we can do so with the following rule: “If the pixel intensity is less than 11, call the pixel background.” This rule implements a decision made about the data based on one of the features—in this case, the pixel

intensity value. Simple decisions like this are at the heart of decision trees, the classification algorithm we'll explore in this section.

For completeness, if we apply the decision rule to each pixel in the image and output 0 or 255 (maximum pixel value) for background versus X-ray data, we get a mask revealing which pixels are part of the image, as Figure 3-6 shows.

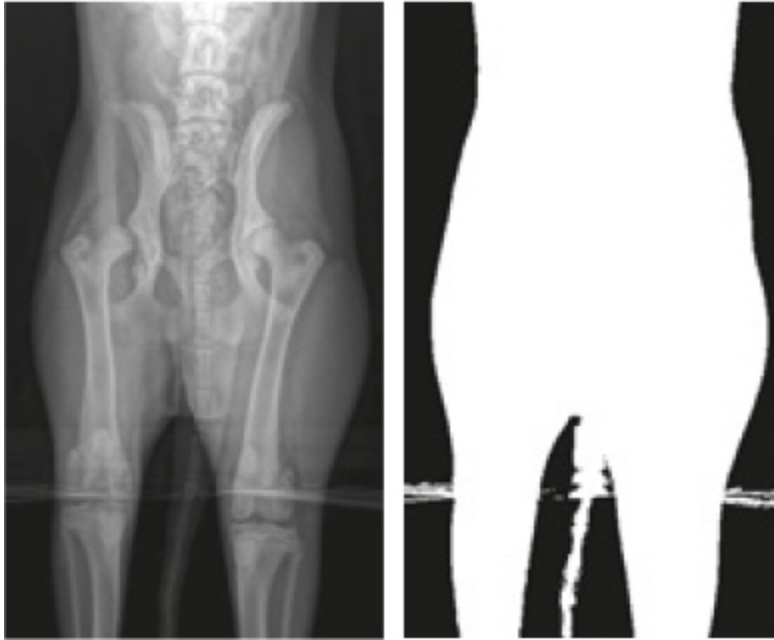


Figure 3-6: An X-ray image of a puppy (left) and the corresponding pixel mask generated by the decision rule, where white pixels are part of the X-ray image (right)

A *decision tree* is a set of nodes. The nodes either define a condition and branch based on the truth or falsehood of the condition, or select a particular class. Nodes that do not branch are *leaf nodes*. Decision trees are called *trees* because, especially for the binary case we'll consider here, they branch like trees. Figure 3-7 shows a decision tree learned by the `sklearn DecisionTreeClassifier` class for the full iris dataset using the first three features (see Chapter 2).

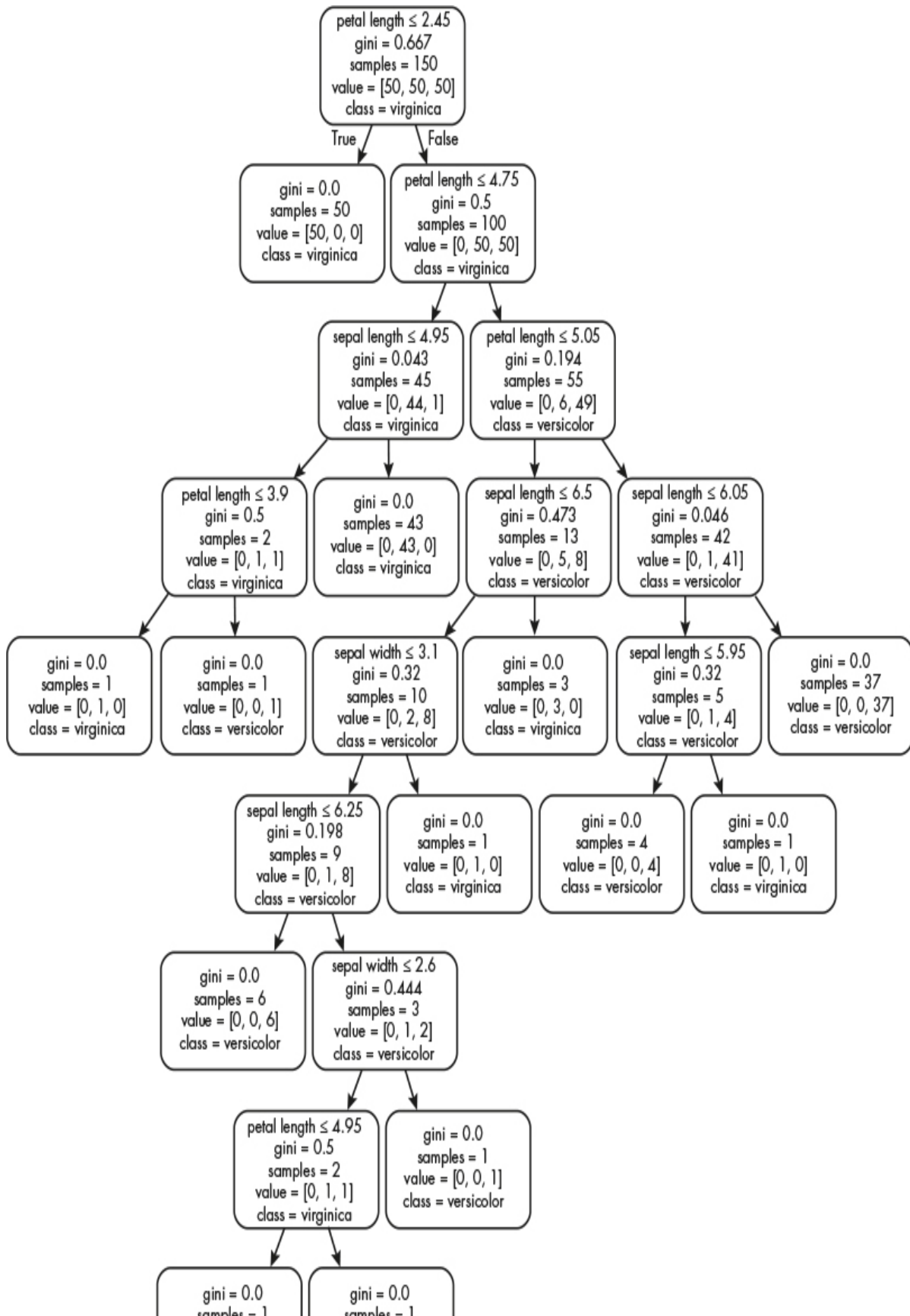


Figure 3-7: A decision tree classifier for the Iris dataset

By convention, the first node in the tree, the *root*, is drawn at the top. For this tree, the root node asks the question, “Is the petal length ≤ 2.45 ?” If it is, the left branch is taken, and the tree immediately reaches a leaf node and assigns a label of *virginica* (class 0). We’ll discuss the other information in the nodes shortly.

If the petal length is not ≤ 2.45 , the right branch is taken, leading to a new node that asks, “Is the petal length ≤ 4.75 ?” If so, we move to a node that asks a question about the sepal length. If not, we move to the right node and consider the petal length again. This process continues until a leaf is reached, which determines the class label.

This process is exactly how we use a decision tree after it’s created. For any new feature vector, the series of questions is asked starting with the root node, and the tree is traversed until a leaf node is reached to decide the class label. This is a human-friendly way to move through the classification process, which is why decision trees are handy when the *why* of the class assignment is as important to know as the actual class assignment. A decision tree can explain itself.

Using a decision tree is simple enough, but how is the tree created in the first place? Unlike the simple algorithms in the previous sections, the treebuilding process is more involved. We’ll follow the main steps to build some intuition as to what goes into defining the tree.

Recursion Primer

Before we talk about the decision-tree algorithm, however, we need to discuss the concept of recursion. Tree-like data structures and recursion go hand in hand. The essence of a recursive algorithm is that it repeats itself at different levels. When implemented as a function in a programming language, this generally means that the function calls itself on a smaller version of the problem. If the function calls itself indefinitely, we’ll have an infinite loop, so the recursion needs a stopping condition—something that says we no longer need to recurse.

Let’s introduce the idea of recursion mathematically. The factorial of an integer, n , denoted $n!$, is defined as follows:

$$n! = n(n-1)(n-2)(n-3) \dots (n-n+1)$$

This just means multiply together all the integers from 1 to n . By definition, $0! = 1$. Therefore, the factorial of 5 is 120:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

If we look at $5!$, we see that it is nothing more than $5 \times 4!$ or, in general, that $n! = n \times (n-1)!$. Now let's write a Python function to calculate factorials recursively using this insight. The code is a hallmark of many recursive functions, as Listing 3-1 shows.

```
def fact(n):
    if (n <= 1):
        return 1
    else:
        return n*fact(n-1)
```

Listing 3-1: Calculating the factorial

The code is a direct implementation of the rule that the factorial of n is n times the factorial of $n-1$. To find the factorial of n , we first ask if n is 1. If it is, we know the factorial is 1 so we return 1; this is our stopping condition. If n is not 1, we know that the factorial of n is n times the factorial of $n-1$, which we find by calling `fact` with $n-1$ as the argument.

Decision Trees

The algorithm to build a decision tree is also recursive. Let's walk through what happens at a high level. The algorithm starts with the root node, determines the proper rule for that node, and then calls itself on the left and right branches. The call to the left branch starts again as if the left branch is the root node. This continues until a stopping condition is met.

For a decision tree, the stopping condition is a leaf node (we'll discuss how a decision tree knows whether to create a leaf node next). Once a leaf node is created, the recursion terminates, and the algorithm

returns to that leaf's parent node and calls itself on the right branch. The algorithm then starts again as if the right branch were the root node. Once both recursive calls terminate, and a node's left and right subtrees are created, the algorithm returns to that node's parent, and so on until the entire tree is constructed.

Let's get a little more specific. How is the training data used to build the tree? When the root node is defined, all the training data is present—say, all n samples. This set of samples is used to pick the rule the root node implements. Once that rule has been selected and applied to the training samples, we have two new sets of samples: one for the left side (the true side) and one for the right side (the false side).

Recursion then works with these nodes, using their respective set of training samples, to define the rule for the left and right branches. Every time a branch node is created, the training set for that branch node gets split into samples that meet the rule and samples that don't meet the rule. A leaf node is declared when the set of training samples is either too small, of a sufficiently high proportion of one class, or the maximum tree depth has been reached.

How do we select the rule for a branch node? The rule relates a single input feature, like the petal length, to a particular value. The decision tree is a *greedy* algorithm, meaning that at every node, it selects the best rule for the current set of information available to it. In this case, this is the current set of training samples available to the node. The best rule is the one that best separates the classes into two groups.

This implies that we need a way to select possible candidate rules and that we have a way to determine that a candidate rule is “best.” The decision tree algorithm uses brute force to locate candidate rules. It runs through all possible combinations of features and values, making continuous values discrete by binning, and evaluates the purity of the left and right training sets after the rule is applied. The best-performing rule is the one kept at that node.

We determine what is *best performing* by the purity of the split into left and right training sample subsets. One way to measure *purity* is to use the *Gini index*; this is the metric sklearn uses. We calculate the Gini index as follows:

$$\text{Gini} = \sum_{i \neq j} P(y_i)P(y_j) = 1 - \sum_i P^2(y_i)$$

Here $P(y_i)$ is the fraction of training examples in the subset for the current node that are of class i . A perfect split between classes, all of one class and none of the other, results in a Gini index of 0. A 50-50 split has a Gini index of 0.5. The algorithm seeks to minimize the Gini index at each node by selecting the candidate rule that results in the smallest index.

For example, in Figure 3-7, the right-hand node below the root has a Gini index of 0.5. This means that the rule for *the node above*, the root, results in a subset of the training data that has petal length > 2.45 , and that subset will be evenly divided between classes 1 and 2. This is the meaning of the value line in the node text. It shows the number of training samples in the subset that defined the node. The class line is the class that would be assigned if the tree were stopped at that node. It's simply the label of the class that has the largest number of training samples in the node's subset. When the tree is used on new, unknown samples, it's run from the root to a leaf, always.

Random Forests

Decision trees are useful when the data is discrete or categorical or has missing values. Continuous data needs to be binned first (sklearn does this for us). Decision trees have a bad habit, however, of overfitting the training data. Therefore, they are likely to learn meaningless statistical nuances of the training data that we happened to use, instead of learning meaningful general features of the data that are useful when applied to unknown data samples. Decision trees also grow very large, as the number of features grows, unless managed by depth parameters.

We can mitigate decision-tree overfitting by using random forests. In fact, unless the problem is simple, we probably want to use a random forest from the start. The following three concepts lead from a decision tree to a random forest:

- Ensembles of classifiers and voting between them

- Resampling of the training set by selecting samples *with replacement*
- Selection of random feature subsets

If we have a set of classifiers, each trained on different data or of a different type, like a k -NN and a naive Bayes, we can use their outputs to vote on the actual category to assign to any particular unknown sample. This is an *ensemble*, and it will generally improve performance over that of any individual classifier, though with diminishing returns as the number of classifiers increases. We can employ a similar idea and imagine an ensemble, or *forest*, of decision trees, but unless we do something more with the training set, we'll have a forest of *exactly* the same tree because a particular set of training examples always leads to the exact same decision tree. The algorithm to create a decision tree is deterministic: It always returns the same results. As an aside, sklearn's approach does introduce some randomness, but a basic decision tree does not.

You might deal with the particular statistical nuances of the training set you have to work with by selecting a new training set from the original training set while allowing the same training-set sample to be selected more than once. This is *selection with replacement*. Think of it as choosing colored marbles from a bag, but before you select the next marble, you put the one you just selected back in the bag so you might pick it again. A new dataset selected in this way is a *bootstrap sample*. *Bagging* is the process of building a collection of new datasets via repeated sampling. The models built from this collection of resampled datasets build the random forest.

If we train multiple trees, each with a resampled training set with replacement, we'll get a forest of trees, each one slightly different from the others. This alone, along with ensemble voting, will probably improve matters. However, there's one issue. Any features that are highly predictive will dominate, and the resulting forest of trees will be very similar to one another and therefore suffer from very similar weaknesses. This is where the *random* of random forest comes into play.

Instead of just bagging, which changes the distribution of samples in the per-tree training set but not the set of features examined, what if we also randomly selected, for each tree in the forest, a subset of the features themselves and trained on only those features? Doing this breaks the correlation between the trees and increases the overall robustness of the forest. In practice, if we have n features per feature vector, each tree will randomly select $\sqrt{n}$ of them over which to build the tree. Random forests are supported in sklearn as well.

Support Vector Machines

Our final classical machine learning model is the one that held neural networks at bay for most of the 1990s, the *support vector machine (SVM)*. If the neural network is a highly data-driven, empirical approach to generating a model, the SVM is a highly elegant and mathematically founded approach. We'll discuss the performance of an SVM at a conceptual level, as the mathematics involved is beyond what we want to introduce here. If you're so inclined, the classic reference is "Support-Vector Networks" by Corinna Cortes and Vladimir Vapnik (1995).

We can summarize what an SVM is doing by gaining intuition about the concepts of margins, support vectors, optimization, and kernels. Let's look at each concept in turn.

Margins

Figure 3-8 shows a two-class dataset with two features.

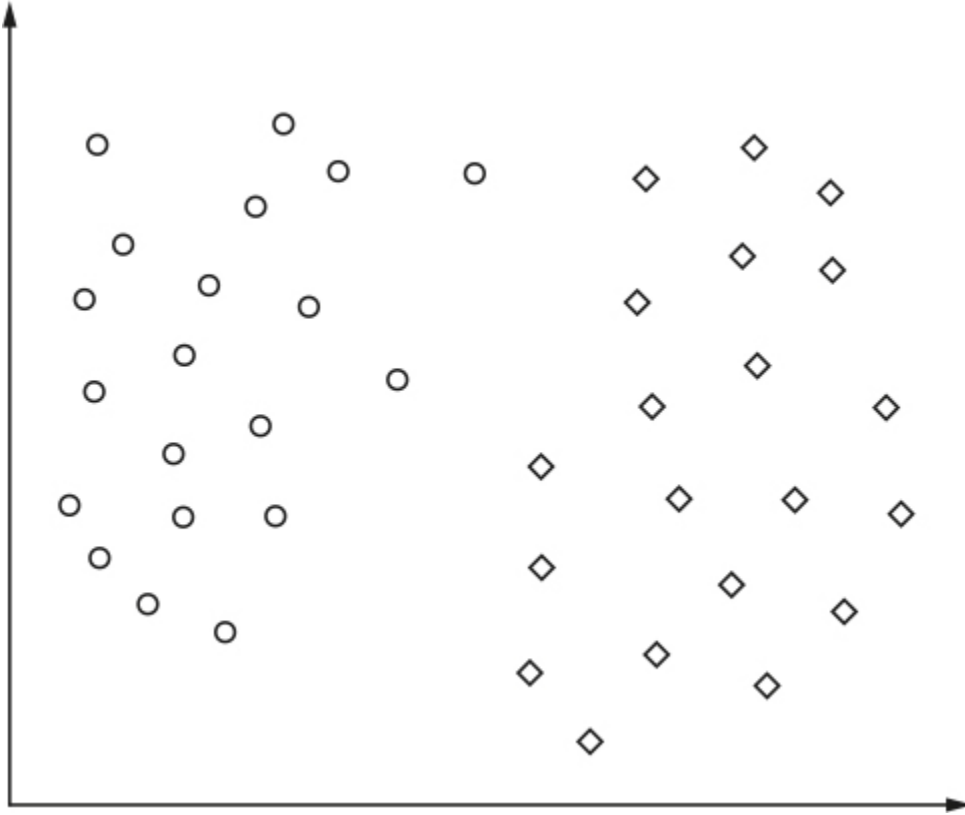


Figure 3-8: A toy dataset with two classes, circles and diamonds, and two features, x-axis and y-axis

I've plotted each sample in the dataset with feature 1 along the x-axis and feature 2 along the y-axis. Circles represent class 0, and diamonds represent class 1. This is a contrived dataset, one that's easily separated by plotting a line between the circles and the diamonds.

Think of a classifier as locating one or more planes that split the space of the training data into homogeneous groups. In the case of Figure 3-8, the separating "plane" is a line. If we had three features, the separating plane would be a 2D plane. With four features, the separating plane would be 3D, and for n dimensions the separating plane is $n - 1$ dimensional. Since the plane is multidimensional, we refer to it as a *hyperplane* and say that the goal of the classifier is to separate the training feature space into groups by using hyperplanes.

Looking at Figure 3-8, we can imagine an infinite set of lines that separate the training data into two groups, with all of class 0 on one side and all of class 1 on the other. Which one do we want to use? Well, let's think a bit about what the position of a line separating the two classes

implies. If we draw a line more to the right side, just before any of the diamonds, we'll have separated the training data, but only barely so. Recall, we're using the training data as a surrogate for the true distribution of samples of each class. The more training data we have, the more faithfully we'll know that true distribution. However, we don't really know it.

A new, unknown sample, which must be of class 0 or class 1, will fall somewhere on the graph. It's reasonable to believe that there are class 1 (diamond) samples in the wild that fall even closer to the circles than any of the samples in the training set. If the separating line is too close to the diamonds, we run the risk of calling valid class 1 samples *class 0* because the separating line is too far to the right. We can make a similar claim if we place the separating line very close to the class 0 points (circles). Then we run the risk of mislabeling class 0 samples as *class 1*.

Therefore, in the absence of more training data, it seems most reasonable to choose the separating line that is as far from both classes as possible. This is the line that is farthest from the rightmost circles while still being as far to the left of the diamonds as possible. This line is the *maximal margin location*, where the *margin* is the distance from the closest sample points. Figure 3-9 shows the maximal margin location as the heavy line, with the two dotted lines indicating the maximum margin.

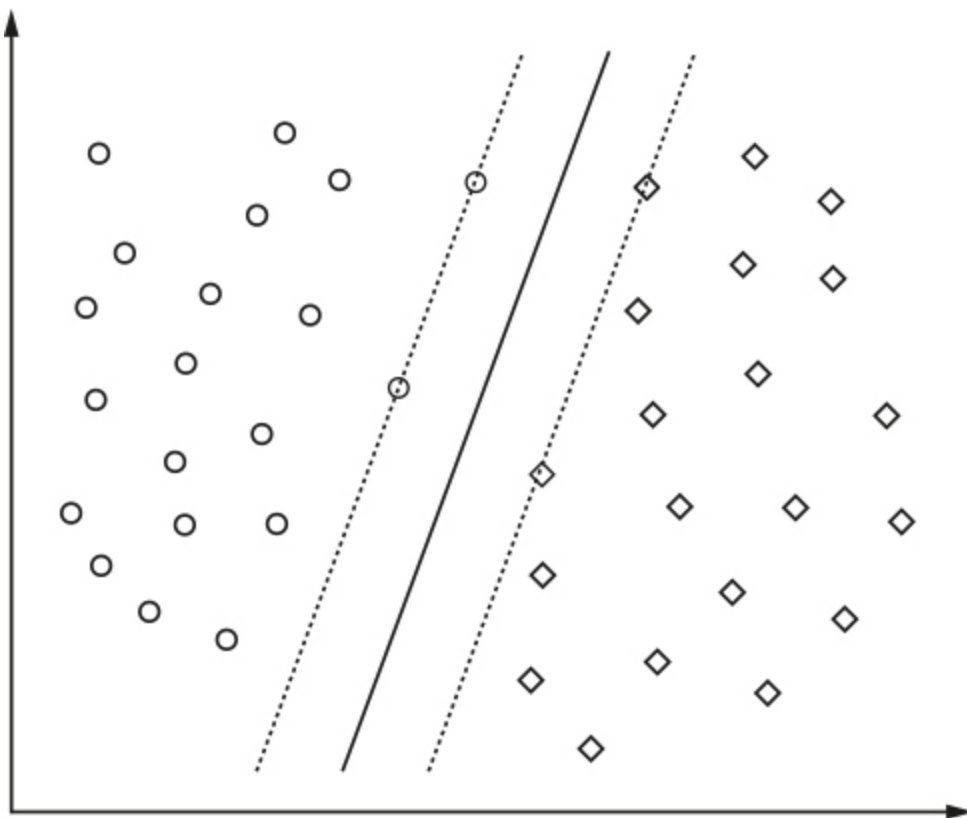


Figure 3-9: The toy dataset of Figure 3-8 with the maximal margin separating line (heavy) and the maximum margins (dotted)

The goal of an SVM is to locate the maximum margin position, as this is the location where we can be most certain to not misclassify new samples, given the knowledge gained from the training set.

Support Vectors

Look again at Figure 3-9, and note the four training data points on the margin. These are the training samples that define the margin, or in other words, support the margin; hence, they are support vectors. This is the origin of the name *support vector machine*.

The support vectors define the margin, but how can we use them to locate the margin position? Here is where we'll simplify the presentation to avoid a lot of complex vector mathematics that will only muddy the waters. For a more mathematical treatment, see "A Tutorial on Support Vector Machines for Pattern Recognition" by Christopher Burges (1998).

Optimization

Mathematically, we can find the maximum margin hyperplane by solving an optimization problem. Recall that in an optimization problem, we have a quantity that depends on certain parameters, and we want to find the set of parameter values that makes the quantity as small or as large as possible.

In the SVM case, we specify the orientation of the hyperplane by a vector, $\vec{w}$. There is also an offset, b , which we must find. Finally, before we can do the optimization, we need to change the way we specify the class labels. Instead of using 0 or 1 for y_i , the label of the i th training sample, x_i , we'll use -1 or $+1$. This lets us define the condition of the optimization problem more simply.

Mathematically, we want to find $\vec{w}$ and b so that the quantity $1/2 \|\vec{w}\|^2$ is as small as possible, given that $y_i (\vec{w} \cdot \vec{x} - b) \geq 1$ for all y_i labels and x_i training vectors in the dataset. We can solve this sort of optimization problem via a technique called *quadratic programming*. (We're ignoring another important mathematical step here: The actual optimization problem uses a Lagrangian to solve the dual form, but again, we're avoiding muddying the waters too much.)

The preceding formulation is used when the dataset, assumed to have only two classes, can be separated by a hyperplane. This is the linearly separable case. In reality, not every dataset can be separated this way. So, the full form of the optimization problem includes a fudge factor, C , which affects the size of the margin found. This factor shows up in the sklearn SVM class and needs to be specified to a certain level. From a practical point of view, C is a *hyperparameter* of the SVM, a value that we need to set in order to get the SVM to train properly. The right value of C is problem dependent. In general, any parameter of a model that is not learned by the model but must be set to use the model, like C for an SVM or k for k -NN, is a hyperparameter.

Kernels

We need to introduce one more mathematical concept, with suitable handwaving. The preceding description is for a linear SVM and uses the training data directly (the $\vec{x}$'s). The nonlinear case maps the training data to another space by passing it through a function, typically called $\phi(\vec{x})$, that produces a new version of the training data vector, $\vec{x}$.

The SVM algorithm uses inner products, $\vec{x}^T \vec{z}$, which means that the mapped version uses $\phi(\vec{x})^T \phi(\vec{z})$. In this notation, vectors are thought of as a column of numbers so that T , the transpose, produces a row vector. Then normal matrix multiplication of a $1 \times n$ row vector and an $n \times 1$ column vector results in a 1×1 output, which is a scalar. The inner product is sometimes written as follows:

$$K(\vec{x}, \vec{z}) = \phi(\vec{x})^T \phi(\vec{z})$$

The function $K(\vec{x}, \vec{z})$ is a *kernel*. The linear kernel is simply $\vec{x}^T \vec{z}$, but other kernels are possible. The Gaussian kernel, also known as a radial basis function (RBF) kernel, is a popular one. In practical use, this kernel introduces a new parameter, apart from C , which is γ . This parameter relates to how spread out the Gaussian kernel is around a particular training point, with smaller values extending the range of influence of the training sample. Typically, we use a grid search over C and, if using the RBF kernel, γ , to locate the best-performing model.

To summarize, a support vector machine uses the training data, mapped through a kernel function, to optimize the orientation and location of a hyperplane that produces the maximum margin between the hyperplane and the support vectors of the training data. The user needs to select the kernel function and associated parameters like C and γ so that the model best fits the training data.

SVMs dominated machine learning in the 1990s and early 2000s, before the advent of deep learning. This is because they're trained efficiently and don't need extensive computational resources to be

successful. Since the arrival of deep learning, however, SVMs have fallen somewhat by the wayside because powerful computers have enabled neural networks to do what previously was not possible with more limited computing resources. Still, SVMs have a place at the table. One popular early deep learning approach uses a large neural network trained on a particular dataset as a preprocessor for a different dataset with an SVM trained on the output of the neural network (minus the top layers).

Summary

This chapter introduced six of the most common classic machine learning models: nearest centroid, k -NN, naive Bayes, decision tree, random forest, and SVMs. These models have been used for decades and are still relevant if the conditions they support best are present. At times, the classic model is still the correct choice. An experienced machine learning practitioner will know when to fall back to the classics.

In the next chapter, we'll use each of these models, via `sklearn`, to perform experiments that will build your intuition of how the models work and when to use them.

4

EXPERIMENTS WITH CLASSICAL MODELS



Chapter 3 introduced several classical machine learning models. In this chapter, we'll take the datasets we built in Chapter 2 and use them with these models to see how well they perform.

We'll use `sklearn` to create the models and compare them by looking at how well they do on the held-out test sets. This will provide an overview of working with `sklearn` and help you build intuition about how the various models perform relative to one another. We'll use three datasets: the iris dataset, both original and augmented; the breast cancer dataset; and the vector form of the MNIST handwritten digits dataset.

Experiments with the Iris Dataset

We'll start with the iris dataset, which has four continuous features (the measurements of the sepal length, sepal width, petal length, and petal width) and three classes (representing the three iris species). The dataset has 150 samples, 50 in each class. In Chapter 2, we applied PCA augmentation to the dataset, so we'll work with two versions: the original 150 samples and the 1,200 augmented training samples. Both can use the same test set.

We'll use `sklearn` to implement versions of our models from Chapter 3: nearest centroid, k -NN, naive Bayes, decision tree, random forest, and SVM. We'll quickly see how powerful and elegant the `sklearn` toolkit is since all our

tests are virtually identical across the models; only the class that's instantiated changes.

Testing the Classical Models

The code for our initial tests is in Listing 4-1.

```
iris_experiments.py
import numpy as np
from sklearn.neighbors import NearestCentroid, KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB, MultinomialNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

❶ def run(x_train, y_train, x_test, y_test, clf):
    clf.fit(x_train, y_train)
    print("    predictions  :", clf.predict(x_test))
    print("    actual labels:", y_test)
    print("    score = %0.4f" % clf.score(x_test, y_test))
    print()

def main():
    ❷ x = np.load("../data/iris/iris_features.npy")
    y = np.load("../data/iris/iris_labels.npy")
    N = 120
    x_train = x[:N]; x_test = x[N:]
    y_train = y[:N]; y_test = y[N:]
    ❸ xa_train=np.load("../data/iris/iris_train_features_augmented.npy")
    ya_train=np.load("../data/iris/iris_train_labels_augmented.npy")
    xa_test =np.load("../data/iris/iris_test_features_augmented.npy")
    ya_test =np.load("../data/iris/iris_test_labels_augmented.npy")

    print("Nearest centroid:")
    ❹ run(x_train, y_train, x_test, y_test, NearestCentroid())
    print("k-NN classifier (k=3):")
    run(x_train, y_train, x_test, y_test,
        KNeighborsClassifier(n_neighbors=3))
    print("Naive Bayes classifier (Gaussian):")
    ❺ run(x_train, y_train, x_test, y_test, GaussianNB())
    print("Naive Bayes classifier (Multinomial):")
    run(x_train, y_train, x_test, y_test, MultinomialNB())
```

```

❹ print("Decision tree classifier:")
    run(x_train, y_train, x_test, y_test, DecisionTreeClassifier())

    print("Random forest classifier (estimators=5):")
    run(xa_train, ya_train, xa_test, ya_test,
        RandomForestClassifier(n_estimators=5))

❺ print("SVM (linear, C=1.0):")
    run(xa_train, ya_train, xa_test, ya_test, SVC(kernel="linear", C=1.0))
    print("SVM (RBF, C=1.0, gamma=0.25):")
    run(xa_train, ya_train, xa_test, ya_test,
        SVC(kernel="rbf", C=1.0, gamma=0.25))
    print("SVM (RBF, C=1.0, gamma=0.001, augmented)")
    run(xa_train, ya_train, xa_test, ya_test,
        SVC(kernel="rbf", C=1.0, gamma=0.001))
❻ print("SVM (RBF, C=1.0, gamma=0.001, original)")
    run(x_train, y_train, x_test, y_test,
        SVC(kernel="rbf", C=1.0, gamma=0.001))

```

Listing 4-1: Classic models using the iris dataset

First, we import the necessary classes and modules. Notice that each class represents a single type of model (classifier). For the naive Bayes classifier, we're using two versions: the Gaussian version, `GaussianNB`, because the features are continuous values; and `MultinomialNB` for the discrete case to illustrate the effect of choosing a model that's inappropriate for the dataset we're working with. Because `sklearn` has a uniform interface for its classifiers, we can simplify this process by using the same function to train and test any particular classifier. That function is `run` ❶. We pass in the training features (`x_train`) and labels (`y_train`) along with the test features and labels (`x_test`, `y_test`). We also pass in the particular classifier object (`clf`).

Inside `run`, we fit the model to the data by calling `fit` with the training data samples and labels. This is the training step. After the model is trained, we can test how well it does by calling the `predict` method with the held-out test data. This method returns the predicted class label for each sample in the test data. We held back 30 samples from the original 150, so `predict` returns a vector of 30 class label assignments, which we print. Next, we print the actual test labels so we can compare them visually with the predictions. Finally, we use the `score` method to apply the classifier to the test data (`x_test`) by using the known test labels (`y_test`) to calculate the overall accuracy.

The accuracy is returned as a fraction between 0 and 1. If every test sample were given the wrong label, the accuracy would be 0. Even random guessing will do better than that, so a return value of 0 means something is amiss. Since the iris dataset has three classes, we'd expect a classifier that guesses the class at random to be right about one-third of the time and return a value close to 0.3333. We calculate the actual score as follows:

$$\text{score} = N_c / (N_c + N_w)$$

Here N_c is the number of test samples for which the predicted class is correct; that is, it matches the class label in `y_test`. N_w is the number of test samples where the predicted class does not match the actual class label.

Now that we have a way to train and test each classifier, we need to load the datasets and run a series of experiments by creating different classifier objects and passing them to `run`. Back inside `main`, we begin by loading the original iris dataset and separating it into train and test cases ❷. We also load the augmented iris dataset that we created in Chapter 2 ❸. By design, the two test sets are identical, so regardless of which training set we use, the test set is the same. This simplifies our comparisons.

We then define and execute the nearest-centroid classifier ❹. The output is as follows:

```
Nearest centroid:
predictions :[011202120211112202201101102211]
actual labels:[011202120211112202201101102211]
score = 1.0000
```

We've removed spaces to facilitate visually comparing the predicted and actual class labels. If an error has occurred, the corresponding value, 0–2, won't match in the two lines. The score is also shown. In this case, it's 1.0, which tells us that the classifier was perfect in its predictions on the heldout test set. This isn't surprising; the iris dataset is a simple one. Because the iris dataset was randomized at its creation in Chapter 2, you might get a different overall score. However, unless your randomization was particularly unfortunate, you should have a high test score.

Based on what you learned in Chapter 3, if the nearest-centroid classifier is perfect on the test data, we should expect that all the other more sophisticated models will likewise be perfect. This is generally the case here, but as we'll see,

careless selection of model type or model hyperparameter values results in inferior performance even from a more sophisticated model.

Look again at Listing 4-1, where we train a Gaussian naive Bayes classifier by passing an instance of `GaussianNB` to run ❸. This classifier is also perfect and returns a score of 1.0. This is the correct way to use continuous values with a naive Bayes classifier. What happens if we instead use the discrete case even though we have continuous features? This is the `MultinomialNB` classifier, which assumes the features are selected from a discrete set of possible values. For the iris dataset, we can get away with defining such a classifier because the feature values are nonnegative. However, because the features aren't discrete, this model isn't perfect and returns the following:

```
Naive Bayes classifier (Multinomial):
predictions :[011202220211122202202101102221]
actual labels:[011202120211112202201101102211]
score = 0.8667
```

Here we see that the classifier is only 86.7 percent accurate on our test samples. If we need discrete counts for the probabilities, why did this approach work at all in this case? The answer is evident in the sklearn source code for the `MultinomialNB` classifier. The method that counts feature frequencies per class uses `np.dot` so that even if the feature values are continuous, the output will be a valid number, though not an integer. Still, mistakes were made, so we should be careful to select the proper classifier type for the actual data we're working with.

The next model we train in Listing 4-1 is a decision tree ❹. This classifier is perfect on this dataset, as is the random forest trained next. Note, the random forest is using five estimators, meaning five random trees are created and trained; voting between the individual outputs determines the final class label. Note also that the random forest is trained on the augmented iris dataset, `xa_train`, because of the limited number of training samples in the unaugmented dataset.

We then train several SVM classifiers ❺, also on the augmented dataset. Recall that SVMs have two parameters we control: the margin constant, `c`, and `gamma` used by the Gaussian kernel.

The first is a linear SVM, meaning we need a value for the margin constant (`c`). We define `c` to be 1.0, the default value for sklearn. This classifier is perfect on the test data, as is the following classifier using the Gaussian kernel, for

which we also set γ to 0.25. The `svc` class defaults to `auto` for `gamma`, which sets γ to $1/n$, where n is the number of features. For the iris dataset, $n = 4$ so $\gamma = 0.25$.

Next, we train a model with very small γ . The classifier is still perfect on the test data. Lastly, we train the same type of SVM, but instead of the augmented training data, we use the original training data ③. This classifier is not perfect:

```
SVM (RBF, C=1.0, gamma=0.001, original)
predictions :[0222020202222220220220220220]
actual labels:[011202120211112202201101102211]
score = 0.5667
```

In fact, it's rather dismal. It never predicts class 1 and is right only 56.7 percent of the time. This shows that data augmentation is valuable, as it turned a lousy classifier into a good one—at least as far as we can know from the small test set we're using!

Implementing a Nearest-Centroid Classifier

If we were stranded on a deserted island and didn't have access to `sklearn`, could we still quickly build a suitable classifier for the iris dataset? The answer is yes, as Listing 4-2 shows.

```
iris_centroids.py
import numpy as np

❶ def centroids(x, y):
    c0 = x[np.where(y == 0)].mean(axis=0)
    c1 = x[np.where(y == 1)].mean(axis=0)
    c2 = x[np.where(y == 2)].mean(axis=0)
    return [c0, c1, c2]

❷ def predict(c0, c1, c2, x):
    p = np.zeros(x.shape[0], dtype="uint8")
    for i in range(x.shape[0]):
        d = [((c0-x[i])**2).sum(),
              ((c1-x[i])**2).sum(),
              ((c2-x[i])**2).sum()]
        p[i] = np.argmin(d)
    return p
```

```

def main():
    ❸ x = np.load("../data/iris/iris_features.npy")
      y = np.load("../data/iris/iris_labels.npy")
      N = 120
      x_train = x[:N]; x_test = x[N:]
      y_train = y[:N]; y_test = y[N:]
      c0, c1, c2 = centroids(x_train, y_train)
      p = predict(c0, c1, c2, x_test)
      nc = len(np.where(p == y_test)[0])
      nw = len(np.where(p != y_test)[0])
      acc = float(nc) / (float(nc)+float(nw))
      print("predicted:", p)
      print("actual    :", y_test)
      print("test accuracy = %0.4f" % acc)

```

Listing 4-2: A quick-and-dirty nearest-centroid classifier for the iris dataset

We load the iris data and separate it into train and test sets as before ❸. The `centroids` function returns the centroids of the three classes ❶. We calculate these by finding the per-feature means of each training sample of the desired class. This is all it takes to train this model. If we compare the returned centroids with those in the preceding trained `NearestCentroid` classifier (see the `centroids_member` variable), we get precisely the same values.

Using the classifier is straightforward, as `predict` shows ❷. First, we define the vector of predictions, one per test sample (`x`). The loop defines `d`, a vector of Euclidean distances from the current test sample, `x[i]`, to the three class centroids. The index of the smallest distance in `d` is the predicted class label (`p[i]`).

Let's unpack `d` a bit more. We set `d` to a list of three values, the distances from the centroids to the current test sample. The following expression is a bit dense:

```
((c0-x[i])**2).sum()
```

The phrase `c0-x[i]` returns a vector of four numbers—four because we have four features. These are the differences between the centroid of class 0 and the test sample feature value. This quantity is squared, which squares each of the four values. This squared vector is summed, element by element, to return the distance measure.

Strictly speaking, we're missing a final step. The actual distance between c_0 and $x[i]$ is the square root of this value. Since we're simply looking for the smallest distance to each of the centroids, we don't need to calculate the square root. The smallest value will still be the smallest value, whether we take the square root of all the values or not. Running this code produces the same output as we saw previously for the nearest-centroid classifier, which is encouraging.

The iris dataset is extremely simple, so we shouldn't be surprised by the excellent performance of our models even though we saw that careless selection of model type and hyperparameters will cause us trouble. Let's now look at a larger dataset with more features, one that wasn't meant as a toy.

Experiments with the Breast Cancer Dataset

The two-class breast cancer dataset we developed in Chapter 2 has 569 samples, each with 30 features, all measurements from a histology slide. These samples include 212 malignant cases (class 1) and 357 benign cases (class 0). Let's train our classic models on this dataset and see what sort of results we get. As all the features are continuous, we'll use the normalized version of the dataset. Recall that a normalized dataset is one where, per feature in the feature vector, each value has the mean for that feature subtracted and then is divided by the standard deviation:

$$x' = \frac{x - \bar{x}}{\sigma}$$

Normalization of the dataset maps all the features into the same overall range so that the value of one feature is similar to the value of another. This helps many model types and is a typical data-preprocessing step, as we discussed in Chapter 1.

Running Two Initial Tests

First, we'll do a quick run with a single test split, as we did in the previous section. The code is in Listing 4-3 and mimics the code we described previously, where we pass in the model instance, train it, and then score it using the testing data.

```
bc_experiments.py
import numpy as np
from sklearn.neighbors import NearestCentroid, KNeighborsClassifier
```

```

from sklearn.naive_bayes import GaussianNB, MultinomialNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

def run(x_train, y_train, x_test, y_test, clf):
    clf.fit(x_train, y_train)
    print("    score = %0.4f" % clf.score(x_test, y_test))
    print()

def main():
    x = np.load("../data/breast/bc_features_standard.npy")
    y = np.load("../data/breast/bc_labels.npy")
    ❶ N = 455
    x_train = x[:N]; x_test = x[N:]
    y_train = y[:N]; y_test = y[N:]

    print("Nearest centroid:")
    run(x_train, y_train, x_test, y_test, NearestCentroid())
    print("k-NN classifier (k=3):")
    run(x_train, y_train, x_test, y_test,
        KNeighborsClassifier(n_neighbors=3))
    print("k-NN classifier (k=7):")
    run(x_train, y_train, x_test, y_test,
        KNeighborsClassifier(n_neighbors=7))
    print("Naive Bayes classifier (Gaussian):")
    run(x_train, y_train, x_test, y_test, GaussianNB())
    print("Decision tree classifier:")
    run(x_train, y_train, x_test, y_test, DecisionTreeClassifier())
    print("Random forest classifier (estimators=5):")
    run(x_train, y_train, x_test, y_test,
        RandomForestClassifier(n_estimators=5))
    print("Random forest classifier (estimators=50):")
    run(x_train, y_train, x_test, y_test,
        RandomForestClassifier(n_estimators=50))
    print("SVM (linear, C=1.0):")
    run(x_train, y_train, x_test, y_test, SVC(kernel="linear", C=1.0))
    print("SVM (RBF, C=1.0, gamma=0.03333):")
    run(x_train, y_train, x_test, y_test,
        SVC(kernel="rbf", C=1.0, gamma=0.03333))

```

Listing 4-3: Initial models using the breast cancer dataset

As before, we load the dataset and split it into training and testing data. We keep 455 of the 569 samples for training (80 percent), and the remaining 114 samples are the test set (74 benign, 40 malignant). The dataset is already randomized, so we skip that step here. We then train nine models: nearest centroid (1), k -NN (2), naive Bayes (1), decision tree (1), random forest (2), linear SVM (1), and an RBF SVM (1). For the SVMs, we use the default C value, and for γ , we use $1/30 = 0.03333$ since we have 30 features. Running this code gives us the scores in Table 4-1.

Table 4-1: Breast Cancer Model Scores

Model type	Score
Nearest centroid	0.9649
3-NN classifier	0.9912
7-NN classifier	0.9737
Naive Bayes (Gaussian)	0.9825
Decision tree	0.9474
Random forest (5)	0.9298
Random forest (50)	0.9737
Linear SVM ($C = 1$)	0.9737
RBF SVM ($C = 1, \gamma = 0.03333$)	0.9825

The number in parentheses for the random-forest classifiers is the number of estimators (number of trees in the forest).

A few results jump out at us. First, perhaps surprisingly, the simple nearest-centroid classifier is right nearly 97 percent of the time. We also see that all the other classifiers are doing better than the nearest centroid, except for the decision tree and the random forest with five trees. Also somewhat surprisingly, the naive Bayes classifier does very well, matching the RBF SVM. The $k = 3$ nearest-neighbor classifier does best of all with 99 percent accuracy, even though we have 30 features, meaning our 569 samples are points scattered in a 30-dimensional space. Recall, a weakness of k -NN is the curse of dimensionality: It requires more training samples as the number of features

increases. The results with all the classifiers are good, which hints that the separation between malignant and benign is, for this dataset, distinct. The two classes using these features don't have much overlap.

If we run the code a second time, do we get the same scores? Would we expect not to? A second run gives us Table 4-2.

Table 4-2: Breast Cancer Scores, Second Run

Model type	Score
Nearest centroid	0.9649
3-NN classifier	0.9912
7-NN classifier	0.9737
Naive Bayes (Gaussian)	0.9825
Decision tree	0.9386
Random forest (5)	0.9474
Random forest (50)	0.9649
Linear SVM ($C = 1$)	0.9737
RBF SVM ($C = 1, \gamma = 0.03333$)	0.9825

I've highlighted the scores that changed. As the random forest is just that, random, we should expect different results from run to run. In sklearn, the decision-tree classifier randomly selects a feature and finds the best split, so different runs also lead to different trees. This is a variation on the basic decision-tree algorithm we discussed in Chapter 3.

All the other algorithms are fixed: For a given training dataset, they can lead to only one model. As an aside, because the SVM implementation in sklearn uses a random number generator, different runs will at times give slightly different results, but, conceptually, we'd expect the same model for the same input data. The tree-based classifiers, however, do change between training runs. We'll explore this variation more next. For now, we need to add some rigor to our quick analysis.

Testing the Effect of Random Splits

Let's change the split between training and testing data and see what happens to our results. We don't need to list all the code again since the only change is to how `x_train` and `x_test` are defined. Before splitting, we randomize the order of the full dataset, but do so by first fixing the pseudorandom number seed so that each run gives the same ordering to the dataset.

Looking again at Listing 4-3, insert the following code before ❶ so that we generate a fixed permutation of the dataset (`idx`):

```
np.random.seed(12345)
idx = np.argsort(np.random.random(y.shape[0]))
x = x[idx]
y = y[idx]
```

It's fixed because we fixed the pseudorandom number generator's seed value. We then reorder the samples (`x`) and labels (`y`) accordingly before splitting into train and test subsets as before. Running this code gives us the results in Table 4-3.

Table 4-3: Breast Cancer Scores After Randomizing the Dataset

Model type	Score
Nearest centroid	0.9474
3-NN classifier	0.9912
7-NN classifier	0.9912
Naive Bayes (Gaussian)	0.9474
Decision tree	0.9474
Random forest (5)	0.9912
Random forest (50)	1.0000
Linear SVM ($C = 1$)	0.9649
RBF SVM ($C = 1, \gamma = 0.03333$)	0.9737

These are entirely different from our earlier results. The k -NN classifiers are both equally good, the SVM classifiers are worse, and the 50-tree random

forest achieves perfection on the test set. Why are we getting all these different results from run to run?

We're seeing the effect of the random sampling that builds the train and test splits. The first split happened to use an ordering of samples that gave good results for one model type and less good results for other model types. The new split favors different model types. Both are correct. Recall what the dataset represents: a sampling from an unknown parent distribution that generates the data that we actually have. If we think in those terms, we see that the dataset we have is an incomplete picture of the true parent distribution. It has biases, though we don't know what they are necessarily, and is deficient in that there are parts of the parent distribution that the dataset doesn't represent well.

Further, when we split the data after randomizing the order, we might end up with a "bad" mix in the train or test portion—a mix of the data that does a poor job of representing the true distribution. If so, we might train a model to recognize a slightly different distribution that doesn't match the true distribution well, or the test set might be a bad mix and not be a fair representation of what the model has learned. This effect is even more pronounced when the proportion of the classes is such that one or more are rare and possibly not present in the train or test split. This is precisely the issue that caused us to introduce the idea of k -fold cross-validation in Chapter 1. With k -fold validation, we'll be sure to use every sample as both train and test at some point and buy ourselves some protection against a bad split by averaging across all the folds.

However, before we apply k -fold validation to the breast cancer dataset, we should notice one essential point. We modified the code of Listing 4-3 to fix the pseudorandom number seed so that we could reorder the dataset in exactly the same way each time we run. We then ran the code and saw the results. If we rerun the code, we'll get *exactly* the same output, even for the tree-based classifiers. This is not what we saw earlier. The tree classifiers are *stochastic*—they will generate a unique tree or forest each time—so we should expect the results to vary somewhat from run to run. But now they don't vary; we get the same output each time.

By setting the NumPy pseudorandom number seed explicitly, we fixed not only the ordering of the dataset, but also the ordering of the pseudorandom sequence that sklearn will use to generate the tree models. This is because sklearn is also using the NumPy pseudorandom number generator. This is a

subtle effect with potentially serious consequences and in a larger project might be very difficult to pick up as a bug.

The solution is to set the seed to a random value after we're done reordering the dataset. We can do this by adding one line after `y = y[idx]` so that the pseudorandom number generator is reset by using the system state, typically read from `/dev/urandom`:

```
np.random.seed()
```

Now when we run again, we'll get different results for the tree models, as before.

Adding k-Fold Validation

To implement k -fold validation, we first need to pick a value for k . Our dataset has 569 samples. We want to split it so that each fold has a decent number of samples because we want to make the test set a reasonable representation of the data. This argues toward making k small. However, we also want to average out the effect of a bad split, so we might want k to be larger.

As with most things in life, a balance must be sought. If we set $k = 5$, we'll get 113 samples per split (ignoring the final four samples, which should have no meaningful impact). This leaves 80 percent for training and 20 percent for test for each combination of folds, a reasonable approach to take. So, we'll use $k = 5$, but we'll write our code so that we can vary k if we want.

We already have an approach for training multiple models on a train/ test split. All we need to add is code to generate each of the k folds and then train the models on them. The code is in Listings 4-4 and 4-5, which show the helper functions and `main` function, respectively. Let's start with Listing 4-4.

bc_kfold.py

```
import numpy as np
from sklearn.neighbors import NearestCentroid
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB, MultinomialNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
import sys

def run(x_train, y_train, x_test, y_test, clf):
```

```

    clf.fit(x_train, y_train)
    return clf.score(x_test, y_test)

def split(x, y, k, m):
    ❶ ns = int(y.shape[0]/m)
    s = []
    for i in range(m):
        ❷ s.append([x[(ns*i):(ns*i+ns)],
                    y[(ns*i):(ns*i+ns)]])
    x_test, y_test = s[k]
    x_train = []
    y_train = []
    for i in range(m):
        if (i == k):
            continue
        else:
            a, b = s[i]
            x_train.append(a)
            y_train.append(b)
    ❸ x_train = np.array(x_train).reshape(((m-1)*ns, 30))
    y_train = np.array(y_train).reshape((m-1)*ns)

    return [x_train, y_train, x_test, y_test]

def pp(z, k, s):
    m = z.shape[1]
    print("%-19s: %0.4f +/- %0.4f | " % (s, z[k].mean(),
        z[k].std()/np.sqrt(m)), end='')
    for i in range(m):
        print("%0.4f " % z[k,i], end='')
    print()

```

Listing 4-4: Using k -fold validation to evaluate the breast cancer dataset (helper functions)

We begin by including all the modules we used before and then define three functions: `run`, `split`, and `pp`. The `run` function takes a train set, test set, and model instance; trains the model; and then scores the model against the test set. The `pp` function is a pretty-print function to show the per-split scores along with the average score across all the splits. The average is shown as the mean $\pm$ the standard error of the mean. Recall that an sklearn score is the

overall accuracy of the model on the test set, or the fraction of times that the model predicted the actual class of the test sample. Perfection is a score of 1.0, and complete failure is 0.0. Complete failure is rare because even random guessing gets it right a fraction of the time.

The only interesting function in Listing 4-4 is `split`. Its arguments are the full dataset, `x`, the corresponding labels, `y`, the current fold number, `k`, and the total number of folds, `m`. We'll divide the full dataset into m distinct sets, the folds, and use the k th fold as test while merging the remaining $m - 1$ folds into a new training set. First, we set the number of samples per fold ❶. The loop then creates a list of folds, `s`. Each element of this list contains the feature vectors and labels of the fold ❷.

The test set is simple, it's the k th fold, so we set those values next (`x_test`, `y_test`). The loop then takes the remaining $m - 1$ folds and merges them into a new training set, `x_train`, with labels, `y_train`.

The two lines after the loop are a bit mysterious ❸. When the loop ends, `x_train` is a *list*, each element of which is another list representing the feature vectors of the fold we want in the training set. So we first make a NumPy array of this list and then reshape it so that `x_train` has 30 columns, the number of features per vector, and $n_s(m - 1)$ rows, where n_s is the number of samples per fold. Thus, `x_train` becomes `x` minus the samples we put into the test fold, those of the k th fold. We also build `y_train` so that the correct label goes with each feature vector in `x_train`.

Listing 4-5 shows us how to use the helper functions.

bc_kfold.py

```
def main():
    x = np.load("../data/breast/bc_features_standard.npy")
    y = np.load("../data/breast/bc_labels.npy")
    idx = np.argsort(np.random.random(y.shape[0]))
    x = x[idx]
    y = y[idx]
    ❶ m = int(sys.argv[1])

    z = np.zeros((8, m))

    for k in range(m):
        x_train, y_train, x_test, y_test = split(x, y, k, m)
        z[0, k] = run(x_train, y_train, x_test, y_test,
                      NearestCentroid())
```

```

z[1, k] = run(x_train, y_train, x_test, y_test,
              KNeighborsClassifier(n_neighbors=3))
z[2, k] = run(x_train, y_train, x_test, y_test,
              KNeighborsClassifier(n_neighbors=7))
z[3, k] = run(x_train, y_train, x_test, y_test,
              GaussianNB())
z[4, k] = run(x_train, y_train, x_test, y_test,
              DecisionTreeClassifier())
z[5, k] = run(x_train, y_train, x_test, y_test,
              RandomForestClassifier(n_estimators=5))
z[6, k] = run(x_train, y_train, x_test, y_test,
              RandomForestClassifier(n_estimators=50))
z[7, k] = run(x_train, y_train, x_test, y_test,
              SVC(kernel="linear", C=1.0))

pp(z, 0, "Nearest"); pp(z, 1, "3-NN")
pp(z, 2, "7-NN");    pp(z, 3, "Naïve Bayes")
pp(z, 4, "Decision tree");    pp(z, 5, "Random forest (5)")
pp(z, 6, "Random forest (50)"); pp(z, 7, "SVM (linear)")

```

Listing 4-5: Using k -fold validation to evaluate the breast cancer dataset (main code)

The first thing we do in `main` is load the full dataset and randomize the ordering. The number of folds, m , is read from the command line ❶ and used to create the output array, `z`. This array holds the per-fold scores for each of the eight models we'll train, so it has the shape $8 \times m$. Recall, when running a Python script from the command line, any arguments passed after the script name are available in `sys.argv`, a list of strings. This is why the argument is passed to `int` to convert it to an integer ❷.

Next, we loop over the m folds, where k is the fold that we'll be using for test data. We create the split and then use it to train the eight model types we trained previously. Each call to `run` trains a model of the type passed in and returns the score found by running that model against the k th fold as test data. We store these results in `z`. Finally, we use `pp` to display the permodel type and per-fold scores along with the average score over all the folds.

A sample run of this code for $k = 5$, showing only the mean score across folds, gives the results in Table 4-4.

Table 4-4: Breast Cancer Scores as Mean over Five Folds

Model	Mean $\pm$ SE
Nearest centroid	0.9310 $\pm$ 0.0116
3-NN	0.9735 $\pm$ 0.0035
7-NN	0.9717 $\pm$ 0.0039
Naive Bayes	0.9363 $\pm$ 0.0140
Decision tree	0.9027 $\pm$ 0.0079
Random forest (5)	0.9540 $\pm$ 0.0107
Random forest (50)	0.9540 $\pm$ 0.0077
SVM (linear)	0.9699 $\pm$ 0.0096

Here we're showing the average performance of each model over all folds. This is the sort of performance we should expect, per model type, if we were to train the model using *all* the data in the dataset and test it against new samples from the same parent distribution. Indeed, in practice, we would do exactly this, as we can assume that the reason behind making the model in the first place is to use it for a certain purpose going forward.

If we run the code a second time with $k = 5$, a new set of outputs appears. Because we're randomizing the order of the dataset on every run (Listing 4-5), this makes a new set of splits and implies that each model will be trained on a different subset mix of the full dataset on each run. So, we should expect different results. Let's run the code 1,000 times with $k = 5$. Note, training this many models takes about 20 minutes on a standard desktop computer. For each run, we'll get an average score over the five folds. We then compute the mean of these averages, the *grand mean*. Table 4-5 shows the results.

Table 4-5: Breast Cancer Scores as Grand Mean over 1,000 Runs with Five Folds

Model	Grand mean $\pm$ SE
Nearest centroid	0.929905 $\pm$ 0.000056
3-NN	0.966334 $\pm$ 0.000113
7-NN	0.965496 $\pm$ 0.000110
Naive Bayes	0.932973 $\pm$ 0.000095

Model	Grand mean $\pm$ SE
Decision tree	0.925706 $\pm$ 0.000276
Random forest (5)	0.948378 $\pm$ 0.000213
Random forest (50)	0.958845 $\pm$ 0.000135
SVM (linear)	0.971871 $\pm$ 0.000136

These grand means are an indication of how well we'd expect each model to do against a new set of unknown feature vectors. The small standard errors of the mean are an indication of how well the mean value is known, not how well a model of that type trained on a dataset will necessarily perform. We use the grand mean to help us order the models so we can select one over another.

Ranking the models from highest score to lowest gives the following:

1. SVM (linear)
2. k -NN ($k = 3$)
3. k -NN ($k = 7$)
4. Random forest (50)
5. Random forest (5)
6. Naive Bayes (Gaussian)
7. Nearest centroid
8. Decision tree

This is interesting given that we might expect the SVM to be best but would likely assume the random forests to do better than k -NN. The decision tree was not as good as we thought and was less accurate than the nearest-centroid classifier.

Some comments are in order here. First, note that these results are derived from the training of 8,000 models on 1,000 orderings of the dataset. When we study neural networks, we'll see much longer training times. Experimenting with classical machine learning models is generally easy since each change to a parameter doesn't require a lengthy training session.

Second, we didn't try to optimize any of the model hyperparameters. Some hyperparameters are indirect, like assuming that the features are normally


```
z[7, k] = run(x_train, y_train, x_test, y_test,  
             KNeighborsClassifier(n_neighbors=15))
```

The grand mean of the scores for 1,000 repetitions of the five-fold crossvalidation code using a different random ordering of the full dataset each time gives the results in Table 4-6.

Table 4-6: Breast Cancer Scores as Grand Mean for Different k Values and Five-Fold Validation

k	Grand mean $\pm$ SE		
1	0.951301	$\pm$	0.000153
3	0.966282	$\pm$	0.000112
5	0.965998	$\pm$	0.000097
7	0.96520	$\pm$	0.000108
9	0.967011	$\pm$	0.000100
11	0.965069	$\pm$	0.000107
13	0.962400	$\pm$	0.000106
15	0.959976	$\pm$	0.000101

We've bolded the $k = 9$ because it returned the highest score, indicating that we might want to use $k = 9$ for this dataset.

Fine-Tuning the Random Forest

Let's look at the random-forest model. The `sklearn RandomForestClassifier` class has quite a few hyperparameters that we could manipulate. To avoid being excessively pedantic, we'll seek only an optimal number of trees in the forest. This is the `n_estimators` parameter. As we did for k in k -NN, we'll search over a range of forest sizes and select the one that gives the best grand mean score for 1,000 runs at five folds each per run.

This is a one-dimensional (1D) grid-like search. We varied k by one, but for the number of trees in the forest, we need to cover a larger scale. We don't expect there to be a meaningful difference between 10 trees in the forest or 11, especially considering that each random-forest training session will lead to a

different set of trees even if the number of trees is fixed. We saw this effect several times in the previous section. Instead, let's vary the number of trees by selecting from $n_t \in \{5, 20, 50, 100, 200, 500, 1,000, 5,000\}$, where n_t is the number of trees in the forest (number of estimators). Running this search gives us the grand means in Table 4-7.

Table 4-7: Breast Cancer Scores as Grand Mean for Different Random Forest Sizes and Five-Fold Validation

n_t	Grand mean $\pm$ SE
5	0.948327 $\pm$ 0.000206
20	0.956808 $\pm$ 0.000166
50	0.959048 $\pm$ 0.000139
100	0.959740 $\pm$ 0.000130
200	0.959913 $\pm$ 0.000122
500	0.960049 $\pm$ 0.000117
750	0.960147 $\pm$ 0.000118
1,000	0.960181 $\pm$ 0.000116

The differences are very small, though if you run the Mann–Whitney U test, you'll see that the difference between $n_t = 5$ (worst) and $n_t = 1,000$ (best) is statistically significant. However, the difference between $n_t = 200$ and $n_t = 1,000$ is not significant. Here we need to make a judgment call. While setting $n_t = 1,000$ gave the best result, it's indistinguishable, for practical purposes, from $n_t = 500$ or even $n_t = 100$. Since runtime for a random forest scales linearly in the number of trees, using $n_t = 100$ results in a classifier that is on average 10 times faster than using $n_t = 1,000$. So, depending on the task, we might select $n_t = 100$ over $n_t = 1,000$ for that reason.

Fine-Tuning the SVMs

Let's turn our attention to the linear SVM. For the linear kernel, we'll adjust C . Note, sklearn has other parameters, as it did for the random forest, but we'll

leave them at their default settings.

The range of C to search over is problem dependent, but the sklearn default value of $C = 1$ is a good starting point. We'll select C values around 1 but over several orders of magnitude. Specifically, we'll select from $C \in \{0.001, 0.01, 0.1, 1.0, 2.0, 10.0, 50.0, 100.0\}$. Running 1,000 five-fold validations, each for a different random ordering of the full dataset, gives grand means as in Table 4-8.

Table 4-8: Breast Cancer Scores as Grand Mean for Different SVM C Values and Five-Fold Validation

C	Grand mean $\pm$ SE
0.001	0.938500 $\pm$ 0.000066
0.01	0.967151 $\pm$ 0.000089
0.1	0.975943 $\pm$ 0.000101
1.0	0.971890 $\pm$ 0.000141
2.0	0.969994 $\pm$ 0.000144
10.0	0.966239 $\pm$ 0.000154
50.0	0.959637 $\pm$ 0.000186
100.0	0.957006 $\pm$ 0.000189

$C = 0.1$ gives the best accuracy. Though the difference between $C = 0.1$ and $C = 1$ is statistically meaningful, in practice the difference is only about 0.4 percent, so the default value of $C = 1$ is likewise a reasonable choice. Further refinement of C is possible because we see that $C = 0.01$ and $C = 2$ give the same accuracy, while $C = 0.1$ is higher than either, implying that if the C curve is smooth, there's a maximum accuracy for a certain C in $[0.01, 2.0]$.

Finding the right C for our dataset is a crucial part of successfully using a linear SVM. Our preceding rough run used a 1D grid search. Since C is continuous, we expect that a plot of the accuracy as a function of C will also be smooth. If that's the case, we can imagine searching for the right C , not with a grid search but with an optimization algorithm. In practice, however, the randomness of the ordering of the dataset and its effect on the output of k -fold cross-validation results will probably make any C found by an optimization

algorithm too specific to the problem at hand. Grid search over a larger scale, with possibly one level of refinement, is sufficient in most cases. The take-home message is: spend some time looking for the proper C value to maximize the effectiveness of the linear SVM.

The preceding analysis has ignored the RBF kernel SVM, so we'll revisit it now and see how to do a simple 2D grid search over C and γ , where γ is the parameter associated with the RBF (Gaussian) kernel. The sklearn library has the `GridSearchCV` class to perform sophisticated grid searching. We aren't using it here in order to demonstrate how to do simple grid searches directly. It's especially important for this kernel to select good values for both of these parameters.

For the search, we'll use the same range of C values as we used for the linear case. For γ we'll use powers of two, 2^p , times the sklearn default value, $1/30 = 0.03333$ for $p \in [-4, 3]$. The search will, for the current C value, do five-fold validation over the dataset for each γ value before moving to the next C value so that all pairs of (C, γ) are considered. The pair that results in the largest score (accuracy) will be output. The code is in Listing 4-6.

bc_rbf_svm_search.py

```
import numpy as np
```

```
from sklearn.svm import SVC
```

```
def run(x_train, y_train, x_test, y_test, clf):
```

```
    clf.fit(x_train, y_train)
```

```
    return clf.score(x_test, y_test)
```

```
def split(x, y, k, m):
```

```
    ns = int(y.shape[0]/m)
```

```
    s = []
```

```
    for i in range(m):
```

```
        s.append([x[(ns*i):(ns*i+ns)], y[(ns*i):(ns*i+ns)])])
```

```
    x_test, y_test = s[k]
```

```
    x_train = []
```

```
    y_train = []
```

```
    for i in range(m):
```

```
        if (i == k):
```

```
            continue
```

```
        else:
```

```
            a, b = s[i]
```

```

        x_train.append(a)
        y_train.append(b)
    x_train = np.array(x_train).reshape(((m-1)*ns, 30))
    y_train = np.array(y_train).reshape((m-1)*ns)
    return [x_train, y_train, x_test, y_test]

def main():
    m = 5
    x = np.load("../data/breast/bc_features_standard.npy")
    y = np.load("../data/breast/bc_labels.npy")
    idx = np.argsort(np.random.random(y.shape[0]))
    x = x[idx]
    y = y[idx]

    ❶ Cs = np.array([0.01, 0.1, 1.0, 2.0, 10.0, 50.0, 100.0])
    gs = (1./30)*2.0**np.array([-4, -3, -2, -1, 0, 1, 2, 3])
    zmax = 0.0
    ❷ for C in Cs:
        for g in gs:
            z = np.zeros(m)
            for k in range(m):
                x_train, y_train, x_test, y_test = split(x, y, k, m)
                z[k] = run(x_train, y_train, x_test, y_test,
                           SVC(C=C, gamma=g, kernel="rbf"))
            ❸ if (z.mean() > zmax):
                zmax = z.mean()
                bestC = C
                bestg = g
    print("best C      = %0.5f" % bestC)
    print("      gamma = %0.5f" % bestg)
    print("  accuracy= %0.5f" % zmax)

```

Listing 4-6: A 2D grid search for C and γ for an RBF kernel SVM (breast cancer dataset)

The two helper functions, `run` and `split`, are the same as we used before (see Listing 4-4); all the action is in `main`. We fix the number of folds at five and then load and randomize the full dataset.

We then define the specific C and γ values to search over ❶. Note how `gs` is defined. The first part is $1/30$, the reciprocal of the number of features. This is the default value for γ used by `sklearn`. We then multiply this factor by an

array, $(2^{-4}, 2^{-3}, 2^{-1}, 2^0, 2^1, 2^2, 2^3)$, to get the final γ values we'll search over. Notice that one of the γ values is exactly the default sklearn uses since $2^0 = 1$.

The double loop ❷ iterates over all possible pairs of C and γ . For each one, we do five-fold validation to get a set of five scores in z . We then ask whether the mean of this set is greater than the current maximum ($z_{\max}$) and if so, update the maximum and keep the C and γ values as our current bests ❸. When the loops over C and γ exit, we have our best values in `bestc` and `bestg`.

If we run this code repeatedly, we'll get different outputs each time. This is because we're randomizing the order of the full dataset, which alters the subsets in the folds, leading to a different mean score over the folds. For example, 10 runs produced the output in Table 4-9.

Table 4-9: Breast Cancer Scores for an RBF SVM with Different C and γ Values Averaged over 10 Runs

C	γ	Accuracy
1	0.03333	0.97345
2	0.03333	0.98053
10	0.00417	0.97876
10	0.00417	0.97699
10	0.00417	0.98053
10	0.01667	0.98053
10	0.01667	0.97876
10	0.01667	0.98053
1	0.03333	0.97522
10	0.00417	0.97876

These results hint that $(C, \gamma) = (10, 0.00417)$ is a good combination. If we use these values to generate a grand mean over 1,000 runs of five-fold validation as before, we get an overall accuracy of 0.976991, or 97.70 percent, which is the

highest grand mean accuracy of any model type we trained on the breast cancer histology dataset.

The breast cancer dataset is not large. We were able to use k -fold validation to find a good model that worked well with it. Now let's move from a pure vector-only dataset to one that is image based and much larger: the MNIST dataset.

Experiments with the MNIST Dataset

The last dataset we'll work with in this chapter is the vector version of the MNIST handwritten digit dataset (see Chapter 2). Recall, this dataset consists of 28×28 -pixel grayscale images of handwritten digits, $[0, 9]$, one digit centered per image. This dataset is by far the most common workhorse in machine learning, especially in deep learning, and we'll use it throughout the remainder of the book.

Testing the Classical Models

MNIST contains 60,000 training images, roughly evenly split among the digits, and 10,000 test images. Since we have a lot of training data, at least for classic models like those we're concerned with here, we won't use k -fold validation, though we certainly could. We'll train on the training data and test on the testing data and trust that the two come from a common parent distribution (they do).

Since our classic models expect vector inputs, we'll use the vector form of the MNIST dataset we created in Chapter 2. The images are unraveled so that the first 28 elements of the vector are row 0, the next 28 are row 1, and so on for an input vector of $28 \times 28 = 784$ elements. The images are stored as 8-bit grayscale, so the data values run from 0 to 255.

We'll consider three versions of the dataset. The first is the raw byte version. The second version scales the data to $[0, 1)$ by dividing by 256, the number of possible grayscale values. The third is a normalized version where, per "feature" (really, pixel), we subtract the mean of that feature across the dataset and then divide by the standard deviation. This lets us explore how the range of the feature values affects results, if at all.

Figure 4-1 shows examples of the original images and the resulting normalized vectors turned back into images and scaled $[0, 255]$.



Figure 4-1: Original MNIST digits (top) and normalized versions used by the models (bottom)

Normalizing affects the appearance but does not destroy spatial relationships among the parts of the digit images. Just scaling the data to $[0, 1)$ will result in images that look the same as those on the top of Figure 4-1.

The code we'll use is very similar to what we used previously, but for reasons that will be explained next, we replace the `svc` class with a new SVM class, `LinearSVC`. First, take a look at the helper functions in Listing 4-7.

```
mnist_experiments.py
import time
import numpy as np
from sklearn.neighbors import NearestCentroid

from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB, MultinomialNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import LinearSVC
from sklearn import decomposition

def run(x_train, y_train, x_test, y_test, clf):
    s = time.time()
    clf.fit(x_train, y_train)
    e_train = time.time() - s
    s = time.time()
    score = clf.score(x_test, y_test)
    e_test = time.time() - s
    print("score = %0.4f (time, train=%8.3f, test=%8.3f)"
          % (score, e_train, e_test))

def train(x_train, y_train, x_test, y_test):
    print("    Nearest centroid          : ", end='')
    run(x_train, y_train, x_test, y_test, NearestCentroid())
    print("    k-NN classifier (k=3)       : ", end='')
    run(x_train, y_train, x_test, y_test,
```

```

        KNeighborsClassifier(n_neighbors=3))
print("    k-NN classifier (k=7)      : ", end='')
run(x_train, y_train, x_test, y_test,
    KNeighborsClassifier(n_neighbors=7))
print("    Naive Bayes (Gaussian)     : ", end='')
run(x_train, y_train, x_test, y_test, GaussianNB())
print("    Decision tree               : ", end='')
run(x_train, y_train, x_test, y_test, DecisionTreeClassifier())
print("    Random forest (trees= 5) : ", end='')
run(x_train, y_train, x_test, y_test,
    RandomForestClassifier(n_estimators=5))
print("    Random forest (trees= 50) : ", end='')
run(x_train, y_train, x_test, y_test,
    RandomForestClassifier(n_estimators=50))
print("    Random forest (trees=500) : ", end='')
run(x_train, y_train, x_test, y_test,
    RandomForestClassifier(n_estimators=500))
print("    Random forest (trees=1000): ", end='')
run(x_train, y_train, x_test, y_test,
    RandomForestClassifier(n_estimators=1000))
print("    LinearSVM (C=0.01)           : ", end='')
run(x_train, y_train, x_test, y_test, LinearSVC(C=0.01))
print("    LinearSVM (C=0.1)            : ", end='')
run(x_train, y_train, x_test, y_test, LinearSVC(C=0.1))
print("    LinearSVM (C=1.0)            : ", end='')

run(x_train, y_train, x_test, y_test, LinearSVC(C=1.0))
print("    LinearSVM (C=10.0)           : ", end='')
run(x_train, y_train, x_test, y_test, LinearSVC(C=10.0))

```

Listing 4-7: Training differently scaled versions of the MNIST dataset by using classic models (helper functions)

The `run` function of Listing 4-7 is also similar to those used previously, except we've added code to time how long training and testing takes. These times are reported along with the score. We added this code for MNIST because, unlike the tiny iris and breast cancer datasets, MNIST has a larger number of training samples so that runtime differences among the model types will start to show themselves. The `train` function is new; it wraps calls to `run` for the different model types.

Now take a look at Listing 4-8, which contains the `main` function.

mnist_experiments.py

```
def main():
    x_train = np.load("mnist_train_vectors.npy").astype("float64")
    y_train = np.load("mnist_train_labels.npy")
    x_test = np.load("mnist_test_vectors.npy").astype("float64")
    y_test = np.load("mnist_test_labels.npy")

    print("Models trained on raw [0,255] images:")
    train(x_train, y_train, x_test, y_test)
    print("Models trained on raw [0,1) images:")
    train(x_train/256.0, y_train, x_test/256.0, y_test)

    ❶ m = x_train.mean(axis=0)
    s = x_train.std(axis=0) + 1e-8
    x_ntrain = (x_train - m) / s
    x_ntest = (x_test - m) / s

    print("Models trained on normalized images:")
    train(x_ntrain, y_train, x_ntest, y_test)

    ❷ pca = decomposition.PCA(n_components=15)
    pca.fit(x_ntrain)
    x_ptrain = pca.transform(x_ntrain)
    x_ptest = pca.transform(x_ntest)

    print("Models trained on first 15 PCA components of normalized images:")
    train(x_ptrain, y_train, x_ptest, y_test)
```

Listing 4-8: Training differently scaled versions of the MNIST dataset by using classic models (main function)

This `main` function loads the data and then trains the models by using the raw byte values. It then repeats the training by using a scaled $[0,1)$ version of the data and a scaled version of the testing data. These are the first two versions of the dataset we'll use.

Normalizing the data requires knowledge of the per-feature means and standard deviations ❶. Note, we add a small value to the standard deviations to make up for pixels that have a standard deviation of zero. We can't divide by zero, after all. We need to normalize the test data, but which means and which standard deviations should we use?

Generally, we have more training data than testing data, so using the means and standard deviations from the training data makes sense; they are a better representation of the true means and standard deviations of the parent distribution that generated the data in the first place. However, at times slight differences might occur between the training and testing data distributions, in which case it might make sense to consider the testing means and standard deviations.

In this case, because the MNIST training and test datasets were created together, there's no difference, so we'll use the training values. The same per-feature means and standard deviations will need to be used for all new, unknown samples, too.

Next, we apply PCA to the dataset just as we did for the iris data in Chapter 2 ❷. Here we're keeping the first 15 components. These account for just over 33 percent of the variance in the data and reduce the feature vector from 784 features (the pixels) to 15 features (the principal components). Then we train the models by using these features.

Running this code, which takes about 90 minutes on my modest test machine, produces a wealth of output that we can learn from. I recommend adding `-W ignore` on the command line after `python3` to silence warning messages from the SVM.

Let's first consider the scores per model type and data source. These are in Table 4-10; values in parentheses are the number of trees in the random forest.

Table 4-10: MNIST Model Scores for Different Preprocessing Steps

Model	Raw [0, 255]	Scaled [0, 1)	Normalized	PCA
Nearest centroid	0.8203	0.8203	0.8092	0.7523
k -NN ($k = 3$)	0.9705	0.9705	0.9452	0.9355
k -NN ($k = 7$)	0.9694	0.9694	0.9433	0.9370
Naive Bayes	0.5558	0.5558	0.5239	0.7996
Decision tree	0.8773	0.8784	0.8787	0.8403
Random forest (5)	0.9244	0.9244	0.9220	0.8845

Model	Raw [0, 255]	Scaled [0, 1)	Normalized	PCA
Random forest (50)	0.9660	0.9661	0.9676	0.9215
Random forest (500)	0.9708	0.9709	0.9725	0.9262
Random forest (1,000)	0.9715	0.9716	0.9719	0.9264
Linear SVM ($C = 0.01$)	0.8494	0.9171	0.9158	0.8291
Linear SVM ($C = 0.1$)	0.8592	0.9181	0.9163	0.8306
Linear SVM ($C = 1.0$)	0.8639	0.9182	0.9079	0.8322
Linear SVM ($C = 10.0$)	0.8798	0.9019	0.8787	0.7603

The nearest-centroid scores make sense as we move from left to right across the versions of the dataset. For the raw data, the center location of each of the 10 classes leads to a simple classifier with an accuracy of 82 percent—not too bad, considering random guessing would have an accuracy closer to 10 percent (1/10 for 10 classes). Scaling the data by a constant won't change the relative relationship between the per-class centroids, so we'd expect the same performance in column 2 of Table 4-10 as in column 1.

Normalizing, however, does more than divide the data by a constant. We saw the effect clearly in Figure 4-1. This alteration, at least for the MNIST dataset, changes the centroids' relationships to each other and results in a decrease in accuracy to 80.9 percent.

Finally, using PCA to reduce the number of features from 784 to 15 has a severe negative impact, resulting in an accuracy of only 75.2 percent. Note the word *only*. In the past, before the advent of deep learning, an accuracy of 75 percent on a problem with 10 classes generally would've been considered pretty good. Of course, it really isn't. Who would get in a self-driving car that has an accident once every four trips? We want to do better.

Let's consider the k -NN classifiers next. We see similar performance for both $k = 3$ and $k = 7$ and the same sort of trend as we saw with the nearestcentroid classifier. This is to be expected, given the similarity of the two types of models. The difference in accuracy between the two (centroid and k -NN) is dramatic, however. An accuracy of 97 percent is generally regarded as good. But still, who would opt for elective surgery with a 3 percent failure rate?

Things get interesting when we look at the naive Bayes classifier. Here all the versions of the dataset perform poorly, though still five times better than guessing. We see a large jump in accuracy with the PCA processed dataset, from 56 percent to 80 percent. This is the only model type to improve after using PCA. Why might this be? Remember, we're using Gaussian naive Bayes, which means our independence assumption is coupled with an assumption that the continuous feature values are, per feature, really drawn from a normal distribution whose parameters, the mean and standard deviation, we can estimate from the feature values themselves.

Now recall what PCA does, geometrically. It's the equivalent of rotating the feature vectors onto a new set of coordinates aligned with the largest orthogonal directions derivable from the dataset. The word *orthogonal* implies that no part of a direction overlaps with any other part of any other direction. Think of the x -, y -, and z -axes of a 3D plot. No part of the x is along the y or z , and so forth. This is what PCA does. Therefore, PCA makes the first assumption of naive Bayes, that the new features are indeed independent of one another, more likely to be true. Add in the Gaussian assumption as to the distribution of the per-pixel values, and we have an explanation for what we see in Table 4-10.

The tree-based classifiers, decision tree and random forest, perform much the same until we get to the PCA version of the dataset. Indeed, there is no difference between the raw data and the data scaled by 256. This is to be expected, as scaling by a constant simply scales the decision thresholds for each of the nodes in the body of the tree or trees. As before, working with reduced dimensionality vectors via PCA results in a loss of accuracy because potentially important information has been discarded.

For any data source, we see scores that make sense relative to one another. As before, the single decision tree performs worst, which it should except for simple cases since it's competing against a collection of trees via the random forests. For the random forests, the score improves as the number of trees in the forest increases—again expected. However, the improvement comes with

diminishing returns. A significant improvement occurs when going from 5 trees to 50 trees, but a minimal improvement in going from 500 trees to 1,000 trees.

Before we look at the SVM results, let's understand why we made the switch from the `svc` class to `LinearSVC`. As the name suggests, `LinearSVC` implements only a linear kernel. The `svc` class is more generic and can implement other kernels, so why switch?

The reason has to do with runtime. Computer science provides specific definitions of complexity and has an entire branch devoted to the analysis of algorithms and how they perform as their inputs scale. All we'll concern ourselves with here is *big-O notation*. This is a way of characterizing how the runtime of an algorithm changes as the input (or the number of inputs) gets increasingly larger.

For example, a classic bubble-sort algorithm works just fine on a few dozen numbers. But as the input gets larger (more numbers to be sorted), the runtime increases not linearly but quadratically: The time to sort the numbers, t , is proportional to the *square* of the number of numbers to be sorted, $t \propto n^2$, which is written as $\mathcal{O}(n^2)$. So, the bubble sort is an order n^2 algorithm. In general, we want algorithms that are better than n^2 , more like n , written as $\mathcal{O}(n)$, or even independent of n , written as $\mathcal{O}(1)$. It turns out that the kernel algorithm for training an SVM is *worse* than $\mathcal{O}(n^2)$ so that when the number of training samples increases, the runtime explodes. This is one reason for the switch from the `svc` class to `LinearSVC`, which doesn't use kernels.

The second reason for the switch has to do with SVMs being designed for binary classification—only two classes. The MNIST dataset has 10 classes, so we have to do something different. There are multiple approaches. According to the sklearn documentation, the `svc` class uses a *one-versus-one* approach that trains pairs of classifiers, one class versus another: class 0 versus class 1, class 1 versus class 2, class 0 versus class 2, and so on. This means it ends up training not one but $m(m-1)/2$ classifiers for $m = 10$ classes, or $10(10-1)/2 = 45$ separate classifiers. This isn't efficient in this case. The `LinearSVC` classifier uses a *one-versus-rest* approach: It trains an SVM to classify “0” versus “1–9,” then “1” versus “0, 2–9,” and so on, for a total of only 10 classifiers, one for each digit.

With the SVM classifiers, we see a definite benefit to scaling the data versus the raw byte inputs. We also see that the optimal C value is likely between $C = 0.1$ and $C = 1.0$. Note that simple $[0, 1)$ scaling leads to SVM models that outperform (for this one dataset!) the models trained on the

normalized data. The effect is small but consistent for different C values. And, as we saw before, dropping the dimensionality from 784 features to only 15 features via PCA leads to a rather large loss of accuracy. PCA seems not to have helped in this case. We'll return to this in a bit and see if we can understand why.

Analyzing Runtimes

Let's now look at the runtime performance of the algorithms. Table 4-11 shows the train and test times, in seconds, for each model type and dataset version.

Table 4-11: Training and Testing Times (in Seconds) for Each Model Type

Model	Raw [0, 255]		Scaled [0, 1)		Normalized		PCA	
	Train	Test	Train	Test	Train	Test	Train	Test
Nearest centroid	0.23	0.03	0.24	0.03	0.24	0.03	0.01	0.00
k -NN ($k = 3$)	33.24	747.34	33.63	747.22	33.66	699.58	0.09	3.64
k -NN ($k = 7$)	33.45	746.00	33.69	746.65	33.68	709.62	0.09	4.65
Naive Bayes	0.80	0.88	0.85	0.90	0.83	0.94	0.02	0.01
Decision tree	25.42	0.03	25.41	0.02	25.42	0.02	2.10	0.00
Random forest (5)	2.65	0.06	2.70	0.06	2.61	0.06	1.20	0.03
Random forest (50)	25.56	0.46	25.14	0.46	25.27	0.46	12.06	0.25
Random forest (500)	252.65	4.41	249.69	4.47	249.19	4.45	121.10	2.51

Model	Raw [0, 255]		Scaled [0, 1)		Normalized		PCA	
	Train	Test	Train	Test	Train	Test	Train	Test
Random forest (1,000)	507.52	8.86	499.23	8.71	499.10	8.91	242.44	5.00
Linear SVM ($C = 0.01$)	169.45	0.02	5.93	0.02	232.93	0.02	16.91	0.00
Linear SVM ($C = 0.1$)	170.58	0.02	36.00	0.02	320.17	0.02	37.46	0.00
Linear SVM ($C = 1.0$)	170.74	0.02	96.34	0.02	488.06	0.02	66.49	0.00
Linear SVM ($C = 10.0$)	170.46	0.02	154.34	0.02	541.69	0.02	86.87	0.00

The test times indicate how long each model takes to classify all 10,000 digit images in the test set. First, note that k -NN is slow; classifying the test set takes over 10 minutes when full feature vectors are used! It's only when we drop down to the first 15 PCA components that we see reasonable k -NN runtimes. This is a good example of the price we pay for a seemingly simple idea. Recall, the k -NN classifier finds the k closest training samples to the unknown sample we wish to classify. Here *closest* means in a Euclidean sense, like the distance between two points on a graph, except in this case we don't have 2 or 3 dimensions but 784.

Therefore, for each test sample, we need to find the $k = 3$ or $k = 7$ closest points in the training data. The naive approach is to calculate the distance between the unknown sample and each of the 60,000 training samples, sort them, look at the k smallest distances, and vote to decide the output class label. This is a lot of work because we have 60,000 training samples and 10,000 test samples for a total of 600,000,000 distance calculations.

However, it isn't actually as bad as all that because sklearn automatically selects the algorithm used to find the nearest neighbors, and decades of

research has uncovered “better than brute force” approaches. Curious readers should investigate the terms *k-d-tree* and *ball tree* (sometimes called *metric tree*). See “An Empirical Comparison of Exact Nearest Neighbor Algorithms” by Ashraf M. Kibriya and Eibe Frank (2007). Still, because of the extreme difference in runtimes between the other model types and *k*-NN, it’s necessary to remember just how slow *k*-NN can be if the dataset is large.

The next slowest test times are for the random-forest classifiers. We understand why a forest with 500 trees takes 10 times longer to run than a forest with 50 trees; we have 10 times as many trees to evaluate. Training times also scale linearly. Reducing the size of the feature vectors with PCA improves runtime but not by a factor of 50 (784 features divided by 15 PCA features $\approx$ 50), so the performance difference is not primarily influenced by the size of the feature vector.

The linear SVMs are the next slowest to train after the random forests, but their execution time is extremely low. Long training times and short classification (inference) times are a hallmark of many model types. The simplest models are quick to train and quick to use, like nearest centroid or naive Bayes, but in general, “slow to train, quick to use” is a safe assumption. It’s especially true of neural networks.

Using PCA hurt the performance of the models except for the naive Bayes classifier. Let’s do an experiment to see the effect of PCA as the number of PCA components changes.

Experimenting with PCA Components

For Tables 4-10 and 4-11, we selected 15 PCA components that represented about 33 percent of the variance in the dataset. This value was selected at random. You could imagine training models using another number of principal components.

Let’s examine the effect of the number of PCA components used on the accuracy of the resulting model. We’ll vary the number of components from 10 to 780, which is basically all the features in the image. For each number of components, we’ll train a naive Bayes classifier, a random forest of 50 trees, and a linear SVM with $C = 1.0$. The code to do this is in Listing 4-9. Patience is required when running this experiment. The runtime on my low-end test system was 10.4 hours.

```
mnist_pca.py
def main():
```

```

x_train = np.load("../data/mnist/mnist_train_vectors.npy")
            .astype("float64")

y_train = np.load("../data/mnist/mnist_train_labels.npy")
x_test = np.load("../data/mnist/mnist_test_vectors.npy").astype("float64")
y_test = np.load("../data/mnist/mnist_test_labels.npy")
m = x_train.mean(axis=0)
s = x_train.std(axis=0) + 1e-8
x_ntrain = (x_train - m) / s
x_ntest = (x_test - m) / s

n = 78
pcomp = np.linspace(10, 780, n, dtype="int16")
nb = np.zeros((n, 4))
rf = np.zeros((n, 4))
sv = np.zeros((n, 4))
tv = np.zeros((n, 2))

for i, p in enumerate(pcomp):
    ❶ pca = decomposition.PCA(n_components=p)
        pca.fit(x_ntrain)
        xtrain = pca.transform(x_ntrain)
        xtest = pca.transform(x_ntest)
        tv[i,:] = [p, pca.explained_variance_ratio_.sum()]
    ❷ sc, etrn, etst = run(xtrain, y_train, xtest, y_test, GaussianNB())
        nb[i,:] = [p, sc, etrn, etst]
        sc, etrn, etst = run(xtrain, y_train, xtest, y_test,
                            RandomForestClassifier(n_estimators=50))
        rf[i,:] = [p, sc, etrn, etst]
        sc, etrn, etst = run(xtrain, y_train, xtest, y_test, LinearSVC(C=1.0))
        sv[i,:] = [p, sc, etrn, etst]

np.save("mnist_pca_tv.npy", tv)
np.save("mnist_pca_nb.npy", nb)
np.save("mnist_pca_rf.npy", rf)
np.save("mnist_pca_sv.npy", sv)

```

Listing 4-9: Model accuracy as a function of the number of PCA components used

First, we load the MNIST dataset and compute the normalized version. This is the version that we'll use with PCA. Next, we set up storage for the results. The variable `pcomp` stores the specific number of PCA components that

will be used from 10 to 780 in steps of 10. Then we start a loop over the number of PCA components. We find the requested number of components (p) and map the dataset to the actual dataset trained and tested (x_{train} , x_{test}) ❶.

We also store the actual amount of variance in the dataset explained by the current number of principal components (t_v). We'll plot this value later to see how quickly the number of components covers the majority of the variance in the dataset.

Next, we train and test a Gaussian naive Bayes classifier using the current number of features ❷. The `run` function called here is virtually identical to that used in Listing 4-7 except that it returns the score, the training time, and the testing time. These are captured and put into the appropriate output array (`nb`). Then we do the same for the random forest and linear SVM.

When the loop completes, we have all the data we need, and we store the NumPy arrays on disk for plotting. Running this code takes some time, but the output, when plotted, leads to Figure 4-2.

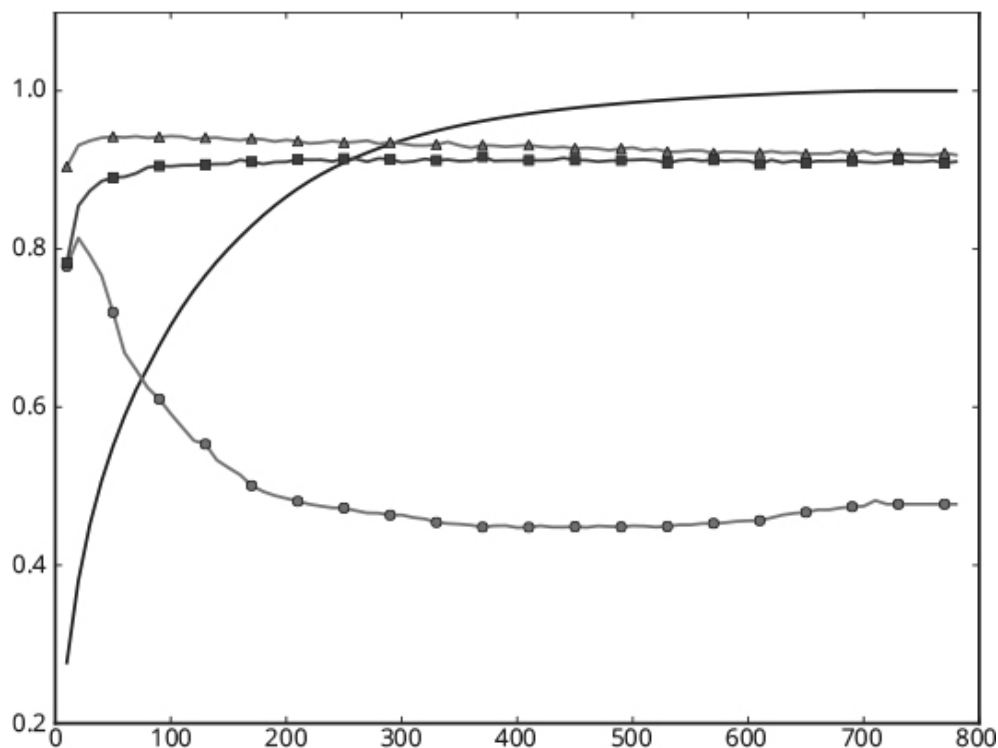


Figure 4-2: The results of the PCA search

The solid curve shows the fraction of the total variance in the dataset explained by the current number of PCA components (x -axis). This curve reaches a maximum of 1.0 when all the features in the dataset are used. It's

helpful in this case because it shows how quickly adding new components explains major orientations of the data. For MNIST, about 90 percent of the variance is explained by using less than half the possible number of PCA components.

The remaining three curves plot the accuracy of the resulting models on the test data. The best-performing model, in this case, is the random forest with 50 trees (triangles). This is followed by the linear SVM (squares) and then naive Bayes (circles). These curves show how the number of PCA components tracks with accuracy, and while the random forest and SVM change only slowly as PCA changes, the naive Bayes classifier rapidly loses accuracy as the number of PCA components increases. Even the random forest and SVM decrease as the number of PCA components increases, which we might expect because the curse of dimensionality will eventually creep in. The dramatically different behavior of the naive Bayes classifier is likely due to violations of the independence assumption as the number of components used increases.

The maximum accuracy and the number of PCA components where it occurs are shown in Table 4-12.

Table 4-12: Maximum Accuracy on MNIST by Model and Number of Components

Model	Accuracy	Components	Variance
Naive Bayes	0.81390	20	0.3806
Random forest (50)	0.94270	100	0.7033
Linear SVM ($C = 1.0$)	0.91670	370	0.9618

Table 4-12 tracks with the plot in Figure 4-2. Interestingly, the SVM does not reach a maximum until nearly all the features in the original dataset are used. Also, the best accuracy found for the random forest and SVM is not as good as seen previously for other versions of the dataset that did not use PCA. So, for these models, PCA is not a benefit; it is, however, for the naive Bayes classifier.

Scrambling the Dataset

Before we leave this section, let's look at one more experiment that we'll come back to in Chapter 6 and again in Chapter 9. In Chapter 2, we made a version of the MNIST dataset that scrambled the order of the pixels in the digit images. The scrambling wasn't random: The same pixel in each input image was moved to the same position in the output image, resulting in images that, at least to us, no longer look like the original digit, as Figure 4-3 shows. How might this scrambling affect the accuracy of the models we've been using in this chapter?



Figure 4-3: The original MNIST digits (top) and scrambled versions of the same digit (bottom)

Let's repeat the experiment code of Listing 4-8, this time running only the scaled $[0, 1)$ version of the scrambled MNIST images. Since the only difference from the original code is the source filenames and the fact that we call `run` only once, we'll forgo a new listing.

Placing the accuracy results side by side gives us Table 4-13.

Table 4-13: MNIST Scores by Model Type for Unscrambled and Scrambled Digits

Model	Unscrambled $[0, 1)$	Scrambled $[0, 1)$
Nearest centroid	0.8203	0.8203
k -NN ($k = 3$)	0.9705	0.9705
k -NN ($k = 7$)	0.9694	0.9694
Naive Bayes	0.5558	0.5558
Decision tree	0.8784	0.8772
Random forest (5)	0.9244	0.9214
Random forest (50)	0.9661	0.9651
Random forest (500)	0.9709	0.9721
Random forest (1,000)	0.9716	0.9711

Model	Unscrambled [0, 1)	Scrambled [0, 1)
Linear SVM ($C = 0.01$)	0.9171	0.9171
Linear SVM ($C = 0.1$)	0.9181	0.9181
Linear SVM ($C = 1.0$)	0.9182	0.9185
Linear SVM ($C = 10.0$)	0.9019	0.8885

Here we see virtually no difference between the scrambled and unscrambled results. For several models, the results are identical. For stochastic models, like the random forests, the results are still very similar. Is this surprising? Perhaps at first, but if we think about it for a bit, it really shouldn't be.

All the classic models are holistic: They operate on the entire feature vector as a single entity. While we can't see the digits anymore because our vision does not operate holistically, the information present in the image is still there, so the models are just as happy with the scrambled as unscrambled inputs. When we get to Chapter 9, we'll encounter a different result of this experiment.

Classical Model Summary

What follows is a summary of the pros and cons related to each of the classical model types we have explored in this chapter. This can be used as a quick list for future reference. It will also take some of the observations we made via our experiments and make them more concrete.

Nearest Centroid

Nearest centroid is the simplest of all the models and can serve as a baseline. It's seldom adequate unless the task at hand is particularly easy. The single centroid for each class is needlessly restrictive. You could use a more generalized approach that first finds an appropriate number of centroids for each class and then groups them together to build the classifier. In the extreme, this approaches k -NN but is still simpler in that the number of centroids is likely far less than the number of training samples. I leave the implementation of this variation as an exercise for the motivated reader.

Pros

As we saw in this chapter, the implementation of a nearest-centroid classifier takes only a handful of code. Additionally, nearest centroid isn't restricted to binary models and readily supports multiclass models, like the irises. Training is very fast and since only one centroid is stored per class, the memory overhead is likewise very small. When used to label an unknown sample, runtime is also very small because only the distance from the sample to each class centroid needs to be computed.

Cons

Nearest centroid makes a simplistic assumption about the distribution of the classes in the feature space—one that's seldom met in practice. As a consequence of this assumption, the nearest-centroid classifier is highly accurate only when the classes form a single tight group in the feature space and the groups are distant from one another like isolated islands.

k-Nearest Neighbors

This is the simplest model to train since there's no training: We store the training set and use it to classify new instances by finding the k -nearest training-set vectors and voting.

Pros

As mentioned, no training required makes k -NN particularly attractive. It can also perform quite well, especially if the number of training samples is large relative to the dimensionality of the feature space (that is, the number of features in the feature vector). Multiclass support is implicit and doesn't require a special approach.

Cons

The simplicity of “training” comes at a cost: Classification is slow because of the need to look at every training example to find the nearest neighbors to the unknown feature vector. Decades of research, still underway, have sped up the search to improve the naive implementation of looking at every training sample every time, but, as we saw in this chapter, classification is still slow, especially when compared to the speed of other model types (for example, SVM).

Naive Bayes

Naive Bayes is conceptually simple and efficient, and surprisingly valid even when the core assumption of feature independence isn't met.

Pros

Naive Bayes is fast to train and fast to classify with, both positives. It also supports multiclass models instead of just binary, and other than continuous features. As long as the probability of a particular feature value can be computed, we can apply naive Bayes.

Cons

The feature-independence assumption central to naive Bayes is seldom true in practice. The more correlated the features (the more a change in, say, feature x_2 implies that x_3 will change), the poorer the performance of the model (in all likelihood).

While naive Bayes works directly with discrete valued features, using continuous features often involves a second level of assumption, as when we assumed that the continuous breast cancer dataset features were well represented as samples from a Gaussian distribution. This second assumption, which is also likely seldom true in practice, means that we need to estimate the parameters of the distribution from the dataset instead of using histograms to stand in for the actual feature probabilities.

Decision Trees

A decision tree is useful when it's important to be able to understand, in human terms, why a particular class was selected.

Pros

Decision trees are reasonably fast to train. They're also fast to use for classifying. Multiclass models are not a problem and aren't restricted to using just continuous features. A decision tree can justify its answer by showing the particular steps used to reach a decision: the series of questions asked from the root to the leaf.

Cons

Decision trees are prone to overfitting—to learning elements of the training data that are not generally true of the parent distribution. Also, interpretability degrades as the tree increases in size. Tree depth needs to be balanced with the

quality of the decisions (labels) as the leaves of the tree. This directly affects the error rate.

Random Forests

A random forest is a more powerful form of decision tree that uses randomness to reduce the overfitting problem. Random forests are one of the best performing of the classic model types and apply to a wide range of problem domains.

Pros

Like decision trees, random forests support multiclass models and other continuous features. They are reasonably fast to train and to use for inference. Random forests are also robust to differences in scale among features in the feature vector. In general, the accuracy improves, with diminishing returns, as the size of the forest grows.

Cons

The easy interpretability of a decision tree disappears with a random forest. While each tree in the forest can justify its decision, the combined effect of the forest as a whole can be difficult to understand.

The inference runtime of a forest scales linearly with the number of trees. However, this can be mitigated by parallelization since each tree in the forest is making a calculation that doesn't depend on any other tree until combining the output of all trees to make an overall decision.

As stochastic models, the overall performance of a forest varies from training session to training session for the same dataset. In general, this isn't an issue, but a pathological forest could exist; if possible, train the forest several times to get a sense of the actual performance.

Support Vector Machines

Before the "rebirth" of neural networks, SVMs were generally considered the pinnacle of model performance when they were applicable and well tuned.

Pros

SVMs can show excellent performance when properly tuned. Inference is very fast once trained.

Cons

Multiclass models aren't directly supported. Extensions for multiclass problems require training multiple models, whether using one-versus-one or one-versus-rest approaches. Additionally, SVMs expect only continuous features, and feature scaling matters; normalization or other scaling is often necessary to get good performance.

Large datasets are difficult to train when using other than linear kernels, and SVMs often require careful tuning of margin and kernel parameters (C , γ), though this can be mitigated somewhat by search algorithms that seek the best hyperparameter values.

When to Use Classical Models

The classical models may be classic, but they are still appropriate under the right conditions. In this section, we'll discuss when you should consider a classical model instead of a more modern approach.

Handling Small Datasets

One of the best reasons for working with a classic model is that the dataset is small. If you have only a few tens or hundreds of examples, a classic model might be a good fit, whereas a deep learning model might not have enough training data to condition itself to the problem.

Of course, exceptions arise. A deep neural network can, via transfer learning, sometimes learn from relatively few examples. Other approaches, like zero-shot or few-shot learning, may also allow a deep network to learn from a small dataset. However, these techniques are far beyond the scope of what we want to address in this book. For us, the rule of thumb is that when the dataset is small, consider using a classic model.

Dealing with Reduced Computational Requirements

Another reason to consider a classic model is that computational requirements may need to be kept to a minimum. Deep neural networks are notoriously demanding of computational resources. The thousands, millions, and even billions of connections in a deep network all require extensive calculation. Implementing such a model on a small handheld device or on an embedded microcontroller will not work, at least not without considerable manipulation.

In such cases, you might consider a classic model that doesn't require a lot of overhead. Simple models like nearest centroid or naive Bayes are good candidates, as are decision trees and SVMs, once trained. From the previous experiments, k -NN is probably not a good candidate unless the feature space or training set is small. This leads to our next rule of thumb: When computation must be kept to a minimum, consider using a classic model.

Having Explainable Models

Some classic models can explain themselves by revealing exactly *how* they arrived at their answer for a given unknown input. This includes decision trees, by design, but also k -NN (by showing the labels of the k voters), nearest centroid (by virtue of the selected centroid), and even naive Bayes (by the selected posterior probability).

By way of contrast, deep neural networks are black boxes; they do not explain themselves. Learning how to get a deep network to give a reason for its decision remains an active area of research. This research hasn't been entirely unsuccessful, to be sure, but it's still far from looking like the decision path in a tree classifier. Therefore, we can give another rule of thumb: When it's essential to know how the classifier makes its decision, consider using a classic model.

Working with Vector Inputs

Our final rule of thumb, acknowledging, of course, that we could indeed give others, has to do with the form of the inputs to the model. Modern deep learning systems often work with inputs that aren't an amalgamation of separate features put into a single vector but instead are multidimensional inputs, such as images, where the "features" (pixels) are not different from one another but are of the same kind and often highly correlated (the red pixel of the apple likely has a red pixel next to it, for example). A color image is a 3D beast: There are three color images, one for the red channel, one for the blue channel, and one for the green channel. If the inputs are images from other sources, like satellites, there might be four to eight or more channels per image. A convolutional neural network is designed precisely for inputs such as these and will look for spatial patterns characteristic of the classes the network is trying to learn about. See Chapter 9 for more details.

But if the input to the model is a vector, especially one whose particular features aren't related to one another (the key assumption of the naive Bayes classifier), then a classic model might be appropriate, since there's no need to

PART III

NEURAL NETWORKS

Deep learning is synonymous with neural networks, though the latter existed long before the former. This part introduces traditional neural networks, which we might consider a stepping stone that allows us to cross the stream from classical machine learning on one side to deep learning on the other. What is true for traditional neural networks is true for deep neural networks.

OceanofPDF.com

5

INTRODUCTION TO NEURAL NETWORKS



Neural networks are the heart of deep learning. Before we dive into what we'll later call traditional neural networks in Chapter 6, we'll spend this chapter introducing the anatomy of a neural network, followed by a quick example.

Specifically, we'll present the components of a fully connected feedforward neural network. You can visually imagine the network as shown in Figure 5-1. We'll refer to this figure often throughout this chapter and the next. Your mission, should you choose to accept it, is to commit this figure to memory to save wear and tear on the book by flipping back to it repeatedly.

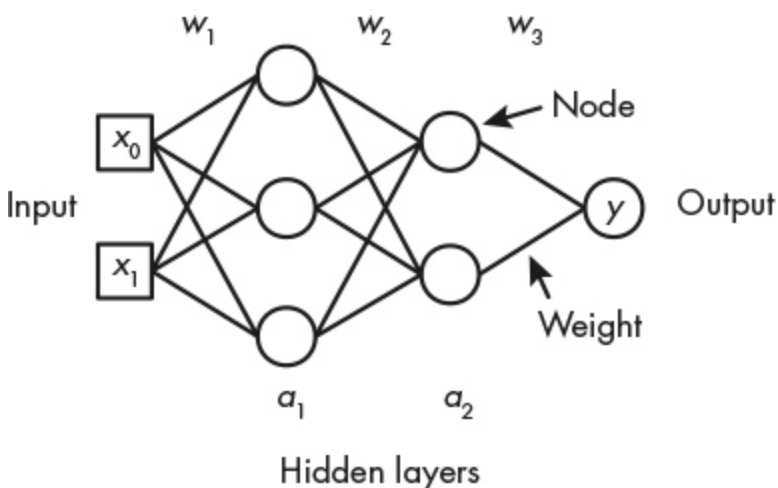


Figure 5-1: A sample neural network

After discussing the structure and parts of a neural network, we'll explore training our example network to classify irises. From this initial experiment, Chapter 6 will lead us to gradient descent and the backpropagation algorithm—the standard way that neural networks, including advanced deep neural networks, are trained. This chapter is intended as a warm-up. The heavy lifting starts in Chapter 6.

Anatomy of a Neural Network

A neural network is a graph. In computer science, a *graph* is a series of nodes, universally drawn as circles, connected by edges (short line segments). This abstraction is useful for representing many kinds of relationships: roads between cities, who knows whom on social media, the structure of the internet, or a series of basic computational units that can be used to approximate any mathematical function.

The last example is, of course, deliberate. Neural networks are universal function approximators. They use a graph structure to represent a series of computational steps mapping an input feature vector to an output value, typically interpreted as a probability.

Neural networks are built in layers. Conceptually, they act from left to right, mapping an input feature vector to the output(s) by passing values along the edges to the nodes. We often refer to the nodes of a neural network as *neurons*. We'll see why shortly. The nodes calculate

new values based on their inputs, which are then passed to the next layer of nodes and so on until the output nodes are reached. Figure 5-1 shows an input layer on the left, a hidden layer to its right, another hidden layer right of that, and a single node in the output layer.

The start of this chapter included the phrase *fully connected feed-forward neural network* without much explanation. Let's break it down. *Fully connected* means that every node of a layer has its output sent to every node of the next layer. *Feed-forward* means that information passes from left to right through the network without being sent back to a previous layer; no feedback, no looping, occurs in the network structure. This leaves only the neural network part.

The Neuron

I have a love-hate relationship with the term *neural network*. The term comes from the fact that, in a crude approximation, the basic unit of the network resembles a neuron in a brain. Consider Figure 5-2, which we'll describe in detail shortly.

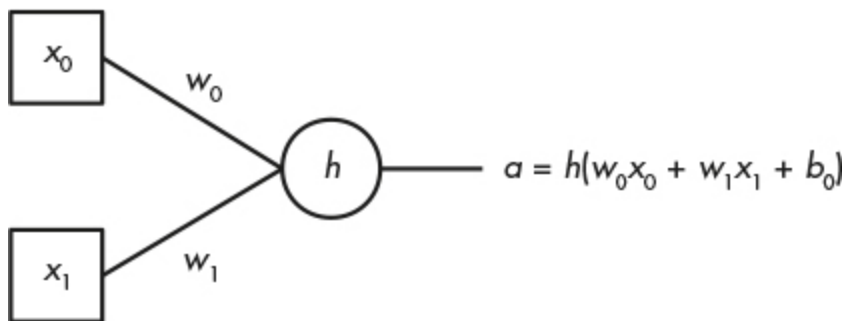


Figure 5-2: A single neural-network node

Recalling that our visualization of a network always moves from left to right, we see that the node (the circle) accepts input from the left and has a single output on the right. This network has two inputs, but it could have hundreds.

Many inputs mapped to a single output echo the way a neuron in the brain works: Structures called *dendrites* accept input from many other neurons, and the single axon is the output. I love this analogy because it leads to a cool way of talking and thinking about the networks, but I

hate it because these artificial neurons are, operationally, quite different from real ones, so the analogy quickly falls apart. Actual neurons are anatomically similar, but they're not the same, and it leads to confusion on the part of those who aren't familiar with machine learning, causing some to believe that computer scientists are truly building artificial brains or that the networks think. The meaning of the word *think* is hard to pin down, but I don't believe it applies in general to what a neural network does. The large language models of Chapter 18 might disagree.

Returning now to Figure 5-2, we see two squares on the left, two lines, a circle, a line on the right, and a bunch of labels with subscripts. Let's sort this out. Once you understand this illustration, you'll be well on your way to understanding neural networks. Later, we'll see an implementation of our visual model in code and will learn how simple it can be.

Everything in Figure 5-2 focuses on the circle. This is the node. In reality, it implements a mathematical function called the *activation function*, which calculates the output of the node, a single number. The two squares are the inputs to the node. This node accepts features from an input feature vector; we use squares to differentiate from circles, but the input might just as well have come from another group of circular nodes in a previous network layer.

Each input is a number, a single scalar value, which we're calling x_0 and x_1 . These inputs move to the node along the two line segments labeled w_0 and w_1 . These line segments represent *weights*, the strength of the connection. Computationally, the inputs (x_0, x_1) are multiplied by the weights (w_0, w_1) , summed, and given to the activation function of the node. Here we're calling the activation function h , which is fairly common.

The value of the activation function is the output of the node. Here we're calling this output a . The inputs, multiplied by the weights, are added together and given to the activation function to produce an output value. The b_0 value, which is also added in and passed to the activation function, is the *bias* term. It's an offset used to adjust the input

range to make it suitable for the activation function. In Figure 5-2, we added a zero subscript. There is a bias value for each node in each layer, so the subscript here implies that this node is the first node in the layer. (Remember, computer people always count from zero, not one.)

A neural network node accepts multiple inputs, $x_0, x_1, \dots$; multiplies each by a weight value, $w_0, w_1, \dots$; sums these products along with the bias term, b ; and passes this sum to the activation function, h , to produce a single scalar output value, a :

$$a = h(w_0x_0 + w_1x_1 + \dots + b)$$

That's it. Get a bunch of nodes together, link them appropriately, figure out how to train them to set the weights and biases, and you have a useful neural network. As you'll see in the next chapter, training a neural network is no easy feat. But once trained, they're simple to use: Feed it a feature vector, and out comes a classification.

As an aside, we've been calling these graphs *neural networks* and will continue to do so, sometimes using the abbreviation *NN*. If you read other books or papers, you might see them called *artificial neural networks* (*ANNs*) or even *multilayer perceptrons* (*MLPs*), as in the sklearn `MLPClassifier` class name.

Activation Functions

The activation function for a node takes a single scalar input, the sum of the inputs times the weights plus the bias, and does something to it. In particular, we need the activation function to be nonlinear so that the model can learn complex functions. Mathematically, it's easiest to see what a nonlinear function is by stating what a linear function is and then saying that any mapping that is not linear is . . . nonlinear.

A *linear function*, g , has output that is directly proportional to the input, $g(x) \propto x$, where $\propto$ means *proportional to*. Alternatively, the graph of a linear function is a straight line. Therefore, any function whose graph is not a straight line is a nonlinear function.

For example, the following function is linear because its graph is a straight line:

$$g(x) = 3x + 2$$

A constant function like $g(x) = 1$ is also linear. However, the following is a nonlinear function because the exponent of x is 2:

$$g(x) = x^2 + 2$$

Transcendental functions are also nonlinear. They are functions like $g(x) = \log x$, or $g(x) = e^x$, where $e = 2.718\dots$ is the base of the natural logarithm. *Trigonometric functions* like sine and cosine, their inverses, and functions like tangent that are built from sine and cosine are also transcendental functions. These functions are transcendental because you can't form them as finite combinations of elementary algebra operations. They are nonlinear because their graphs aren't straight lines.

The network needs nonlinear activation functions; otherwise, it will be able to learn only linear mappings, and linear mappings aren't sufficient to make the networks generally useful. Consider a trivial network of two nodes, each with one input. This means there's one weight and one bias value per node, and the output of the first node is the input of the second. If we set $h(x) = 5x - 3$, a linear function, then for input x the network computes output a_1 to be the following:

$$\begin{aligned} a_1 &= h(w_1 a_0 + b_1) \\ &= h(w_1 h(w_0 x + b_0) + b_1) \\ &= h(w_1 (5(w_0 x + b_0) - 3) + b_1) \\ &= h(w_1 (5w_0 x + 5b_0 - 3) + b_1) \\ &= h(5w_1 w_0 x + 5w_1 b_0 - 3w_1 + b_1) \\ &= 5(5w_1 w_0 x + 5w_1 b_0 - 3w_1 + b_1) - 3 \\ &= (25w_1 w_0)x + (25w_1 b_0 - 15w_1 + 5b_1 - 3) \\ &= Wx + B \end{aligned}$$

Here $W = 25w_1w_0$ and $B = 25w_1b_0 - 15w_1 + 5b_1 - 3$, which is also a linear function, another line with slope W and intercept B since neither W nor B depend on x . Therefore, a neural network with linear activation functions would learn only a linear model since the composition of linear functions is also linear. It's precisely this limitation of linear activation functions that caused the first neural network “winter” in the 1970s: Research into neural networks was effectively abandoned because they were thought to be too simple to learn complex functions.

Okay, so we want nonlinear activation functions. Which ones? We have an infinite number of possibilities. In practice, a few have risen to the top because of their proven usefulness or nice properties. Traditional neural networks used either sigmoid activation functions or hyperbolic tangents. A *sigmoid* is as follows:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

And here is the *hyperbolic tangent*:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

Plots of both functions are in Figure 5-3, with the sigmoid on the top and the hyperbolic tangent on the bottom.

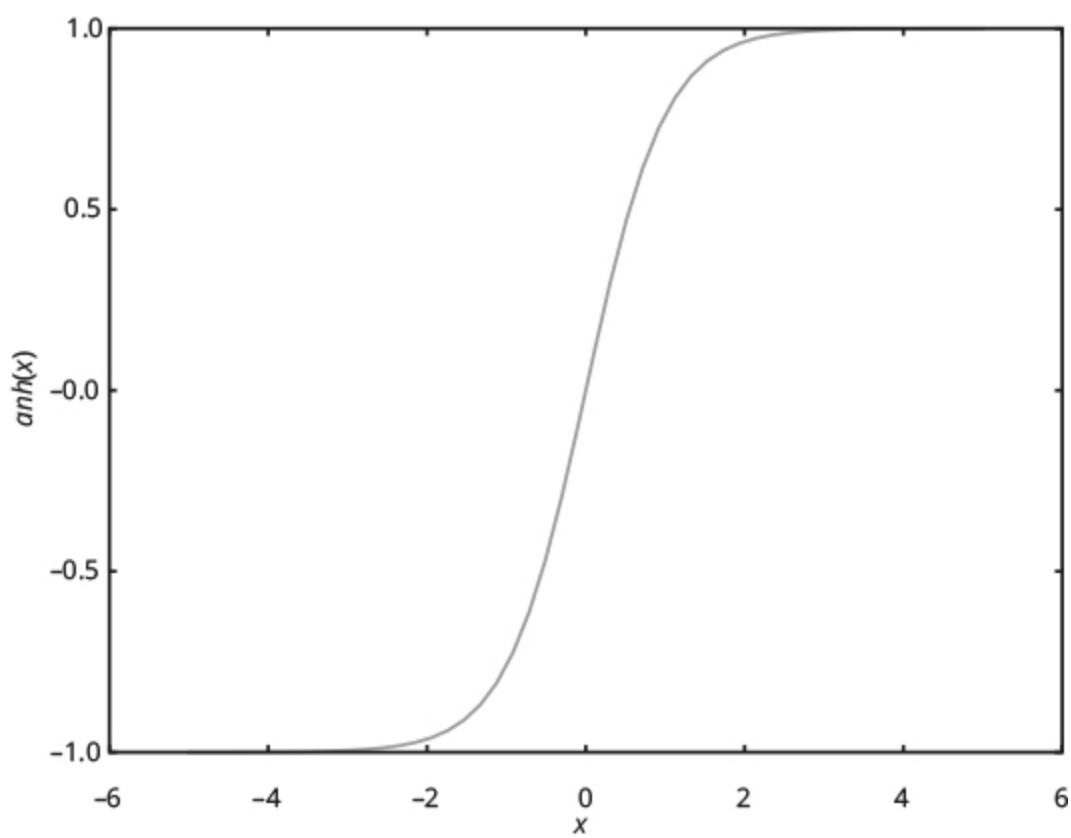
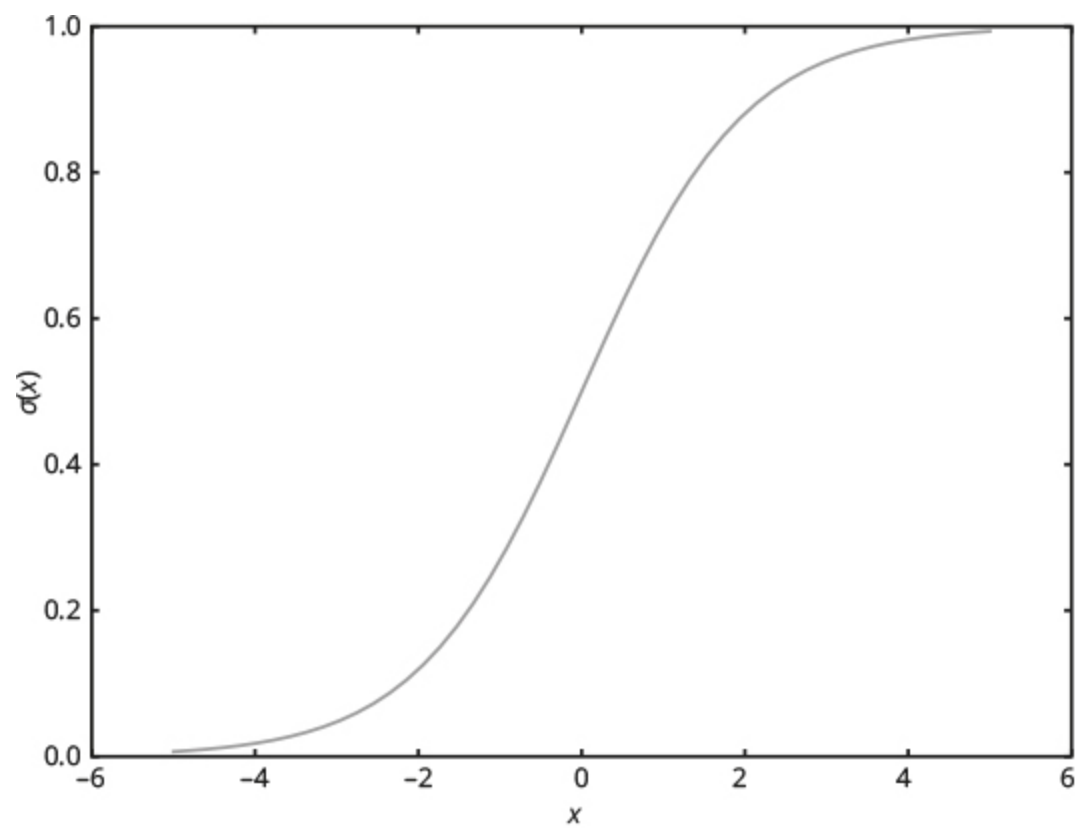


Figure 5-3: A sigmoid function (top) and a hyperbolic tangent function (bottom). Note that the y-axis scales are not the same.

Both of these functions have roughly the same S shape. The sigmoid runs from 0 as you go farther left along the x-axis to 1 as you go to the right. At 0, the function value is 0.5. The hyperbolic tangent does the same but goes from -1 to $+1$ and is 0 at $x = 0$.

The sigmoid and hyperbolic tangent have been largely replaced by the rectified linear unit (ReLU). The ReLU is simple and has convenient properties for neural networks. Even though the word *linear* is in the name, the ReLU is a nonlinear function; its graph is not a straight line. When we discuss backpropagation training of neural networks in Chapter 6, we'll learn what caused this change.

The ReLU is as follows:

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{otherwise} \end{cases}$$

Figure 5-4 shows the ReLU.

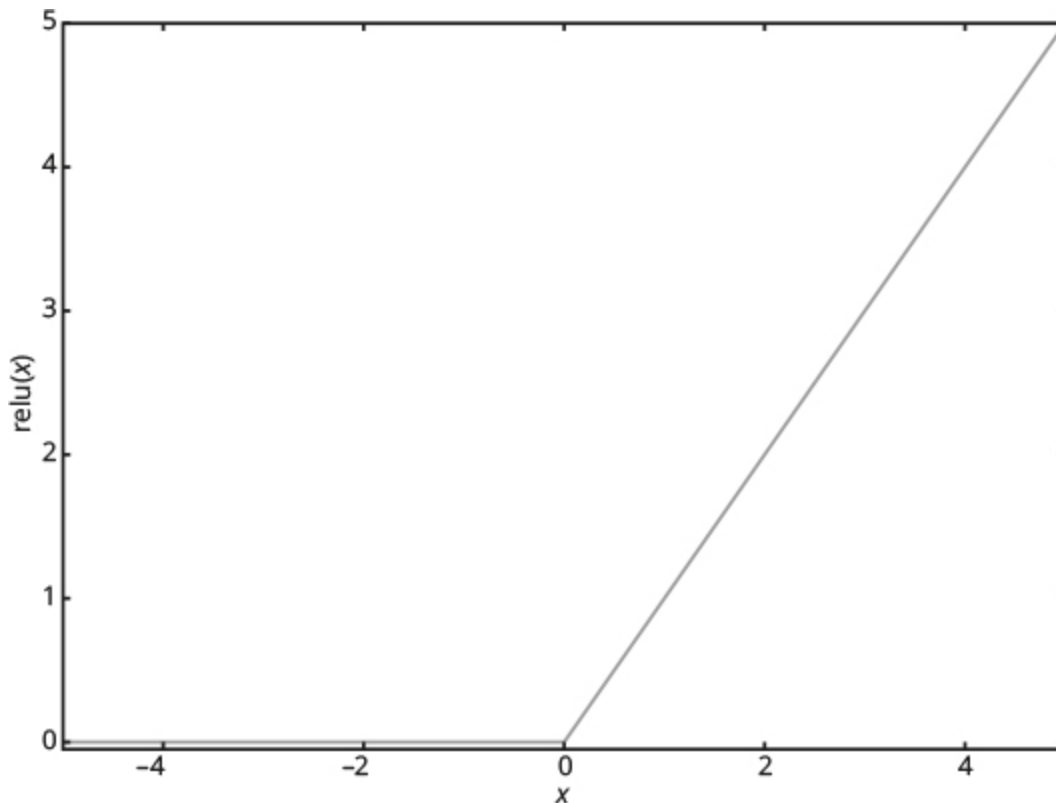


Figure 5-4: The rectified linear activation function, $\text{ReLU}(x) = \max(0, x)$

The ReLU is *rectified* because it removes the negative values and replaces them with 0. In truth, the machine learning community uses several versions of this function, but all are essentially replacing negative values with a constant or another value. The piecewise nature of the ReLU is what makes it nonlinear and therefore suitable for use as a neural network activation function.

The ReLU is also computationally simple, far faster to calculate than either the sigmoid or the hyperbolic tangent. This is because the latter functions use e^x , which in computer terms means a call to the `exp` function. This function is typically implemented as a sum of terms of a series expansion, translating into dozens of floating-point operations in place of the single `if` statement necessary to implement a ReLU. Small savings like this add up in an extensive network with potentially thousands of nodes.

We've discussed nodes and how they work, and hinted that nodes are connected to form networks. Let's look more closely at how nodes

are connected, the architecture of the network.

The Architecture of a Network

Standard neural networks like the ones we're working with in this chapter are built in layers, as you saw in Figure 5-1. We don't need to do this, but as we'll see, this buys us some computational simplicity and greatly simplifies training. A feed-forward network has an input layer, one or more hidden layers, and an output layer. The input layer is simply the feature vector, and the output layer is the prediction (probability). If the network is for a multiclass problem, the output layer might have more than one node, with each node representing the model's prediction for each of the possible classes of inputs.

The hidden layers are made of nodes, and the nodes of layer i accept as input the output of the nodes of layer $i - 1$ and pass their outputs to the inputs of the nodes of layer $i + 1$. The layers are typically fully connected, meaning every output of every node of layer $i - 1$ is used as an input to every node of layer i . Again, we don't need to do this, but it simplifies the implementation.

The number of hidden layers and the number of nodes in each hidden layer define the architecture of the network. It has been proven that a single hidden layer with enough nodes can learn any function mapping. This is good because it means neural networks are applicable to machine learning problems since, in the end, the model acts as a complex function mapping inputs to output labels and probabilities. However, like many theoretical results, this doesn't mean that it's practical for a single-layer network to be used in all situations. As the number of nodes (and layers) in a network grows, so, too, does the number of parameters to learn (weights and biases), and therefore the amount of training data needed goes up as well. It's the curse of dimensionality again.

Issues like these stymied neural networks for a second time in the 1980s. Computers were too slow to train large networks and too little data was usually available to train the network anyway. Practitioners knew that if both of these situations changed, it would become possible to train large networks that would be far more capable than the small

networks of the time. Fortunately for the world, the situation changed in the early 2000s.

Selecting the proper neural network architecture has a huge impact on whether your model will learn anything. This is where experience and intuition come in. Selecting the right architecture is the dark art of using neural networks. Let's give some (crude) rules of thumb:

- If your input has definite spatial relationships, like the parts of an image, you might want to use a convolutional neural network instead (Chapter 9).
- Use no more than three hidden layers. Recall, in theory, one sufficiently large hidden layer is all that is needed, so use as few hidden layers as necessary. If the model learns with one hidden layer, add a second to see if that improves performance.
- The number of nodes in the first hidden layer should match or (ideally) exceed the number of input vector features.
- Except for the first hidden layer (see previous rule), the number of nodes per hidden layer should be the same as or a value between the number of nodes in the preceding layer and the following layer. If layer $i - 1$ has N nodes and layer $i + 1$ has M nodes, then layer i might be good with $N \leq x \leq M$ nodes.

The first rule says that a traditional neural network best applies when your input doesn't have spatial relationships—that is, you have a feature vector, not an image. Also, when your input dimension is small, or when you don't have a lot of data (which makes it hard to train a larger convolutional network), you should give a traditional network a try. If a traditional neural network might be called for, start small, and grow it as long as performance improves.

Output Layers

The last layer of a neural network is the output layer. If the network is modeling a continuous value, known as *regression*, then the output layer is a node that doesn't use an activation function; it simply reports the

argument to h in Figure 5-2. This is the same as saying that the activation function is the identity function, $h(x) = x$.

Our neural networks are for classification; we want them to output a decision value. If we have two classes labeled 0 and 1, we make the activation function of the final node a sigmoid. This outputs a value between 0 and 1 that we can interpret as a likelihood or probability that the input belongs to class 1. We make our classification decision based on the output value with a simple rule: If the activation value is less than 0.5, call the input class 0; otherwise, call it class 1. We'll see in Chapter 8 how changing this threshold of 0.5 can be used to tune a model's performance for the task at hand.

If we have more than two classes, we need to take a different approach. Instead of a single node in the output layer, we'll have N output nodes, one for each class, each one using the identity function for h . Then we apply a softmax operation to these N outputs and select the output with the largest softmax value.

Let's illustrate what we mean by softmax. Suppose we have a dataset containing four classes. What they represent doesn't really matter; the network doesn't know what they represent either. The classes are labeled 0, 1, 2, and 3. So, $N = 4$ means our network will have four output nodes, each one using the identity function for h . This looks like Figure 5-5, which also shows the softmax operation and the resulting output vector.

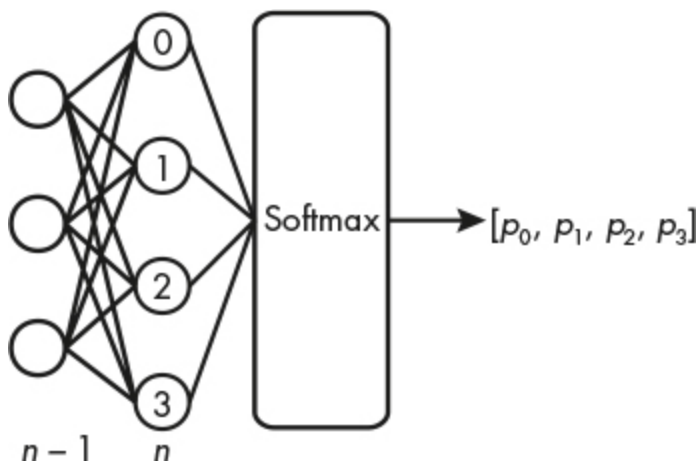


Figure 5-5: The last hidden layer $n - 1$ and output layer (n , nodes numbered) for a neural network with four classes. The softmax operation is applied, producing a four-element output vector, $[p_0, p_1, p_2, p_3]$.

We select the index of the largest value in this output vector as the class label for the given input feature vector. The softmax operation ensures that the elements of this vector sum to 1, so we can again be a bit sloppy and call these values the probability of belonging to each of the four classes. That is why we take only the largest value to decide the output class label.

The softmax operation is straightforward; the probability for each of the outputs is as follows:

$$p_i = \frac{e^{a_i}}{\sum_j e^{a_j}}$$

Here a_i is the i th output, and the denominator is the sum over all the outputs. For the example, $i = 0, 1, 2, 3$, and the index of the largest value will be the class label assigned to the input.

As an example, assume the output of the four last layer nodes is as follows:

$$a_0 = 0.2$$

$$a_1 = 1.3$$

$$a_2 = 0.8$$

$$a_3 = 2.1$$

Then calculate the softmax as follows:

$$p_0 = e^{0.2} / (e^{0.2} + e^{1.3} + e^{0.8} + e^{2.1}) = 0.080$$

$$p_1 = e^{1.3} / (e^{0.2} + e^{1.3} + e^{0.8} + e^{2.1}) = 0.240$$

$$p_2 = e^{0.8} / (e^{0.2} + e^{1.3} + e^{0.8} + e^{2.1}) = 0.146$$

$$p_3 = e^{2.1} / (e^{0.2} + e^{1.3} + e^{0.8} + e^{2.1}) = 0.534$$

Select class 3 because p_3 is the largest. Note that the sum of the p_i values is 1.0, as we would expect.

Two points should be mentioned here. In the preceding equations, we used the sigmoid to calculate the output of the network. If we set the number of classes to 2 and calculate the softmax, we get two output values: One is some p , and the other is $1 - p$. This is identical to the sigmoid alone, selecting the probability of the input being of class 1.

The second point has to do with implementing the softmax. If the network outputs, the a values, are large, then e^a might be very large, which the computer won't like. Precision will be lost, at least, or the value might overflow and make the output meaningless. Numerically, if we subtract the largest a value from all the others before calculating the softmax, we'll take the exponential over smaller values that are less likely to overflow. Doing this for the preceding example gives new a values:

$$a'_0 = 0.2 - 2.1 = -1.9$$

$$a'_1 = 1.3 - 2.1 = -0.8$$

$$a'_2 = 0.8 - 2.1 = -1.3$$

$$a'_3 = 2.1 - 2.1 = 0.0$$

We subtract 2.1 because that is the largest a value. This leads to precisely the same p values we found before, but this time protected against overflow in the case that any of the a values are too large.

Weight and Bias Representation

Before we move on to an example neural network, let's revisit the weights and biases and see that we can greatly simplify the

implementation of a neural network by viewing it in terms of matrices and vectors.

Consider the mapping from an input feature vector of two elements to the first hidden layer with three nodes (a_1 in Figure 5-1). Let's label the edges between the two layers (the weights) as W_{ij} with $i = 0, 1$ for the inputs x_0 and x_1 and $j = 0, 1, 2$ for the three hidden-layer nodes numbered from top to bottom in the figure. Additionally, we need three bias values that aren't shown in the figure, one for each hidden node. We'll call these b_0 , b_1 , and b_2 , again, top to bottom.

To calculate the outputs of the activation functions, h , for the three hidden nodes, we need to find the following:

$$\begin{aligned}a_0 &= h(w_{00}x_0 + w_{10}x_1 + b_0) \\a_1 &= h(w_{01}x_0 + w_{11}x_1 + b_1) \\a_2 &= h(w_{02}x_0 + w_{12}x_1 + b_2)\end{aligned}$$

But, remembering how matrix multiplication and vector addition work, we see that this is exactly as follows:

$$\vec{a} = h\left(\begin{bmatrix} w_{00} & w_{10} \\ w_{01} & w_{11} \\ w_{02} & w_{12} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}\right) = h\left(W\vec{x} + \vec{b}\right)$$

Here $\vec{a} = (a_0, a_1, a_2)$, $\vec{x} = (x_0, x_1)$, $\vec{b} = (b_0, b_1, b_2)$, and W is a 3×2 matrix of weight values. In this case, the activation function, h , is given a vector of input values and produces a vector of output values. This means it simply applies h to every element of $W\vec{x} + \vec{b}$. For example, applying h to a vector $\vec{x}$ with three elements is

$$h(\vec{x}) = h\left((x_0, x_1, x_2)\right) = (h(x_0), h(x_1), h(x_2))$$

with h applied separately to each element of $\vec{x}$.

Since the NumPy Python module is designed to work with arrays, and matrices and vectors are arrays, we arrive at the pleasant conclusion that the weights and biases of a neural network can be stored in NumPy arrays and we need only simple matrix operations (calls to `np.dot`) and addition to work with a fully connected neural network. This is why we want to use fully connected networks: Their implementation is straightforward.

To store the network of Figure 5-1, we need a weight matrix and bias vector between each layer, giving us three matrices and three vectors: a matrix and vector each for the input to the first hidden layer, the first hidden layer to the second, and the second hidden layer to the output. The weight matrices have the dimensions 3×2 , 2×3 , and 1×2 , respectively. The bias vectors have lengths of 3, 2, and 1.

A Simple Neural Network Implementation

In this section, we'll implement the sample neural network of Figure 5-1 and train it on two features from the iris dataset. We'll implement the network from scratch but use `sklearn` to train it. The goal of this section is to see how straightforward it is to implement a simple neural network. Hopefully, this clears some of the fog that might be hanging around from the discussion of the previous sections.

The network of Figure 5-1 accepts an input feature vector with two features. It has two hidden layers, one with three nodes and the other with two nodes. It has one sigmoid output. The activation functions of the hidden nodes are also sigmoids.

Building the Dataset

Before we look at the neural network code, let's build the dataset we'll train against and see what it looks like. We know the iris dataset already, but for this example, we'll use only two classes and only two of the four features. The code to build the train and test datasets is in Listing 5-1.

```
nn_iris_dataset.py
import numpy as np
```

```

❶ d = np.load("iris_train_features_augmented.npy")
   l = np.load("iris_train_labels_augmented.npy")
   d1 = d[np.where(l == 1)]
   d2 = d[np.where(l == 2)]

❷ a = len(d1)
   b = len(d2)
   x = np.zeros((a+b, 2))
   x[:a,:] = d1[:,2:]
   x[a:,:] = d2[:,2:]

❸ y = np.array([0]*a+[1]*b)
   i = np.argsort(np.random.random(a+b))
   x = x[i]
   y = y[i]

❹ np.save("iris2_train.npy", x)
   np.save("iris2_train_labels.npy", y)

❺ d = np.load("iris_test_features_augmented.npy")
   l = np.load("iris_test_labels_augmented.npy")
   d1 = d[np.where(l == 1)]
   d2 = d[np.where(l == 2)]
   a = len(d1)
   b = len(d2)
   x = np.zeros((a+b, 2))

   x[:a,:] = d1[:,2:]
   x[a:,:] = d2[:,2:]
   y = np.array([0]*a+[1]*b)
   i = np.argsort(np.random.random(a+b))
   x = x[i]
   y = y[i]
   np.save("iris2_test.npy", x)
   np.save("iris2_test_labels.npy", y)

```

Listing 5-1: Building the simple example dataset

This code is straightforward data munging. We start with the augmented dataset and load the samples and labels ❶. We want only class 1 and class 2, so we find the indices of those samples and pull them

out. We keep only features 2 and 3 and put them in x ❷. Next, we build the labels (y) ❸. Note, we recode the class labels to 0 and 1. We scramble the order of the samples and write the new dataset to disk ❹. Last of all, we repeat this process to build the test samples ❺.

Figure 5-6 shows a subset of the training set. We can plot it in this case because we have only two features.

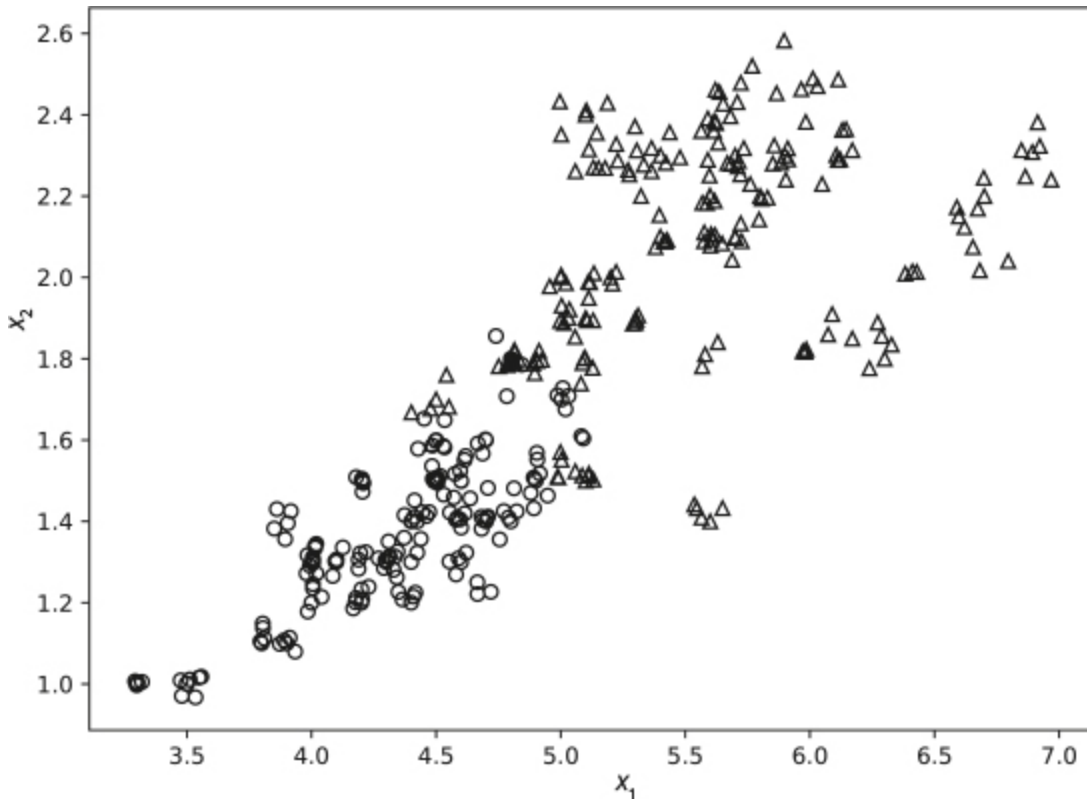


Figure 5-6: A subset of the training data for the two-class, two-feature iris dataset

We immediately see that this dataset is not trivially separable. We can't draw a simple line to correctly split the training set into two groups, one all class 0 and the other all class 1. This makes life a little more interesting.

Implementing the Neural Network

Let's see how to implement the network of Figure 5-1 in Python using NumPy. We'll assume that it's already trained, meaning we already know all the weights and biases. Run the code in Listing 5-3

(*nn_iris_mlpclassifier.py*) to find the weights and biases before running the code in Listing 5-2.

```
nn_iris_evaluate.py
import numpy as np
import pickle
import sys

def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def evaluate(x, y, w):
    ❶ w12, b1, w23, b2, w34, b3 = w
    nc = nw = 0
    prob = np.zeros(len(y))
    for i in range(len(y)):
        a1 = sigmoid(np.dot(x[i], w12) + b1)
        a2 = sigmoid(np.dot(a1, w23) + b2)
        prob[i] = sigmoid(np.dot(a2, w34) + b3)
        z = 0 if prob[i] < 0.5 else 1
    ❷ if (z == y[i]):
        nc += 1
    else:
        nw += 1
    return [float(nc) / float(nc + nw), prob]

❸ xtest = np.load("iris2_test.npy")
ytest = np.load("iris2_test_labels.npy")
❹ weights = pickle.load(open("iris2_weights.pkl", "rb"))
score, prob = evaluate(xtest, ytest, weights)
print()
for i in range(len(prob)):
    print("%3d: actual: %d predict: %d prob: %0.7f" %
          (i, ytest[i], 0 if (prob[i] < 0.5) else 1, prob[i]))
print("Score = %0.4f" % score)
```

Listing 5-2: Using the trained weights and biases to classify held-out test samples

Notice how short the code is. The `evaluate` function implements the network. We also need to define `sigmoid` as NumPy doesn't have it natively. The main code loads the test samples (`xtest`) and associated labels (`ytest`) ❸. These are the files generated by the preceding code, so we know that `xtest` has a shape of 23×2 because we have 23 test samples, and each has two features. Similarly, `ytest` is a vector with 23 labels.

When we train this network, we'll store the weights and biases as a list of NumPy arrays. The Python way to store a list on disk is via the `pickle` module, so we use `pickle` to load the list from disk ❹. The list `weights` has six elements representing the three weight matrices and three bias vectors that define the network. These are the "magic" numbers that our training has conditioned to the dataset. Finally, we call `evaluate` to run each of the test samples through the network. This function returns the score (accuracy) and the output probabilities for each sample (`prob`). The remainder of the code displays the sample number, actual label, predicted label, and associated output probability of being class 1. Finally, the score is shown.

The network is implemented in `evaluate`; let's see how. First, pull the individual weight matrices and bias vectors from the supplied weight list ❶. These are NumPy arrays: `w12` is a 2×3 matrix mapping the two-element input to the first hidden layer with three nodes, `w23` is a 3×2 matrix mapping the first hidden layer to the second hidden layer, and `w34` is a 2×1 matrix mapping the second hidden layer to the output. The bias vectors are `b1`, three elements; `b2`, two elements; and `b3`, a single element (a scalar).

Notice the weight matrices are not of the same shape as we previously indicated they would be. They are transposes. This is because we're multiplying vectors, which are treated as 1×2 matrices, by the weight matrices. Because scalar multiplication is commutative, meaning $ab = ba$, we see that we're still calculating the same argument value for the activation function.

Next, `evaluate` sets the number correct (`nc`) and number wrong (`nw`) counters to 0. These are for calculating the overall score across the entire test set. Similarly, we define `prob`, a vector to hold the output probability value for each of the test samples.

The loop applies the entire network to each test sample. First, we map the input vectors to the first hidden layer and calculate a_1 , a vector of three numbers, the activation for each of the three hidden nodes. We then take these first hidden-layer activations and calculate the second hidden-layer activations, a_2 . This is a two-element vector as there are two nodes in the second hidden layer. Next, we calculate the output value for the current input vector and store it in the `prob` array. The class label, z , is assigned by checking whether the output value of the network is < 0.5 . Finally, we increment the correct (`nc`) or incorrect (`nw`) counters based on the actual label for this sample (`y[i]`) ❷. When all samples have been passed through the network, the overall accuracy is returned as the number of correctly classified samples divided by the total number of samples.

This is all well and good; we can implement a network and pass input vectors through it to see how well it does. If the network had a third hidden layer, we would pass the output of the second hidden layer (a_2) through it before calculating the final output value.

Training and Testing the Neural Network

Listing 5-2 applies the trained model to the test data. To train the model in the first place, we'll use `sklearn`. The code to train the model is in Listing 5-3.

```
nn_iris_mlpclassifier.py
import numpy as np
import pickle
from sklearn.neural_network import MLPClassifier

xtrain = np.load("iris2_train.npy")
ytrain = np.load("iris2_train_labels.npy")

xtest = np.load("iris2_test.npy")
ytest = np.load("iris2_test_labels.npy")
```

```
❶ clf = MLPClassifier(
    ❷ hidden_layer_sizes=(3, 2),
```

```

        ❸ activation="logistic",
            solver="adam", tol=1e-9,
            max_iter=5000,
            verbose=True)
clf.fit(xtrain, ytrain)
prob = clf.predict_proba(xtest)
score = clf.score(xtest, ytest)

4 w12 = clf.coefs_[0]
  w23 = clf.coefs_[1]
  w34 = clf.coefs_[2]
  b1 = clf.intercepts_[0]
  b2 = clf.intercepts_[1]
  b3 = clf.intercepts_[2]
  weights = [w12, b1, w23, b2, w34, b3]
  pickle.dump(weights, open("iris2_weights.pkl", "wb"))

print()
print("Test results:")
print(" Overall score: %0.7f" % score)
print()
for i in range(len(ytest)):
    p = 0 if (prob[i,1] < 0.5) else 1
    print("%03d: %d - %d, %0.7f" % (i, ytest[i], p, prob[i,1]))
print()

```

Listing 5-3: Using sklearn to train the iris neural network

First, we load the training and testing data from disk. These are the same files we created previously. Then we set up the neural network object, an instance of `MLPClassifier` ❶. The network has two hidden layers, the first with three nodes and the second with two nodes ❷. This matches the architecture in Figure 5-1. The network is also using logistic layers ❸. This is another name for a sigmoid layer. We train the model by calling `fit` just as we did for other sklearn model types. Since we set `verbose` to `True`, we'll get output showing us the loss for each iteration.

Calling `predict_proba` gives us the output probabilities on the test data. This method is also supported by most other sklearn models. This is the model's certainty as to the assigned output label. We then call `score` to calculate the score over the test set as we've done before.

We want to store the learned weights and biases so we can use them with our test code. We can pull them directly from the trained model ④. These are packed into a list (`weights`) and dumped to a Python pickle file.

The remaining code prints the results of running the sklearn trained model against the held-out test data. For example, a particular run of this code gives the following:

Test results:

Overall score: 1.0000000

```
000: 0 - 0, 0.0705069
001: 1 - 1, 0.8066224
002: 0 - 0, 0.0308244
003: 0 - 0, 0.0205917
004: 1 - 1, 0.9502825
005: 0 - 0, 0.0527558
006: 1 - 1, 0.9455174
007: 0 - 0, 0.0365360
008: 1 - 1, 0.9471218
009: 0 - 0, 0.0304762
010: 0 - 0, 0.0304762
011: 0 - 0, 0.0165365
012: 1 - 1, 0.9453844
013: 0 - 0, 0.0527558
014: 1 - 1, 0.9495079
015: 1 - 1, 0.9129983
016: 1 - 1, 0.8931552
017: 0 - 0, 0.1197567
018: 0 - 0, 0.0406094
019: 0 - 0, 0.0282220
020: 1 - 1, 0.9526721
021: 0 - 0, 0.1436263
022: 1 - 1, 0.9446458
```

This indicates that the model was perfect against the small test dataset. The output shows the sample number, the actual class label, the assigned class label, and the output probability of being class 1. If we run the pickle file holding the sklearn network's weights and biases through our evaluation code, we see that the output probabilities are precisely the same as the preceding code, indicating that our hand-generated neural network implementation is working correctly.

Summary

In this chapter, we discussed the anatomy of a neural network. We described the architecture, the arrangement of nodes, and the connections between them. We discussed the output-layer nodes and the functions they compute. We then saw that all the weights and biases could be conveniently represented by matrices and vectors. Finally, we presented a simple network for classifying a subset of the iris data and showed how it could be trained and evaluated.

Now that we have our feet wet, let's move on and dive into the theory behind neural networks.

6

TRAINING A NEURAL NETWORK



In this chapter, we'll discuss how to train a neural network. We'll look at the standard approaches and tricks being used in the field today. There will be some math, some hand-waving, and a whole host of new terms and concepts. But you don't need to follow the math at a deep level: We'll gloss over details as needed to get the main point across.

This chapter is perhaps the most challenging in the book, at least conceptually. It certainly is mathematically. While it's crucially important to building intuition and understanding, sometimes we get impatient and like to dive into topics first to test the waters. Thanks to preexisting libraries, we can do that here. If you want to play around with neural networks before learning how they work, jump to Chapter 7 before coming back here to fill in the theory. But do come back.

It's possible to learn to use powerful toolkits like `sklearn` and `Keras` without understanding how they work. That approach should not satisfy anyone, though the temptation is real. Understanding how these algorithms work is well worth your time.

A High-Level Overview

Let's begin with an overview of the concepts we'll discuss. Read on, but don't fret if the concepts are unclear. Instead, try to get a feel for the overall process.

The first step in training a neural network is selecting intelligent initial values for the weights and biases. We then use gradient descent to modify these weights and biases so that we reduce the error over the training set. We'll use the average value of the loss function to measure the error, which tells us how wrong the network currently is located. We know if the network is right or wrong because we have the expected output for each input sample in the training set (the class label).

Gradient descent is an algorithm that requires gradients. For now, think of gradients as measures of steepness. The larger the gradient, the steeper the function is at that point. To use gradient descent to search for the smallest value of the loss function, we need to be able to find gradients. For that, we'll use backpropagation. This is the fundamental algorithm of neural networks that allows them to learn successfully. It gives us the gradients we need by starting at the output of the network and moving back through the network toward the input. Along the way, it calculates the gradient value for each weight and bias.

With the gradient values, we can use the gradient-descent algorithm to update the weights and biases so that the next time we pass the training samples through the network, the average of the loss function will be less than it was before. In other words, our network will be less wrong. This is the goal of training, and we hope it results in a network that has learned general features of the data.

Learning general features of the dataset requires regularization. This has many approaches, and we'll discuss the main ones. Without regularization, the training process is in danger of overfitting, and we could end up with a network that doesn't generalize. But with regularization, we can be successful and get a useful model.

The following sections introduce gradient descent, backpropagation, loss functions, weight initialization, and regularization. These are the main components of successful neural network training. We don't need to understand these in all their gory mathematical details; instead, we need to understand them conceptually so we can build an intuitive

approach to what it means to train a neural network. With this intuition, we'll be able to make meaningful use of the parameters that sklearn and Keras give us for training.

Gradient Descent

The standard way to train a neural network is to use gradient descent. Let's parse this term. The word *descent* means to go down from somewhere higher up. A *gradient*, in short, indicates how something changes with respect to how something else changes. Measuring how much one thing changes as another changes is something we're all familiar with. For example, speed measures how position changes as time changes. We even say it in words: miles per hour or kilometers per hour.

You might already be familiar with the gradient in another context. Consider the equation of a line, where m is the slope and b is the y-axis intercept:

$$y = mx + b$$

The slope is how quickly the line's y position changes with each change in the x position. If we know two points that are on the line, (x_0, y_0) and (x_1, y_1) , then we can calculate the slope as follows:

$$m = \frac{y_0 - y_1}{x_0 - x_1}$$

In words, we might say, “y’s per x .” It’s a measure of the steepness or shallowness of the line: its gradient. In mathematics, when talking about a change in a variable, we put a Δ (delta) in front of the variable. So, we might write the slope of a line as follows to drive home the point that the slope is the change in y for each change in x :

$$m = \frac{\Delta y}{\Delta x}$$

Fortunately for us, it turns out that not only do lines have a slope at each point, but most functions do as well. However, except for straight lines, this slope changes from point to point. A visual should help here; consider Figure 6-1.

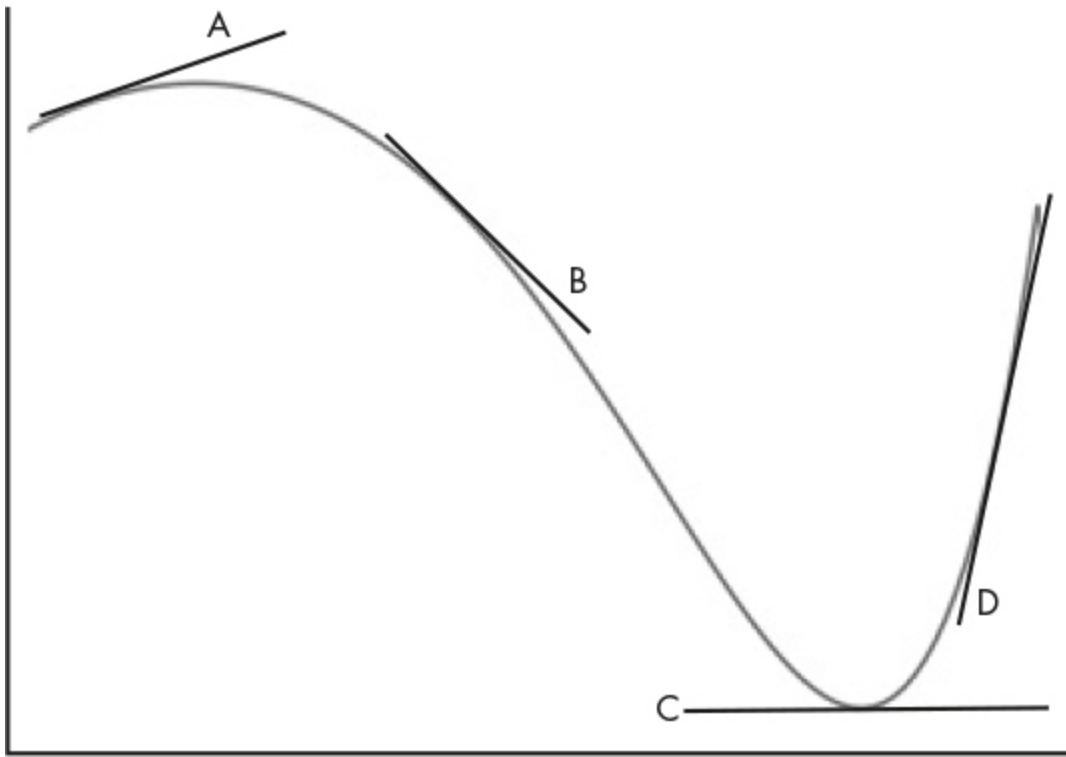


Figure 6-1: A function with several tangent lines indicated

The graph in Figure 6-1 is of a polynomial. The lines drawn on the figure that are just touching the function are tangent lines. As lines, they have a slope we can see in the plot. Now imagine moving one of the lines over the function so that it continues to touch the function at only one point; imagine how the slope of the line changes as it moves.

It turns out that how the slope changes over the function is itself a function called the *derivative*. Given a function and an x value, the derivative tells us the slope of the function at that point, x . The fact that functions have derivatives is a fundamental insight of calculus and of fundamental importance to us.

The notion of a derivative is essential because for single-variable functions, the derivative at the point x is the gradient at x ; it's the direction in which the function is changing. If we want to find the

minimum of the function, the x that gives us the smallest y , we want to move in the direction *opposite* to the gradient as that will move us in the direction of the minimum.

We can write the derivative in many ways, but the following approach echoes the idea of the slope, how y changes for a change in x :

$$\frac{dy}{dx}$$

We'll return to this form next when discussing the backpropagation algorithm. That's it for the gradient; now let's take a closer look at descent.

Finding Minimums

Since we want a model that makes few mistakes, we need to find the set of parameters that lead to a small value for the loss function. In other words, we need to find a minimum of the loss function.

Look again at Figure 6-1. The minimum is on the right, where tangent line C is. We can see it's the minimum, and that the gradient is 0 there. This tells us we're at a minimum (or maximum). If we start at B, we see that the slope of the tangent line is negative (down and to the right). Therefore, we need to move to an x value in the positive direction because this is opposite to the sign of the gradient. Doing this takes us closer to the minimum at C. Similarly, if we start at D, the slope of the tangent line is positive (up and to the right), meaning we need to move in the negative x direction, again toward C, to move closer to the minimum. All of this hints at an algorithm for finding the minimum of a function: Pick a starting point (an x value) and use the gradient to move to a lower point.

For simple functions of just x , like those of Figure 6-1, this approach works nicely, assuming we start in a good place like B or D. When we move to more than one dimension, this approach continues to work nicely, provided we start in a good place with our initial guess.

Working still with Figure 6-1 and assuming we're starting at B, we see that the gradient tells us to move to the right, toward C. But how do we select the next x value to consider, to move us closer to C? This is

the *step size*, and it tells us how big a jump we make from one x position to the next. Step size is a parameter we have to choose; in practice, this value, called the *learning rate*, is often fluid and gets smaller as we move, under the assumption that as we move, we get closer to the minimum value and therefore need smaller steps.

This is all well and good, but we have a small problem. What if instead of starting at B or D, we start at A? The gradient at A points us to the left, not the right. In this case, our simple algorithm fails: It moves us to the left, and we never reach C. The figure shows only one minimum, at C, but we can imagine a second minimum (say, to the left of A) that doesn't go as low (doesn't have as small a y value) as C. If we start at A, we move toward this minimum, and not the one at C. Our algorithm falls into a local minimum. Once in, our algorithm can't get us out, and we won't be able to find the global minimum at C. We'll see that this is a genuine issue for neural networks, but one that for modern deep networks is, almost magically, not much of an issue after all.

How does all this help us train a neural network? The gradient tells us how a small change in x changes y . If x is one of the parameters of our network, and y is the error given by the loss function, then the gradient tells us how much a change in that parameter affects the overall error of the network. Once we know that, we're in a position to modify the parameter by an amount based on the gradient, which will move us toward a minimum error. When the error over the training set is at a minimum, we can claim that the network has been trained.

Let's talk a bit more about the gradients and parameters. All our discussion to this point, based on Figure 6-1, has been rather 1D; our functions are functions of x only. We talked about changing one value, the position along the x -axis, to see how it affects the y position. In reality, we're not working with just one dimension. Every weight and bias in our network is a parameter, and the loss function value depends upon all of them. The simple network in Figure 5-1 (see page 130) alone has 20 parameters, meaning that the loss function is a 20-dimensional function. Regardless, our approach remains much the same: If we know the gradient for each parameter, we can still apply our

algorithm in an attempt to locate a set of parameters minimizing the loss.

Updating the Weights

We'll talk about how to get gradient values in a bit, but for the time being let's assume we have them already. We'll say that we have a set of numbers that tells us how, given the current configuration of the network, a change in any weight or bias value changes the loss. With that knowledge, we can apply gradient descent: We adjust the weight or bias by a fraction of that gradient value to move us, collectively, toward a minimum of the entire loss function.

Mathematically, we update each weight and bias by using a simple rule:

$$w \leftarrow w - \eta \Delta w$$

Here w is one of the weights (or biases), η (eta) is the learning rate (the step size), and Δw is the gradient value.

The five (deceptively) simple steps for using gradient descent to train a neural network are as follows:

1. Pick intelligent starting values for the weights and biases.
2. Run the training set through the network by using its current weights and biases and calculate the average loss.
3. Use this loss to get the gradient for each weight and bias.
4. Update the weight or bias value by the step size times the gradient value.
5. Repeat from step 2 until the loss is low enough.

The algorithm appears simple, but as they say, the devil is in the details. We have to make choices at every step, and every choice prompts further questions. For example, step 1 says to pick intelligent starting values. What should they be? It turns out that successfully training a neural network depends critically on choosing good initial

values. We already saw how this might be so in our preceding example using Figure 6-1, where if we start at A, we won't find the minimum at C. Much research has been conducted over the years related to step 1.

Step 2 is straightforward; it's the forward-pass through the network. We haven't talked in detail about the loss function itself; for now, just think of it as a function measuring the effectiveness of the network on the training set.

Step 3 is a black box for the time being. We'll explore how to do it shortly. For now, assume we can find the gradient values for each parameter.

Step 4 follows the form of the previous equation that moves the parameter from its current value to one that reduces the overall loss. In practice, the simple form of this equation is not sufficient; other terms, like momentum, preserve a fraction of the previous weight change for the next iteration (next pass of the training data through the network) so that parameters don't change too wildly. We'll revisit momentum later. For now, let's look at a variation of gradient descent that's actually used to train deep networks.

Stochastic Gradient Descent

The steps in our algorithm describe gradient-descent training of a neural network. In practice, this basic idea has many flavors. One that's in widespread use and works well empirically is stochastic gradient descent (SGD), where *stochastic* refers to a random process. We'll see next why the word *stochastic* goes before *gradient descent* in this case.

Batches and Minibatches

Step 2 of our gradient-descent algorithm says to run the complete training set through the network with the current values of the weights and biases. This approach is called *batch training*, so named because we use all the training data to estimate the gradients. Intuitively, this is reasonable to do: We've carefully constructed the training set to be a fair representation of the unknown parent process that generates the

data, and it's this parent process we want the network to successfully model for us.

If our dataset is small, like the original iris dataset of Chapter 2, then batch training makes sense. But what if our training dataset is hundreds of thousands or even millions of samples? We'll be facing significantly longer training times.

We've run into a problem. We want a large training set that will (hopefully) better represent the unknown parent process that we want to model. But the larger the training set, the longer it takes to pass each sample through the network, get an average value for the loss, and update the weights and biases. We call passing the entire training set through the network an *epoch*, and we'll need many dozens to hundreds of epochs to train the network. Doing a better job of representing what we want to model means longer computation times because of all the samples that must be passed through the network.

This is where SGD comes into play. Instead of using all the training data on each pass, let's alternatively select a small subset of the training data and use the average loss calculated from it to update the parameters. We'll calculate an "incorrect" gradient value because we're estimating the loss over the full training set with only a small sample, but we'll save a lot of time.

Let's see how this sampling plays out with a simple example. We'll use NumPy to define a vector of 100 random bytes:

```
>>> d = np.random.normal(128, 20, size=100).astype("uint8")
>>> d
130, 141,  99, 106, 135, 119,  98, 147, 152, 163, 118, 149, 122,
133, 115, 128, 176, 132, 173, 145, 152,  79, 124, 133, 158, 111,
139, 140, 126, 117, 175, 123, 154, 115, 130, 108, 139, 129, 113,
129, 123, 135, 112, 146, 125, 134, 141, 136, 155, 152, 101, 149,
137, 119, 143, 136, 118, 161, 138, 112, 124,  86, 135, 161, 112,
117, 145, 140, 123, 110, 163, 122, 105, 135, 132, 145, 121,  92,
118, 125, 154, 148,  92, 142, 118, 128, 128, 129, 125, 121, 139,
152, 122, 128, 126, 126, 157, 124, 120, 152
```

Here the byte values are normally distributed around a mean of 128. The actual mean of the 100 values is 130.9. Selecting subsets of these values, 10 at a time, gives us an estimate of the actual mean value, with repeated subsets leading to estimated means of 135.7, 131.7, 134.2, 128.1, and so forth:

```
>>> i = np.argsort(np.random.random(100))
>>> d[i[:10]].mean()
138.9
```

None of the estimated means are the actual mean, but they're all close to it. If we can estimate the mean from a random subset of the full dataset, we can see by analogy that we should be able to estimate the gradients of the loss function with a subset of the full training set. Since the sample is randomly selected, the resulting gradient values are randomly varying estimates. This is why we add the word *stochastic* in front of *gradient descent*.

Because passing the full training set through the network on each weight and bias update step is known as *batch training*, passing a subset through is called *minibatch training*. A *minibatch* is the subset of the training data used for each step of the SGD. Training is usually a certain number of epochs, where the relationship between epochs and minibatches is as follows:

$$1 \text{ epoch} = \left(\frac{\text{number of training-set samples}}{\text{minibatch size}} \right) \text{ minibatches}$$

In practice, we don't want to select the minibatches at random from the full training set. Doing so runs the risk of not using all the samples: Some might never be selected, and others might be selected too often. Typically, we randomize the order of the training samples and select fixed-size blocks of samples sequentially whenever a minibatch is required. When all available training samples are used, we can shuffle the full training-set order before repeating the process.

Convex vs. Nonconvex Functions

SGD sounds like a concession to practicality. In theory, it seems that we'd never want to use it, and we might expect our training results to suffer because of it. However, the opposite is generally true. In some sense, gradientdescent training of a neural network shouldn't work at all because we're applying an algorithm meant for convex functions to one that is nonconvex. Figure 6-2 illustrates the difference between a convex function and a nonconvex function.

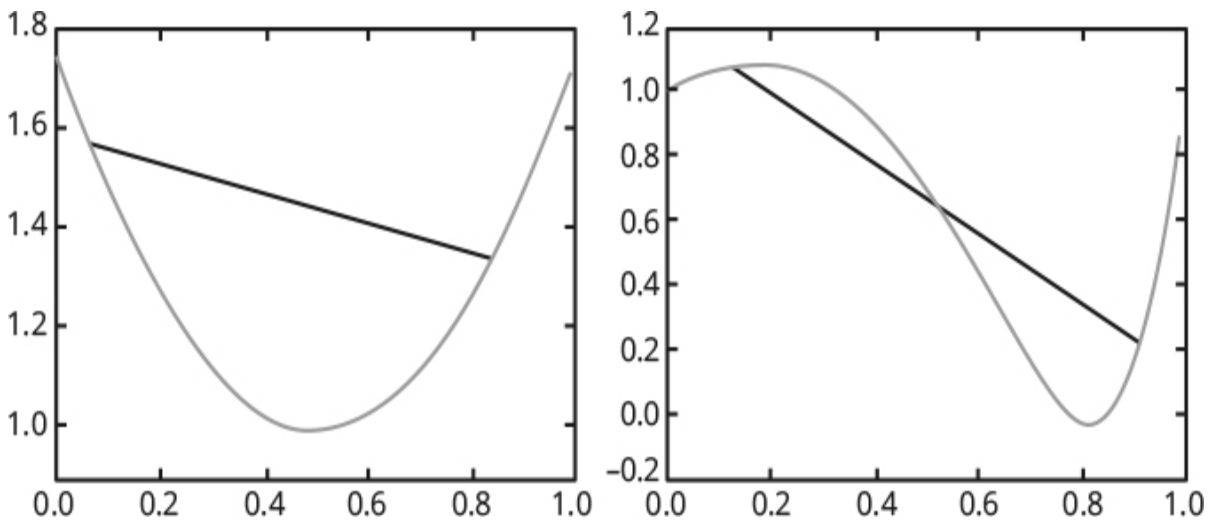


Figure 6-2: A convex function of x (left) and a nonconvex function of x (right)

A *convex function* is such that a line segment between any two points on the function does not cross the function at any other point. The black line on the left of Figure 6-2 is one example, and any such segment will not cross the function at any other point, indicating that this is a convex function. However, the same can't be said of the curve on the right of Figure 6-2. This is the curve from Figure 6-1. Here the black line does cross the function.

Gradient descent is designed to find the minimum when the function is convex. Because this approach relies only on the gradient (the first derivative), it's sometimes known as a *first-order optimization* method. Gradient descent should not work, in general, with nonconvex functions because it runs the risk of getting trapped in a local minimum instead of finding the global minimum. We saw this with the example in Figure 6-1.

Here's where SGD helps. In multiple dimensions, the gradient will point in a direction that isn't necessarily toward the nearest minimum of the loss function. Our step will be in a slightly wrong direction, but that direction might help us avoid getting trapped somewhere we don't want to be.

The situation is more complicated, of course, and more mysterious. The machine learning community has been struggling with the contradiction between the obvious success of using first-order optimization on the nonconvex loss function and the fact that it shouldn't work at all.

Two ideas are emerging. The first is what I just stated, that SGD helps by actually moving us in a slightly wrong direction. The second idea, which seems to be pretty much proven now, is that, the loss functions used in deep learning have many local minimums and these are all basically the same, so landing in almost any one of them results in a network that performs well.

Some researchers argue that most gradient-descent learning winds up on a *saddle point*; this is a place that looks like a minimum but isn't. Imagine a saddle for a horse and place a marble in the middle. The marble will sit in place, but you could push the marble in a certain direction and have it roll off the saddle. The argument, not without some justification, is that most training ends on a saddle point, and better results are possible with a better algorithm. However, even the saddle point is still for practical purposes a good place to be, so the model is successful regardless.

In practice, then, we should use SGD because it leads to better overall learning and reduces the training time by not requiring full batches. It does introduce a new hyperparameter, the minibatch size, that we must select at some point before training.

When to Stop Training

We haven't yet discussed a critical question: When should we stop training? Remember that in Chapter 2, we went through some effort to create training sets, validation sets, and test sets. This is where we'll use

the validation sets. While training, we can use the accuracy, or another metric, on the validation set to decide when to stop. If using SGD, we typically run the validation set through the network for each minibatch or set of minibatches to compute the accuracy. By tracking the accuracy on the validation set, we can decide when to stop training.

If we train for a long time, two things usually happen eventually. First, the error on the training set goes toward zero; we get increasingly better on the training set. Second, the error on the validation set goes down and then, eventually, starts to go back up.

These effects are due to overfitting. The training error goes down as the model learns more to represent the parent distribution that generated the dataset. But, eventually, it stops learning general details about the training set. At this point, we're overfitting, and we want to stop training because the model is no longer learning general features and is instead learning minutiae about the particular training set we're using. We can watch for this by using the validation set while training. Since we don't use the samples in the validation set to update the weights and biases of the network, it should give us a fair test of the current state of the network. When overfitting starts, the error on the validation set will begin to go up from a minimum value. At this point, we can keep the weights and biases that produced the minimum value on the validation set and claim that those represent the best model.

We don't want to use any data that has influenced training to measure the final effectiveness of our network. We use the validation set to decide when to stop training, so characteristics of the samples in the validation set have also influenced the final model; this means we can't strongly rely on the validation set to give us an idea of how the model will behave on new data. It's only the held-out test set, unused until we declare victory over training, that gives us some idea of how we might expect the model to perform on data in the wild. So, just as it is anathema to report training-set accuracy as a measure of model quality, it's also anathema to report the validation-set accuracy.

This picture hides a fascinating, recently understood phenomenon, that of double descent. A neural network's complexity (think *size*) affects what happens during training. The error on the validation set may go

down for large models as the number of training epochs increases, only to go up again before eventually going down a second time, often to a lower level. The details behind why this happens are beyond our scope, but double descent due to the number of training epochs is related to the large network memorizing the training set while still adapting the weights and biases so that it continues to improve performance on the validation and, ultimately, the held-out test set.

The Learning Rate

In our generic update equation for changing the weights and biases based on the gradient, we introduced a hyperparameter, η (eta): the learning rate, or step size. It's a scale factor indicating how much we should update the weight or bias based on the gradient value.

We previously stated that the learning rate doesn't need to be fixed and that it could, and even should, get smaller as we train under the assumption that we need smaller steps to get to the actual minimum value of the loss function. We didn't state how we should actually update the learning rate.

We could update the step size in more than one way, but some are more helpful than others. The `MLPClassifier` class of `sklearn`, which uses SGD solvers, has three options. The first is to never change the learning rate; just leave η at its initial value, η_0 . The second is to scale η so that it decreases with epochs (minibatches) according to this:

$$\eta = \frac{\eta_0}{t^p}$$

Here η_0 is set by the user, t is the iteration (epoch, minibatch), and p is an exponent on t , also picked by the user. The `sklearn` default p is 0.5—that is, scale by $\sqrt{t}$, which seems a reasonable default.

The third option is to adapt the learning rate by watching the loss function value. As long as the loss is decreasing, leave the learning rate where it is. When the loss stops decreasing for a set number of minibatches, divide the learning rate by a value like 5, the `sklearn` default. If we never change the learning rate, and it's too large, we might

end up moving around the minimum without ever being able to reach it because we're consistently stepping over it. It's a good idea then to decrease the learning rate when using SGD. Later in the book, we'll encounter other optimization approaches that automatically adjust the learning rate for us.

Momentum

SGD has one last wrinkle that we have to cover. As we saw previously, the weight update equation for both gradient descent and SGD is as follows:

$$w \leftarrow w - \eta \Delta w$$

We update the weight by the learning rate (η) times the gradient, which we represent here as Δw .

A common and powerful trick is to introduce a *momentum* term that adds back a fraction of the previous Δw , the update of the prior minibatch. The momentum term prevents the w parameter from changing too quickly in response to a particular minibatch. Adding in this term gives us this:

$$w_{i+1} \leftarrow w_i - \eta \Delta w_i + \mu \Delta w_{i-1}$$

The added subscripts indicate the next pass through the network ($i + 1$), the current pass (i), and the previous pass ($i - 1$). The previous pass Δw is the one we need to use. A typical value for μ (mu), the momentum, is around 0.9. Virtually all toolkits implement momentum in some form, including sklearn.

Backpropagation

We've been operating under the assumption that we know the gradient value for each parameter. Let's discuss how the backpropagation algorithm gives us these magic numbers. This algorithm is perhaps the single most important development in the history of neural networks, as

it enables the training of large networks with hundreds, thousands, millions, and even billions of parameters. This is especially true of the convolutional networks we'll work with in Chapter 9.

The backpropagation algorithm itself was published by David E. Rumelhart et al. in their 1986 paper, "Learning Representations by Backpropagating Errors." It's a careful application of the chain rule for derivatives. The algorithm is called *backpropagation* because it works backward from the output layer of the network toward the input layer, propagating the error from the loss function down to each parameter of the network. Colloquially, the algorithm is known as *backprop*; we'll use that term here.

Adding backprop into the training algorithm for gradient descent, and tailoring it to SGD, gives us the following algorithm:

1. Pick intelligent starting values for the weights and biases.
2. Run a minibatch through the network by using its current weights and biases and calculate the average loss.
3. Use this loss and backprop to get the gradient for each weight and bias.
4. Update the weight or bias value by the step size times the gradient value.
5. Repeat from step 2 until the loss is low enough.

Step 2 of this algorithm is referred to as the *forward pass*; step 3 is the *backward pass*. The forward pass is also how we'll use the network after it's finally trained. The backward pass is backprop calculating the gradients for us so that we can update the parameters in step 4.

We'll describe backprop twice. First, we'll do so with a simple example and work with the derivatives. Second, we'll work with a more abstract notation to see how backprop applies to neural networks in a general sense. There's no way to sugarcoat this: this section involves derivatives, but we already have a good intuitive sense of what those are from our discussion of gradient descent, so we should be in good shape to proceed.

A Simple Example

Suppose we have two functions, $z = f(y)$ and $y = g(x)$, meaning $z = f(g(x))$. We know that the derivative of the function g gives us dy/dx , which tells us how y changes when x changes. Similarly, we know that the derivative of the function f gives us dz/dy . The value of z depends upon the composition of f and g , meaning the output of g is the input to f , so if we want to find an expression for dz/dx , how z changes with x , we need a way to link through the composed functions. This is what the chain rule for derivatives gives us:

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$$

This notation is especially nice because we can imagine the dy “term” canceling just as it would if these were actual fractions.

How does this help us? In a neural network, the output of one layer is the input to the next, which is composition, so we can see intuitively that the chain rule might apply. Remember that we want the values that tell us how the loss function changes with respect to the weights and biases. Let’s call the loss function L and any given weight or bias w . We want to calculate $\partial\mathcal{L}/\partial w$ for all the weights and biases.

Alarm bells should be going off in your head. The previous paragraph slipped in new notation. So far, we’ve been writing derivatives as dy/dx , but the derivative for the loss with respect to a weight was written as $\partial\mathcal{L}/\partial w$. What is this fancy ∂ ?

When we had a function of one variable, just x , there was only one slope at a point to talk about. As soon as we have a function with more than one variable, the idea of the slope at a point becomes ambiguous. An infinite number of lines are tangent to the function at any point. So we need the idea of the *partial derivative*, which is the slope of the line in the direction of the variable we’re considering when all other variables are treated as fixed. This tells us how the output will change as we change only the one variable. To note that we’re using a partial derivative, we shift from d to ∂ , which is just a script d .

Let's set up a straightforward network so that we can see how the chain rule leads directly to the expressions we want. We're looking at the network in Figure 6-3, which consists of an input, two hidden layers of a single node each, and an output layer.

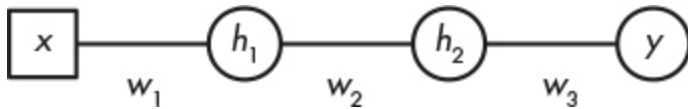


Figure 6-3: A simple network to illustrate the chain rule

For simplicity, we'll ignore any bias values. Additionally, let's define the activation function to be the identity function, $b(x) = x$. This simplification removes the derivative of the activation function to make the presentation more transparent.

For this network, the forward pass computes:

$$\begin{aligned}h_1 &= w_1 x \\h_2 &= w_2 h_1 \\y &= w_3 h_2\end{aligned}$$

This follows the form we've used previously, chaining functions together by making the output of one layer the input to the next, giving us the output of the network, y , for input x . If we're looking to train the network, we'll have a training set, a set of pairs, $(x_i, \hat{y}_i)$, $i = 0, 1, \dots$, that are examples of what the output should be for a given input. Note that the forward pass moved from the input, x , to the output, y . We'll next see why the backward pass moves from the output to the input.

Now let's define the loss function, $\mathcal{L}$, to be the squared error between y , the network output for a given input x , and $\hat{y}$, the output we should get. Functionally, the loss looks like the following:

$$\mathcal{L} = \frac{1}{2} (y - \hat{y})^2$$

For simplicity, we're ignoring that the loss is a mean over the training set or a minibatch drawn from it. The factor of $1/2$ is not strictly necessary but is commonly used to make the derivative a bit nicer. Since

we're looking to minimize the loss for a particular set of weights, it doesn't matter that we're always multiplying the loss by a constant factor of 1/2; the smallest loss will still be the smallest loss regardless of its actual numeric value.

To use gradient descent, we need to find how the loss changes with the weights. In this simple network, that means we need to find three gradient values, one each for w_1 , w_2 , and w_3 . This is where the chain rule comes into play. We'll write the equations first and then talk about them:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_3} &= \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial w_3} \\ \frac{\partial \mathcal{L}}{\partial w_2} &= \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial h_2} \frac{\partial h_2}{\partial w_2} \\ \frac{\partial \mathcal{L}}{\partial w_1} &= \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial h_2} \frac{\partial h_2}{\partial h_1} \frac{\partial h_1}{\partial w_1}\end{aligned}$$

The order of the equations shows why this algorithm is called *backpropagation*. To get the partial derivative for the output layer parameter, we need only the output and the loss, y and $\mathcal{L}$. To get the partial derivative for the middlelayer weight, we need the following two partial derivatives from the output layer:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial y} \\ \frac{\partial y}{\partial h_2}\end{aligned}$$

Finally, to get the partial derivative for the input-layer weight, we need partial derivatives from the output and middle layer. In effect, we've moved backward through the network, propagating values from later layers.

For each of these equations, the right-hand side matches the left-hand side if we imagine the “terms” canceling like fractions. Since we selected a particularly simple form for the network, we can calculate the actual gradients by hand. We need the following gradients, from the right-hand side of the preceding equations:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial y} &= (y - \hat{y}) \\
\frac{\partial y}{\partial w_3} &= h_2 = w_2 h_1 = w_2 w_1 x \\
\frac{\partial y}{\partial h_2} &= w_3 \\
\frac{\partial h_2}{\partial w_2} &= h_1 = w_1 x \\
\frac{\partial h_2}{\partial h_1} &= w_2 \\
\frac{\partial h_1}{\partial w_1} &= x
\end{aligned}$$

The $\partial \mathcal{L} / \partial y$ comes from the form we selected for the loss and the rules of differentiation from calculus.

Putting these back into the equations for the gradients of the weights gives us the following:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_3} &= (y - \hat{y}) w_2 w_1 x \\
\frac{\partial \mathcal{L}}{\partial w_2} &= (y - \hat{y}) w_3 w_1 x \\
\frac{\partial \mathcal{L}}{\partial w_1} &= (y - \hat{y}) w_3 w_2 x
\end{aligned}$$

After a forward pass, we have numeric values for all the quantities on the right-hand side of these equations. Therefore, we know the numeric value of the gradients. The update rule from gradient descent then tells us to change the weights as follows, where η is the learning-rate parameter defining how large a step to take when updating:

$$\begin{aligned}
w_3 &\leftarrow w_3 - \eta \frac{\partial \mathcal{L}}{\partial w_3} = w_3 - \eta (y - \hat{y}) w_2 w_1 x \\
w_2 &\leftarrow w_2 - \eta \frac{\partial \mathcal{L}}{\partial w_2} = w_2 - \eta (y - \hat{y}) w_3 w_1 x \\
w_1 &\leftarrow w_1 - \eta \frac{\partial \mathcal{L}}{\partial w_1} = w_1 - \eta (y - \hat{y}) w_3 w_2 x
\end{aligned}$$

To recap, we use the chain rule, the heart of the backprop algorithm, to find the gradients we need in order to update the weights during training. For our simple network, we were able to explicitly work out

the value of these gradients by moving backward through the network from the output toward the input. As this is just a toy network, let's now take a second look at how to use backprop in a more general sense to calculate the necessary gradients for any fully connected feed-forward network.

An Abstract Example

Let's begin by revisiting the loss function and introducing some new notation. The *loss function* is a function of all the parameters in the network, meaning that every weight and bias value contributes to it. For example, the loss for the network of Figure 5-1 on page 130, which has 20 weights and biases, could be written as shown here:

$$\begin{aligned} \text{loss} = \mathcal{L} & \left(w_{00}^{(1)}, w_{01}^{(1)}, w_{02}^{(1)}, w_{10}^{(1)}, w_{11}^{(1)}, w_{12}^{(1)}, b_0^{(1)}, b_1^{(1)}, b_2^{(1)}, \right. \\ & w_{00}^{(2)}, w_{01}^{(2)}, w_{10}^{(2)}, w_{11}^{(2)}, w_{20}^{(2)}, w_{21}^{(2)}, b_0^{(2)}, b_1^{(2)}, \\ & \left. w_{00}^{(3)}, w_{10}^{(3)}, b_0^{(3)} \right) \end{aligned}$$

Note we've introduced a new notation for the parameters:

$$w_{jk}^{(i)}$$

This represents the weight that links the j th input, an output of the $i - 1$ layer, to the k th node of layer i . We also have the following to represent the bias value for the k th node of the i th layer:

$$b_k^{(i)}$$

Here layer 0 is the input layer itself. The parentheses on the exponent are a label, the layer number; they should not be interpreted as actual exponents. Therefore, the following is the weight from the third output of the first layer to the first node of the second layer:

$$w_{20}^{(2)}$$

This is the highlighted weight in Figure 6-4. Remember that we always number nodes top to bottom, starting with 0.

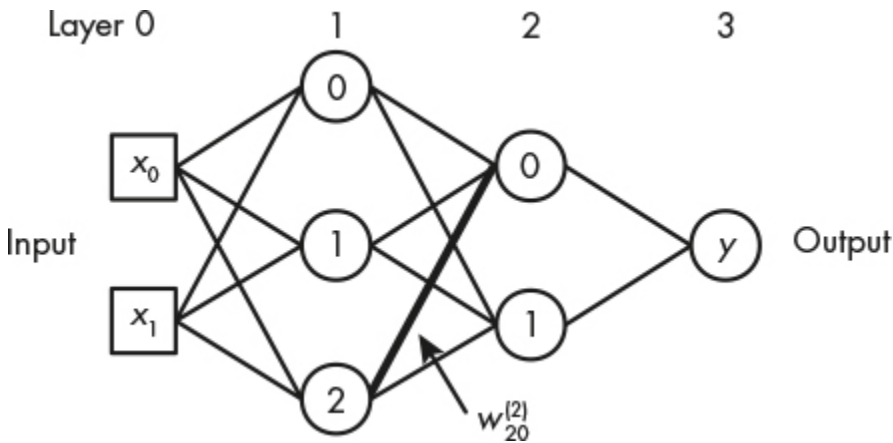


Figure 6-4: The network of Figure 5-1 with weight $w_{20}^{(2)}$ marked with a bold line

This notation is a bit daunting, but it lets us precisely reference any weight or bias of the network. The number we need in order to use backprop is the partial derivative of the loss with respect to each weight or bias. Therefore, what we want to find ultimately is written, in all its glorious mathematical notation, as follows:

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{(i)}}$$

This gives us the slope: the amount the loss will change for a change in the weight linking the k th node of the i th layer to the j th output of the $i - 1$ layer. A similar equation gives us the partial derivatives of the biases.

We can simplify this cumbersome notation by dealing only with the layer number, understanding that buried in the notation is a vector (biases, activations) or matrix (weights) so that we want to find the following:

$$\frac{\partial \mathcal{L}}{\partial w^{(i)}} \text{ and } \frac{\partial \mathcal{L}}{\partial b^{(i)}}$$

These correspond to a matrix for all the weights linking layer $i - 1$ to i , and a vector for all the biases of layer i , respectively.

We'll protect our notational sanity by looking at vectors and matrices. Let's start with the output layer and see what that buys us. We know that the activations of the output layer, layer L , are found via

$$a^{(L)} = h(W^{(L)}a^{(L-1)} + b^{(L)})$$

with a being the activations from layer $L - 1$, b the bias vector for layer L , and W the weight matrix between layers $L - 1$ and L . The activation function is h .

Additionally, we'll define the argument to h to be $z^{(L)}$:

$$z^{(L)} \equiv W^{(L)}a^{(L-1)} + b^{(L)}$$

We'll call $\partial\mathcal{L}/\partial z^{(l)}$ the *error*, the contribution to the loss from the inputs to layer l . Next, we define the following so that we can work with δ (delta) from now on:

$$\delta^{(l)} \equiv \frac{\partial\mathcal{L}}{\partial z^{(l)}}$$

For the output layer, we can write δ as shown here:

$$\delta^{(L)} = \frac{\partial\mathcal{L}}{\partial z^{(L)}} = \frac{\partial\mathcal{L}}{\partial a^{(L)}} \cdot h'(z^{(L)})$$

The notation $h'(z^{(L)})$ is another way to write the derivative of h (with respect to z) evaluated at $z^{(L)}$. The $\cdot$ represents elementwise multiplication. This is the way NumPy works when multiplying two arrays of the same size so that if $C = A \cdot B$, then $C_{ij} = A_{ij}B_{ij}$. Technically, this product is called the *Hadamard product*, named for the French mathematician Jacques Hadamard.

The preceding means that to use backpropagation, we need a loss function that can be differentiated—a loss function for which a derivative exists at every point. This isn't too much of a burden; the loss functions we'll examine in the next section meet this criterion. We also need an activation function that can be differentiated so we can find h'

(z). Again, the activation functions we have considered so far are essentially all differentiable.

NOTE

I say “essentially” because the derivative of the ReLU is undefined at $x = 0$. The derivative from the left is 0, while the derivative from the right is 1. In practice, implementations choose a particular value to return should the argument to the derivative of the ReLU be exactly 0. For example, TensorFlow simply asks whether the argument is less than or equal to 0 and, if it is, returns 0 as the derivative. Otherwise, it returns 1. This works because, numerically, so much rounding off is happening to floating-point values during calculations that it’s unlikely the value passed to the derivative of the ReLU function was actually meant to be identically 0.

The equation for δ tells us the error due to the inputs to a particular layer. We’ll see next how to use this to get the error from each weight of a layer.

With $\delta^{(L)}$ in hand, we can propagate the error down to the next layer via the following, where, for the next-to-last layer, $l + 1 = L$:

$$\delta^{(l)} = \left((W^{(l+1)})^T \delta^{l+1} \right) \cdot h' \left(z^{(l)} \right)$$

The T represents *matrix transpose*. This is a standard matrix operation that involves a reflection across the diagonal so that if

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

then:

$$A^T = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

We need the transpose of the weight matrix because we are going in the opposite direction from the forward pass. If there are three nodes in layer l and two in layer $l + 1$, then the weight matrix between them, W , is a 2×3 matrix, so Wx is a two-element vector. In backprop, we are going from layer $l + 1$ to layer l , so we transpose the weight matrix to map the two-element vector, here δ , to a three-element vector for layer l .

The $\delta^{(l)}$ equation is used for every layer moving backward through the network. The output layer values are given by $\delta^{(L)}$, which starts the process.

Once we have the errors per layer, we can finally find the gradient values we need. For the biases, the values are the elements of δ for that layer for the j th element of the bias for the l th layer:

$$\frac{\partial \mathcal{L}}{\partial b_j^{(l)}} = \delta_j^{(l)}$$

For the weights, we need the following, linking the k th output of the previous layer to the j th error for the current layer, l :

$$\frac{\partial \mathcal{L}}{\partial w_{kj}^{(l)}} = a_k^{(l-1)} \delta_j^{(l)}$$

Using the preceding equations for each layer of the network gives us the set of weight and bias gradient values needed to continue applying gradient descent.

As I hope you can see from this rather dense section, we can use a convenient mathematical definition of the error to set up an iterative process that moves the error from the output of the network back through the layers of the network to the input layer. We cannot calculate the errors for a layer without already knowing the errors for

the layer after it, so we end up propagating the error backward through the network, hence the name *backpropagation*.

Loss Functions

The *loss function* is used during training to measure how poorly the network is doing. The goal of training is to make this value as small as possible, while still generalizing to the true characteristics of the data. In theory, we can create any loss function we want if we feel it's relevant to the problem at hand. In deep learning literature, papers do this all the time.

Still, most research falls back on a few standard loss functions that, empirically, do a good job most of the time. We'll discuss three of those here: absolute loss (sometimes called L_1 loss), mean squared error (sometimes called L_2 loss), and cross-entropy loss.

Absolute and Mean Squared Error Loss

Let's start with the absolute and mean squared error loss functions. We'll discuss them together because they're similar mathematically.

We've seen mean squared error already in our discussion of backprop. Absolute loss is new. Mathematically, the two equations are as follows:

$$\begin{aligned}\mathcal{L}_{\text{abs}} &= |y - \hat{y}| \\ \mathcal{L}_{\text{MSE}} &= \frac{1}{2} (y - \hat{y})^2\end{aligned}$$

I've labeled them *abs* for *absolute value* and *MSE* for *mean squared error*, respectively. We'll always use y for the network output, the output from the forward pass with input x . We'll always use $\hat{y}$ for the known training class label, which is always an integer label starting with 0.

Even though we're writing the loss functions in a simple form, remember that when used, the value is really the mean of the loss over the training set or minibatch. This is also the origin of *mean* in *mean squared error*. Therefore, we really should be writing this:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^N \frac{1}{2} (y_i - \hat{y}_i)^2 = \frac{1}{2N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

Here we're finding the average of the squared error loss over the N values in the training set (or minibatch).

Both of these loss functions are reasonable if we consider what they are measuring. We want the network to output a value that matches the expected value, the sample label. The difference between these two is an indication of how wrong the network output is. For the absolute loss, we find the difference and drop the sign, which is what the absolute value does. For the MSE loss, we find the difference and then square it. This also makes the difference positive because multiplying a negative number by itself always results in a positive number. As mentioned in “Backpropagation” on page 157, the $1/2$ factor on the MSE loss simplifies the derivative of the loss function but doesn't change how it works.

The absolute loss and MSE are different, however. The MSE is more sensitive to outliers. This is because we're squaring the difference, and a plot of $y = x^2$ grows quickly as x , the difference, gets larger. For the absolute loss, this effect is minimized because no squaring occurs; the difference is merely the difference.

In truth, neither of these loss functions is commonly used for neural networks when the goal of the network is classification, which is our implicit assumption in this book. It's more common to use the cross-entropy loss, presented next. We want the network output to lead to the correct class label for the input. However, it's entirely possible to train a network to output a continuous real value instead. This is called *regression*, and both of these loss functions are quite useful in that context.

Cross-Entropy Loss

Even though we can use the absolute and MSE loss functions in training a neural network for classification, the most commonly used loss is the crossentropy loss (closely related to the log-loss). This loss function assumes that the output of the network is a softmax (vector) for

the multiclass case or a sigmoid (logistic, scalar) for the binary case. Mathematically, it looks like this for M classes in the multiclass case:

$$\mathcal{L}_{\text{ent}} = -\sum_i^M \hat{y}_i \log y_i \quad (\text{multiclass case})$$
$$\mathcal{L}_{\text{ent}} = -\hat{y} \log(y) + (1 - \hat{y}) \log(1 - y) \quad (\text{binary case})$$

What is the cross-entropy doing that often makes it a better choice for training a neural network for classification? Let's think about the multiclass case with softmax outputs. *Softmax* means that the network outputs can be thought of as probability estimates that the input represents each of the possible classes. In most cases, the softmax outputs are not true probabilities because the models are not calibrated, but we can live with the imprecision for now.

If we have three classes, we might get a softmax output that looks like this:

$$y = (0.03, 0.87, 0.10)$$

This output roughly means that the network thinks there is a 3 percent chance the input is of class 0, an 87 percent chance it is of class 1, and a 10 percent chance it is of class 2. This is the output vector, y . We compute the loss by supplying the actual label via a vector, where 0 means *not this class* and 1 means *this class*. So, the $\hat{y}$ vector associated with the input that led to this y would be

$$\hat{y} = (0, 1, 0)$$

for an overall loss value of:

$$\mathcal{L}_{\text{ent}} = -(0(\log 0.03) + 1(\log 0.87) + 0(\log 0.10)) = 0.139262$$

The three predictions of the network can be thought of together as a probability distribution, just like the one we get when we sum together the likelihoods of various outcomes for throwing two dice. We also have a known probability distribution from the class label. For the preceding example, the actual class is class 1, so we made a probability distribution

that assigns no chance to classes 0 and 2, and 100 percent probability to class 1, the actual class. As the network trains, we expect the output distribution to be closer and closer to (0, 1, 0), the distribution for the label.

Minimizing the cross-entropy drives the network toward increasingly better predictions of the probability distribution for the classes we want the network to learn about. Ideally, these output distributions will look like the training labels: 0 for all classes except the actual class, which has an output of 1.

For classification tasks, we usually use the cross-entropy loss. The `sklearn.MLPClassifier` class uses cross-entropy. Keras supports cross-entropy loss as well but provides many others, including absolute and MSE.

Weight Initialization

Before we can train a neural network, we need to initialize the weights and biases. Step 1 of our gradient-descent algorithm on page 152 says to “Pick intelligent starting values for the weights and biases.”

The initialization techniques examined here all depend on selecting random numbers in a certain range. More than that, the random numbers need to be either uniform over that range or normally distributed. *Uniformly distributed* means that all the values in the range are equally likely to be selected. This is what you get for each number, 1 through 6, if you roll a fair die many times over. *Normally distributed* values were introduced in Chapter 1. These are values with a particular mean, the most likely value returned, and a range around the mean over which the likelihood of a value being selected falls off gradually toward 0 according to a parameter known as the *standard deviation*. This is the classic bell-curve shape.

Either distribution can be used. The main point is that the initial weights are not all the same value (like 0) because if they are, all gradients will be the same during backprop, and each weight will change in the same way. The initial weights need to be different to break this

symmetry and allow individual weights to adapt themselves to the training data.

In the early days of neural networks, people initialized the weights and biases by choosing values in $[0, 1)$ uniformly ($U(0, 1)$) or by drawing them from the standard normal distribution, $N(0, 1)$, with a mean of 0 and a standard deviation of 1. These values were often multiplied by a small constant, like 0.01. In many cases, this approach works, at least for simple networks. However, as networks became more complex, this simple approach fell apart. Networks initialized in this way had trouble learning, and many failed to learn at all.

Let's fast-forward several decades and a great deal of research later. Researchers realized that precisely how the weights of a particular layer should be initialized depended primarily on a few parameters: the type of activation function used and the number of weights coming into the layer (f_{in}) and, possibly, going out (f_{out}). These realizations led to the main initialization approaches in use today.

The `sklearn.MLPClassifier` class uses *Glorot initialization*. This is also sometimes called *Xavier initialization*, though some toolkits mean something different when they use that term. (For more on these, see Xavier Glorot and Yoshua Bengio's 2010 paper, "Understanding the Difficulty of Training Deep Feedforward Neural Networks.")

Let's see how `sklearn` uses Glorot initialization. The key method in `MLPClassifier` for initializing the weights is `_init_coef`. This method uses a uniform distribution and sets the range for it so that the weights are in the following form:

$$\left[-\sqrt{\frac{A}{f_{in}+f_{out}}}, \sqrt{\frac{A}{f_{in}+f_{out}}} \right]$$

Here the bracket notation indicates the smallest possible value selected (left) to the largest possible value (right). As the distribution is uniform, every value in that range is equally likely to be selected.

We did not yet specify what A is. This value depends upon the activation function used. According to the literature, if the activation

function is a sigmoid (logistic), then $A = 2$ is suggested. Otherwise, $A = 6$ is recommended.

Now to confuse things. Some toolkits, like Caffe, use an alternate form of Xavier initialization by which they mean a multiplier on samples from a standard normal distribution. In that case, we initialize the weights with draws from the following:

$$N\left(0, 1\right) \sqrt{\frac{1}{f_{in}}} \quad \left(\text{alternate Xavier initialization}\right)$$

To add even more confusion, the introduction of the ReLU resulted in a further recommended change. This is known now as *He initialization*, and it replaces the 1 in Xavier initialization with a 2:

$$N\left(0, 1\right) \sqrt{\frac{2}{f_{in}}} \quad \left(\text{He initialization, ReLU only}\right)$$

For more on this, see “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification” by Kaiming He et al. (2015).

The key point with these initialization schemes is that the old-school “small random value” is replaced by a more principled set of values that take the network architecture into account via f_{in} and f_{out} .

The preceding discussion intentionally ignores bias values. While it might be okay to initialize the bias values instead of leaving them all 0, prevailing wisdom, which is fickle and fluid, currently says it’s best to initialize them all to 0. That said, `sklearn.MLPClassifier` initializes the bias values in the same way as the weights.

Managing Model Complexity and Generalization

The goal of training a model is for it to learn essential, general features of the parent distribution the dataset is sampled from. That way, when the model encounters new inputs, it’s prepared to interpret them correctly. As we’ve seen in this chapter, the primary method for training a neural network involves optimization—looking for the “best” set of

parameters so that the network makes as few errors as possible on the training set.

However, it's not enough to simply look for the best set of values that minimizes the training error. If we make no mistakes when classifying the training data, it's often the case that we've overfitted and haven't actually learned general features of the data. This is more likely the situation with traditional models, neural network or classical, and less so with deep models like the convolutional networks of Chapter 9, in part because of the double descent phenomenon.

Overfitting

We've mentioned overfitting from time to time but haven't gained any intuition as to what it is. One way to think of overfitting is to consider a separate problem, the problem of fitting a function to a set of points. This is *curve fitting*, and one approach to it is to optimize a measure of error over the points by finding parameters to the function that minimize the error. This should sound familiar, as it's exactly what we do when training a neural network.

As an example of curve fitting, consider the following points:

x	y
0.00	50.0
0.61	-17.8
1.22	74.1
1.83	29.9
2.44	114.8
3.06	55.3
3.67	66.0
4.28	89.1
4.89	128.3
5.51	180.8

x	y
6.12	229.7
6.73	229.3
7.34	227.7
7.95	354.9
8.57	477.1
9.18	435.4
9.79	470.1

We want to find a function, $y = f(x)$, that describes these points—a function that might have been the parent function these points were measured from, albeit noisily.

Typically, when curve fitting, we already know the form of the function; it's the parameters we're looking for. But what if we didn't know the exact form of the function, only that it was some kind of polynomial? In general, a polynomial looks like this for a maximum exponent, n :

$$y = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_nx^n$$

The goal of fitting a polynomial to a dataset is to find the parameters, $a_0, a_1, a_2, \dots, a_n$. The method for doing this usually minimizes the squared difference between y , a given output for a given x position, and $f(x)$, the function output at the same x for the current set of parameters. Note that we discussed using precisely this type of loss function for training a neural network.

How does this relate to overfitting? Let's plot the previous dataset, along with the result of fitting two functions to it. The first is a quadratic function, the type of function you may have learned to hate as a beginning algebra student:

$$y = a_0 + a_1x + a_2x^2$$

The second function is a 15th-degree polynomial:

$$y = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_{14}x^{14} + a_{15}x^{15}$$

Figure 6-5 shows the results.

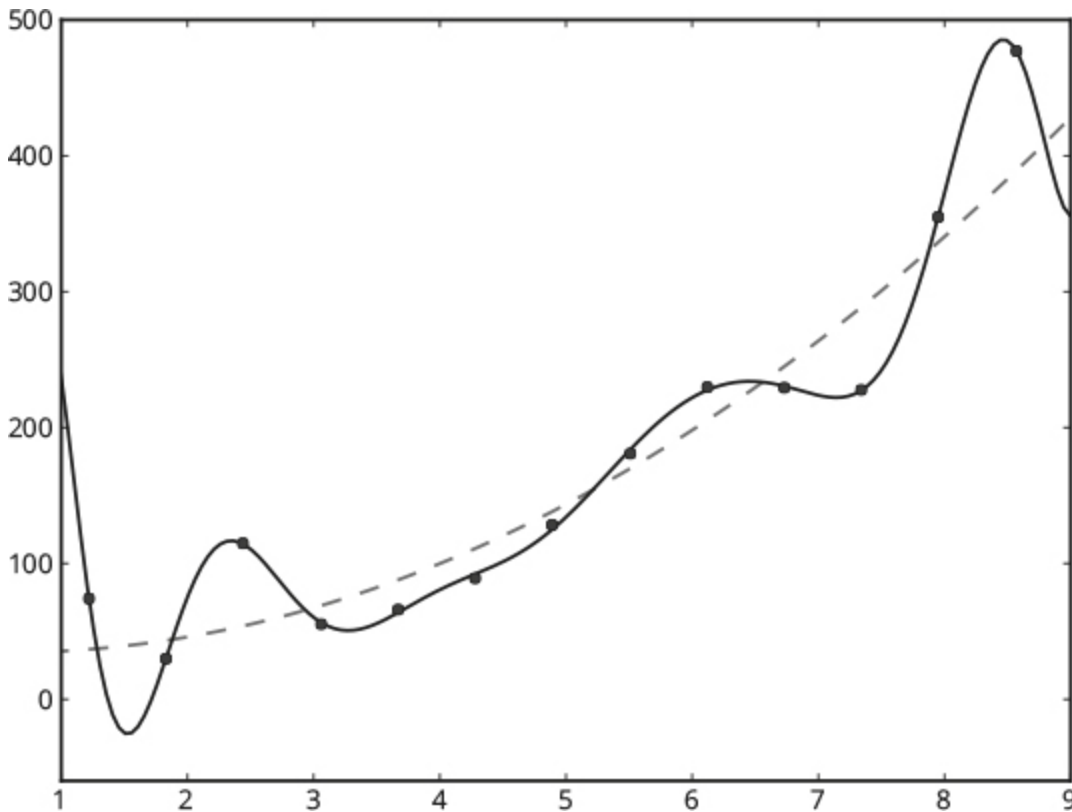


Figure 6-5: A dataset and two functions fit to it: a quadratic (dashed) and a 15th-degree polynomial (solid)

Which function better captures the general trend of the dataset? The quadratic clearly follows the general trend of the data, while the 15th-degree polynomial is all over the place. If all we use to decide that we have fit the data well is the distance between the data points and the corresponding function value, we'd say that the 15th-degree polynomial is the better fit; it passes through nearly all the data points, after all. This is analogous to training a neural network and achieving perfection on the training set. The cost of that perfection might well be a poor ability to generalize to new inputs. The quadratic fit of Figure 6-5 didn't hit the data points, but it did capture the general trend of the data,

making it more useful should we want to make predictions about the y values we'd expect to get for a new x value.

When a human wants to fit a curve to something like our sample dataset, they usually look at the data and, noticing the general trend, select a function to fit. The expected functional form might already be known from theory. If we want to be analogous to neural networks, however, we'll find ourselves not knowing the proper function to fit and needing to find a "best" one from the space of functions of x along with its parameters.

Hopefully, this example drives home the idea that training a neural network isn't akin to other optimization problems. We need something to push the function the network is learning in a direction that captures the essence of the data without falling into the trap of paying too much attention to specific features of the training data. That something is regularization, and you need it, especially for large networks that have a huge capacity.

Regularization

Regularization is anything that pushes the network to learn the relevant features of the parent distribution and not the details of the training set. The best form of regularization is increasing the size and representative nature of the training set. The larger the dataset and the better it represents all the types of samples the network will encounter in the wild, the better it will learn. Of course, we're typically forced to work with a finite training set. The machine learning community has spent, and is spending, untold time and energy learning how to get more from smaller datasets.

In Chapter 2, we encountered perhaps the second-best way to regularize a model: data augmentation. This is a proxy for having a larger dataset, as we use the data we do have to generate new training samples that are plausibly from the parent distribution. For example, we considered increasing a limited set of training images by simple rotations, flips, and shifts of the images already in the training set. Data augmentation is powerful, and you should use it when possible. It's particularly easy to apply when working with images as inputs, though

in Chapter 2 we also saw a way to augment a dataset consisting of continuously valued vectors.

We now have two tricks in our regularization toolbox: more data and data augmentation. These are the best tricks to know, but others exist that you should use when available. Let's look at two more: L2 regularization and dropout. The former is now standard and widely supported by the toolkits, including sklearn and Keras. The latter is powerful and was a game changer when it appeared in 2012.

L2 Regularization

A model with a few weights that have large values is somehow less simple than a model that has smaller weights. Therefore, keeping the weights small will hopefully allow the network to implement a simpler function better suited to the task we want it to learn.

We can encourage the weights to be small by using *L2 regularization*. This technique adds a term to the loss function so that the loss becomes

$$\mathcal{L} = \mathcal{L}(x, y, w, b) + \frac{\lambda}{2} \sum_i w_i^2$$

where the first term is whatever loss we're already using, and the second term is the new L2 regularization term. Notice that the loss is a function of the input (x), the label (y), the weights (w), and the biases (b), where we mean all the weights and all the biases of the network. The regularization term is a sum over all the weights in the network, and only the weights. The L2 label is what causes us to square the weights.

Here *L2* refers to the type of norm or distance. You might be familiar with the equation for the distance between two points on the plane: $d^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$. This is the Euclidean distance, also known as the *L2 distance*, because the values are squared. This is why the regularization term is called *L2* and the weight values are squared. It's also possible to use an L1 loss term, where instead of squaring the weights, we use the absolute value. In practice, L2 regularization is more common and, at least empirically, seems to work better for neural network classifiers.

The λ (lambda) multiplier sets the importance of this term; the larger it is, the more it dominates the overall loss used to train the network. Typical values of λ are around 0.0005. We'll see in a little bit why the multiplier is $\lambda/2$ and not just λ .

What is the L2 term doing? Recall that the loss is the value we want to minimize while training. The new L2 term sums the squares of the weights of the network. If weights are large, the loss is large, and we don't want that while training. Smaller weights make the L2 term smaller, so gradient descent will favor small weights, whether they're positive or negative, since we square the weight value. If all the weights of the network are relatively small, and none strongly dominate, then the network will use all the weights to represent the function, which is a good thing when preventing overfitting.

L2 regularization is also known as *weight decay* because of what the L2 term does during backprop. Backprop gives us the partial derivative of the loss function with respect to w_i . Adding L2 regularization means that the partial derivative of the total loss now adds in the partial derivative of the L2 term itself with respect to any particular weight, w_i . The derivative of $\lambda/2 w^2$ is λw ; the 1/2 cancels the factor of 2 that would otherwise be there. Also, since we want the partial derivative with respect to a specific weight, w_i , all the other parts of the L2 term go to 0. The net effect is that the update for weight w_i during gradient descent becomes

$$w_i \leftarrow w_i - \eta \frac{\partial \mathcal{L}}{\partial w_i} - \eta \lambda w_i$$

where η (eta) is the learning rate, and we're ignoring any additional momentum term. The $\eta \lambda w_i$ term is new. It is due to L2 regularization, and we can see that it's pushing the weights toward 0 as training progresses because both η and λ are < 1 , so on each minibatch, we're subtracting a small fraction of the weight value. The weight can still increase, but to do so, the gradient of the original loss must be large.

I previously stated that the form of the loss function is up to the developer of the network. A regularization term isn't the only kind of

term we can add to the loss function. As we did with the L2 term, we can create and add terms to change the behavior of the network during training and help it learn what we want it to learn. This is a powerful technique that we can use to customize various aspects of what a neural network learns.

Dropout

Dropout took the machine learning community by storm when it appeared in 2012 (see “ImageNet Classification with Deep Convolutional Neural Networks” by Alex Krizhevsky et al., 2017). As of this writing, this paper has been cited over 135,000 times, and as one well-known machine learning researcher told me privately at the time, “If we had had dropout in the 1980s, this would be a different world now.” So, what is dropout, and why was everyone so excited by it?

To answer that question, we need to review the concept of ensembles of models. We talked about them a bit in Chapter 3. An *ensemble* is a group of models, all slightly different and all trained on the same dataset or a slightly different version of the dataset. The idea is straightforward: Because training most models involves randomness, training multiple similar models should result in a set that is mutually reinforcing; the sets of outputs can be combined to produce a result that is better than that produced by any one model alone.

Ensembles are useful, and we use them often, but they come at a price in terms of runtime. If it takes x milliseconds to run a sample through a neural network, and we have an ensemble of 20 networks, then our evaluation time (inference time) has jumped to $20x$ milliseconds, ignoring the possibility of parallel execution. In some situations, that amount of time is unacceptable (to say nothing of the storage and power requirements for 20 big networks versus 1). Since the net result of an ensemble of models is better overall performance, we can say that an ensemble is a kind of regularizer as well since it embodies the “wisdom of the crowd.”

Dropout takes the ensemble idea to an extreme but does so only during training and without creating a second network so that in the end, we still have one model to deal with. Like many good ideas in

statistics, this one requires randomness. Right now, when we train the network, we do a forward pass using the current weights and biases. What if, during that forward pass, we randomly assign a 0 or a 1 to each node of the network so that nodes with a 1 are used in the next layer, while nodes with a 0 are dropped out? We'd effectively be running the training samples through a different neural network configuration each time. For example, see Figure 6-6.

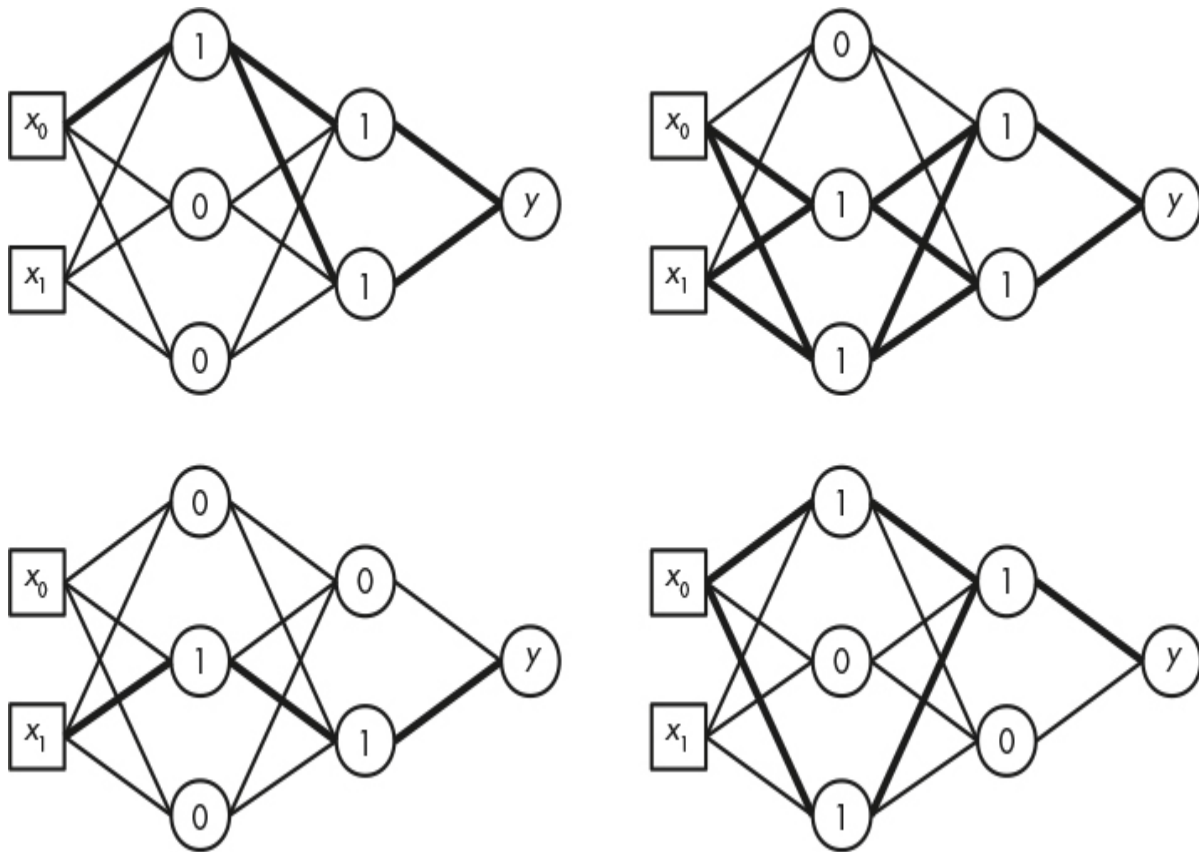


Figure 6-6: Possible networks used when applying dropout during training

Here we show the network of Figure 5-1 from page 130 but with a 0 or 1 for each of the hidden nodes. This 0 or 1 determines whether the output is used. The heavy lines in the network show the connections that are still valid. In other words, the heavy lines show the network that was actually used to create the output accumulated for backprop.

If we do this for each training sample, we can readily see that we'll be training a vast number of neural networks, each trained on a single sample. Moreover, since the weights and biases persist between forward

passes, all the networks will share those weights in the hope that the process reinforces good weight values that represent the essence of the dataset. As I've mentioned several times in this chapter, learning the essence of the data is the goal of training; we want to generalize well to new data from the same virtual parent distribution that generated the training set in the first place. Dropout is serious regularization.

I previously said that we “randomly assign a 0 or a 1” to the nodes. Do we assign them equally? The probability with which we drop nodes in a layer is something we get to specify. Let's call it p . Typically, $p = 0.5$, meaning about 50 percent of the nodes in a layer will be dropped for each training sample. Setting $p = 0.8$ would drop 80 percent of the nodes, while $p = 0.1$ would drop only 10 percent. Sometimes a different probability is used for different layers of the network, especially the first input layer, which should use a smaller probability than the hidden nodes. If we drop too many of the inputs, we'll lose the source of the signal we're trying to get the network to recognize. Think of dropout applied to the input layer as a form of data augmentation.

Conceptually, dropout is training a large set of networks that share weights. We can combine the output of each of these networks with the others via a geometric mean, assuming we use a softmax output. The *geometric mean* of two numbers is the square root of their product, meaning the geometric mean of n numbers is the n th root of their product. In the case of dropout, this can be approximated by using the entire network with all the weights multiplied by the probability that they would be included. Given we said p is the probability that a node is dropped, the weights need to be multiplied by $1 - p$, as that is the probability the node would not be dropped. So, if we fix $p = 0.5$ and use it for all the nodes, the final network is the one where all the weights are divided by 2.

As of this writing, sklearn's `MLPClassifier` class does not support dropout, but Keras most certainly does, so we'll see dropout again in Chapter 9.

Summary

Because this is an important chapter, let's review what you've learned in a little more depth. This chapter described how to train a neural network with gradient descent and backpropagation. The overall sequence of steps is as follows:

1. Select the architecture of the model. This means the number of layers, their sizes, and the type of activation function.
2. Initialize the weights and biases of the network with intelligently selected initial values.
3. Run a minibatch of training samples through the network and compute the mean loss over the minibatch. We discussed common loss functions.
4. Using backpropagation, calculate the contribution of each weight and bias to the overall loss for the minibatch.
5. Using gradient descent, update the weight and bias values of the model based on the contributions found via backpropagation. We discussed stochastic gradient descent and its relationship to the concept of minibatches.
6. Repeat from step 3 until the desired number of epochs or minibatches has been processed, or the loss has dropped below a certain threshold or has stopped changing much, or when the score on a validation set of samples has reached its minimum value.
7. If the network isn't learning well, apply regularization and train again. We looked at L2 regularization and dropout in this chapter. You can also think of data augmentation, or increasing the size or representativeness of the training set, as regularization.

The goal of training a neural network is to learn the parameters of a model that generalizes well to unseen inputs. This is the goal of all supervised machine learning. For a neural network, we know it's able to approximate any function, with enough capacity and enough training data. We may think that we're doing nothing more than ordinary optimization, but, in an important sense, we are not. Perfection on the

training set is often not a good thing but a sign of overfitting (or not—recall double descent). Instead, we want the model to learn a function that captures the essential nature of the function implied by the training set. We use the test data to give us confidence that we've learned a useful function.

In the next chapter, we'll get real and explore traditional neural networks through a series of experiments using sklearn.

OceanofPDF.com

7

EXPERIMENTS WITH NEURAL NETWORKS



In Chapter 6, we discussed the theory behind neural networks. In this chapter, we'll trade equations for code and run experiments designed to increase your intuition regarding the essential parameters of neural networks: architecture and activation functions, batch size, base learning rate, training-set size, L2 regularization, momentum, weight initialization, feature ordering, and the precision of the weights and biases.

To save space and eliminate tedious repetition, we won't show the specific code for each experiment. In most cases, the code is trivially different from the previous example; we're usually changing only the particular argument to the `MLPClassifier` constructor we're interested in. The code for each experiment is included on the book's GitHub site, and we'll list the network parameters and the name of the file. When necessary, we'll provide code to clarify a particular approach. We'll show the code for the first experiment in its entirety.

The Dataset

We'll be working with the MNIST dataset's vector form, which we assembled in Chapter 2. Recall that this dataset consists of 28×28-pixel 8-bit grayscale images of handwritten digits, [0, 9]. In vector form, each 28×28 image is unraveled into a vector of $28 \times 28 = 784$ elements, all bytes ([0, 255]). The unraveling lays each row end to end. Therefore, each sample has 784 elements and an associated label. The training set has 60,000 samples, while the test set has 10,000.

For our experiments, we won't use all the data in the training set. This is to help illustrate the effect of network parameters and to keep our training times reasonable. Refer to Figure 2-3 on page 49 for representative MNIST digits.

The MLPClassifier Class

The `MLPClassifier` class follows the same format as the other sklearn classifiers. There is a constructor and the expected methods: `fit` for training, `score` for applying the classifier to test data, and `predict` to make a prediction on unknown inputs. We'll also use `predict_proba` to return the actual predicted per-class probabilities. The constructor has many options:

```
MLPClassifier(hidden_layer_sizes=(100, ), activation='relu',
              solver='adam', alpha=0.0001, batch_size='auto',
              learning_rate='constant', learning_rate_init=0.001,
              power_t=0.5, max_iter=200, shuffle=True,
              random_state=None, tol=0.0001, verbose=False,
              warm_start=False, momentum=0.9, nesterovs_momentum=True,
              early_stopping=False, validation_fraction=0.1, beta_1=0.9,
              beta_2=0.999, epsilon=1e-08)
```

I've provided the default values for each parameter. See the sklearn documentation at <https://scikit-learn.org> for a complete description of each parameter. We'll set some of these to specific values, and others will be changed for the experiments, while still others are relevant only

in specific situations. The key parameters we'll work with are in Table 7-1.

Table 7-1: Important `MLPClassifier` Constructor Keywords and Their Descriptions

Keyword	Description
<code>hidden_layer_sizes</code>	Tuple giving the hidden-layer sizes
<code>activation</code>	Activation function type; for example, ReLU
<code>alpha</code>	L2 parameter—we called it λ (lambda)
<code>batch_size</code>	Minibatch size
<code>learning_rate_init</code>	The learning rate, η (eta)
<code>max_iter</code>	Number of training epochs
<code>warm_start</code>	Continue training or start again
<code>momentum</code>	Momentum
<code>solver</code>	Solver algorithm (SGD)
<code>nesterovs_momentum</code>	Use Nesterov momentum (<code>False</code>)
<code>early_stopping</code>	Use early stopping (<code>False</code>)
<code>learning_rate</code>	Learning rate schedule (constant)
<code>tol</code>	Stop early if loss change < tol (1e-8)
<code>verbose</code>	Output to console while training (<code>False</code>)

The following set of experiments explores the effect of various `MLPClassifier` parameters. As mentioned, I'll show all the code used for the first experiment, understanding that only small changes are needed to

perform the other experiments. At times, I'll show code snippets to make the change concrete.

Architecture and Activation Functions

When designing a neural network, we immediately face two fundamental questions: What architecture, and what activation function? These are arguably the most important deciding factors for a model's success. Let's explore what happens when we train a model by using different architectures and activation functions while holding the training dataset fixed.

The Code

As promised, for this first experiment we'll show the code in its entirety, starting with the helper functions in Listing 7-1.

mnist_nn_experiments.py

```
import numpy as np
import time
from sklearn.neural_network import MLPClassifier

def run(x_train, y_train, x_test, y_test, clf):
    s = time.time()
    clf.fit(x_train, y_train)
    e = time.time()-s
    loss = clf.loss_
    weights = clf.coefs_
    biases = clf.intercepts_
    params = 0
    for w in weights:
        params += w.shape[0]*w.shape[1]

    for b in biases:
        params += b.shape[0]
    return [clf.score(x_test, y_test), loss, params, e]

def nn(layers, act):
```



```
return MLPClassifier(solver="sgd", verbose=False, tol=1e-8,
                    nesterovs_momentum=False, early_stopping=False,
                    learning_rate_init=0.001, momentum=0.9, max_iter=200,
                    hidden_layer_sizes=layers, activation=act)
```

Listing 7-1: Helper functions for experimenting with the architecture and activation function

We import the usual modules and then define two helper functions, `run` and `nn`. Starting with `nn`, we see that it returns an instance of `MLPClassifier` using the hidden-layer sizes and the given activation function type.

The hidden-layer sizes are given as a tuple, and each element is the number of nodes in the corresponding layer. Recall that sklearn works with fully connected layers, so we need only a single number to specify the size. The input samples given for training determine the size of the input layer. Here the input samples are vectors representing the digit images, so the input layer has $28 \times 28 = 784$ nodes.

What about the output layer? It's not specified explicitly because it depends on the number of classes in the training labels. The MNIST dataset has 10 classes, so the output layer will have 10 nodes. When the `predict_proba` method is called to get an output probability, sklearn applies a softmax over the 10 outputs. If the model is binary, meaning the only class labels are 0 and 1, then there is only one output node, a logistic (sigmoid), representing the probability of belonging to class 1.

Now let's look at the parameters we passed in to `MLPClassifier`. First, we explicitly state that we want to use the SGD solver. The solver is the approach used to modify the weights and biases during training. All the solvers use backprop to calculate the gradients; how we use those gradients varies. Plain vanilla SGD is good enough for us right now.

Next, we set a low tolerance so that we'll train the requested number of epochs (`max_iter`). We also turn off Nesterov momentum (a variant of standard momentum) and early stopping (generally useful but not desired here).

The initial learning rate is set to the default value of 0.001, as is the value of standard momentum, 0.9. The number of epochs is arbitrarily

set to 200 (the default), but we'll explore this more in the experiments that follow. Please indulge your curiosity at all times to learn how changing these values affects training. For consistency's sake, we'll use these values as defaults throughout unless they are the parameters we want to experiment with.

The other helper function in Listing 7-1 is `run`. This function will train and test the classifier object it's passed, using the standard sklearn `fit` and `score` methods. It also does some other things that we've not seen before.

In particular, after timing how long training takes, we extract the final training loss value, the network weights, and the network biases from the `MLPClassifier` object so that we can return them. The `MLPClassifier` class minimizes the log-loss (see Chapter 6). The log-loss is stored in the `loss_` member variable. The size of this value, and how it changes during training, gives us a clue as to how well the network is learning. In general, the smaller the log-loss, the better the network is doing. As you explore neural networks, you'll begin to develop intuition for what a good loss value is and whether the training process is learning quickly by how rapidly the loss changes.

The weights and biases are stored in the `coefs_` and `intercepts_` member variables. These are lists of NumPy matrices (weights) and vectors (biases), respectively. Here we use them to calculate the number of parameters in the network by summing the number of elements in each matrix and vector with the two small loops in the `run` function. Finally, we return all this information, including the score against the test set, to the `main` function. Listing 7-2 shows the `main` function.

mnist_nn_experiments.py

```
def main():
    x_train = np.load("mnist_train_vectors.npy").astype("float64")/256.0
    y_train = np.load("mnist_train_labels.npy")
    x_test = np.load("mnist_test_vectors.npy").astype("float64")/256.0
    y_test = np.load("mnist_test_labels.npy")

    N = 1000
    x_train = x_train[:N]
```

```

y_train = y_train[:N]
x_test = x_test[:N]
y_test = y_test[:N]

layers = [
    (1,), (500,), (800,), (1000,), (2000,), (3000,),
    (1000,500), (3000,1500),
    (2,2,2), (1000,500,250), (2000,1000,500),
]

for act in ["relu", "logistic", "tanh"]:
    print("%s:" % act)
    for layer in layers:
        scores = []
        loss = []
        tm = []
        for i in range(10):
            s, l, params, e = run(x_train, y_train, x_test, y_test,
                                nn(layer, act))

            scores.append(s)
            loss.append(l)
            tm.append(e)
        s = np.array(scores)
        l = np.array(loss)

        t = np.array(tm)
        n = np.sqrt(s.shape[0])
        print("    layers: %14s, score= %0.4f +/- %0.4f,
              loss = %0.4f +/- %0.4f (params = %6d, time = %0.2f s)" % \
              (str(layer), s.mean(), s.std()/n, l.mean(),
               l.std()/n, params, t.mean()))

```

Listing 7-2: The main function for experimenting with the architecture and activation function

We first load the MNIST train and test data stored in `x_train` (samples) and `y_train` (labels), and `x_test` and `y_test`. Notice that we divide

the samples by 256.0 to make them floats in the range [0, 1). This normalization is the only preprocessing we'll do in this chapter.

As the full training set has 60,000 samples and we want to run many training sessions, we'll use only the first 1,000 samples for training. We'll likewise keep the first 1,000 test samples. Our goal in this chapter is to see relative differences as we change parameters, not to build the best model possible, so we'll sacrifice the quality of the model to get results in a reasonable timeframe. With 1,000 training samples, we'll have only 100 instances of each digit type, on average. We'll vary the number of training samples for specific experiments.

The `layers` list holds the architectures we'll explore. Ultimately, we'll pass these values to the `hidden_layer_sizes` argument of the `MLPClassifier` constructor. We'll examine architectures ranging from a single hidden layer with a single node to three hidden layers with up to 2,000 nodes per layer.

The `main` loop runs over three activation function types: ReLU, logistic (sigmoid) unit, and the hyperbolic tangent. We'll train a model for each combination of activation function type and architecture (`layers`). Moreover, since we know neural network training is stochastic, we'll train 10 models for each combination and report the mean and standard error of the mean, so we're not thrown off by a particularly bad model that isn't representative.

When you run the code in the experiments that follow, you'll likely generate warning messages from `sklearn` like this one:

```
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reached
and the optimization hasn't converged yet.
```

The messages are `sklearn`'s way of telling you the number of training iterations completed before `sklearn` felt that the network had converged to a good set of weights. The warnings are safe to ignore, and you can add `-W ignore` to the command line when you run the code to disable them completely:

```
> python3 -W ignore mnist_nn_experiments.py
```

The Results

Running this code takes hours to complete; it's best to let it run overnight. The code produces output with lines that look something like the following:

```
layers:(3000,1500), score=0.8822+/-0.0007, loss=0.2107+/-0.0006  
(params=6871510, time=253.42s)
```

This tells us that using a ReLU activation function, and an architecture with two hidden layers of 3,000 and 1,500 nodes each, the models had an average score of 88.2 percent and an average final training loss of 0.21 (remember that lower is better). The output also tells us that the neural network had a total of nearly 6.9 million parameters and took, on average, a little more than four minutes to train.

Table 7-2 summarizes the scores for the various network architectures and activation function types.

Table 7-2: The Mean Score (Mean $\pm$ SE) on the MNIST Test Set as a Function of the Architecture and Activation Function Type

Architecture	ReLU	Tanh	Logistic (sigmoid)
1	0.2066 $\pm$ 0.0046	0.2192 $\pm$ 0.0047	0.1718 $\pm$ 0.0118
500	0.8616 $\pm$ 0.0014	0.8576 $\pm$ 0.0011	0.6645 $\pm$ 0.0029
800	0.8669 $\pm$ 0.0014	0.8612 $\pm$ 0.0011	0.6841 $\pm$ 0.0030
1,000	0.8670 $\pm$ 0.001	0.8592 $\pm$ 0.0014	0.6874 $\pm$ 0.0028
2,000	0.8682 $\pm$ 0.0008	0.8630 $\pm$ 0.0012	0.7092 $\pm$ 0.0029
3,000	0.8691 $\pm$ 0.0005	0.8652 $\pm$ 0.0011	0.7088 $\pm$ 0.0024
1,000; 500	0.8779 $\pm$ 0.0011	0.8720 $\pm$ 0.0011	0.1184 $\pm$ 0.0033
3,000; 1,500	0.8822 $\pm$ 0.0007	0.8758 $\pm$ 0.0009	0.1221 $\pm$ 0.0001
1,000; 500; 250	0.8829 $\pm$ 0.0011	0.8746 $\pm$ 0.0012	0.1220 $\pm$ 0.0000

Architecture	ReLU	Tanh	Logistic (sigmoid)
2,000; 1,000; 500	0.8850 ± 0.0007	0.8771 ± 0.0010	0.1220 ± 0.0000

In each case, we show the mean score on the reduced test set averaged over the 10 models trained (plus or minus the standard error of the mean). This table contains quite a bit of information, so let's look at it carefully.

If we look at the activation type, we see something is off. The results for the logistic activation function show improved scores as the single hidden layer gets larger, something we might expect to see, but when we move to more than one hidden layer, the network fails to train. We know that it was unable to train because the scores on the test set are abysmal. If you check the output, you'll see that the loss values do not go down. If the loss value doesn't decrease while training proceeds, something is wrong.

It's not immediately evident why training failed for the logistic activation function case. One possibility is a bug in sklearn, but this is rather unlikely given how widely used the toolkit is. The most likely culprit has to do with network initialization. The sklearn toolkit uses the standard, commonly used initialization schemes (see Chapter 5). But these are tailored for ReLU and tanh activation functions and may not be performing well for the logistic case.

For our purposes, we can view this failure as a glaring sign that the logistic activation function is not a good one to use for the hidden layers. Sadly, this is precisely the activation function that was widely used throughout much of the early history of neural networks, so we were shooting ourselves in the foot from the beginning. No wonder it took so long for neural networks to finally find their proper place! From here on out, we'll ignore the logistic activation function results.

Consider again the scores for the single-hidden-layer networks (see Table 7-2, rows 1–6). For the ReLU and tanh activation functions, we see a steady improvement in the performance of the networks. Also,

note that in each case, the ReLU activation function slightly outperforms tanh for the same number of nodes in the hidden layer, though these differences are likely not statistically significant with only 10 models per architecture. Still, it follows a general observation prevalent in the community: ReLU is preferred to tanh.

Looking at the remaining rows of Table 7-2, we see that adding a second and even third hidden layer continues to improve the test scores but with diminishing returns. This is also a widely experienced phenomenon that we should look at a little more closely. In particular, we should consider the number of parameters in the models of Table 7-2. This makes the comparison a bit unfair. If, instead, we train models that have closely matched numbers of parameters, we can more fairly compare the models' performance. Any differences in performance we see can be plausibly attributed to the number of layers used since the overall number of parameters will be virtually the same.

By modifying the layers array in Listing 7-2, we can train multiple versions of the architectures in Table 7-3. The number of nodes per layer was selected to parallel the overall number of parameters in the models.

Table 7-3: The Model Architectures Tested to Produce Figure 7-1

Architecture	Number of parameters
1,000	795,010
2,000	1,590,010
4,000	3,180,010
8,000	6,360,010
700; 350	798,360
1,150; 575	1,570,335
1,850; 925	3,173,685
2,850; 1425	6,314,185

Architecture	Number of parameters
660; 330; 165	792,505
1,080; 540; 270	1,580,320
1,714; 857; 429	3,187,627
2,620; 1,310; 655	6,355,475

Where did the magic numbers in Table 7-3 come from? I first picked the single-layer sizes we wanted to test. I then determined the number of parameters in models with those architectures. Next, I crafted two-layer architectures using the rules of thumb from Chapter 5 so that the number of parameters in those models will be close to the corresponding number of parameters in the single-layer models. Finally, I repeated the process for three-layer models. This approach lets us compare the performance of the models for very similar numbers of parameters. In essence, we're fixing the number of parameters in the model and altering only the way they interact with one another.

Training models as we did in Listing 7-2, but this time averaging 25 models instead of just 10, gives us Figure 7-1.

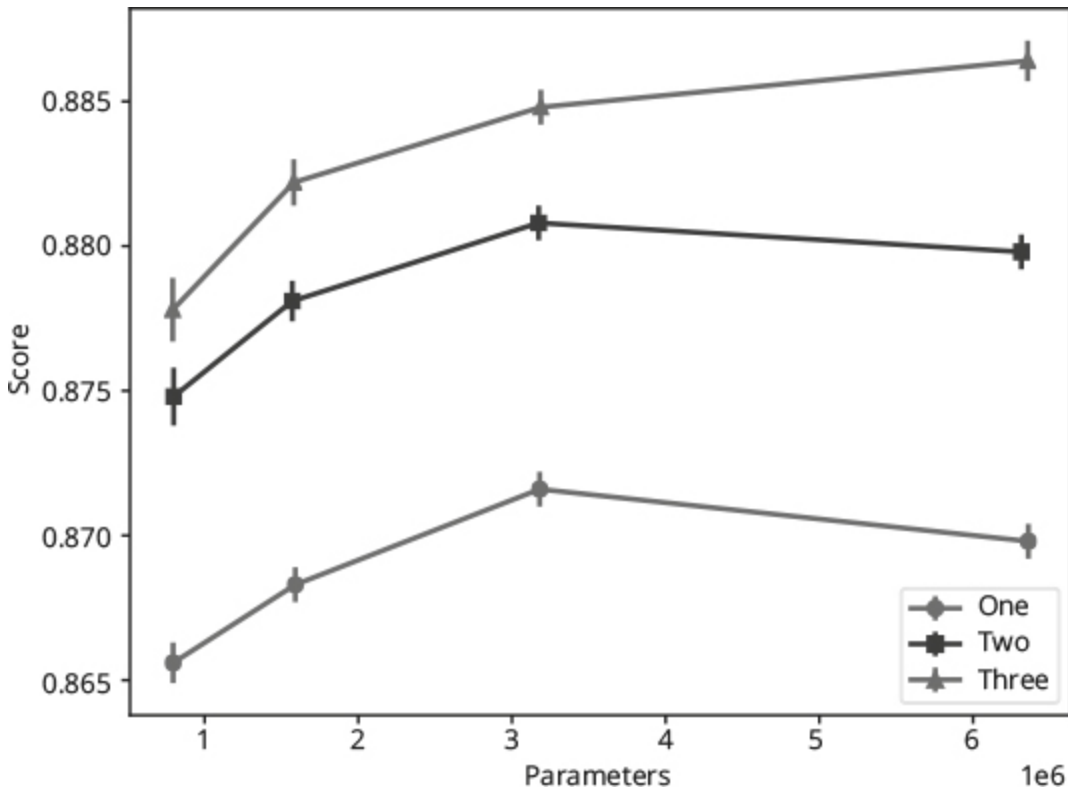


Figure 7-1: The scores (mean $\pm$ SE) on the MNIST test set for the architectures of Table 7-3 as a function of the number of parameters in the network

Let's parse Figure 7-1. First, note that the x-axis, the number of parameters in the model, is given in millions. Second, we can compare the three lines going vertically, as those models all have similar numbers of parameters. The legend tells us which plot represents models with one, two, or three hidden layers.

Looking at the leftmost points, representing the smallest models in each case, we see that changing from a single layer to two layers gives us a jump in model performance. Also, moving from two layers to three results in another, smaller rise. This repeats for all the layer sizes moving left to right. We'll address the dip in performance between the two largest models for single- and double-layer architectures in a bit. Fixing the number of parameters but increasing the depth of the network (number of layers) results in better performance, though this won't work in some cases. Still, it's worth remembering: More layers can help, not just a wider layer with more nodes.

What about the dip for the largest models in the one- and two-layer cases? These are the rightmost points of Figure 7-1. Recall, the models used to make the plot were trained with only 1,000 samples each. That likely wasn't enough data to adequately train the largest models. If we were to increase the number of training samples, which we can do because we have 60,000 to choose from for MNIST, we might see the dip go away. I leave this as an exercise for you.

Batch Size

Let's turn our attention to how batch size affects training. Recall that here *batch size* means minibatch size, a subset of the full training set used in the forward pass to calculate the average loss over the minibatch. From this loss, we use backprop to update the weights and biases. Processing a single minibatch, then, results in a single gradient-descent step—a single update to the parameters of the network.

We'll train a fixed-size subset of MNIST for a set number of epochs with different minibatch sizes to see how that affects the final test scores. Before we do that, however, we need to understand, for epochs and minibatches, the process sklearn uses to train a neural network.

Let's look briefly at the sklearn source code for the `MLPClassifier` class, in the `_fit_stochastic` method, found at https://github.com/scikit-learn/scikit-learn/blob/7389dba/sklearn/neural_network/multilayer_perceptron.py. Understanding that this method is an internal one and might change from version to version, we see code that looks like this:

```
for it in range(self.max_iter):
    X, y = shuffle(X, y, random_state=self._random_state)
    accumulated_loss = 0.0
    for batch_slice in gen_batches(n_samples, batch_size):
        activations[0] = X[batch_slice]
        batch_loss, coef_grads, intercept_grads = self._backprop(
            X[batch_slice], y[batch_slice], activations, deltas,
            coef_grads, intercept_grads)
        accumulated_loss += batch_loss * (batch_slice.stop -
```

```

                                batch_slice.start)
    grads = coef_grads + intercept_grads
    self._optimizer.update_params(grads)
    self.n_iter_ += 1

```

This code has two `for` loops, the first over the number of epochs (`max_iter`), and the second over the number of minibatches present in the training data. The `gen_batches` function returns minibatches from the training set. In reality, it returns slice indices with `x[batch_slice]` returning the actual training samples, but the effect is the same. The calls to `_backprop` and `update_params` complete the gradient-descent step for the current minibatch.

An *epoch* is a full pass through the minibatches present in the training set. The minibatches themselves are groupings of the training data so that looping over the minibatches uses all the samples in the training set once. If the number of training samples is not an integer multiple of the minibatch size, the final minibatch will be smaller than expected, but that won't affect training in the long run.

We can view this graphically as in Figure 7-2, which shows how an epoch is built from the minibatches in the training set.

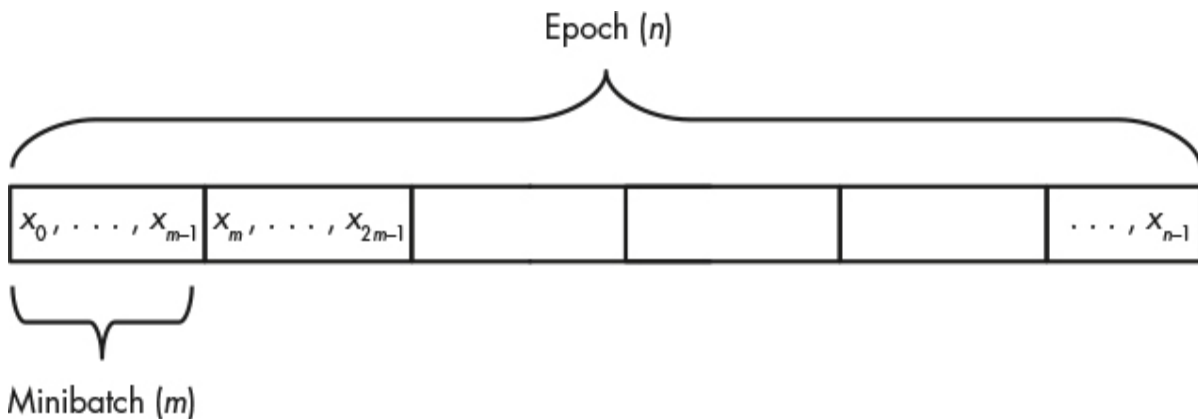


Figure 7-2: The relationship between epochs (n), minibatches (m), and samples $\{x_0, x_1, \dots, x_{n-1}\}$

In Figure 7-2, the entire training set is represented as the epoch with n samples. A minibatch has m samples, as indicated. The last minibatch is smaller than the rest to indicate that the n/m might not be an integer.

The figure also implies that the order of the samples in the training set is essential, which is why we shuffled the datasets when we made them. The sklearn toolkit will also rearrange the samples after every epoch during training if desired. As long as a minibatch is, statistically, a random sample from the training set as a whole, everything should be okay. If the minibatch is not, it might give a biased view of the gradient direction during backprop.

Our minibatch experiment will fix the number of MNIST training samples at 16,384 while we vary the minibatch size. We'll also fix the number of epochs at 100. The scores we report are the mean and standard error for five runs of the same model, each with a different random initialization. The `MLPClassifier` object is therefore instantiated via the following:

```
MLPClassifier(solver="sgd", verbose=False, tol=1e-8,  
              nesterovs_momentum=False, early_stopping=False,  
              learning_rate_init=0.001, momentum=0.9, max_iter=100,  
              hidden_layer_sizes=(1000,500), activation="relu",  
              batch_size=bz)
```

This code indicates that all the models have two hidden layers of 1,000 and 500 nodes, respectively, making the architecture of the entire network $784 \times 1,000 \times 500 \times 10$ when adding in the nodes of the input and output layers. The only parameter that varies when defining a network is the `batch_size`. We'll use the batch sizes in Table 7-4 along with the number of gradient-descent steps taken for each epoch (see Figure 7-2).

Table 7-4: The Minibatch Sizes and Corresponding Gradient-Descent Steps

Minibatch size	SGD steps per epoch
2	8,192
4	4,096
8	2,048
16	1,024

Minibatch size	SGD steps per epoch
32	512
64	256
128	128
256	64
512	32
1,024	16
2,048	8
4,096	4
8,192	2
16,384	1

When the minibatch size is 2, more than 8,000 gradient-descent steps are taken per epoch, but when the minibatch size is 8,192, only two gradient-descent steps are taken. Fixing the number of epochs should favor a smaller minibatch size since there will be correspondingly more gradient-descent steps, implying more opportunity to move toward the optimal set of network parameters.

Figure 7-3 plots the mean score as a function of the minibatch size.

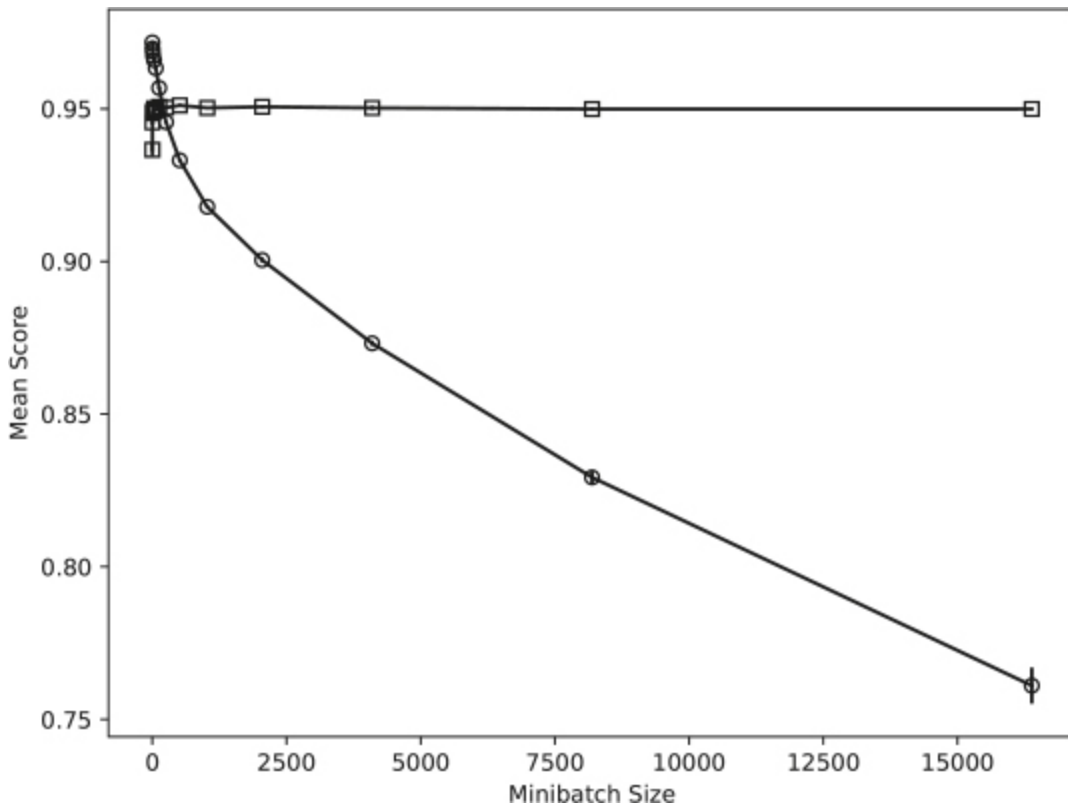


Figure 7-3: The average score on the MNIST test set as a function of minibatch size (mean $\pm$ SE) for a fixed number of epochs (100) regardless of the minibatch size (circles) or a fixed number of minibatches (squares)

The code that generated the data for the plot is in the `mnist_nn_experiments_batch_size.py` file, and the plotting code itself is in `mnist_nn_experiments_batch_size_plot.py`. The curve that concerns us for the moment is the one using circles. We'll explain the square-symbol curve shortly.

Here we've fixed the number of epochs at 100, so by varying the minibatch size, we vary the number of gradient steps: The larger the minibatch, the *fewer* gradient steps we take. Because the minibatch is larger, the steps themselves are based on a more faithful representation of the actual gradient direction; however, the number of steps is reduced because there are fewer minibatches per epoch, leading to poorer convergence: We are not reaching a good minimum of the loss function.

A more "fair" test might be to see what happens when we adjust the number of epochs so that the number of *minibatches* examined is constant regardless of the minibatch size. We may do this by noting that

the number of minibatches per epoch is n/m , where n is the number of training samples and m is the number of minibatches. If we call the overall number of minibatches we want to run M , then, to hold it fixed, we need to set the number of *epochs* to the following so that regardless of m , we perform a total of M gradient-descent steps during training:

$$E = \frac{Mm}{n}$$

Let's keep the same set of minibatches but alter the number of epochs according to the preceding equation. We need to select M , the overall number of minibatches (gradient-descent steps). Let's set it to $M = 8,192$ so that the number of epochs is an integer in each case. When the minibatch size is 2, we use one epoch to get 8,192 minibatches. And when the minibatch size is 16,384 (n is still also 16,384 samples), we get 8,192 epochs.

If we do this, we get a completely different set of results, the square-symbol curve in Figure 7-3, where we see that the mean score is pretty much a constant representing the constant number of gradient-descent updates performed during training. When the minibatch size is small, corresponding to points near 0 in Figure 7-3, we do see a degradation in performance. However, after a certain minibatch size, the performance levels off, reflecting the constant number of gradient-descent updates combined with a reasonable estimate of the true gradient from using a large enough minibatch.

For the set of base neural network parameters, specifically for a fixed learning rate, fixing the number of epochs results in reduced performance because of the design of sklearn. Fixing the number of minibatches examined results in mainly constant performance.

Base Learning Rate

In Chapter 6, we introduced the basic equation for updating the weights of a neural network during training:

$$w \leftarrow w - \eta \Delta w$$

Here η (eta) is the learning rate, the parameter that controls the step size based on the gradient value, Δw . In sklearn, η is specified via the `learning_rate_init` parameter. During training, the learning rate is often reduced so that the step sizes get smaller the closer we get to the training minimum (hopefully!). For our experiments, however, we're using a constant learning rate, so whatever value we set `learning_rate_init` to persists throughout the entire training session. Let's see how this value affects learning.

For this experiment, we fix the minibatch size at 64 samples and the architecture to (1,000, 500), meaning two hidden layers with 1,000 and 500 nodes, respectively. We then look at two main effects. We get the first when we fix the number of epochs regardless of the base learning rate. In this case, we'll always take a set number of gradient-descent steps during training. The second case fixes the *product* of the base learning rate and the number of epochs. This case is interesting because it looks at the effect on the test score of fewer large steps versus many small steps. The code for these experiments is in *mnist_experiments_base_lr.py*. The training set is the first 20,000 MNIST samples.

The first experiment fixes the epochs at 50 and loops over multiple base learning rates:

```
[0.2, 0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001]
```

The second uses the same base learning rates but varies the number of epochs so that in each case the product of the base learning rate and epochs is 1.5. This leads to the following number of epochs matched to the preceding base learning rates:

```
[8, 15, 30, 150, 300, 1500, 3000, 15000]
```

Running the two experiments again takes some time. When they're complete, we can plot the test score as a function of the base learning rate size. Doing this gives us Figure 7-4; the circles represent the fixed epochs case, and the squares are the fixed product of the base learning rate and the epochs case.

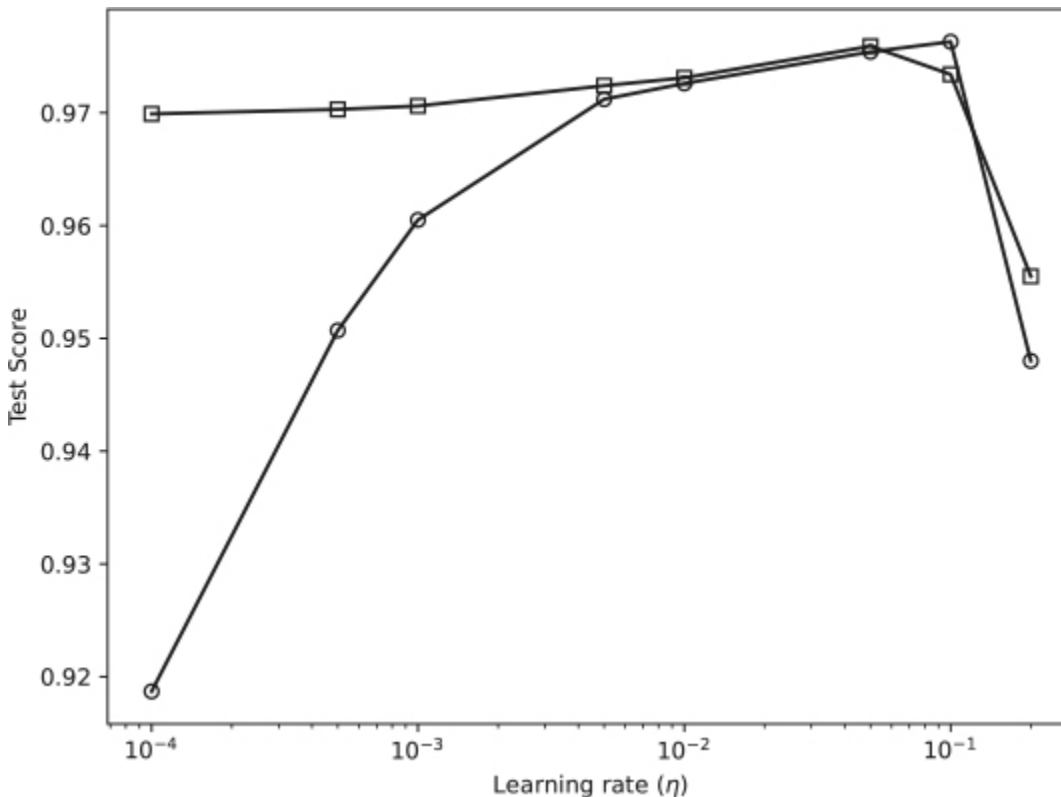


Figure 7-4: The MNIST test scores as a function of the base learning rate

Figure 7-4 shows two plots. In the first plot, using circles, the number of epochs was fixed at 50. Fixing the number of epochs fixes the number of gradient-descent steps taken during training. We then vary the learning rate. The larger the learning rate, the bigger the steps we'll take.

Imagine walking over a football field, attempting to get to the very center from one of the corners in a limited number of steps. If we take large steps, we move over a lot of ground quickly but won't be able to zero in on the center because we'll keep stepping past it. If we take tiny steps, we'll cover only a short distance from the corner toward the center. We might be on track, but since we're allowed only a certain number of steps, we can't reach the center. Intuitively, we can perhaps convince ourselves that there is a sweet spot where the step size and the number of steps we get to take combine to get us to the center.

We see this effect in the circle plot of Figure 7-4. The leftmost point represents the case of tiny steps. We do relatively poorly because we haven't traversed enough of the error space to find the minimum.

Similarly, the rightmost point represents taking very large steps. We do poorly because we keep stepping past the minimum. The best score happens when the number of steps we get to make and the size of those steps work together to move us to the minimum. In the figure, this happens when the base learning rate is 0.1.

Now let's look at the square-symbol plot in Figure 7-4. This plot comes from the scores found when the product of the base learning rate and the number of epochs is constant, meaning small learning rates will run for a large number of epochs. For the most part, the test scores are the same for all base learning rates except the very largest. In our walking over the football field thought experiment, the square-symbol plot corresponds to taking a few large steps or very many small steps. We can imagine both approaches getting us close to the center of the field, at least until our step size is too large to let us land at the center.

Some readers might object at this point. If we compare the first three points of both the circle and square plots in Figure 7-4, we see a large gap. For the circles, the performance improves as the base learning rate increases. For the squares, however, the performance remains high and constant regardless of the base learning rate.

For the circles, we trained for 50 epochs, always. This is a more significant number of epochs than were used for the squares for the corresponding base learning rates. This means that in the circles' case, we stomped around quite a bit after we got near the center of the field. For the squares, however, we limited the number of epochs, so we stopped walking when we were near the center of the field, hence the improved performance. This implies that we need to adjust the number of epochs (gradient-descent steps taken) to match the learning rate so that we get near to the minimum of the loss function quickly, but not so quickly that we're taking large steps that won't let us converge on the minimum.

Thus far we've been holding the learning rate constant throughout training. Because of space considerations, we can't fully explore the effect of changing the learning rate during training. Still, we can use our football field thought experiment to help us visualize why changing the learning rate during training makes sense.

Recall, the network is initialized intelligently but randomly. This means we start somewhere on the field at random. The odds are low that this arbitrary position is near the center, the minimum of the error surface, so we need to apply gradient descent to move us closer to the center. At first, we might as well take significant steps to move quickly through the field. Since we're following the gradient, this moves us toward the center.

If we keep taking large steps, however, we might overshoot the center. After taking a few large steps, we might think it wise to start taking smaller steps, believing that we are now closer to our goal of reaching the center. The more we walk, the smaller our steps so we can get as close to the center as possible. This is why the learning rate is typically reduced during training with SGD.

Training-Set Size

I've mentioned that the number of samples in the training set significantly affects performance. Let's use the MNIST data to quantify this assertion. For this experiment, we'll vary the number of training-set samples while adjusting the number of epochs so that in each case, we take (approximately) 1,000 gradient-descent steps during training. The code for this experiment is in *mnist_nn_experiments_samples.py*. In all cases, the minibatch size is 100, and the architecture of the network has two hidden layers of 1,000 and 500 nodes, respectively. Figure 7-5 shows the results of this experiment.

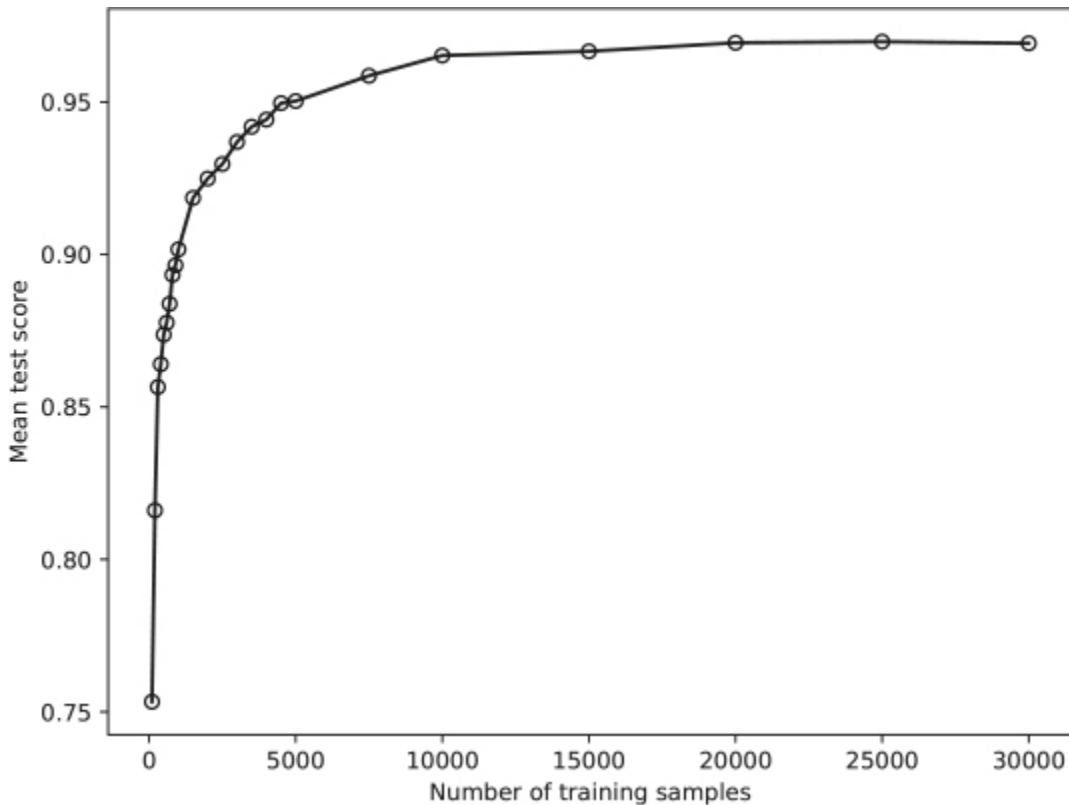


Figure 7-5: The MNIST test scores as a function of the number of training samples

This plot is particularly satisfying because it shows exactly what we'd expect to see. If we have too little training data, we can't learn to generalize well because we're training the model with a very sparse sample from the parent distribution. As we add more training data, we expect a potentially rapid rise in the performance of the network since the training set is a better sample of the parent distribution we're asking the model to learn.

The figure shows that increasing the training-set size results in diminishing returns. Moving from 1,000 to 5,000 training-set samples results in a substantial improvement in performance, but moving from 5,000 to 10,000 samples gives us only a small performance boost, and further increases in the training-set size level off at a certain ceiling performance. Think of this level region as having reached a particular capacity—that the model has pretty much learned all it will learn from the dataset. At this point, we might consider enlarging the network architecture to see whether we get a jump in test-set scores, provided we have enough training samples available.

L2 Regularization

In Chapter 6, we discussed regularization techniques that improve network generalization, including L2 regularization. L2 regularization, which adds a new term to the loss function during training, is functionally equivalent to weight decay and penalizes the network during training if the weights get large.

In `sklearn`, the parameter controlling the strength of L2 regularization is `alpha`. If this parameter is 0, no L2 regularization occurs. The regularization increases in intensity as `alpha` increases. Let's explore the effect of L2 regularization on our MNIST networks.

For this experiment, we'll fix the minibatch size at 64. We'll also set the momentum to 0 so that the effect we see is due to L2 regularization alone. Finally, we'll use a smaller network with two hidden layers of 100 and 50 nodes each and a small training set of the first 3,000 MNIST samples. The code is in *`mnist_nn_experiments_L2.py`*.

Unlike the previous experiments, in this case we'd like to evaluate the test data after each training epoch so that we can watch the network learn over the training process. If it is learning, the error on the test set will go down as the number of training epochs increases. We know that `sklearn` will loop over all the minibatches in the dataset for one epoch, so we can set the number of training epochs to 1. However, if we set `max_iter` to 1 and then call the `fit` method, the next time we call `fit`, we'll start over with a newly initialized network. This won't help us at all; we need to preserve the weights and biases between calls to `fit`.

Fortunately for us, the creators of `sklearn` thought ahead and added the `warm_start` parameter. If this parameter is set to `True`, a call to `fit` will *not* reinitialize the network but will use the existing weights and biases. If we set `max_iter` to 1 and `warm_start` to `True`, we'll be able to watch the network learn by calling `score` after each epoch of training, which gives us the accuracy on the test data. If we want the error, the value we need to track is $1 - \text{score}$. This is the value we plot as a function of epoch. The `alpha` values we'll plot are as follows:

[0.0, 0.1, 0.2, 0.3, 0.4]

We've made these rather large compared to the default so we can see the effect.

Focusing on the test error only, the code for evaluating a single epoch is as follows:

```
def epoch(x_train, y_train, x_test, y_test, clf):
    clf.fit(x_train, y_train)
    val_err = 1.0 - clf.score(x_test, y_test)
    clf.warm_start = True
    return val_err
```

Here `fit` is called to perform one epoch of training. Then we calculate the error on the test set and store it in `val_err`. Setting `warm_start` to `True` after calling `fit` ensures that the first call to `epoch` properly initializes the network, but subsequent calls keep the weights and biases from the previous call.

Training then happens in a simple loop:

```
def run(x_train, y_train, x_test, y_test, clf, epochs):
    val_err = []
    clf.max_iter = 1
    for i in range(epochs):
        verr = epoch(x_train, y_train, x_test, y_test, clf)
        val_err.append(verr)
    return val_err
```

This loop collects the per-epoch results and returns them to the `main` function, which itself loops over the α values we're interested in.

Let's run this code and plot `val_err`, the test error, as a function of the number of epochs for each `alpha`. Figure 7-6 is the result.

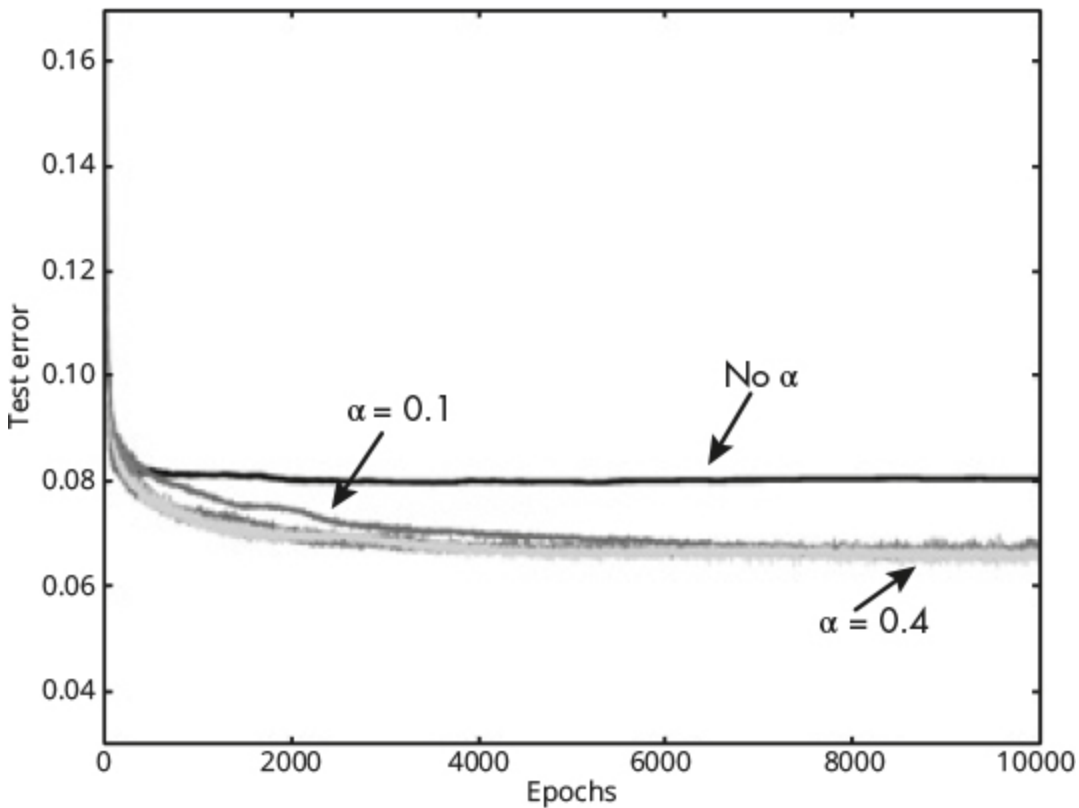


Figure 7-6: The MNIST test error as a function of training epoch for different values of α

Any nonzero value for α produces a lower test error compared to not using L2 regularization, leading us to conclude that L2 regularization is helpful. The different α values all result in approximately the same test error, but larger values are slightly more effective and reach a lower test error sooner. Compare $\alpha = 0.1$ to $\alpha = 0.4$, for example.

Larger α values seem noisier: The plot is thicker as the error jumps around more, relative to the smaller α values. To understand this, think about the total loss minimized during training. When α is large, we're placing more importance on the L2 term relative to the network's error over the minibatch. This means that when we ask the network to adjust the weights and biases during backprop, it'll be more strongly affected by the magnitude of the network's parameters than the training data itself. Because the network is focusing less on reducing the loss due to the training data, we might expect the per-epoch test error to vary more.

Momentum

Momentum alters the weight update during training by adding in a fraction of the gradient value used to update the weight in the previous minibatch. The fraction is specified as a multiplier on the previous gradient value, $[0, 1]$. We covered momentum in Chapter 6.

Let's see how changing this parameter affects training. In this case, the setup for the experiment is simple. It's identical to the approach used previously for L2 regularization, but instead of fixing the momentum parameter (μ) and varying the L2 weight (α), we'll fix $\alpha = 0.0001$ and vary μ . All the other parts remain the same: training by single epochs, the network configuration, and so forth. See the file *mnist_nn_experiments_momentum.py*.

We'll explore these momentum values:

$[0.0, 0.3, 0.5, 0.7, 0.9, 0.99]$

They range from no momentum term ($\mu = 0$) to a large momentum term ($\mu = 0.99$). Running the experiment produces Figure 7-7.

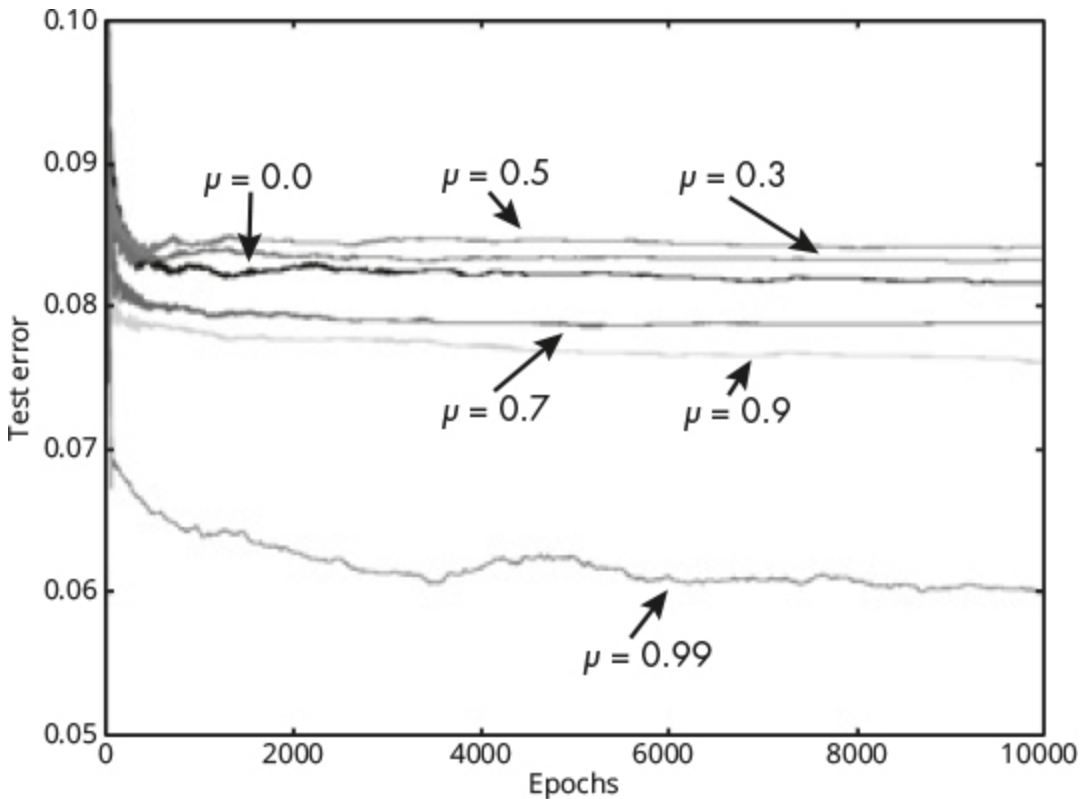


Figure 7-7: The MNIST test error as a function of training epoch for different values of μ

In Figure 7-7, we see three distinct regions. The first, represented by no momentum or relatively small momentum values ($\mu = 0.3, \mu = 0.5$), shows the highest test-set error. The second shows improvement with moderate momentum values ($\mu = 0.7, \mu = 0.9$), including the “standard” (sklearn default) value of 0.9. In this case, however, a large momentum of 0.99 lowers the test-set error from about 7.5 percent to about 6 percent. Momentum helps and should be used, especially with values near the standard of 0.9. In practice, people seldom seem to alter the momentum much, but as this example shows, sometimes it makes a big difference to the results.

We severely limited the training set to a mere 3,000 samples, about 300 per digit, which likely made momentum matter more because the training set was a small and less complete sample of the parent distribution that we want the model to learn. Increasing the training-set size to 30,000 results in a different, and more typical, ordering of the plot, where a momentum of 0.9 is the best option.

Weight Initialization

Once treated rather cavalierly, the initial set of values used for the weights and biases of a network is now known to be extremely important. The simple experiment of this section shows this plainly.

The sklearn toolkit initializes the weights and biases of a neural network by calling the `_init_coef` method of the `MLPClassifier` class. This method selects weights and biases randomly according to the Glorot algorithm (see Chapter 6). This algorithm sets the weights and biases to values sampled uniformly from the following range:

$$\left[-\sqrt{\frac{A}{f_{in}+f_{out}}}, \sqrt{\frac{A}{f_{in}+f_{out}}} \right]$$

Here f_{in} is the number of inputs and f_{out} is the number of outputs for the current layer being initialized. If the activation function is a sigmoid, $A = 2$; otherwise, $A = 6$.

If we play a little trick, we can change the way that sklearn initializes the network and thereby experiment with alternative initialization schemes. The trick uses Python's object-oriented programming abilities. If we make a subclass of `MLPClassifier`, let's call it simply `Classifier`, we can override the `_init_coef` method with our own. Python also allows us to arbitrarily add new member variables to a class instance, which gives us all we need.

The remainder of the experiment follows the format of the previous sections. We'll ultimately plot the test error by epoch of the MNIST digits trained on a subset of the full data for various initialization approaches. The model itself will use the first 6,000 training samples, a minibatch size of 64, a constant learning rate of 0.01, a momentum of 0.9, an L2 regularization parameter of 0.2, and an architecture with two hidden layers of 100 and 50 nodes each. See *mnist_nn_experiments_init.py* for this experiment's code.

We'll test four new weight initialization schemes along with the standard Glorot approach of sklearn. Table 7-5 shows the schemes.

Table 7-5: Weight Initialization Schemes

Name	Equation	Description
Glorot	$\left[-\sqrt{\frac{6}{f_{in} + f_{out}}}, \sqrt{\frac{6}{f_{in} + f_{out}}} \right]$	sklearn default
He	$N(0, 1) \sqrt{\frac{2}{f_{in}}}$	He initialization for ReLU
Xavier	$N(0, 1) \sqrt{\frac{1}{f_{in}}}$	Alternate Xavier
Uniform	$0.01(U(0, 1) - 0.5)$	Classic small uniform
Gaussian	$0.005N(0, 1)$	Classic small Gaussian

Recall that $N(0, 1)$ refers to a sample from a bell curve with a mean of 0 and a standard deviation of 1, while $U(0, 1)$ refers to a sample drawn uniformly from $[0,1)$, meaning all values in that range are equally likely except 1.0. Each of the new initialization methods sets the bias values to 0, always. However, sklearn's Glorot implementation sets the bias values in the same way it sets the weights.

NOTE

As mentioned in Chapter 6, both Xavier and Glorot refer to the same person, Xavier Glorot. We differentiate them here because the form we're calling Xavier is referred to as such in other machine learning toolkits like Caffe, and the equation is different from the one used in the original paper.

This all sounds nice, but how do we implement it in code? First, we define a new Python class, `Classifier`, which is a subclass of `MLPClassifier`. As a subclass, the new class immediately inherits all the functionality of the superclass (`MLPClassifier`) while allowing us the freedom to override any of the superclass methods with our own implementation. We simply

need to define our own version of `_init_coef` with the same arguments and return values. In code, it looks like this:

```
class Classifier(MLPClassifier):
    def _init_coef(self, fan_in, fan_out, dtype):
        if (self.init_scheme == 0):
            weights, biases = super(Classifier, self)._init_coef(fan_in, fan_out,
dtype)
        elif (self.init_scheme == 1):
            weights = 0.01*(np.random.random((fan_in, fan_out))-0.5)
            biases = np.zeros(fan_out)
        elif (self.init_scheme == 2):
            weights = 0.005*(np.random.normal(size=(fan_in, fan_out)))
            biases = np.zeros(fan_out)
        elif (self.init_scheme == 3):
            weights = np.random.normal(size=(fan_in,
fan_out))*np.sqrt(2.0/fan_in)
            biases = np.zeros(fan_out)
        elif (self.init_scheme == 4):
            weights = np.random.normal(size=(fan_in,
fan_out))*np.sqrt(1.0/fan_in)
            biases = np.zeros(fan_out)

        return weights.astype(dtype, copy=False), biases.astype(dtype,
copy=False)
```

The initialization we perform depends on the value of `init_scheme`. This is a new member variable that we use to select the initialization method (see Table 7-6).

Table 7-6: The `init_scheme` Value and Initialization Scheme

Value	Initialization method
0	sklearn default
1	Classic small uniform

Value	Initialization method
2	Classic small Gaussian
3	He initialization
4	Alternate Xavier

We set the variable immediately after creating the `Classifier` object.

We know that training a network more than once results in slightly different performance because of the way the network is initialized. Therefore, training a single network for each initialization type will likely lead to a wrong view of how well the initialization performs because we might hit a bad set of initial weights and biases. To mitigate this, we need to train multiple versions of the network and report the average performance. Since we want to plot the test error as a function of the training epoch, we need to track the test error at each epoch for each training of each initialization scheme. This suggests a 3D array:

```
test_err = np.zeros((trainings, init_types, epochs))
```

We have `trainings` trainings of each initialization type (`init_types`) for a maximum of `epochs` epochs.

With all this in place, the generation and storage of the experiment output is straightforward, if rather slow, taking the better part of a day to run:

```
for i in range(trainings):
    for k in range(init_types):
        nn = Classifier(solver="sgd", verbose=False, tol=0,
                        nesterovs_momentum=False, early_stopping=False,
                        learning_rate_init=0.01, momentum=0.9,
                        hidden_layer_sizes=(100,50), activation="relu", alpha=0.2,
                        learning_rate="constant", batch_size=64, max_iter=1)
        nn.init_scheme = k
        test_err[i,k,:] = run(x_train, y_train, x_test, y_test, nn, epochs)
np.save("mnist_nn_experiments_init_results.npy", test_err)
```

Here `nn` is the classifier instance to train, `init_scheme` sets the initialization scheme to use, and `run` is the function we defined earlier to train and test the network incrementally.

If we set the number of training sessions to 10, the number of epochs to 4,000, and plot the mean test error per epoch, we get Figure 7-8.

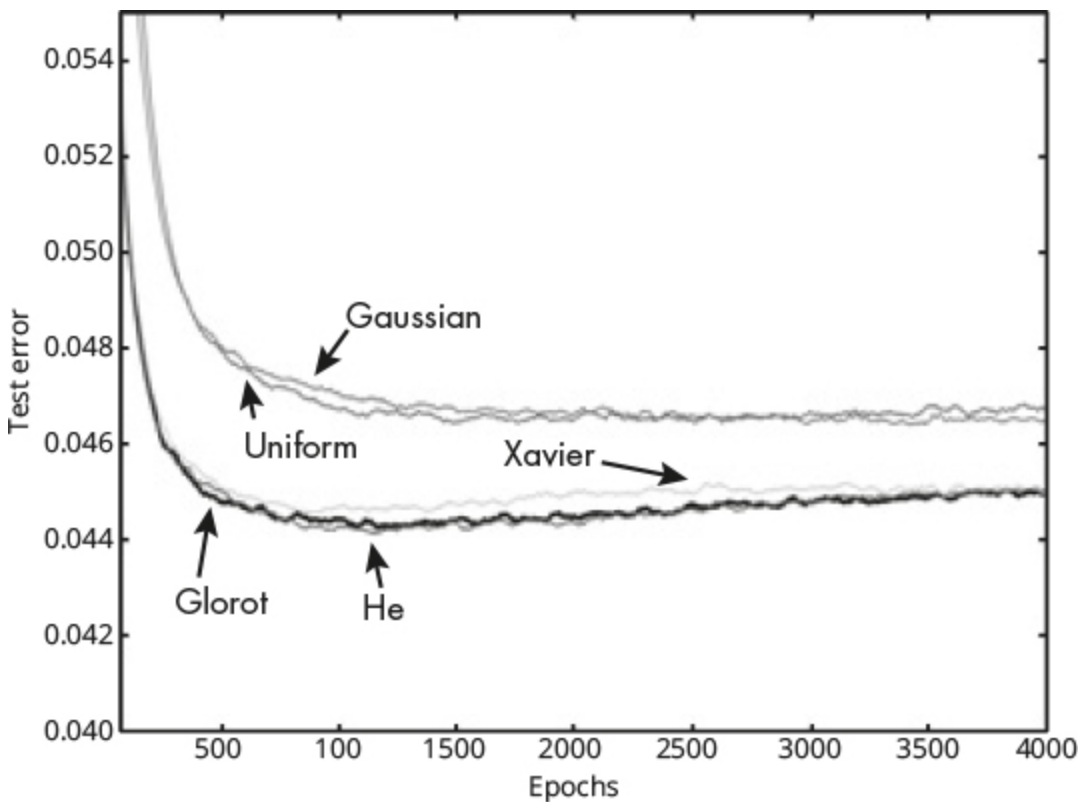


Figure 7-8: The MNIST test error as a function of training epoch for various weight initialization methods (mean over 10 training runs)

The five initialization approaches are marked, each pointing to one of the five curves in the figure. The curves themselves are familiar to us by now; they show the test-set error as a function of the training epoch. In this case, the value plotted for each curve is the average over 10 training runs of the same network architecture initialized with the same approach but different random values.

We see two distinct groups of results. On the top, we have the test error for the classic initialization approaches using small uniform or normally distributed values (Gaussian). On the bottom, we have the

results for the more principled initialization in current use. Even this basic experiment clearly shows the effectiveness of modern initialization approaches. Recall, the classic approaches were part of the reason neural networks had a bad name a few decades ago. Networks were finicky and difficult to train in large part because of improper initialization.

Looking at the bottom set of results, we see that for this experiment, there is little difference between the sklearn default initialization, which we're calling Glorot, and the initialization approach of He. The two plots are virtually identical. The plot labeled Xavier is slightly worse at first, but toward the end of our training runs matches the other two. Sklearn is using a good initialization strategy.

The plot also shows us something else. For the classic initialization approaches, the test-set error levels off and remains more or less constant. For the modern initialization approaches, we observe the test error increase slightly with the training epoch. This is particularly true for the Glorot and He methods. This increase is a telltale sign of overfitting: As we keep training, the model stops learning general features of the parent distribution and starts to focus on specific features of the training set. We didn't plot the training-set error, but it would be going down even as the test-set error starts to rise. The lowest test-set error is at about 1,200 epochs. Ideally, this would be where we stop training because we have the most reliable evidence that the model is in a good place to correctly predict new, unseen inputs. Further training tends to degrade model generalization.

The likely cause of the increase in the test error is too small of a training set, only 6,000 samples. Also, the model architecture isn't very large, with only 100 and 50 nodes in the hidden layers.

This section dramatically demonstrates the benefit of using current, state-of-the-art network initialization. When we explore convolutional neural networks (CNNs) in Chapter 9, we'll use these approaches exclusively.

Feature Ordering

We'll end our MNIST experiments with a bit of fun that we'll return to when exploring CNNs. All the experiments so far use the MNIST digits as a vector made by laying the rows of the digit images end to end.

When we do this, we know that the elements of the vector are related to one another in a way that will reconstruct the digit should we take the vector and reshape it into a 28×28 element array. This means, except for the end of one row and the beginning of the next, that the pixels in the row are still part of the digit; the spatial relationship of the image components is preserved.

However, if we scramble the pixels of the image, but always scramble them in the same way, we'll destroy the local spatial relationship among the pixels. We use this local relationship when we look at the image to decide what digit it represents. We look for the top part of a 5 to be a straight-line segment and the bottom portion to curve on the right side, and so forth.

Look at Figure 4-3 on page 120, which shows MNIST digit images on the top row and what the same digit images look like after scrambling on the bottom. In Chapter 4, we showed that this scrambling doesn't affect the accuracy of classic machine learning models; the models consider the inputs holistically, not by local spatial relationships as we do. Is this true for neural networks as well? Also, if true, will the network learn as quickly with the scrambled inputs as it does with the original images? Let's find out.

You can find the code for this experiment in *mnist_nn_experiments_scrambled.py*, where we define our now expected neural network model and train it on the first 6,000 MNIST digit samples—first as usual, and then using the scrambled versions:

```
MLPClassifier(solver="sgd", verbose=False, tol=0,
              nesterovs_momentum=False, early_stopping=False,
              learning_rate_init=0.01, momentum=0.9,
              hidden_layer_sizes=(100,50), activation="relu",
              alpha=0.2, learning_rate="constant", batch_size=64, max_iter=1)
```

We compute the test-set error as a function of the epoch and average the results over 10 runs before plotting. The result is Figure 7-9.

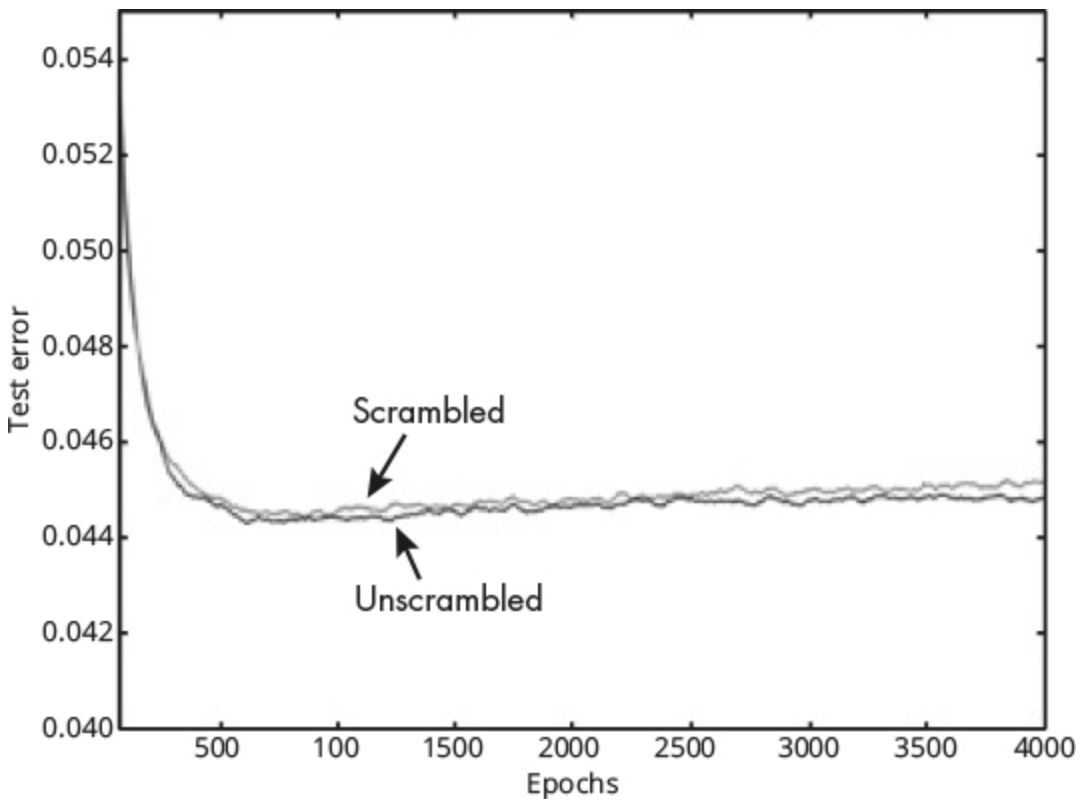


Figure 7-9: The MNIST test error as a function of training epoch for scrambled and unscrambled digits

The figure contains the answers to our earlier questions. Yes, traditional neural networks interpret their input vectors holistically, like the classic models. And yes, the network learns just as rapidly with the scrambled digits as it does with the unscrambled ones. The difference between the scrambled and unscrambled curves in Figure 7-9 is not statistically significant.

These results indicate that (traditional) neural networks “understand” their inputs in their entirety and don’t look for local spatial relationships. We’ll encounter a different outcome to this experiment when we work with CNNs in Chapter 9. It’s precisely this lack of spatial awareness (assuming images as inputs) that limited neural networks for so long and led to the development of CNNs, which are spatially aware.

Summary

In this chapter, we explored the concepts developed in Chapters 5 and 6 via experiments with the MNIST dataset. By varying key parameters associated with the network architecture and gradient-descent learning process, we increased our intuition as to how the parameters influence the overall performance of the network. Space considerations prevented us from thoroughly exploring all the `MLPClassifier` options, so I encourage you to experiment more on your own. In particular, experiment with using the various solvers, Nesterov momentum, early stopping, and, particularly crucial for training CNNs, nonconstant learning rates.

The next chapter explores techniques and metrics for evaluating the performance of machine learning models. This interlude before we jump to CNNs will supply us with tools we can use to help understand the performance of more-advanced model types.

OceanofPDF.com

8

EVALUATING MODELS



So far, we've evaluated models by looking at their accuracy on a held-out test set. This is natural and intuitive, but as you'll learn in this chapter, it's not all that we can, or should, do to evaluate a model.

We'll begin this chapter by defining metrics and delineating some basic assumptions. Then we'll look at why we need more than just accuracy. We'll introduce the concept of a confusion matrix and spend time discussing the metrics we can derive from it. From there, we'll jump to performance curves, which are the best way to compare models. Finally, we'll extend the idea of a confusion matrix to the multiclass case.

We won't say all there is to say about performance metrics, as this area is still somewhat evolving. However, by the end of this chapter, you'll be familiar with the sorts of numbers that people involved in machine learning will throw around and have a good understanding of what they mean.

Definitions and Assumptions

We can use many other metrics besides accuracy to help us evaluate how well a model is performing and to reasonably compare models. Let's

start by defining the word *metric*. For us, it is a number or set of numbers that represents something about how well the model is doing.

The value of the metric increases or decreases as the performance of the model increases or decreases, or possibly vice versa. At times, we'll be a bit sloppy and refer to graphs as metrics as well since we use them to judge the performance of a model.

We're concerned with evaluating a model for which we have a single held-out test set. We'll assume that we've followed the advice of Chapter 1 and built three datasets: a training set to teach the model, a validation set to decide when the model is done training, and a held-out test set to evaluate the trained model. We've now trained our model, thereby utilizing the training and validation sets, and want to know how well we've done.

We have another, implicit assumption in this chapter. It's a crucial one: We assume that the held-out test set is a good representation of the parent distribution that generated the data. Put another way, the held-out test set must represent the sort of data the model will encounter in the wild in as many ways as possible. For example, the frequency with which particular classes appear in the test set should match, as far as is practical, the expected rates that will be encountered when the model is used.

This is necessary because the training set is conditioning the model to expect a particular distribution, a particular set of characteristics, and if the data given to the model when it's used has different characteristics, the model won't perform well. A difference in distribution between the training set and the set of data presented to the model when it's used is one of the most common reasons that deployed machine learning models fail in actual use.

Why Accuracy Is Not Enough

A binary classifier outputs a single decision for a particular input: class 0 or class 1. Let's define the following:

N_c The number of test examples the model correctly classified

N_w The number of test examples the model got wrong

Then, the overall accuracy of this model, a number between 0 and 1, is as follows:

$$\text{ACC} = \frac{N_c}{N_c + N_w}$$

This is the accuracy as we've been using it throughout the book. In this chapter, we'll use ACC when we mean the overall accuracy.

This seems like a reasonable metric, but there are a couple of good reasons not to trust this number too much. For example, N_c and N_w tell us nothing about the relative frequency of each class. Let's see how it might affect results if one class is rare.

If the model is 95 percent accurate ($\text{ACC} = 0.95$), we might be happy. However, let's say the frequency (read *prior probability*) of class 1 is only 5 percent, meaning that on average, if we draw 100 samples from the test set, about 5 of them will be of class 1 and the other 95 will be of class 0. We see that a model that predicts all inputs are of class 0 will be right 95 percent of the time. If we stick with the overall accuracy, we might think we have a good model when, in fact, we have a terrible model that we could implement in two lines of Python:

```
def predict(x):  
    return 0
```

In this code, we say that the class is 0 regardless of the input feature vector, x . No one would be satisfied with such a model.

The prior probabilities of the classes affect the way we should think about the overall accuracy. However, if we know the following

- N_0 , the number of class 0 instances in our test set
- N_1 , the number of class 1 instances in our test set
- C_0 , the number of class 0 instances our model found
- C_1 , the number of class 1 instances our model found

we can easily compute the accuracy per class:

$$\begin{aligned}\text{ACC}_0 &= \frac{C_0}{N_0} \\ \text{ACC}_1 &= \frac{C_1}{N_1} \\ \text{ACC} &= \frac{C_0 + C_1}{N_0 + N_1}\end{aligned}$$

The final expression is just another way to compute the overall accuracy because it tallies all the correct classifications divided by the number of samples tested.

The per-class accuracy is better than the overall accuracy because it accounts for any imbalance in the frequency of the respective classes in the test set. For our previous hypothetical test set with the frequency of class 1 at 5 percent, if the classifier were predicting class 0 for all inputs, we would detect it because our per-class accuracies would be $\text{ACC}_0 = 1.0$ and $\text{ACC}_1 = 0.0$. This makes sense. We'd get every class 0 sample correct and every class 1 sample wrong (we'd call them class 0 anyway). Per-class accuracies will show up again when we consider evaluating multiclass models.

A more subtle reason to not just use the overall accuracy is that being wrong might bring a much higher cost than being right. This introduces something outside just the test set: the meaning we assign to class 0 and class 1. For example, if our model is testing for breast cancer, perhaps using the dataset we created in Chapter 2, reporting class 1 (malignant) when, in fact, the sample does not represent a malignant case might cause anxiety for the person waiting for their test results. With further testing, however, they'll be shown to not have breast cancer after all. But consider the other case. A benign result that is actually malignant might mean that they won't receive treatment, or receive it too late, which could be fatal. The relative cost of one class versus another isn't the same and might mean the difference between life and death. The same could be said of a self-driving car that thinks the child playing in the middle of the road is an empty soda can, or any number of other real-world examples.

We use models in the real world, so their outputs are connected to the real world, and sometimes the cost associated with an output is significant. Using just the overall accuracy of a model can be misleading because it doesn't take the cost of an error into account.

The 2x2 Confusion Matrix

All the models we've worked with so far have ultimately assigned each input a class label. For example, we interpret a neural network with a logistic output as a probability of membership in class 1. Using a typical threshold of 0.5 lets us assign a class label: If the output is < 0.5 , call the input class 0; otherwise, call it class 1. For other model types, the decision rule is different (for example, voting in k -NN), but the effect is the same: We get a class assignment for the input.

If we run our entire test set through our model and apply the decision rule, we get the assigned class label along with the true class label for each sample. Again, thinking only of the binary classifier case, we have four possible outcomes for each input sample in regards to the assigned class and the true class; see Table 8-1.

Table 8-1: Possible Relationships Between the Assigned Class Label and the True Class Label for a Binary Classifier

Assigned class	True class	Case
0	0	True negative (TN)
0	1	False negative (FN)
1	0	False positive (FP)
1	1	True positive (TP)

The case label defines how we talk about these situations. If the actual class of the input is class 0 and the model assigns class 0, we have a correctly identified negative case, so we have a *true negative (TN)*. If the actual class is class 1 and the model assigns class 1, we have a correctly identified positive case, so we have a *true positive (TP)*. However, if the

actual class is class 1 and the model assigns class 0, we have a positive case wrongly called a negative case, so we have a *false negative (FN)*. Finally, if the actual class is 0 and the model assigns class 1, we have a negative case wrongly called a positive case, so we have a *false positive (FP)*.

We can place each input in our test set into one, and only one, of these cases. This lets us tally the number of times each case appears in the test set, which we can present nicely as a table, such as Table 8-2.

Table 8-2: The Definitions of the Class Labels in the 2×2 Table

	Actual class 1	Actual class 0
Model assigns class 1	TP	FP
Model assigns class 0	FN	TN

I placed the case labels (TP, FP, and so forth) in the location where the actual tally counts would go for each case.

This table is called a *2×2 confusion matrix* (or *contingency table*), as it has two rows and two columns. It is a confusion matrix because it shows us at a glance how the classifier is performing and, especially, where it is confused. The classifier is confused when it assigns an instance of one class to the other class. In the 2×2 table, this confusion shows up as counts that are not along the main diagonal of the table (upper left to lower right). These are the FP and FN entries. A model that performs flawlessly on the test set has $FP = 0$ and $FN = 0$; it makes no mistakes in assigning class labels.

In Chapter 4, we experimented with the breast cancer dataset built in Chapter 2. We reported the performance of classic models against this dataset by looking at their overall accuracy. This is what the `sklearn.score` method returns. Let's now instead look at some 2×2 tables generated from the test set for these models.

The code we’re looking at is in *bc_experiments.py*; it trains multiple classic model types. Instead of using the overall accuracy, however, we’ll introduce a new function that computes the entries in the 2×2 table, as in Listing 8-1.

```
bc_experiments.py
def tally_predictions(clf, x, y):
    p = clf.predict(x)
    score = clf.score(x, y)
    tp = tn = fp = fn = 0
    for i in range(len(y)):
        if (p[i] == 0) and (y[i] == 0):
            tn += 1
        elif (p[i] == 0) and (y[i] == 1):
            fn += 1
        elif (p[i] == 1) and (y[i] == 0):
            fp += 1
        else:
            tp += 1
    return [tp, tn, fp, fn, score]
```

Listing 8-1: Generating tally counts

This function accepts a trained sklearn model object (*clf*), the test samples (*x*), and the corresponding actual test labels (*y*). The function first uses the sklearn model to predict a class label for each of the test samples; it stores the result in *p*. It then calculates the overall score, loops over each test sample, and compares the predicted class label (*p*) to the actual known class label (*y*) to see whether that sample is a true positive, true negative, false positive, or false negative. When done, all these values are returned.

Applying *tally_predictions* to the output of *bc_experiments.py* gives us Table 8-3. Here the sklearn model type is given.

Table 8-3: 2×2 Tables for the Test Set

Nearest centroid			3-NN		
	Actual 1	Actual 0		Actual 1	Actual 0

Nearest centroid	Actual 1	Actual 0
Model assigns 1	43	4
Model assigns 0	2	65

3-NN	Actual 1	Actual 0
Model assigns 1	45	1
Model assigns 0	0	68

Decision tree	Actual 1	Actual 0
Model assigns 1	44	6
Model assigns 0	1	63

SVM (linear)	Actual 1	Actual 0
Model assigns 1	45	4
Model assigns 0	0	65

We have four 2×2 tables corresponding to the test set applied to the respective models: nearest centroid, 3-NN, decision tree, and linear SVM. From the tables alone, the best-performing model was the 3-NN, as it had only one false positive and no false negatives. This model never called a true malignant case benign and only once called a benign case malignant. Given our discussion in the previous section, this is an encouraging result.

The overall accuracies for the nearest-centroid and the decision-tree models are 94.7 percent and 93.9 percent, respectively. From the accuracy alone, we might be tempted to say that the nearest-centroid model is better. However, if we look at the 2×2 tables, we see that even though the decision tree had more false positives (6), it had only one false negative, while the nearest centroid had two false negatives. In this case, a false negative means a missed cancer detection with potentially serious consequences. So, for this dataset, we want to minimize false negatives even if that means we need to tolerate a small increase in false positives. Therefore, we'll select the decision tree over the nearest-centroid model.

Metrics Derived from the 2×2 Confusion Matrix

Looking at the raw 2×2 table is helpful, but even more helpful are the metrics derived from it. We'll look at several in this section to see how they can help us interpret the information in the 2×2 table. Before we start, however, keep in mind that the metrics we'll discuss are sometimes a bit controversial. There is still healthy academic debate as to which are best to use when.

Our intention is to introduce them via examples and to describe what they are measuring. As a machine learning practitioner, you'll encounter virtually all of these from time to time, so it's wise to at least be familiar with them.

Deriving Metrics from the 2×2 Table

The first metrics are derived directly from the values in the 2×2 table: TP, TN, FP, FN. Think of these as the bread-and-butter metrics. They're easy to compute and to understand. Recall the general form of the 2×2 table from Table 8-2. We'll now define two other quantities:

$$\begin{aligned} \text{TPR} &= \frac{\text{TP}}{\text{TP} + \text{FN}} \\ \text{TNR} &= \frac{\text{TN}}{\text{TN} + \text{FP}} \end{aligned}$$

The *true-positive rate (TPR)* is the probability that an actual instance of class 1 will be correctly identified by the model. The TPR is frequently known by other names: sensitivity, recall, and hit rate. You will likely see it referred to as *sensitivity* in medical literature.

The *true-negative rate (TNR)* is the probability that an actual instance of class 0 will be correctly identified by the model. The TNR is also known as the specificity, particularly in medical literature.

Both of these quantities, as probabilities, have a value from 0 to 1; higher is better. A perfect classifier has $\text{TPR} = \text{TNR} = 1.0$, which happens when it makes no mistakes so that $\text{FP} = \text{FN} = 0$, always.

To assess a model, we need to understand the TPR and TNR together. For example, I previously mentioned that if class 1 is rare and the model always predicts class 0, it will have high accuracy. If we look at

TPR and TNR in that case, we'll see that the TNR is 1 because the model never assigns an instance of class 0 to class 1 (FP = 0). However, the TPR is 0 for the very same reason: All actual instances of class 1 are misidentified as false negatives; they get assigned to class 0. Therefore, the two metrics together immediately indicate that the model is not a good one.

How do we want the TPR and TNR to look in the breast cancer case where a false negative might be fatal? Ideally, we want them both to be as high as possible, but we might be willing to use the model anyway if the TPR is very high while the TNR might be lower. In that situation, we know that actual breast cancers, when presented, are detected almost always. Why? Because the false-negative count (FN) is virtually 0, so the denominator of the TPR is about TP, which implies a TPR of about 1.0. If, on the other hand, we tolerate false positives (actual negative instances called malignant by the model), the TNR might be well below 1.0 because the denominator of the TNR includes the FP counts.

The TPR and TNR tell us something about the likelihood that the model will pick up actual class 1 and class 0 instances. What it does not tell us, however, is how much faith we should put into the output of the model. For example, if the model says "class 1," should we believe it? To make that assessment, we need two other metrics derived directly from the 2x2 table:

$$\begin{aligned}\text{PPV} &= \frac{\text{TP}}{\text{TP} + \text{FP}} \\ \text{NPV} &= \frac{\text{TN}}{\text{TN} + \text{FN}}\end{aligned}$$

The *positive predictive value (PPV)* is most often known as the *precision*. It's the probability that when the model says the instance is of class 1, it is of class 1. Similarly, the *negative predictive value (NPV)* is the probability that the model is correct when it claims an instance is of class 0. Both values are also numbers from 0 to 1, where higher is better.

The only difference between the TPR and the PPV is whether we consider false negatives or false positives in the denominator. By including the false positives, the instances the model says are of class 1

when they are really of class 0, we get the probability that the model output is correct.

If a model always predicts class 0, the PPV is undefined because both the TP and FP are zero. All the class 1 instances are pushed into the FN count, and the TN count includes all the actual class 0 instances. If the TPR is high, but TNR is not, we have a nonzero FP count so that the PPV goes down.

Let's make up an example to see why this is so and how we might understand it. Say that our breast cancer model has produced the 2x2 table in Table 8-4.

Table 8-4: A Hypothetical 2x2 Table for a Breast Cancer Dataset

	Actual 1	Actual 0
Model assigns 1	312	133
Model assigns 0	6	645

For this example, the metrics we've covered so far are listed here:

$$\text{TPR} = 0.9811$$

$$\text{TNR} = 0.8398$$

$$\text{PPV} = 0.7011$$

$$\text{NPV} = 0.9908$$

This means a truly malignant case will be called malignant by the model 98 percent of the time, but a benign case will be called benign only 84 percent of the time. The PPV of 70 percent implies that when the model says "malignant," there's only a 70 percent chance that the case is malignant; however, because of the high TPR, we know that buried in the "malignant" outputs are virtually all the actual breast cancer cases. Notice also that this implies a high NPV, so when the model says "benign," we have very high confidence that the instance is not breast cancer. This makes the model useful even if the PPV is less than 100 percent. In a clinical setting, this model will warrant further testing

when it says “malignant” but in general, no further testing will likely be needed if it says “benign.”

Of course, defining acceptable levels of these metrics depends upon the use case for the model. Some might call an NPV of only 99.1 percent too low given the potentially very high cost of missing a cancer detection. Thoughts like these likely also motivate the recommended frequency of screening.

We can easily derive two additional basic metrics from the 2×2 table, the *false-positive rate (FPR)* and the *false-negative rate (FNR)*:

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$
$$\text{FNR} = \frac{\text{FN}}{\text{FN} + \text{TP}}$$

These metrics tell us the likelihood that a sample will be a false positive if the actual class is class 0 or a false negative if the actual class is class 1, respectively. The FPR will show up again later when we talk about using curves to assess models. Notice that $\text{FPR} = 1 - \text{TNR}$ and $\text{FNR} = 1 - \text{TPR}$.

Calculating these basic metrics is straightforward, especially if we use the output of the `tally_predictions` function defined previously as the input (Listing 8-2).

```
def basic_metrics(tally):
    tp, tn, fp, fn, _ = tally
    return {
        "TPR": tp / (tp + fn),
        "TNR": tn / (tn + fp),
        "PPV": tp / (tp + fp),
        "NPV": tn / (tn + fn),
        "FPR": fp / (fp + tn),
        "FNR": fn / (fn + tp)
    }
```

Listing 8-2: Calculating basic metrics

We break up the list that `tally_predictions` returns, disregarding the accuracy, and then build and return a dictionary containing each of the

six basic metrics we described. While robust code would check for pathological cases where the denominators are zero, we've ignored that code here to preserve clarity in the presentation.

Using Metrics to Interpret Models

Let's use `tally_predictions` and `basic_metrics` to interpret some models. We'll work with the vector form of the MNIST data but keep only digits 3 and 5 so that we have a binary classifier. The code is similar to that found in *mnist_experiments.py* (see Chapter 4).

Keeping only digits 3 and 5 leaves us with 11,552 training samples (6,131 3s and 5,421 5s) and 1,902 test samples of which 1,010 are 3s and 892 are 5s. The actual code is in *mnist_2x2_tables.py* with selected output in Table 8-5.

Table 8-5: Selected Output from MNIST 3 vs. 5 Models and Corresponding Basic Metrics

	Nearest centroid	3-NN	Naive Bayes	RF 500	Linear SVM
TP	760	878	612	884	853
TN	909	994	976	1,003	986
FP	101	16	34	7	24
FN	132	14	280	8	39
TPR	0.8520	0.9843	0.6851	0.9910	0.9563
TNR	0.9000	0.9842	0.9663	0.9931	0.9762
PPV	0.8827	0.9821	0.9474	0.9921	0.9726
NPV	0.8732	0.9861	0.7771	0.9921	0.9620
FPR	0.1000	0.0158	0.0337	0.0069	0.0238
FNR	0.1480	0.0157	0.3139	0.0090	0.0437

In Table 8-5, the raw counts are at the top and the metrics defined in this section are at the bottom. Let's understand what's happening. We'll concentrate on the metrics at the bottom of the table. The first two columns show the true-positive rate (sensitivity, recall) and the true-negative rate (specificity). We should examine these values together.

Looking at the nearest-centroid results, $TPR = 0.8520$ and $TNR = 0.9000$. Here class 1 is a 5, and class 0 is a 3. So, the nearest-centroid classifier calls 85 percent of the 5s it sees "five." Similarly, it calls 90 percent of the 3s it sees "three." While not too shabby, we should not be impressed. Looking down the columns, we see that two models performed very well for these metrics: 3-NN and the random forest with 500 trees. In both cases, the TPR and TNR were nearly identical and quite close to 1.0. This is a sign of the model performing well. Absolute perfection would be $TPR = TNR = PPV = NPV = 1.0$ and $FPR = FNR = 0.0$. The closer we get to perfection, the better. If attempting to pick the best model for this classifier, we'd likely choose the random forest because it's the closest to perfection on the test set.

For the naive Bayes results, the TNR (specificity) is reasonably high, about 97 percent. However, the TPR (sensitivity) of 68.5 percent is pathetic. Roughly speaking, only two out of every three 5s presented to this model will be correctly classified. If we examine the next two columns, the positive and negative predictive values, we see a PPV of 94.7 percent, meaning when the model says the input is a five, we can be somewhat confident that it is a five. However, the negative predictive value isn't so good at 77.7 percent. The top portion of Table 8-5 shows what's happening in this case. The FP count is only 34 out of 1,010 3s in the test set, but the FN count is high: 280 of the 5s were labeled "three." This is the source of the low NPV for this model.

Here is a good rule of thumb for these metrics: A well-performing model has TPR, TNR, PPV, and NPV very close to 1.0, and FPR and FNR very close to 0.0.

Look again at Table 8-5, particularly the lower metrics for the random forest. As their names suggest, the FPR and FNR values are rates. We can use them to estimate how often FP and FN will occur when using the model. For example, if we present the model with $N =$

1,000 cases that are 3s (class 0), we can use the FPR to estimate how many of them the model will call 5s (class 1):

$$\text{estimated number of FP} = \text{FPR} \times N = 0.0069(1,000) \approx 7$$

A similar calculation gives us the estimated number of FN for $N = 1,000$, instances that are really 5s:

$$\text{estimated number of FN} = \text{FNR} \times N = 0.0090(1,000) = 9$$

The same holds for the TPR and TNR, which also have *rate* in their names ($N = 1,000$ each for actual 3s and 5s):

$$\text{estimated number of TP} = \text{TPR} \times N = 0.9910(1,000) = 991$$

$$\text{estimated number of TN} = \text{TNR} \times N = 0.9931(1,000) = 993$$

These calculations show how well this model performs on the test data.

More-Advanced Metrics

In this section, we'll look at what I'm arbitrarily calling *more-advanced metrics*. They are more advanced because instead of using the 2×2 table entries directly, they're built from values calculated from the table itself. In particular, we'll examine five advanced metrics: informedness, markedness, F1 score, Cohen's kappa, and the Matthews correlation coefficient (MCC).

Informedness and Markedness

Informedness and markedness go together. They are somewhat less well known than other metrics in this section, but this will hopefully change in the future. Earlier, you learned that TPR (sensitivity) and TNR (specificity) should be interpreted together. The *informedness* (also called *Youden's J statistic*) does just that:

$$\text{informedness} = \text{TPR} + \text{TNR} - 1$$

Informedness is a number in $[-1, +1]$ that combines the TPR and TNR. The higher the informedness, the better. An informedness of 0 implies random guessing, while an informedness of 1 implies perfection (on the test set). An informedness of less than 0 might suggest a model that's worse than random guessing. An informedness of -1 implies that all true-positive instances were called negatives, and vice versa. In that case, we could swap the label the model wants to assign to each input and get a quite good model. Only pathological models lead to negative informedness values.

The *markedness* combines the positive and negative predictive values in the same way that informedness combines TPR and TNR:

$$\text{markedness} = \text{PPV} + \text{NPV} - 1$$

Markedness has the same range as informedness. The informedness speaks to how well the model correctly labels inputs from each class, and the markedness speaks to how well the model is doing at being correct when it claims a particular label for a particular input, be it class 0 or class 1. Random guessing gives a markedness near 0, and perfection a markedness near 1.

The informedness and markedness each capture essential aspects of the model's performance in a single number. Some claim that these metrics are unbiased by the prior probabilities of the particular classes. This means if class 1 is significantly less common than class 0, the informedness and markedness are not affected. For in-depth details, see "Evaluation: From Precision, Recall and F-measure to ROC, Informedness, Markedness, and Correlation" by David Martin Powers (2011).

F1 Score

The *F1 score*, rightly or wrongly, is widely used, and we should be familiar with it. It combines two basic metrics, precision (PPV) and recall (TPR), into one:

$$\text{F1} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{2 \times \text{PPV} \times \text{TPR}}{\text{PPV} + \text{TPR}}$$

The F1 score is a number in $[0, 1]$, where higher is better. It's not obvious in this form, but the F1 score is the harmonic mean of the precision and recall. A *harmonic mean* is the reciprocal of the arithmetic mean of the reciprocals. For example:

$$\begin{aligned} \text{F1} &= \left(\frac{1}{2} \left(\frac{1}{\text{precision}} + \frac{1}{\text{recall}} \right) \right)^{-1} \\ &= \left(\frac{\text{precision} + \text{recall}}{2 \times \text{precision} \times \text{recall}} \right)^{-1} \\ &= \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \end{aligned}$$

One criticism of the F1 score is that it doesn't take the true negatives into account as informedness does (via the TNR). Both the PPV and TPR quantities depend entirely on the TP, FP, and FN counts from the 2×2 table, but not the TN count. Additionally, the F1 score places equal weight on the precision and the recall. Precision is affected by false positives, while recall is affected by false negatives.

From the previous breast cancer model, we saw that the human cost of a false negative might be substantially higher than a false positive. Some argue that this must be taken into account when evaluating model performance, and indeed it should. However, if the relative costs of a false positive and a false negative are the same, the F1 score will have more meaning.

Cohen's Kappa

Cohen's kappa is another statistic commonly found in machine learning that attempts to account for the possibility that the model might put the input into the correct class by accident. Mathematically, we define the metric as follows:

$$\kappa = \frac{p_o - p_e}{1 - p_e}$$

Here p_o is the observed accuracy, and p_e is the accuracy expected by chance. For a 2×2 table, we define these values to be as follows, with N being the total number of samples in the test set:

$$p_o = (TP + TN)/N$$

$$p_e = \frac{(TP+FN)(TP+FP)}{N^2} + \frac{(TN+FP)(TN+FN)}{N^2}$$

Cohen's kappa is generally from 0 to 1, where 0 means a complete disagreement between the assigned class labels and the given class labels. A negative value indicates worse-than-chance agreement. A value near 1 indicates strong agreement.

Matthews Correlation Coefficient

Our final metric is *Matthews correlation coefficient (MCC)*, which is the geometric mean of the informedness and markedness. In that sense, like the F1 score, it's a combination of two metrics into one.

We define the MCC as the following:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$$

Mathematically, this works out to the geometric mean of the informedness and markedness:

$$MCC = \sqrt{\text{informedness} \times \text{markedness}}$$

The MCC is favored by many because it takes the full 2x2 table into account, including the relative frequency of the two classes (the class prior probabilities). The F1 score doesn't do this because it ignores the true negatives.

The MCC is a number from -1 to 1, with higher being better. If considering only one value as a metric for evaluating a binary model, make it the MCC. Note, there are four sums in the denominator of the MCC. If one of these sums is 0, the entire denominator is 0, which is a problem since we can't divide by 0. Fortunately, in that case, we can replace the denominator with 1 to give a still meaningful result. A well-performing model has MCC close to 1.0.

Metric Implementation

Let's write a function to construct these metrics from a given 2×2 table. Listing 8-3 shows the code.

```
from math import sqrt
def advanced_metrics(tally, m):
    tp, tn, fp, fn, _ = tally
    n = tp + tn + fp + fn
    po = (tp+tn) / n
    pe = (tp+fn)*(tp+fp)/n**2 + (tn+fp)*(tn+fn)/n**2

    return {
        "F1": 2.0*m["PPV"]*m["TPR"] / (m["PPV"] + m["TPR"]),
        "MCC": (tp*tn - fp*fn) / sqrt((tp+fp)*(tp+fn)*(tn+fp)*(tn+fn)),
        "kappa": (po - pe) / (1.0 - pe),
        "informedness": m["TPR"] + m["TNR"] - 1.0,
        "markedness": m["PPV"] + m["NPV"] - 1.0
    }
```

Listing 8-3: Calculating advanced metrics

For the sake of simplicity, we're not checking whether the MCC denominator is 0 as a full implementation would.

This code takes the tallies and basic metrics as arguments and returns a new dictionary with the more advanced metrics. Let's see how our MNIST example from Table 8-5 looks when we calculate the advanced metrics.

Table 8-6 shows the metrics of this section for the MNIST 3 versus 5 models.

Table 8-6: Selected Output from MNIST 3 vs. 5 Models and Corresponding Advanced Metrics

Model	F1	MCC	Cohen's kappa	Informedness	Markedness
Nearest centroid	0.8671	0.7540	0.7535	0.7520	0.7559
3-NN	0.9832	0.9683	0.9683	0.9685	0.9682

Model	F1	MCC	Cohen's kappa	Informedness	Markedness
Naive Bayes	0.7958	0.6875	0.6631	0.6524	0.7244
RF 500	0.9916	0.9842	0.9842	0.9841	0.9842
LinearSVM	0.9644	0.9335	0.9334	0.9325	0.9346

First, notice that the F1 score is always higher than the MCC or Cohen's kappa. In a way, the F1 score is overly optimistic. As previously noted, the F1 score doesn't take the true negatives into account, while both the MCC and Cohen's kappa do.

Another point to note is that well-performing models, like 3-NN and the random forest, score highly in all these metrics. When the model performs well, the difference between the F1 score and MCC is smaller than when the model is doing poorly (naive Bayes, for example). Notice also that the MCC is always between the informedness and markedness, as a geometric mean will be. Finally, from the values in Tables 8-5 and 8-6, we see that the best-performing model is the random forest, based on the MCC of 0.9842.

In this section, and the two before it, we saw how to calculate and interpret quite a few metrics. A well-performing model scores highly on all of them. It's when the models we're evaluating are less than sterling that the relative differences between the metrics, and the meaning of the metrics, really comes into play. That's when we need to consider specific metric values and the cost associated with the mistakes the models are making (FP and FN). In those cases, we have to use our judgment and problem-specific factors to decide which model to ultimately select.

Now let's shift gears and take a look at a graphical way of evaluating model performance.

The Receiver Operating Characteristics Curve

They say that a picture is worth a thousand words. In this section, we'll learn that a picture—more accurately, a curve—can be worth upward of a dozen numbers. That is, we'll learn how to turn the output of a model into a curve that captures more of the performance than the metrics of the previous sections can. Specifically, we'll learn about the widely used receiver operating characteristics (ROC) curve: what it is, how to plot it, and how to use `sklearn` to plot it for us.

Gathering the Models

To make the curve, we need a model that outputs a probability of belonging to class 1. In the previous sections, we used models that output a class label so that we could tally the TP, TN, FP, and FN counts. For our ROC curves, we still need these counts, but instead of the class label as model output, we need the probability of class 1 membership. We'll apply various thresholds to these probabilities to decide what label to give the input.

Fortunately for us, traditional neural networks (and the deep networks you'll meet in Chapter 9) output the necessary probability. If we're using `sklearn`, other classical models can also be made to output a probability estimate, but we'll ignore that fact here to keep life simple.

Our test case is a series of neural networks trained to decide between even MNIST digits (class 0) and odd MNIST digits (class 1). Our inputs are the vector form of the digits that we've been using up to this point in the book. We can use the training and test data we created in Chapter 2; we only need to recode the labels so that digits 0, 2, 4, 6, and 8 are class 0, while digits 1, 3, 5, 7, and 9 are class 1. That is easily accomplished with a few lines of code:

```
old = np.load("mnist_train_labels.npy")
new = np.zeros(len(old), dtype="uint8")
new[np.where((old % 2) == 0)] = 0
new[np.where((old % 2) == 1)] = 1
np.save("mnist_train_even_odd_labels.npy", new)

old = np.load("mnist_test_labels.npy")
new = np.zeros(len(old), dtype="uint8")
```

```
new[np.where((old % 2) == 0)] = 0

new[np.where((old % 2) == 1)] = 1
np.save("mnist_test_even_odd_labels.npy", new)
```

The directory paths point to the same place the other MNIST data is stored. We use the fact that the remainder when an even number is divided by 2 is always 0 or 1 depending on whether the number is even or odd.

To emphasize the difference between the respective models, we'll intentionally train models that we know are far from ideal. In particular, we'll use the following code to generate the models and produce the probability estimates:

```
mnist_even_odd.py
import numpy as np
from sklearn.neural_network import MLPClassifier

def run(x_train, y_train, x_test, y_test, clf):
    clf.fit(x_train, y_train)
    return clf.predict_proba(x_test)

def nn(layers):
    return MLPClassifier(solver="sgd", verbose=False, tol=1e-8,
                        nesterovs_momentum=False, early_stopping=False, batch_size=64,
                        learning_rate_init=0.001, momentum=0.9, max_iter=200,
                        hidden_layer_sizes=layers, activation="relu")

def main():
    x_train = np.load("mnist_train_vectors.npy").astype("float64")/256.0
    y_train = np.load("mnist_train_even_odd_labels.npy")
    x_test = np.load("mnist_test_vectors.npy").astype("float64")/256.0
    y_test = np.load("mnist_test_even_odd_labels.npy")
    x_train = x_train[:1000]
    y_train = y_train[:1000]
    layers = [(2,), (100,), (100,50), (500,250)]
    mlayers = ["2", "100", "100x50", "500x250"]
    for i, layer in enumerate(layers):
```



```
prob = run(x_train, y_train, x_test, y_test, nn(layer))
np.save("mnist_even_odd_probs_%s.npy" % mlayers[i], prob)
```

The `run` and `nn` functions should be familiar. We used virtually identical versions in Chapter 7, where `nn` returns a configured `MLPClassifier` object and `run` trains the classifier and returns the prediction probabilities on the test set. The `main` function loads the train and test sets, limits the training set to the first 1,000 samples (about 500 even and 500 odd), then loops over the hidden layer sizes we'll train. The first two are single-hidden-layer networks with 2 and 100 nodes, respectively. The last two are two-hidden-layer networks with 100×50 and 500×250 nodes per layer.

Plotting the Metrics

The output of `clf.predict_proba` is a matrix with as many rows as there are test samples (10,000 in this case). The matrix has as many columns as there are classes; since we're dealing with a binary classifier, there are two columns per sample. The first is the probability that the sample is even (class 0), and the second is the probability that the sample is odd (class 1). For example, Table 8-7 shows the first 10 outputs for one of the models.

Table 8-7: The Example Model Output Probabilities and Actual Label

Class 0	Class 1	Actual label
0.009678	0.990322	3
0.000318	0.999682	3
0.001531	0.998469	7
0.007464	0.992536	3
0.011103	0.988897	1
0.186362	0.813638	7
0.037229	0.962771	7

Class 0	Class 1	Actual label
0.999412	0.000588	2
0.883890	0.116110	6
0.999981	0.000019	6

The first column is the probability of being even, and the second is the probability of being odd. The third column is the actual class label for the sample, showing that the predictions are spot on. The odd digits have high class 1 probabilities and low class 0 probabilities, while the opposite is true for the even samples.

When we build a 2×2 table from the performance of a model on a held-out test set, we get a collection of TP, TN, FP, and FN numbers from which we can calculate all the metrics of the previous sections. This includes the true-positive rate (TPR, sensitivity) and the false-positive rate (FPR, equal to $1 - \text{specificity}$). Implicit in the table is the threshold we used to decide when the model output should be considered class 1 or class 0. In the previous sections, this threshold was 0.5. If the output is ≥ 0.5 , we assign the sample to class 1; otherwise, we assign it to class 0. Sometimes you'll see this threshold added as a subscript like $\text{TPR}_{0.5}$ or $\text{FPR}_{0.5}$.

Mathematically, we can consider the TPR and FPR calculated from a 2×2 table to be a point on the FPR (x-axis) versus TPR (y-axis) plane—specifically, the point (FPR, TPR). Since both FPR and TPR range from 0 to 1, the point (FPR, TPR) will lie somewhere within a square of length 1 with the lower-left corner of the square at the point (0,0) and the upper-right corner at the point (1,1).

Every time we change our decision threshold, we get a new 2×2 table leading to a new point on the FPR versus TPR plane. For example, if we change our decision threshold from 0.5 to 0.3 so that each output class 1 probability of 0.3 or higher is called class 1, we'll get a new 2×2 table and a new point, $(\text{FPR}_{0.3}, \text{TPR}_{0.3})$, on the plane. As we systematically change the decision threshold from high to low, we generate a sequence of points that we can connect to form a curve.

We call curves generated by changing a parameter in this way *parametric curves*. The points are functions of the threshold. Let's call the threshold value θ (theta) and vary it from near 1 to near 0. Doing so lets us calculate a set of points, $(FPR_{\theta}, TPR_{\theta})$, which, when plotted, lead to a curve in the FPR versus TPR plane. As noted earlier, this curve has a name: the receiver operating characteristics (ROC) curve. Let's look at an ROC curve and explore what it can tell us.

Exploring the ROC Curve

Figure 8-1 shows the ROC curve for the MNIST even-versus-odd model with a single hidden layer of 100 nodes.

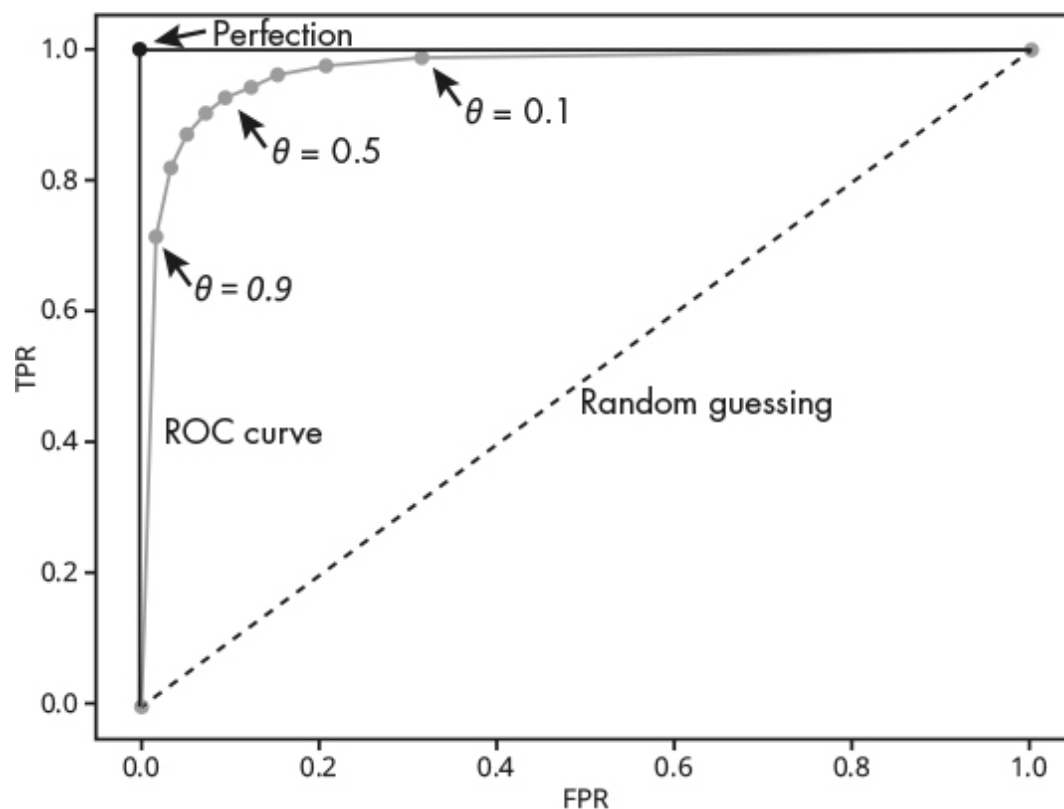


Figure 8-1: An ROC curve with key elements marked

The labeled points represent the FPR and TPR for the given threshold values. The dashed line is the diagonal from (0,0) to (1,1) and represents a classifier that guesses its output randomly. The closer our curve is to this dashed line, the less powerful the model is. If your curve lies on top of the line, you might as well flip a coin and assign the label

that way. Any curve below the dashed line is performing *worse* than random guessing. If the model were entirely wrong, meaning it calls all class 1 instances class 0, and vice versa, a curious thing happens: We can turn the entirely wrong model into a perfectly correct model by changing all class 1 output to class 0 and all class 0 output to class 1. It's unlikely that you'll run across a model this bad.

The ROC curve in Figure 8-1 has a single point labeled Perfection in the upper-left corner of the graph. This is the ideal we are striving for. We want our ROC curve to move up and to the left toward this point. The closer we get the curve to this point, the better the model is performing against our test set. A perfect model has an ROC curve that jumps up vertically to this point and then horizontally to the point (1,1). The ROC curve in Figure 8-1 is going in the right direction and represents a reasonably well-performing model.

Notice the labeled θ values. We can select a level of performance from the model by adjusting θ . In this case, the typical default value of 0.5 gives us the best performance because that threshold value returns a TPR and FPR with the best balance, the point closest to the upper left of the graph. However, there are reasons we might want to use a different θ value.

If we make θ small (say, 0.1), we move along the curve toward the right. Two changes happen. First, the TPR goes up to about 0.99, meaning we correctly assign about 99 percent of the real class 1 instances handed to the model to class 1. Second, the FPR also goes up, to about 0.32, meaning we'll simultaneously call about 32 percent of the true negatives (class 0) class 1 as well. If our problem is such that we can tolerate calling some negative instances "positive," knowing that we now have a meager chance of doing the opposite, calling a positive case "negative," we might choose to change the threshold to 0.1. Think of the previous breast cancer example: We never want to call a positive case "negative," so we tolerate more false positives to know we aren't mislabeling any actual positives.

What does it mean to move the threshold (θ) to 0.9? In this case, we've moved along the curve to the left, to a point with a very low false-positive rate. We might do this if we want to know with a high degree of

confidence that when the model says “class 1,” it is an instance of class 1. This means we want a high positive predictive value (PPV, precision). Recall the definition of the PPV:

$$\text{PPV} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

The PPV is high if FP is low. Setting θ to 0.9 makes the FP low for any given test set. For the ROC curve of Figure 8-1, moving to $\theta = 0.9$ implies an FPR of about 0.02 and a TPR of about 0.71 for a PPV of about 0.97. At $\theta = 0.9$, when the model outputs “class 1,” there is a 97 percent chance that the model is correct. In contrast, at $\theta = 0.1$, the PPV is about 76 percent. A high threshold can be used when we’re interested in definitely locating an example of class 1 without caring that we might not detect all class 1 instances.

Changing the threshold θ moves us along the ROC curve. As we do so, we should expect the metrics of the previous section to also change as a function of θ . Figure 8-2 shows us how the MCC and PPV change with θ .

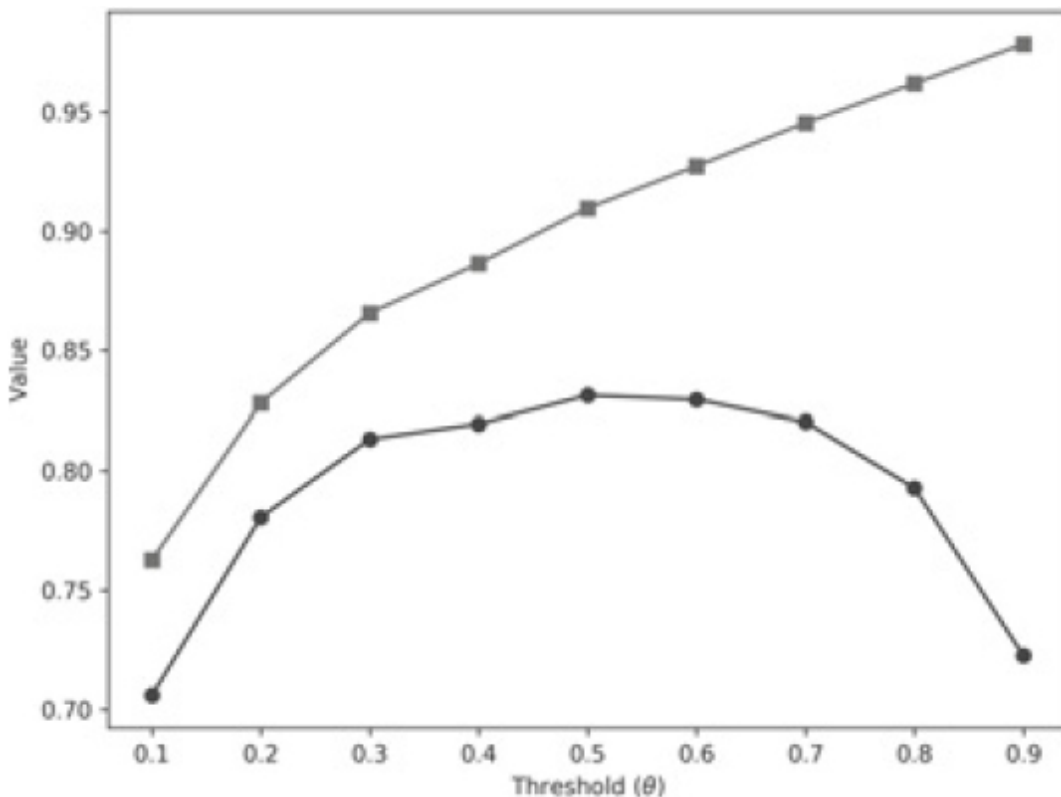


Figure 8-2: How MCC (circles) and PPV (squares) change as the decision threshold (θ) changes for the MNIST even/odd model of Figure 8-1

We can see that as the threshold goes up, so does the PPV. The model becomes more confident when it declares an input a member of class 1. However, this is tempered by the change in MCC, which, as we previously saw, is an excellent single-metric measure of overall model performance. In this case, the highest MCC is at $\theta = 0.5$, with MCC falling off as the threshold increases or decreases.

Comparing Models with ROC Analysis

The ROC curve gives us a significant amount of information. It's also handy for comparing models, even if those models are radically different from one another in architecture or approach. However, we must take care when making the comparison so that the test sets used to generate the curves are ideally the same or very nearly the same.

Let's use ROC analysis to compare the MNIST even/odd digit models that we trained previously. We'll see if this helps us to choose between them.

Figure 8-3 shows the ROC curves for these models with an inset expanding the upper-left corner of the graph to make it easier to distinguish one model from the other. The number of nodes in each hidden layer is indicated to identify the models.

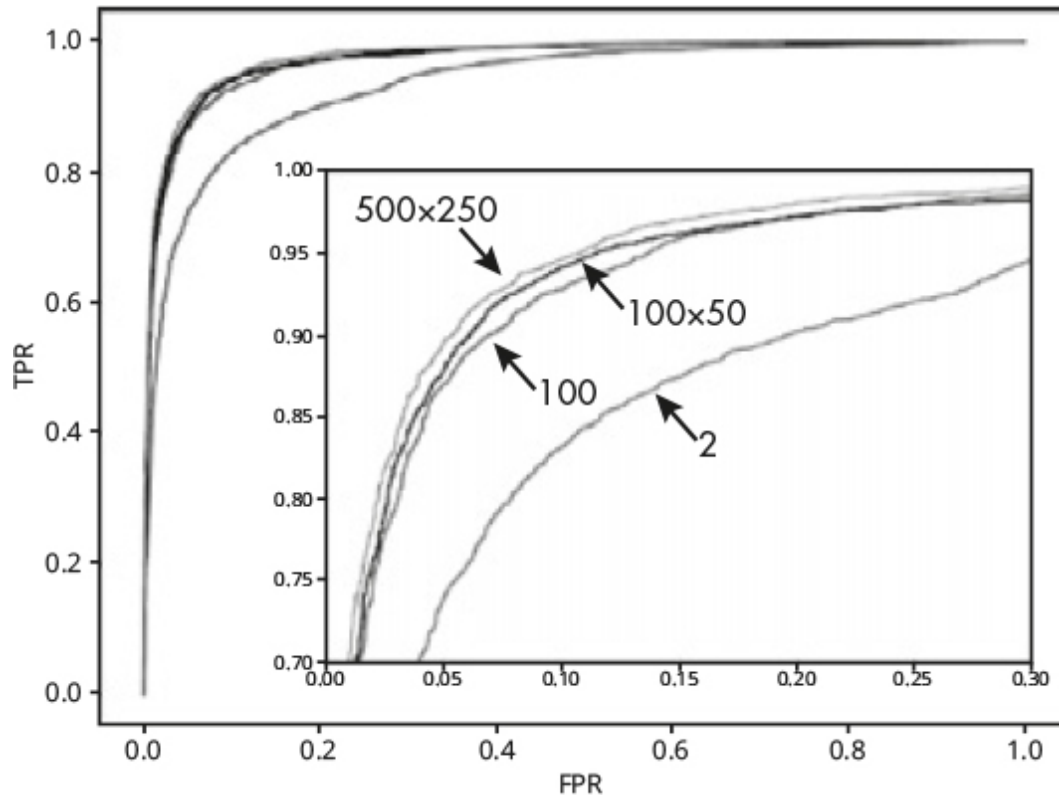


Figure 8-3: ROC curves for the MNIST even/odd models. The model hidden-layer sizes are indicated.

One ROC curve is significantly different from the other three. This is the ROC curve for the model with a single hidden layer with two nodes. All the other ROC curves are above this one. As a general rule, if one ROC curve is entirely above another, then the model that generated the curve can be considered superior. All the larger MNIST even/odd models are superior to the model with only two nodes in its hidden layer.

The other three models are quite close to one another, so how do we choose one? The decision isn't always clear-cut. Following our rule of thumb about ROC curves, we should select the two-layer model with 500 and 250 nodes, respectively, as its ROC curve is above the others.

However, we might hesitate depending on our use case. This model has over 500,000 parameters. Running it requires use of all those parameters. The 100×50 model contains slightly more than 80,000 parameters. That's less than one-fifth the number of the larger model. We might decide that processing speed considerations eclipse the small improvement in the overall performance of the larger model and select the smaller model. The ROC analysis showed us that doing so involves only a minor performance penalty.

Another factor to consider when comparing ROC curves visually is the slope of the curve when the FPR is small. A perfect model has a vertical slope since it jumps immediately from the point (0,0) to (0,1). Therefore, the better model will have an ROC curve that has a steeper slope in the low FPR region.

A commonly used metric derived from the ROC curve is the *area under the curve*. This area is usually abbreviated as *AUC* or, in medical circles, *Az*. A perfect ROC curve has an AUC of 1.0 since the curve jumps from (0,0) to (0,1) and then over to (1,1), forming a square of side 1 with an area of 1. A model that guesses randomly (the diagonal line in the ROC plot) has an AUC of 0.5, the area of the triangle formed by the dashed diagonal line. To calculate the area under an arbitrary ROC curve, one needs to perform numerical integration. Fortunately for us, sklearn knows how to do this, so we don't need to. We'll see this shortly.

People often report the AUC, but as time goes by, I'm less in favor of it. The main reason is that AUC replaces the highly informative graph with a single number, but different ROC curves can lead to the same AUC. If the AUC of two curves is the same, but one leans far to the right while the other has a steep slope in the low FPR region, we might be tempted to think the models are roughly equivalent in terms of performance. However, in reality, the model with the steeper slope is likely the one we want because it will reach a reasonable TPR without too many false positives.

Another caution when using the AUC is that it changes only a small amount for even fairly significant changes in other parameters. This makes it difficult for humans to judge well based on AUC values that are only slightly different from one another. For example, the AUC of the

MNIST even/odd model with two nodes in its hidden layer is 0.9373, while the AUC of the model with 100 nodes is 0.9722. Both are well above 0.9 out of a possible 1.0, so are they both about the same? We know that they are not, since the ROC curves clearly show the two-node model to be well below the other.

Generating an ROC Curve

We are now ready to create an ROC curve. The easy way to get the ROC curve, and AUC, is to use sklearn:

```
import os
import sys
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_auc_score, roc_curve

def main():
    labels = np.load(sys.argv[1])
    probs = np.load(sys.argv[2])
    pname = sys.argv[3]

    auc = roc_auc_score(labels, probs[:,1])
    roc = roc_curve(labels, probs[:,1])
    print("AUC = %0.6f" % auc)

    plt.plot(roc[0], roc[1], color='r')
    plt.plot([0,1],[0,1], color='k', linestyle=':')

    plt.xlabel("FPR")
    plt.ylabel("TPR")
    plt.tight_layout(pad=0, w_pad=0, h_pad=0)
    plt.savefig(pname, dpi=300)
    plt.show()
```

This routine reads a set of labels and the associated per-class probabilities, such as the output generated by the code in the previous section. It then calls the sklearn functions `roc_auc_score` and `roc_curve` to

return the AUC and the ROC points, respectively. The ROC curve is plotted, saved to disk, and displayed.

We need not use sklearn as a black box. We can generate the ROC curve points ourselves quickly enough. We load the same inputs, the labels, and the per-class probabilities, but instead of calling a library function, we loop over the threshold values of interest and calculate TP, TN, FP, and FN for each threshold. From these, we can directly calculate the FPR and TPR, which gives us the set of points we need to plot. The code to do this is straightforward:

```
def table(labels, probs, t):
    tp = tn = fp = fn = 0
    for i, l in enumerate(labels):
        c = 1 if (probs[i,1] >= t) else 0
        if (l == 0) and (c == 0):
            tn += 1
        if (l == 0) and (c == 1):
            fp += 1
        if (l == 1) and (c == 0):
            fn += 1
        if (l == 1) and (c == 1):
            tp += 1
    return [tp, tn, fp, fn]

def main():
    labels = np.load(sys.argv[1])
    probs = np.load(sys.argv[2])
    pname = sys.argv[3]

    th = [0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1]
    roc = []
    for t in th:
        tp, tn, fp, fn = table(labels, probs, t)
        tpr = tp / (tp + fn)
        fpr = fp / (tn + fp)
        roc.append([fpr, tpr])
    roc = np.array(roc)
```

```

xy = np.zeros((roc.shape[0]+2, roc.shape[1]))
xy[1:-1,:] = roc
xy[0,:] = [0,0]
xy[-1,:] = [1,1]
plt.plot(xy[:,0], xy[:,1], color='r', marker='o')
plt.plot([0,1],[0,1], color='k', linestyle=':')
plt.xlabel("FPR")
plt.ylabel("TPR")
plt.savefig(pname)
plt.show()

```

The `main` function loads the labels and probabilities. The loop over `th` applies the various threshold values, accumulating the ROC points in `roc` by calling the `table` function, which calculates the TP, TN, FP, and FN for the current threshold.

The `table` function loops over all the per-class probabilities, assigning a class label of 1 if the class 1 probability is greater than or equal to the current threshold value. This class assignment is then compared to the actual class label, and the appropriate tally counter is incremented.

Once the ROC points are calculated, the plot is made by adding the point (0,0) to the beginning of the point list, and the point (1,1) to the end of the list. Doing this ensures that the plot extends the full range of FPR values. The points are plotted and saved to disk.

EXPLORING THE PRECISION-RECALL CURVE

Before leaving this section, I should mention one other evaluation curve that you'll run across from time to time in machine learning: the *precision-recall (PR) curve*. As the name suggests, it plots the PPV (precision) and TPR (recall, sensitivity) as the decision threshold varies, just like an ROC curve. A good PR curve moves toward the upper right instead of the upper left as a good ROC curve does. We can easily generate the points of this curve in sklearn by using the `precision_recall_curve` function in the `metrics` module.

We aren't spending time with this curve because it doesn't take the true negatives into account. Consider the definition of the PPV and TPR to see that this is so. My bias against the PR curve stems from the same concern as my bias against the F1 score. By not taking the true negatives into account, the PR curve and F1 score give an incomplete picture of the quality of the classifier. The PR curve does have utility when the true-positive class is rare or when the true-negative performance is not essential. However, in general when evaluating classifier performance, I claim it is best to stick to the ROC curve and the metrics we've defined.

Handling Multiple Classes

All the metrics we've discussed so far apply only to binary classifiers. We know that many classifiers are multiclass: They output multiple labels, not just 0 or 1. To evaluate these models, we'll extend our idea of the confusion matrix to the multiclass case and see that we can also extend some of the metrics we're already familiar with as well.

We need some multiclass model results to work with. Thankfully, the MNIST data is already multiclass. Recall, we went to the trouble of recoding the labels to make the dataset binary. Here we'll train models with the same architectures, but this time we'll leave the labels as they are so that the model outputs 1 of 10 labels: the digit it assigned to the test input, the output of the `predict` method of the `MLPClassifier` class. We won't show the code as it's identical to the code in the previous section except that `predict` is called in place of `predict_proba`.

Extending the Confusion Matrix

The basis for our binary metrics was the 2×2 confusion matrix, which is readily extended to the multiclass case. To do so, we let the rows of the matrix represent the actual class labels, while the columns of the matrix represent the model's predictions. The matrix is square with as many rows and columns as there are classes in the dataset. For MNIST, then, we arrive at a 10×10 confusion matrix since there are 10 digits.

We calculate the confusion matrix from the actual known test labels and the predicted labels from the model. We could use a function in the `metrics` module of `sklearn`, `confusion_matrix`, but calculating it ourselves is straightforward enough:

```
def confusion_matrix(y_test, y_predict, n=10):
    cmat = np.zeros((n, n), dtype="uint32")
    for i, y in enumerate(y_test):
        cmat[y, y_predict[i]] += 1
    return cmat
```

Here `n` is the number of classes, fixed at 10 for MNIST. If needed, we could instead determine it from the supplied test labels.

The code is straightforward. The inputs are vectors of the actual labels (`y_test`) and the predicted labels (`y_predict`), and the confusion matrix (`cmat`) is filled in by incrementing each possible index formed from the actual label and the predicted label. For example, if the actual label is 3 and the predicted label is 8, then we add 1 to `cmat[3, 8]`.

Let's look at the confusion matrix for a model with one hidden layer of 100 nodes (Table 8-8).

Table 8-8: The 100-Node Confusion Matrix

Class	0	1	2	3	4	5	6	7	8	9
0	943	0	6	9	0	10	7	1	4	0
1	0	1,102	14	5	1	1	3	1	8	0
2	16	15	862	36	18	1	17	24	41	2
3	3	1	10	937	0	20	3	13	17	6
4	2	8	4	2	879	0	14	1	6	66
5	19	3	3	53	13	719	17	3	44	18
6	14	3	4	2	21	15	894	1	4	0
7	3	21	32	7	10	1	0	902	1	51

Class	0	1	2	3	4	5	6	7	8	9
8	17	14	11	72	11	46	21	9	749	24
9	10	11	1	13	42	5	2	31	10	884

The rows represent the actual test sample label, $[0, 9]$. The columns are the label assigned by the model. If the model is perfect, we have a one-to-one match between the actual label and the predicted label. This is the main diagonal of the confusion matrix. Therefore, a perfect model has entries along the main diagonal, and all other elements are 0. Table 8-8 is not perfect, but the largest counts are along the main diagonal.

The place where row 4 and column 4 meet has the value 879, which means that there were 879 times when the actual class was 4 and the model correctly predicted 4 as the label. If we look along row 4, we see other numbers that are not zero. Each of these represents a case where an actual 4 was called another digit by the model. For example, there were 66 times when a 4 was called a 9 but only one case of a 4 being labeled a 7.

Column 4 represents the cases when the model called the input a 4. As we saw, the model was correct 879 times. However, the model accidentally labeled other digits as 4, like the 21 times a 6 was called a 4 or the one time a 1 was mistaken for a 4. There were no cases of a 3 being labeled a 4.

The confusion matrix tells us at a glance how well the model is doing on the test set. We can quickly see if the matrix is primarily diagonal. If it is, the model is doing a good job on the test set. If not, we need to take a closer look to see what classes are being confused with other classes. A simple adjustment to the matrix can help. Instead of the raw counts, which require us to remember how many examples of each class are in the test set, we can divide the values of each row by the sum of the row. Doing so converts the entries from counts to fractions. We can then multiply the entries by 100 to convert to percents. This transforms the confusion matrix into what we'll call an *accuracy matrix*. The conversion is straightforward:

```
acc = 100.0*(cmat / cmat.sum(axis=1))
```

Here `cmat` is the confusion matrix. This produces an accuracy matrix: Table 8-9.

Table 8-9: A Confusion Matrix Presented as Per-Class Accuracies

Class	0	1	2	3	4	5	6	7	8	9
0	96.2	0.0	0.6	0.9	0.0	1.1	0.7	0.1	0.4	0.0
1	0.0	97.1	1.4	0.5	0.1	0.1	0.3	0.1	0.8	0.0
2	1.6	1.3	83.5	3.6	1.8	0.1	1.8	2.3	4.2	0.2
3	0.3	0.1	1.0	92.8	0.0	2.2	0.3	1.3	1.7	0.6
4	0.2	0.7	0.4	0.2	89.5	0.0	1.5	0.1	0.6	6.5
5	1.9	0.3	0.3	5.2	1.3	80.6	1.8	0.3	4.5	1.8
6	1.4	0.3	0.4	0.2	2.1	1.7	93.3	0.1	0.4	0.0
7	0.3	1.9	3.1	0.7	1.0	0.1	0.0	87.7	0.1	5.1
8	1.7	1.2	1.1	7.1	1.1	5.2	2.2	0.9	76.9	2.4
9	1.0	1.0	0.1	1.3	4.3	0.6	0.2	3.0	1.0	87.6

The diagonal shows the per-class accuracies. The worst-performing class is 8 with an accuracy of 76.9 percent, and the best-performing class is 1 with an accuracy of 97.1 percent. The non-diagonal elements are the percentage of the actual class labeled as a different class by the model. For class 0, the model gave a class 5 label to a true zero 1.1 percent of the time. The row percentages sum to 100 percent (within rounding error).

Why did class 8 do so poorly? Looking across the row for class 8, we see that the model mistook 7.1 percent of the actual 8 instances for a 3 and 5.2 percent of the instances for a 5. Confusing an 8 with a 3 was the biggest single mistake the model made, though 6.5 percent of 4 instances were labeled 9 as well. A moment's reflection makes sense of

the errors. How often do people confuse 8 and 3 or 4 and 9? This model is making errors similar to those humans make.

The confusion matrix can reveal pathological performance as well. Consider the MNIST model in Figure 8-3, with a single hidden layer of only two nodes. Table 8-10 shows the accuracy matrix it produces.

Table 8-10: The Two-Node Accuracy Matrix

Class	0	1	2	3	4	5	6	7	8	9
0	51.0	1.0	10.3	0.7	1.8	0.0	34.1	0.7	0.0	0.4
1	0.4	88.3	0.4	1.1	0.8	0.0	0.0	9.3	1.0	0.0
2	8.6	2.8	75.2	6.9	1.7	0.0	1.4	3.0	0.3	0.6
3	0.2	1.0	4.9	79.4	0.3	0.0	0.0	13.5	0.0	0.2
4	28.4	31.3	7.3	2.1	9.7	0.0	0.3	13.6	1.0	0.5
5	11.4	42.5	2.2	4.9	4.4	0.0	0.1	16.5	0.9	0.3
6	35.4	1.0	5.4	0.2	1.4	0.0	55.0	0.0	0.0	0.1
7	0.4	5.2	2.0	66.2	0.8	0.0	0.0	25.5	0.2	0.3
8	10.5	41.9	2.8	8.0	4.1	0.0	0.1	22.1	1.3	0.4
9	4.7	9.1	5.8	26.2	5.8	0.0	0.2	41.2	2.2	3.1

We can immediately see that this is an inferior model. Column 5 is entirely zero, meaning the model never outputs 5 for any input. Much the same is true for output labels 8 and 9. On the other hand, the model likes to call inputs 0, 1, 2, or 3, as those columns are densely populated for all manner of input digits. Looking at the diagonal, we see that only 1 and 3 stand a reasonable chance of being correctly identified, though many of these will be called 7. Class 8 is rarely labeled correctly (1.3 percent). A poorly performing model will have a confusion matrix like this, with oddball outputs and large off-diagonal values.

Calculating Weighted Accuracy

The diagonal elements of an accuracy matrix tell us the per-class accuracies for the model. We can calculate an overall accuracy by averaging these values. However, this could be misleading if one or more classes is far more prevalent in the test data than the others.

Instead of a simple average, we should use a weighted average. The weights are based on the total number of test samples from each class divided by the total number of test samples presented to the model. Say we have three classes, and their frequencies and per-class accuracies in our test set are as in Table 8-11.

Table 8-11: Hypothetical Accuracies for a Model with Three Classes

Class	Frequency	Accuracy
0	4,004	88.1
1	6,502	76.6
2	8,080	65.2

Here we have $N = 4,004 + 6,502 + 8,080 = 18,586$ test samples. Table 8-12 shows the per-class weights.

Table 8-12: Example Weights per Class

Class	Weight
0	$4,004 / 18,586 = 0.2154$
1	$6,502 / 18,586 = 0.3498$
2	$8,080 / 18,586 = 0.4347$

We can calculate the average accuracy as follows:

$$\text{ACC} = 0.2154 \times 88.1 + 0.3498 \times 76.6 + 0.4347 \times 65.2 = 74.1$$

Philosophically, we should replace the weights with the actual per-class prior probabilities, if we know them. These probabilities are the true likelihood of the class appearing in the wild. However, if we assume that the test set is fairly constructed, we're likely safe using only the per-class frequencies. We claim that a properly built test set will represent the true prior class probabilities reasonably well.

In code, we can succinctly calculate the weighted mean accuracy from the confusion matrix:

```
def weighted_mean_acc(cmat):  
    N = cmatrix.sum()  
    C = cmatrix.sum(axis=1)  
    return ((C/N)*(100*np.diag(cmat)/C)).sum()
```

N is the total number of samples that were tested, which is just the sum of the entries in the confusion matrix since every sample in the test set falls somewhere in the matrix. c is a vector of the number of samples per class—that is, the sum of the rows of the confusion matrix. We calculate the perclass accuracy, as a percentage, from the diagonal elements of the confusion matrix (`np.diag(cmat)`) divided by the number of times each class shows up in the test set, c . Multiply this by 100 to make these percent accuracies.

If we summed these per class and divided by the number of classes, we'd have the (potentially misleading) unweighted mean accuracy. Instead, we first multiply by c/N , the fraction of all test samples that were of each class (recall, c is a vector), and then sum to get the weighted accuracy. This code works for any size confusion matrix.

For the MNIST models of the previous section, we calculate weighted mean accuracies to be those in Table 8-13.

Table 8-13: Weighted Mean Accuracies for the MNIST Models

Architecture	Weighted mean accuracy
2	40.08%

Architecture	Weighted mean accuracy
100	88.71%
100×50	88.94%
500×250	89.63%

These accuracies show the sort of diminishing returns we've seen previously as the model size increases. The single hidden layer of 100 nodes is virtually identical to the two-hidden-layer model with 100 and 50 nodes, and only 1 percent worse than the much larger model with 500 nodes and 250 nodes in its hidden layers. The model with only two nodes in the hidden layer performs poorly. Since there are 10 classes, random guessing tends to have an accuracy of $1/10 = 0.1 = 10$ percent, so even this very strange model that maps 784 input values (28×28 pixels) to only 2 and then to 10 output nodes is still 4 times more accurate than random guessing. However, this is misleading on its own because, as we just saw in Table 8-10, the confusion matrix for this model is quite strange. We certainly wouldn't want to use this model. Nothing beats careful consideration of the confusion matrix.

Considering the Multiclass Matthews Correlation Coefficient

The 2×2 confusion matrix led to many possible metrics. While it's possible to extend several of those metrics to the multiclass case, we'll consider only the main metric here: the MCC. For the binary case, we saw that the MCC was as follows:

$$\text{MCC} = \frac{\text{TP} \times \text{TN} - \text{FP} \times \text{FN}}{\sqrt{(\text{TP} + \text{FP})(\text{TP} + \text{FN})(\text{TN} + \text{FP})(\text{TN} + \text{FN})}}$$

We can extend this to the multiclass case by using terms from the confusion matrix like so

$$\text{MCC} = \frac{c \times s - \sum_k^K p_k \times t_k}{\sqrt{\left(s^2 - \sum_k^K p_k^2\right) \times \left(s^2 - \sum_k^K t_k^2\right)}}$$

where:

$$t_k = \sum_i^K C_{ik}, \quad p_k = \sum_i^K C_{ki}, \quad c = \sum_k^K C_{kk}, \quad s = \sum_i^K \sum_j^K C_{ij}$$

Here K is the number of classes, and C is the confusion matrix. This notation is from the sklearn website's description of the MCC, giving us a direct view of how it's implemented. We don't need to follow the equations in detail; we need to know only that the MCC is built from the confusion matrix in the multiclass case as in the binary case.

Intuitively, this makes sense. The binary MCC is a value in the range $[-1, +1]$. The multiclass case changes the lower bound based on the number of classes, but the upper bound remains 1.0, so the closer the MCC is to 1.0, the better the model is doing.

Calculating the MCC for the MNIST models, as we did for the weighted mean accuracy, gives Table 8-14.

Table 8-14: MCC for the MNIST Models

Architecture	MCC
2	0.3440
100	0.8747
100×50	0.8773
500×250	0.8849

Again, this shows us that the smallest model is inferior, while the other three models are all quite similar in terms of performance. The time to make predictions on the 10,000 test samples, however, varies quite a bit by model. The single-hidden-layer model with 100 nodes takes 0.052 seconds, while the largest model needs 0.283 seconds, over five times longer. If speed is essential, the smaller model might be preferable.

Many factors come into play when deciding which model to use. The metrics discussed in this chapter are guides, but you should follow them

cautiously. In the end, only you know what makes sense for the problem you're trying to solve.

Summary

In this chapter, you learned why accuracy is not a sufficient measure of a model's performance. You learned how to generate the 2×2 confusion matrix for a binary classifier, and what this matrix tells us about the model's performance on the held-out test set. We derived basic metrics from the 2×2 confusion matrix and used them to derive more-advanced metrics. We discussed the utility of the various metrics to build our intuition as to how and when to use them.

We then learned about the receiver operating characteristics (ROC) curve, including what it illustrates about the model and how to interpret it to compare models against one another. Finally, we introduced the multiclass confusion matrix, giving examples of how to interpret it and how to extend some of the binary classifier metrics to the multiclass case.

In the next chapter, we'll reach the pinnacle of our machine learning models: convolutional neural networks (CNNs). The next chapter introduces the basic ideas behind the CNN; later chapters will conduct many experiments using this deep learning architecture.

PART IV

CONVOLUTIONAL NEURAL NETWORKS

Deep learning began with convolutional neural networks, the subject of this part of the book. Convolutional neural networks process inputs by seeking structure within the input as opposed to the holistic view enforced by traditional neural networks. The shift enabled explosive progress in deep learning beginning in 2012, though convolutional networks first appeared in the late 1990s. Convolutional networks form the backbone of many AI systems, especially those that process images. The experiments in these chapters prepare us for more advanced techniques.

OceanofPDF.com

9

INTRODUCTION TO CONVOLUTIONAL NEURAL NETWORKS



In this chapter, we'll introduce a potent approach to dealing with multidimensional information. In particular, we'll work through the theory and high-level operation of CNNs, a cornerstone of modern deep learning.

We'll begin by presenting the motivations behind the development of CNNs. Convolutions are the heart of CNNs, so they'll come next. We'll discuss them in some detail, in particular how they're used by the CNN. We'll then introduce a basic CNN and work through its anatomy. We'll use this basic CNN architecture for the remainder of the chapter.

After we dissect a CNN, we'll work through how convolutional layers work. Then come pooling layers. We'll see what they do, what benefit they offer, and what price they exact in return. To round out our discussion of the fundamental components of a CNN, we'll present the fully connected layers, which, in reality, are just the layers of a traditional, fully connected, feed-forward neural network like those of Chapter 5.

One topic will be conspicuously absent from this chapter: the mechanics of training a CNN. In part, we'll gloss over training because it's messy once convolutional layers are introduced, but primarily because we've already discussed backpropagation in Chapter 6, and we use the same algorithm to train a CNN. We calculate the weights and biases of all layers

from the average loss over the training minibatch and use backprop to determine the derivatives we need in order to update the weights and biases for each step of stochastic gradient descent.

Why Convolutional Neural Networks?

CNNs have several advantages over traditional neural networks. First, the convolutional layers of a CNN require vastly fewer parameters than fully connected neural networks, as you'll see later in the chapter. CNNs require fewer parameters because the convolution operation applies parameters in each layer to small subsets of the input instead of the entire input at once, as is done with a traditional neural network.

Second, CNNs introduce the idea of *spatial invariance*, the ability to detect a spatial relationship in the input regardless of where it appears. For example, if the input to a neural network is an image of a cat, a traditional neural network will take the image in as a single feature vector, meaning that if a cat appears in the upper-left corner of the image, the network will learn that cats can appear in the upper-left corners of the image but not that they can also appear in the lower-right corners (unless the training data contains examples with cats in the lower-right corners). For a CNN, however, the convolution operation can detect cats anywhere they appear.

While CNNs are usually used with 2D inputs, they can also be used with 1D inputs, like the feature vectors we have worked with up to now. However, those feature vectors, like the iris measurements, don't reflect any sort of spatial relationship as the parts of an image of a cat do. There's nothing there for the convolution operation to take advantage of. This doesn't mean that a CNN won't work but that it might not be the best sort of model to use. As always, we need to understand how various model types operate so we select the best model for the task at hand.

NOTE

Depending on who you ask, CNNs were developed either in 1980 by Kunihiko Fukushima to implement the neocognitron model or in 1998 by Y. LeCun et al. as presented in "Gradient-Based Learning Applied to Document Recognition," which as of this writing has been referenced over 72,000 times. My take is that both deserve credit, though LeCun used the

phrase convolutional neural network, or convnet as they are still sometimes called, and the description in the paper is what we will work with in this book. The neocognitron reflected some of the ideas in a CNN but not CNNs themselves.

Convolution

Convolution involves sliding one thing over another. For us, this means sliding a *kernel*, a small 2D array, over the input, which might be the input image to the CNN or the output of a lower convolutional layer.

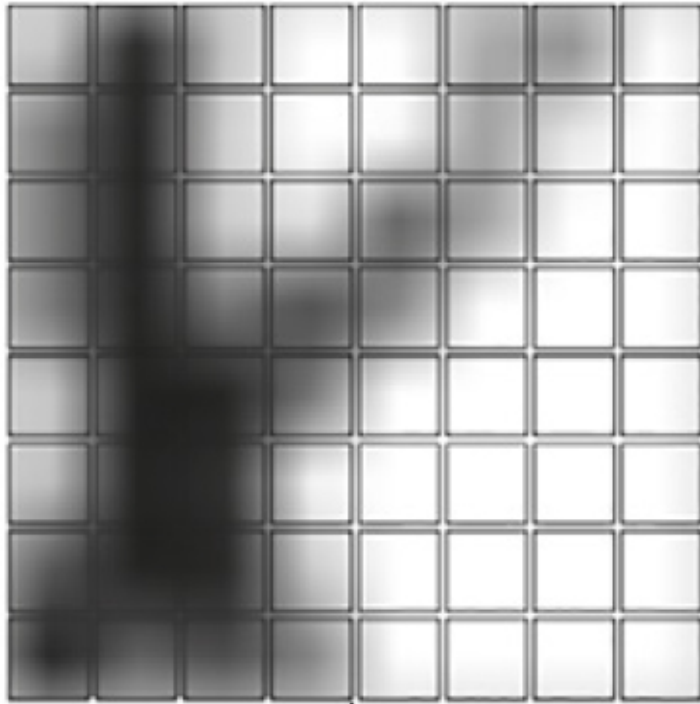
Convolution has a formal mathematical definition, but it won't help us right now. Luckily, all our inputs are discrete, which means we can get away with a bit of hand-waving. For simplicity, we'll focus on only the 2D case.

Scanning with the Kernel

The kernel is the thing we are asking the convolutional layer to learn during training. It's a collection of small 2D arrays that we move over the input. Ultimately, the kernels become the weights of a convolutional layer in a CNN.

Convolution takes a small section of the input that's the same size as the kernel, covers it with the kernel, performs an operation on that set of numbers to produce a single output number, and then repeats the process after moving the kernel to a new position in the input. The *stride* is how far the kernel is moved. Typically, the stride is 1, meaning the kernel slides over one element of the input.

Figure 9-1 shows the effect of convolution on part of an MNIST digit image.



60	248	67	0	0	93	121	0
145	253	54	0	0	105	26	0
145	253	54	33	170	102	0	0
145	254	150	195	130	0	0	0
61	253	253	149	0	0	0	0
68	253	237	11	0	0	0	0
185	253	249	57	0	0	0	0
253	175	201	136	0	0	0	0

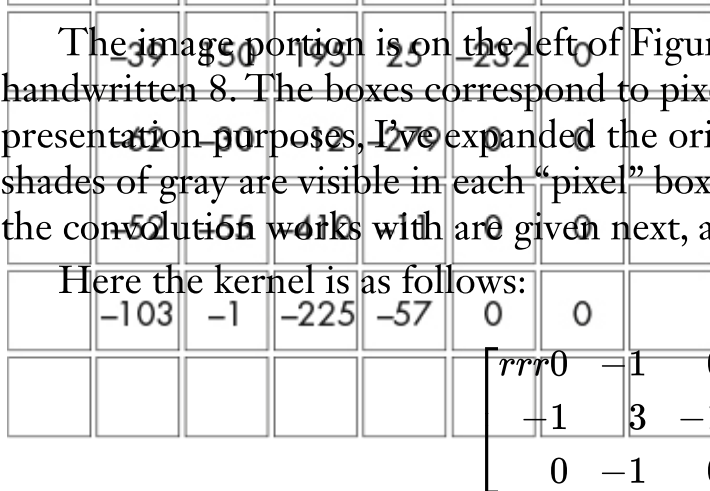
×

0	-1	0
-1	3	-1
0	-1	0

=

	59	-212	-93	-257	88	-148	
	52	228	220	220	21	128	

Figure 9-1: Convolving a kernel with an image



The image portion is on the left of Figure 9-1, which shows part of a handwritten 8. The boxes correspond to pixel intensities, though for presentation purposes, I've expanded the original image so that many shades of gray are visible in each "pixel" box. The actual pixel values that the convolution works with are given next, after the arrow.

Here the kernel is as follows:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

This is the set of numbers we'll slide over the input pixels. This is a 3x3 matrix, so we need to cover a 3x3 region of the input image. The first 3x3 region, the upper-left corner, is shown here:

$$\begin{bmatrix} 60 & 248 & 67 \\ 145 & 253 & 54 \\ 145 & 253 & 54 \end{bmatrix}$$

Convolution performs a straightforward operation with the kernel and the covered region as the input: multiplying corresponding entries and summing them. Finding the first output value of the convolution begins with the following:

$$\begin{bmatrix} 60 & 248 & 67 \\ 145 & 253 & 54 \\ 145 & 253 & 54 \end{bmatrix} \times \begin{bmatrix} 0 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -248 & 0 \\ -145 & 759 & -54 \\ 0 & -253 & 0 \end{bmatrix}$$

When the preceding elements are summed, this gives the output value as follows:

$$0 + (-248) + 0 + (-145) + 759 + (-54) + 0 + (-253) + 0 = 59$$

The output of the first convolution operation is 59, but what do we do with that number? The kernel is 3x3, an odd number along each side. This means that there is a middle element, the one with the 3 in it. The place

where the middle number is in the output array gets replaced with the output value, 59. Figure 9-1 shows the full output of the convolution. Sure enough, the first element of the output is 59, located at the center of the kernel when the kernel is covering the upper-left corner.

The remaining output values are calculated in precisely the same way but by moving the kernel over 1 pixel each time. When the end of a row is reached, the kernel moves back to the left side but down 1 pixel. In this way, it slides over the entire input image to produce the output in Figure 9-1, just like the scan lines of an old analog television.

The next output value is shown here:

$$\begin{bmatrix} 248 & 67 & 0 \\ 253 & 54 & 0 \\ 253 & 54 & 33 \end{bmatrix} \times \begin{bmatrix} 0 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -67 & 0 \\ -253 & 162 & 0 \\ 0 & -54 & 0 \end{bmatrix}$$

This sums to -212, as shown on the right side of Figure 9-1.

Repeating the convolution operation produces the output in Figure 9-1. Notice the empty boxes around the output. These values are empty because the middle of the 3×3 kernel doesn't cover the edge of the input array. Therefore, the output matrix of numbers is two smaller in each dimension than the input. If the kernel were 5×5, there would be a border 2 pixels wide instead of 1.

Implementations of 2D convolution need to make a decision about these border pixels. There are options, and most toolkits support several of them. One is to simply ignore these pixels and make the output smaller than the input, as in Figure 9-1. We often refer to this approach as *exact* or *valid* because we retain only values that are actually output by the operation.

Another approach is to imagine that a border of 0 values surrounds the input image. The border is as thick as is needed so that the kernel fits with its middle value matching the upper-left pixel of the input. For the example in Figure 9-1, this means a border of 1 pixel because the kernel is 3×3 and one element is on either side of the kernel's center value. If the kernel were 5×5, the border would be 2 pixels since there are two values on either side of the kernel center. This is *zero-padding* and gives an output that is the same size as the input. Instead of convolving a 28×28-pixel MNIST digit

image with a 3×3 kernel and getting a 26×26-pixel output as in Figure 9-1, we get an output that is also 28×28 pixels.

If we zero-pad the example image in Figure 9-1, we can fill in the first empty output square like so:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 60 & 248 \\ 0 & 145 & 253 \end{bmatrix} \times \begin{bmatrix} 0 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 180 & -248 \\ 0 & -145 & 0 \end{bmatrix}$$

This sums to −213. This means that we could replace the upper-left corner of the output matrix in Figure 9-1, which currently has an empty box, by −213. Similarly, the rest of the empty boxes would have values, and the output of the convolution operation would be 28×28 pixels.

Using Convolution for Image Processing

When used in a neural network, convolution is sometimes viewed as magical, a special operation that lets CNNs do the wonderful things that they can do. This is more or less true, but the convolution operation is certainly not new. Even if we ignore mathematics entirely and think only of the discrete convolution of 2D images, image scientists were using convolution for image processing decades before convolution was applied to machine learning.

The convolution operation allows for all manner of image processing. For example, consider the images of the moon in Figure 9-2.

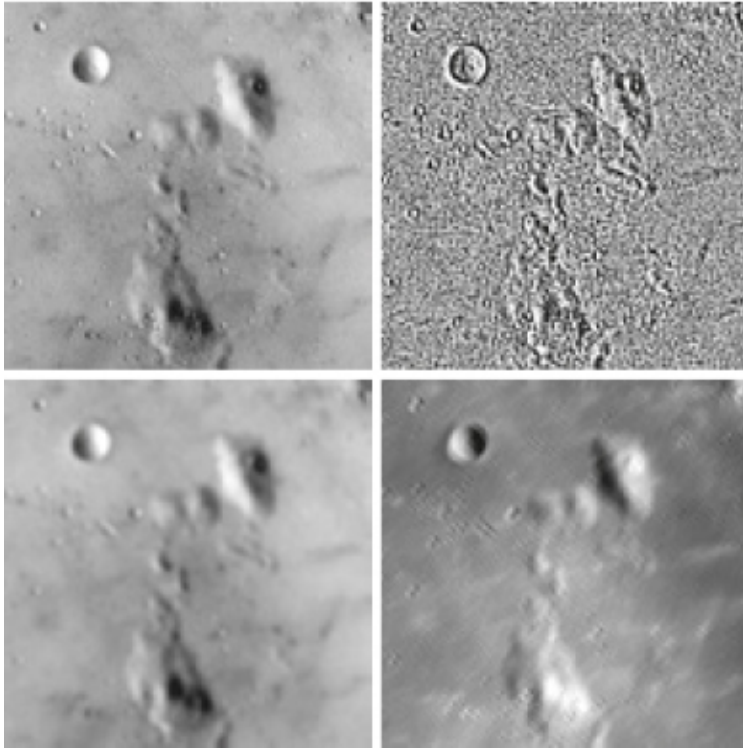


Figure 9-2: Applying 5×5 convolution kernels to an image

The original moon image is on the upper left, while the other three are the output from convolving the moon image with different 5×5 kernels. Moving clockwise from the upper right, the kernels either emphasize edges, emphasize diagonal structures (upper left to lower right), or blur the input image. All this is accomplished by changing the values in the kernel, but the convolution operation remains the same.

From a machine learning perspective, the power of a convolutional approach comes partially from the savings in terms of parameters. If a model can learn a set of kernels, that is a smaller set of numbers to learn than the weights for a fully connected model.

This is a good thing on its own. The fact that a convolution can pull out other information about an image—such as its slowly changing components (the blur of Figure 9-2), its rapidly changing components (the edges of Figure 9-2), or even components along a specific direction (the diagonals of Figure 9-2)—means that the model gains insight as to what is in the input. And, since we move the kernel over the image, we're not dependent upon *where* in the image these structures occur.

Anatomy of a Convolutional Neural Network

Medical students learn about anatomy by dissecting a cadaver to see the parts and how they relate to one another. In similar, though less challenging fashion, we'll start with the body of a CNN, an illustration of its basic architecture, and then pull it apart to learn what each component is and what it does.

Figure 9-3 shows us our body. This is the default example CNN that's used by the Keras toolkit to train a model classifying MNIST digits. We'll use it as our standard for the remainder of this chapter.

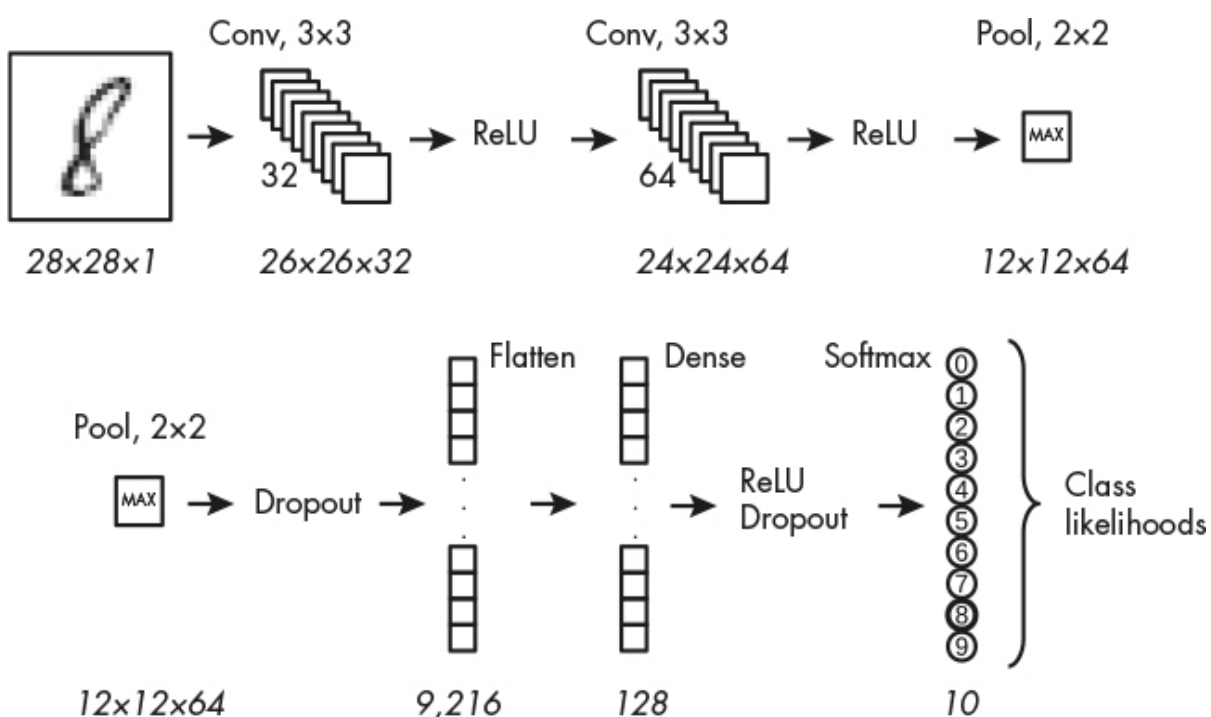


Figure 9-3: The architecture of a basic convolutional neural network

How do we interpret this figure? Like a traditional neural network, a CNN has an input and an output. In this case, the input is the digit image on the upper left. The network then flows left to right, following the arrows. At the end of the top row, the network continues on the following row. Note, I've duplicated the layer at the end of the top row and placed it at the beginning of the next row for presentation purposes.

The flow continues along the bottom row, again left to right, until the output is reached. The output here is a softmax layer to give us the

likelihoods of each of the possible digits, just as we saw for the traditional neural networks of Chapter 7.

Exploring the Types of Layers

Between each arrow in Figure 9-3 is a layer of the network. Unlike a traditional neural network, a CNN has many kinds of layers. I list them here before discussing each in turn:

- Convolutional (Conv)
- ReLU
- Pooling (Pool)
- Dropout
- Flatten
- Dense

Note that we're using the Keras names for the layers. For instance, Keras uses `Dense` for what many other toolkits call *fully connected* or even *InnerProduct* layers.

Several of these layers should be familiar, such as the ReLU layer, which implements a rectified linear unit that takes each of its inputs and asks whether it's greater than or less than 0 (see Chapter 5). If the input is less than 0, the output is 0; otherwise, the output is the input. We can express this mathematically as the following, where the max function returns the largest of its two arguments:

$$\text{ReLU}(x) = \max(0, x)$$

Likewise, in Chapter 6 you learned that dropout selects a percentage of its outputs at random during training and sets them to 0. This provides a powerful form of regularization to help the network learn meaningful representations of the input data. Our basic CNN has two dropout layers. The first uses a probability of 25 percent, meaning during any minibatch pass while training, some 25 percent of the outputs will be set to 0. The second dropout layer uses a probability of 50 percent.

We know the flatten and dense layers by another name and not as independent entities. Our traditional feed-forward neural network uses fully connected layers to process a 1D vector. Here the flatten and dense layers work together to implement a fully connected layer. The flatten layer takes its input—usually a 4D array (you’ll see why later)—and turns it into a vector. It does something similar to what we did to construct the vector form of the MNIST dataset, where we put the pixels of each row end to end to unravel the 2D image. The dense layer implements a traditional neural network layer, where each input value is mapped to each node of the dense layer. Typically, the output of the dense layer is passed to another dense layer or a softmax layer to let the network make predictions.

Internally, many layers of a CNN expect 4D arrays as inputs and produce 4D arrays as outputs. The first dimension is the number of inputs in the minibatch. So, if we have a minibatch of 24, the first dimension of the 4D array will be 24.

The second and third dimensions are the height and width. If the input to a layer is the input to the model (say, an image), these dimensions are the actual height and width dimensions of the image. If the input is really the output of another layer, say, a (yet to be described) convolutional layer, the height and width refer to the output from applying a convolutional kernel to some input. For example, the output in Figure 9-1 has a height and width of 26.

The last dimension is the number of channels (if an input image) or the number of *feature maps* (if the output of a convolutional or pooling layer). The number of channels in an image is simply the number of bands; a grayscale image has a single band, and a color image typically has three bands, one each for red, green, and blue. Some color images also have an alpha channel used to specify the transparency of a pixel, but these are typically dropped before passing the image through a CNN.

The output in Figure 9-1 is called a *feature map* because it’s the response from convolving a kernel over an input. As Figure 9-2 shows, convolving a kernel over an image can pull out features in the image, so the outputs of the kernels used by a convolutional layer are feature maps. This leaves two new layers to investigate: convolutional and pooling.

In our basic CNN, the convolutions operate on sets of 2D inputs. By *sets*, I mean stacks of 2D arrays, and the third dimension is the number of

channels or feature maps. This means that unlike every other model we've looked at in this book, the input here really is the full image, not a vector created from the image. In terms of CNNs, however, the convolutions need not operate on only 2D inputs; 3D convolutions exist, as do 1D, though both are seldom used compared to 2D convolutions.

A *pooling* layer is used to reduce the spatial dimension of its input by combining input values according to a certain rule. The most common rule is *max*: The largest value in the small block moved over the input is kept, and the other values are discarded. We'll cover pooling layers at length in this chapter.

Many other layer types can be used by modern networks, and many of these are directly supported in Keras already, though it's possible to add your own layers. This flexibility is one reason Keras often quickly supports new deep learning developments. As with a traditional neural network, for a layer to have weights that can be learned, the layer needs to be differentiable in a mathematical sense so that the chain rule can continue, and the partial derivatives can be calculated to learn how to adjust the weights during gradient descent (see "Backpropagation" on page 157).

Passing Data Through the CNN

Let's look again at Figure 9-3. A lot is happening here beyond just the order and names of the layers. Many layers have numbers in italics running along the bottom, which represent the dimensions of the output of the layer: the height, width, and number of feature maps. If the layer has only a single number, it outputs a vector with that many elements.

The input to the CNN is a $28 \times 28 \times 1$ image. The output of a convolutional layer is a set of feature maps. Thus the output of the first convolutional layer is $26 \times 26 \times 32$, meaning there are 32 feature maps, each a 26×26 image calculated from the single $28 \times 28 \times 1$ input image. Similarly, the output of the second convolutional layer is $24 \times 24 \times 64$, a set of 64 feature maps derived from the $26 \times 26 \times 32$ input, which was itself the output of the first convolutional layer.

We see that the pooling layer at the end of the first row takes its $24 \times 24 \times 64$ input and reduces it to $12 \times 12 \times 64$. The max label tells us what the pooling is doing; it takes a 2×2 region of the input and returns the largest value. Since the input is 2×2 and it returns only one value, this reduces each

24×24 input to a 12×12 output. This process is applied to each feature map so that the output is 12×12×64.

The bottom row of Figure 9-3 shows us that the `Flatten` layer takes the 12×12×64 output of the pooling layer and turns it into a vector of 9,216 elements. Why 9,216? Because $12 \times 12 \times 64 = 9,216$. Next, the `Dense` layer has 128 nodes, and, finally, our output softmax has 10 nodes because there are 10 classes, the digits 0 through 9.

The `ReLU` and `Dropout` layers have no numbers below them in the figure, meaning these layers don't alter the shape of their inputs. They simply perform an operation on each of the elements regardless of the shape.

The convolutional layers of our basic CNN have other numbers associated with them: 3×3 and 32 or 64. The 3×3 tells us the size of the convolutional kernel, and the 32 or 64 tells us the number of feature maps.

The 2×2 part of the pooling layer represents the size of the pooling kernel, which, much like a convolutional kernel, slides over the input, feature map by feature map (or channel by channel), to reduce the size of the input. Working with a 2×2 pooling kernel implies that, typically, the output will be one-half the size of the input in each of the row and column dimensions.

Figure 9-3 has familiar parts, but the presentation is new, and we have these mysterious new convolutional and pooling layers to think about, so you might be somewhat nebulous in your understanding right now. That is perfectly fine. We have new ideas and some visual indications of how they link together to make a CNN. For now, this is all you need. The remainder of this chapter will, I hope, be a series of “aha!” moments for you as you think back to this figure. When you understand what each layer does, you'll start to see why they are where they are in the processing chain, leading from image input to output softmax predictions.

Convolutional Layers

If our discussion of convolution ended with the preceding sections, you'd understand the essential operation but still be in the dark about exactly *how* a convolutional layer in a CNN works. Bearing this in mind, let's look at how the convolution idea generalizes across the inputs and outputs of a CNN's convolutional layer.

How They Work

We can think of both the input and output of a convolutional layer as stacks of 2D arrays (or matrices). The operation of the convolutional layer is best illustrated with a simple example showing how to map the input stack of arrays to the output stack.

Before I present the example, I need to introduce some terminology. I previously described the convolution operation in terms of applying a kernel to an input, both of which are 2D. We'll continue to use the term *kernel* for this single 2D matrix. When implementing a convolutional layer, however, you'll soon see that we need stacks of kernels, which are typically referred to in machine learning as *filters*. The filter, via its kernels, is applied over the input stack to produce the output stack. During training, the model is learning kernels, so it is fair to say that the model is also learning filters.

For our example, the input is a stack of two 5×5 arrays, the kernel size is 3×3 , and we want an output stack that is three deep. We choose three because, as the designer of the CNN architecture, we believe that learning three outputs will help the network learn the task at hand. The convolution operation determines the width and height of each output array; we select the depth. We'll use valid convolution, losing a border of thickness one on the output, meaning our input will drop two in width and height. Therefore, a 5×5 input convolved with a 3×3 kernel will create a 3×3 output.

That accounts for the change in dimension, but how do we go from a stack of two arrays to a stack of three? The key to mapping the $5 \times 5 \times 2$ input to the desired $3 \times 3 \times 3$ output is the set of kernels (the filter) learned during training. Let's see how the filter gives us the mapping we want.

We'll assume we already know the filters at this point, each of which is a $3 \times 3 \times 2$ stack of kernels. In general, if there are M arrays in the input stack and we want N arrays in the output stack using a kernel that is $K \times K$, then we need a set of N filters, each of which is a stack of $K \times K$ kernels M deep.

Let's explore why. If we break up the stack so we can see each element clearly, our input stack looks like this:

$$\begin{array}{l}
0 : \begin{bmatrix} -1 & 2 & 2 & -2 & -2 \\ -1 & 0 & 2 & 1 & -2 \\ -1 & 2 & 2 & 1 & -2 \\ -1 & 2 & 2 & 2 & 2 \\ -1 & -1 & 1 & 0 & -1 \end{bmatrix} \\
1 : \begin{bmatrix} 2 & -2 & -1 & -2 & -2 \\ 1 & 1 & 2 & 1 & -2 \\ 2 & 2 & -1 & -1 & 0 \\ -1 & -1 & 2 & -2 & 2 \\ -1 & -2 & 0 & -2 & 0 \end{bmatrix}
\end{array}$$

We have two 5×5 matrices labeled 0 and 1. The values were selected at random.

To get an output stack of three, we need a set of three filters. The stack of kernels in each filter is two deep, to mirror the number of arrays in the input stack. The kernels themselves are 3×3 , so we have three $3 \times 3 \times 2$ filters, and we convolve each kernel in the filter with the corresponding input array. The three filters are as follows; I've added 0 and 1 labels to show which kernels are applied to which input stack arrays:

$$\begin{array}{l}
\begin{array}{ccc}
k_0 & k_1 & k_2
\end{array} \\
0 : \begin{bmatrix} 1 & -1 & 1 \\ 1 & -1 & 0 \\ -1 & -1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & -1 & 0 \\ -1 & 1 & -1 \\ -1 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix} \\
1 : \begin{bmatrix} -1 & 0 & 0 \\ 1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} -1 & 1 & 1 \\ 1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}
\end{array}$$

We also have a bias vector, as we did for the traditional neural network layers. This is a vector, one value for each kernel stack, that we add in at the end to help align the output of the convolutional layer to the data, just as we did for the traditional neural network layers. The bias adds one more degree of freedom to the layer—one more thing that can be learned to help the layer learn the most it can from the data. For our example, the bias vector is $b = \{1, 0, 2\}$, selected at random.

To get the output stack, we convolve each kernel of each filter with the corresponding input array, sum the elements of the resulting output, and add the bias value. For filter k_0 , we convolve the first input array with the first kernel to get the following:

$$\begin{bmatrix} -1 & 2 & 2 & -2 & -2 \\ -1 & 0 & 2 & 0 & 2 \\ 1 & 2 & 2 & 1 & -2 \\ -1 & 2 & 2 & 2 & 2 \\ 1 & -1 & 1 & 0 & -1 \end{bmatrix} * \begin{bmatrix} 1 & -1 & 1 \\ 1 & -1 & 0 \\ -1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} -3 & -7 & -1 \\ 1 & -4 & 3 \\ -1 & -1 & -3 \end{bmatrix}$$

We're using $*$ to mean the full convolution operation, which is fairly standard. We repeat this operation for the second kernel in k_0 , applying it to the second array of the input:

$$\begin{bmatrix} 2 & -2 & -1 & -2 & -2 \\ 1 & 1 & 2 & 1 & -2 \\ 2 & 2 & -1 & -1 & 0 \\ -1 & -1 & 2 & -2 & 2 \\ -1 & -2 & 0 & -2 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 0 \\ 1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -3 & -3 & 6 \\ 6 & -1 & -2 \\ -6 & -1 & -1 \end{bmatrix}$$

Finally, we sum the two convolution outputs and add in the bias value:

$$\begin{bmatrix} -3 & -7 & -1 \\ 1 & -4 & 3 \\ -1 & 1 & -3 \end{bmatrix} + \begin{bmatrix} -3 & 3 & 6 \\ 6 & -1 & -2 \\ -6 & -1 & -1 \end{bmatrix} = \begin{bmatrix} -6 & -4 & 5 \\ 7 & -5 & 1 \\ -7 & 0 & -4 \end{bmatrix} + 1 = \begin{bmatrix} -5 & -3 & 6 \\ 8 & -4 & 2 \\ -6 & 1 & -3 \end{bmatrix}$$

This gives us the first output array, the application of filter k_0 to the input stack.

We repeat this process for filters k_1 and k_2 to get their outputs so that the final convolutional layer output for the given input is the following, where we write the stacked arrays side by side, a $3 \times 3 \times 3$ output, as desired:

$$\begin{bmatrix} -5 & -3 & 6 \\ 8 & -4 & 2 \\ -6 & 1 & -3 \end{bmatrix} \begin{bmatrix} -1 & 2 & -6 \\ 4 & -3 & 1 \\ 0 & -3 & -5 \end{bmatrix} \begin{bmatrix} -5 & 0 & -3 \\ 0 & 0 & -5 \\ 7 & -3 & 4 \end{bmatrix}$$

Our convolutional layer example mapped a $5 \times 5 \times 2$ input to a $3 \times 3 \times 3$ output. If we naively used a fully connected layer instead, we'd need a weight matrix that has $50 \times 27 = 1,350$ weights that need to be learned. In contrast, the convolutional layer used only $3 \times 3 \times 2$ weights per filter and three filters for a total of 54 weights, excluding bias values—a significant reduction.

In Action

The preceding example illustrates how a convolutional layer works; now let's see the effect of one. Imagine that we've trained the network in Figure 9-3, so we have the weights and biases we need to run unknown images through the network. (You'll see how to train a CNN in Chapter 10.)

The first layer of the network in Figure 9-3 is a convolutional layer that maps a $28 \times 28 \times 1$ input, the single-channel grayscale digit image, to a $26 \times 26 \times 32$ output using a filter with thirty-two 3×3 kernels. Therefore, we know that the weights between the input image and output fit in an array that is $3 \times 3 \times 1 \times 32$: 3×3 for the kernel size, 1 for the number of input channels, and 32 for the number of kernels in the filter.

After training, what do the thirty-two 3×3 kernels of the filter actually look like? We can extract them from the trained model and print them as a set of thirty-two 3×3 matrices. Here are the first two:

$$\begin{bmatrix} 0.022 & 0.163 & 0.152 \\ 0.032 & 0.104 & 0.290 \\ -0.322 & -0.345 & -0.221 \end{bmatrix} \quad \begin{bmatrix} 0.141 & 0.239 & 0.311 \\ 0.005 & -0.026 & 0.215 \\ -0.158 & -0.370 & -0.207 \end{bmatrix}$$

This is nice, but not particularly helpful for building intuition about what the kernels do.

We can also visualize the kernels of a filter by converting the matrices to images. To get the kernels as images, first note that all the kernel values happen to fit in the range $[-0.5, +0.5]$. If we add 0.5 to each kernel value, we've mapped the range to $[0, 1]$. After this, multiplication by 255 converts the kernel values to byte values, the same values a grayscale image uses. Additionally, a value of 0 is now 127, which is a middle gray value.

After this conversion, the kernels can be shown as grayscale images; negative kernel values are closer to black, and positive kernel values are closer to white. However, the mapped kernels are still only 3×3 pixels. The

last step is to upscale the 3×3 images to 64×64 pixels. We'll upscale in two ways. The first uses nearest-neighbor sampling to show the kernel in blocks. The second uses a Lanczos filter, which smooths the image, making it easier to see the orientation of the kernel. Figure 9-4 shows the kernel images with the block versions on top and the smoothed versions on the bottom.

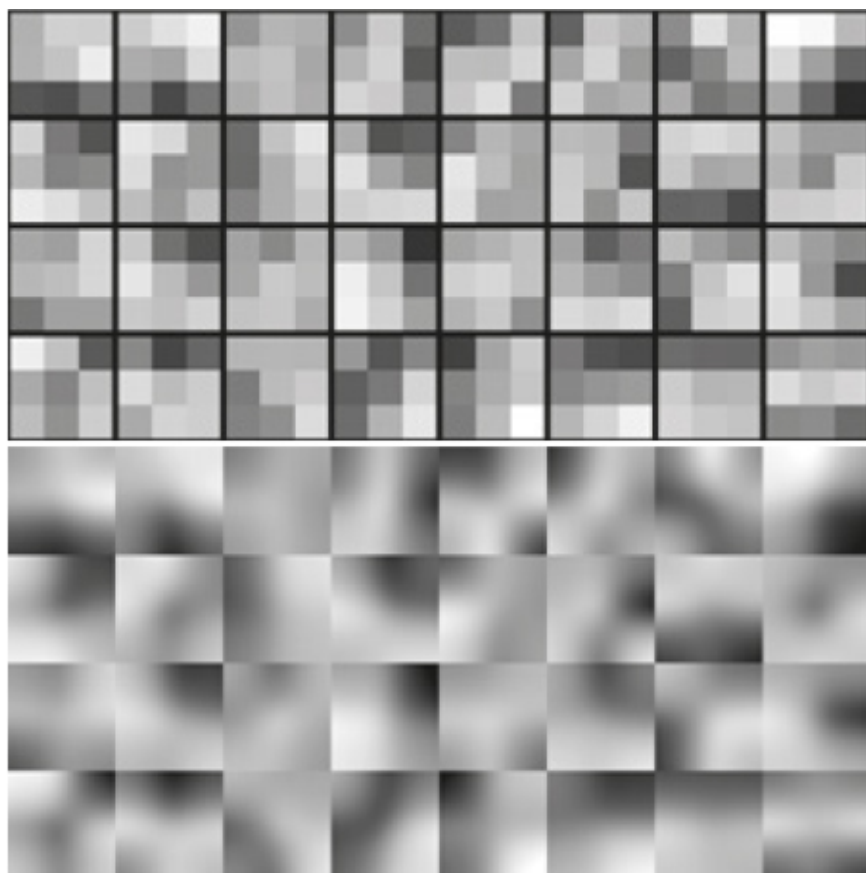


Figure 9-4: The 32 learned kernels of the first convolutional layer (top) and smoothed versions to show the orientations more clearly (bottom)

These images represent the 32 kernels learned by the first convolutional layer of the model in Figure 9-3. The images have just enough detail to hint that the kernels are selecting for structure in specific directions, just like the kernel that produced the image on the lower right of Figure 9-2, which emphasized diagonal structures.

Let's turn our attention now to the effect of the kernels. What do the kernels do to an input MNIST image? We can run a sample MNIST image through the kernels by convolving each kernel with the sample, here a 3,

and following a process similar to the one that produced the preceding kernel images. The result is a set of 32 26×26 images, which we again upscale to 64×64 before displaying them. Figure 9-5 shows the result.

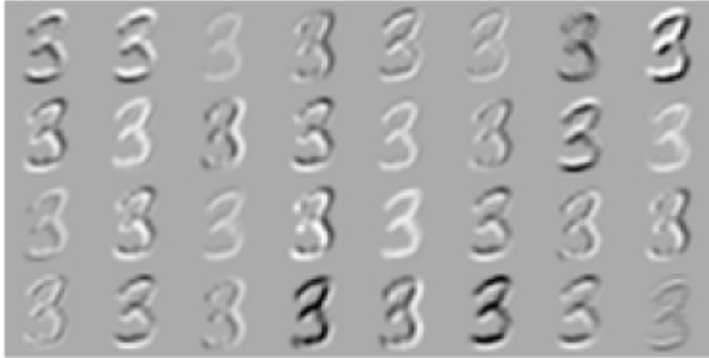


Figure 9-5: The 32 kernels applied to a sample MNIST input

The order of the kernels in Figure 9-4 matches the images in Figure 9-5. For example, the top-right image of Figure 9-4 shows a kernel that is light on the upper left and dark on the lower right, meaning it will detect structures along the diagonal from lower left to upper right. The output from applying this kernel to the sample is the upper-right image of Figure 9-5. We see that the kernel enhanced parts of the three that are primarily diagonal from the lower left to the upper right.

This example is easy to interpret because the input is a grayscale image with a single channel. There is no summing of kernel outputs across channels as we previously saw for the more general operation.

Typically, the first convolutional layer of a CNN learns kernels that select for specific orientations, textures, or, if the input image is RGB, colors. For the grayscale MNIST images, orientation is most important. The kernels learned at higher convolutional layers in the CNN are also selecting for image features, but the interpretation of *what* the kernel is selecting becomes more abstract and difficult to understand.

It's worth noting that the kernels learned by a CNN's first convolutional layer are very similar to the first layer of visual processing in the mammalian brain. This is the primary visual cortex, or V1 layer, that detects lines and edges. Additionally, always keep in mind that the set of convolutional and pooling layers are there to learn a new feature representation: a new representation of the input image. This new

representation does a better job of separating classes so that the fully connected layers can more easily distinguish between them.

Multiple Layers

Most CNNs have more than one convolutional layer. One reason for this is to build up features that are influenced by larger portions of the input as one goes deeper into the network. This introduces the ideas of receptive field and effective receptive field. The two concepts are similar and often confused. We can explain both by looking at Figure 9-6.

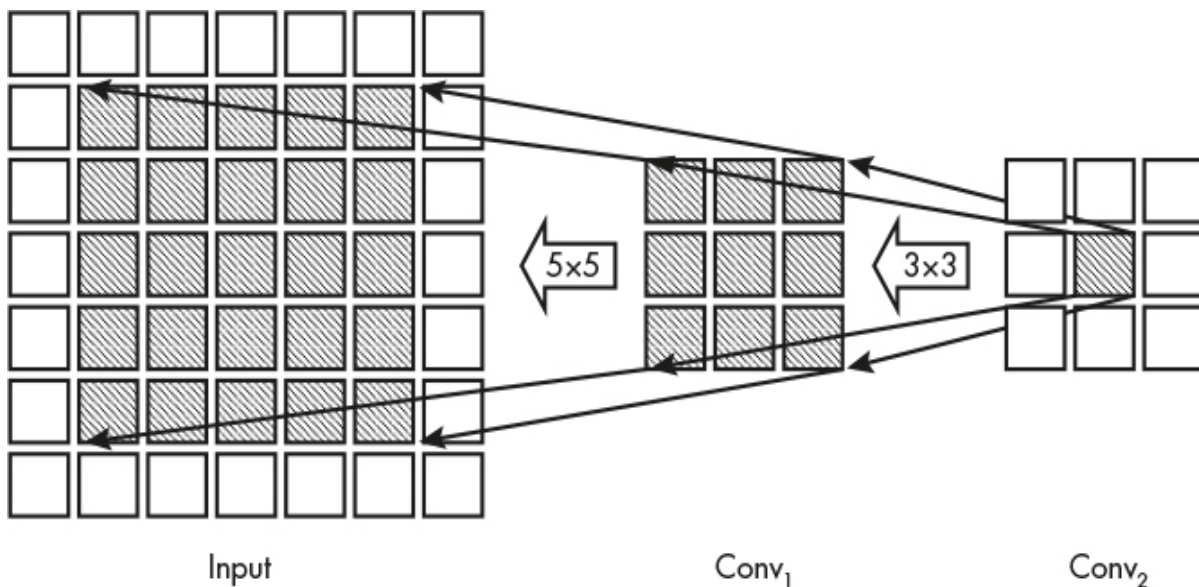


Figure 9-6: Receptive fields

The figure shows the output of two convolutional layers and the input to the model. We're showing only the relevant parts of the output, using a 3x3 kernel. We're also ignoring the depth of the filters since the receptive fields (defined next) are the same across the depth of the convolutional layer outputs.

Read Figure 9-6 right to left as the arrows indicate. This is the opposite direction to the flow of data through the network. Here we're looking back to earlier layers to see what has influenced the output value at a higher layer. The squares are output values, with the rightmost shaded square being one of the outputs of Conv₂. This is our starting point for looking back to see what influences this value. The arrows point to the outputs of Conv₁ that influence the shaded value in Conv₂. The value in Conv₂ then

has a 3×3 receptive field, as it's influenced directly by the 3×3 shaded outputs of Conv_1 . This is how we'll define *receptive field*: the set of outputs from the preceding layer that directly influence the output of the current layer.

If we look at the set of input values that directly influence the 3×3 shaded region of Conv_1 , we see a 5×5 region. This makes sense: Each shaded output of Conv_1 has a receptive field that is a 3×3 region of the input. The receptive field is 3×3 because the kernels of Conv_1 are 3×3 kernels. They overlap so that the shaded 5×5 input region is what influences all the shaded Conv_1 outputs.

Look again at the rightmost shaded output value. If we trace back to the input all the values that can influence it, we see that the shaded 5×5 region of the input can affect its value. This region is the *effective receptive field* for the rightmost shaded output of Conv_2 . Ultimately, this output value responds to what is happening in the input image in the leftmost shaded region. As the CNN gets deeper, with additional convolutional layers, we can see how the effective receptive field can change so that deeper convolutional layers are working with values ultimately derived from larger portions of the input to the model.

Initialization

Chapter 6 showed that the performance of a traditional neural network is strongly influenced by the type of random initialization used for the learned weights and biases. The same is true for CNNs. Recall that the weights of a convolutional layer are the values of the kernels. They are learned during backprop, just like the weights of a traditional neural network.

We need an intelligent way to initialize these values when we set up the network. Fortunately, the best initialization approaches for a traditional neural network apply directly to convolutional layers as well. For example, Keras defaults to Glorot initialization, which, as noted in Chapter 6, is sometimes called *Xavier initialization* in other toolkits.

Let's move on now from convolutional layers to pooling layers. These are simpler but perform an important, if somewhat controversial, function.

Pooling Layers

Our favorite illustration, Figure 9-3, shows a pooling layer after the first two convolutional layers. This pooling layer takes an input stack of $24 \times 24 \times 64$ and produces an output stack of $12 \times 12 \times 64$. The pooling part is marked as 2×2 . What's going on here?

The key is the 2×2 . This means, for each of the 64 24×24 inputs, we move a 2×2 sliding window over the input and perform an operation similar to convolution. Not explicitly called out in Figure 9-3 is that the stride is also 2 so that the sliding 2×2 window jumps by two to avoid overlapping itself. This is typically the case but doesn't need to be. Since the pooling operation is per input in the stack, the output leaves the stack size unchanged. This is contrary to what a convolutional layer often does.

Let's look at the pooling operation applied to a single input in the stack, a 24×24 matrix. Figure 9-7 shows us what's going on.

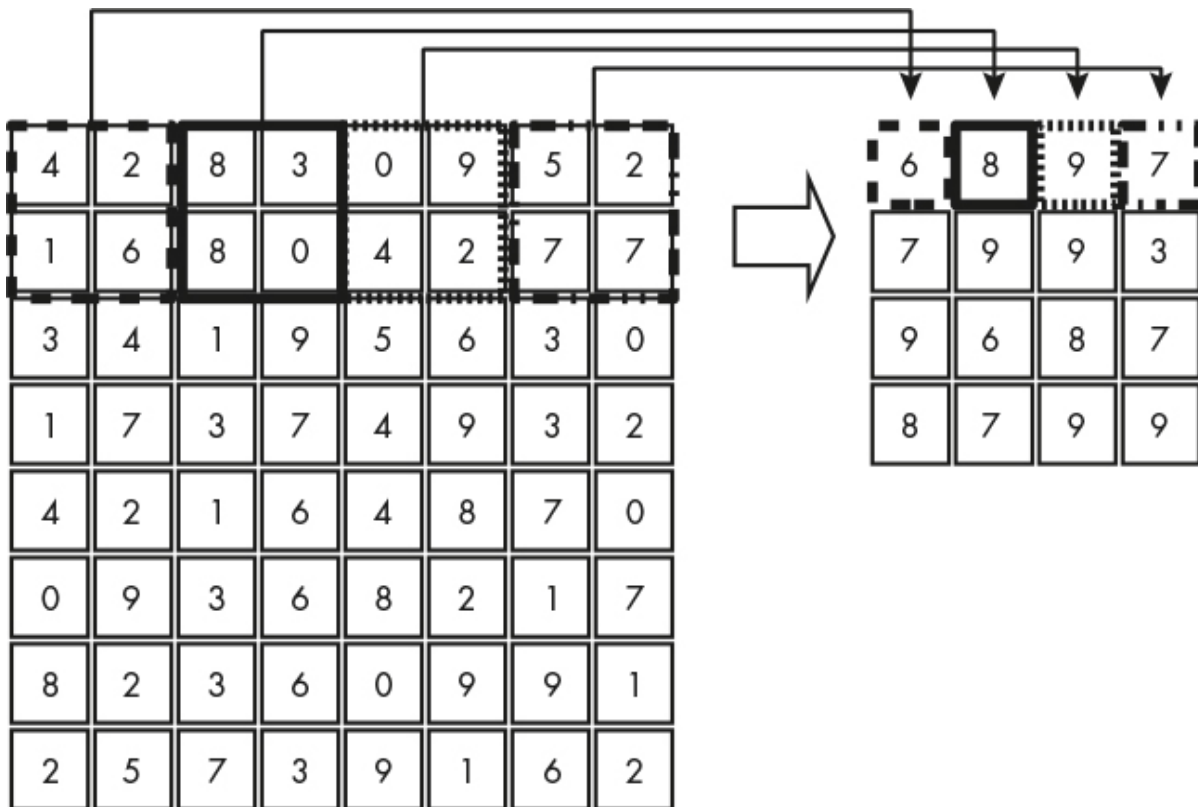


Figure 9-7: Applying 2×2 max pooling to an 8×8 input

The first 2×2 values are mapped to the first output value. Then we move over two and map the next 2×2 region to the output, and so on, until

the entire input is mapped. The operation performed on each 2×2 region is up to the architect of the CNN. The most common operation is “select the largest value,” or *max pooling*, which is what Figure 9-7 shows. This is also the operation the model in Figure 9-3 is performing. Another fairly common pooling operation is to average the values.

We can see from Figure 9-7 that the 8×8 input matrix is mapped to a 4×4 output matrix. This explains why the output of the pooling layer in Figure 9-3 is 12×12 ; each dimension is half the size of the input.

The pooling operation is straightforward but throws information away. So why do it at all? The primary motivation for pooling is to reduce the number of values in the network. Typically, as depth increases, the number of filters used by convolutional layers increases, by design. We see this for even the simple network of Figure 9-3: The first convolutional layer has 32 filters, while the second has 64. Therefore, the second convolutional layer outputs $24 \times 24 \times 64 = 36,864$ values, but after 2×2 pooling, there are only $12 \times 12 \times 64 = 9,216$ values to work with, a 75 percent reduction.

It’s important to note that we’re talking about the number of values present as we move data through the network, not the number of learned parameters in the layers. The second convolutional layer in Figure 9-3 has $3 \times 3 \times 32 \times 64 = 18,432$ learned parameters (ignoring bias values), while the pooling layer has no learned parameters.

This reduction in the number of values in the output, which is our representation of the input, speeds up computation and acts as a regularizer to guard against overfitting. The regularization techniques and rationales of Chapter 6 are equally valid for CNNs. However, since pooling throws information away and selects proxies to represent entire regions of the representation (the convolutional layer outputs), it alters the spatial relationship between parts of the input. This loss of spatial relationships might be critical for some applications and has motivated people like Geoffrey Hinton to eliminate pooling by introducing other types of networks (search for *capsule networks*).

Fully Connected Layers

In the second row of Figure 9-3, all the layers starting with `Flatten` form the fully connected layer of the model. The figure uses Keras terminology;

many people call the Dense layer the fully connected layer and assume there is a flatten operation as part of it along with the activation (ReLU) and optional dropout before the softmax layer. Therefore, the model in Figure 9-3 has only one fully connected layer.

I previously stated that the net effect of the convolutional and pooling layers is to change the representation of the input feature (say, the image) into one that makes it easier for a model to reason about. During training, we are asking the network to learn a different, often more compact, representation of the input to help the model perform better on unseen inputs. For the model in Figure 9-3, all the layers up to and including the pooling layer (and the dropout layer after it for training) are there to learn a new representation of the input image. In this case, the fully connected layer is the model: It will take that new representation and ultimately make a classification based on it.

Fully connected layers are just that: fully connected. The weights between the flattened final pooling layer of 9,216 elements for Figure 9-3 ($12 \times 12 \times 64 = 9,216$) and the dense layer of 128 elements are the same as if we were building a traditional neural network. This means that there are $9,216 \times 128 = 1,179,648$ weights plus an additional 128 bias values that need to be learned during training. Therefore, of the 1,199,882 parameters (weights and biases) in the model of Figure 9-3, 98.3 percent of them are in the transition between the final pooling layer and the fully connected layer.

This illustrates an important point: Fully connected layers are expensive in terms of parameters that need to be learned, just as they are for traditional neural networks. Ideally, if the feature-learning layers (the convolutional and pooling layers) are doing their job well, we might expect to need only one or two fully connected layers.

Fully connected layers have another disadvantage, besides memory use, that can impact their utility. Consider the following scenario: You want to be able to locate digits in grayscale images. Assume for simplicity that the background is black. If you use the model of Figure 9-3 trained on MNIST digits, you'll have a model that is very good at identifying digits centered in 28×28-pixel images. But what if the input images are large and you don't know where the digits are in the image, let alone how many digits there are?

Then things get a little more interesting. The model of Figure 9-3 expects input images that are 28×28 pixels in size and only that size. In Chapter 10, we'll work through this problem in detail as an experiment, but for now, let's discuss fully convolutional layers, a possible solution to this disadvantage of using fully connected layers in CNNs.

Fully Convolutional Layers

In the preceding section, I said that the model of Figure 9-3 expects input images that are 28×28 pixels in size and only that size. Let's see why.

This model has many kinds of layers. Some, like the ReLU and dropout layers, have no impact on the dimensionality of the data flowing through the network. The same cannot be said of the convolutional, pooling, and fully connected layers. Let's look at these layers one by one to see how they're tied to the dimensionality of the input image.

The convolutional layers implement convolutions. By definition, a convolution involves moving a fixed-size kernel over an input image (thinking purely 2D here). Nothing in that operation specifies the size of the input image. The output of the first convolutional layer in Figure 9-3 is $26 \times 26 \times 32$. The 32 comes from the number of filters selected by the architecture. The 26×26 comes from using a 3×3 convolution kernel on a 28×28 input with no padding. If the input image were instead 64×64 pixels, the output of this layer would be $62 \times 62 \times 32$, and we wouldn't need to do anything to alter the architecture of the network. The convolutional layers of a CNN are agnostic to the spatial dimensions of their inputs.

The pooling layer in Figure 9-3 takes a $24 \times 24 \times 64$ input and produces a $12 \times 12 \times 64$ output. As we previously saw, the pooling operation is much like the convolution operation: it slides a fixed-size window over the input, spatially, and produces an output; in this case, the output is half the dimensionality of the input while leaving the depth the same. Again, nothing in this operation fixes the spatial dimensions of the input stack. If the input stack were $32 \times 32 \times 64$, the output of this max-pooling operation would be $16 \times 16 \times 64$ without a change needed to the architecture.

Finally, we have the fully connected layer that maps the $12 \times 12 \times 64 = 9,216$ pooling output to a 128-element, fully connected (Dense) layer. As shown in Chapter 5, fully connected neural networks use matrices of

weights between layers in their implementation. There are 9,216 elements in the output of the pooling layer and a fixed 128 in the dense layer, so we need a matrix that is 9,216×128 elements. This size *is* fixed.

If we use the network with a larger (say, 32×32) input image, by the time we get through the pooling layer, the output size will be $14 \times 14 \times 64 = 12,544$, which would require an existing 12,544×128-weight matrix to map to the fully connected layer. This won't work, as we trained a network that uses a 9,216×128 matrix. The fully connected layers of a CNN fix the input size of the CNN. If we get around this, we could apply inputs of any size to the CNN, assuming memory allows.

We could, naively, slide a 28×28 window over the larger input image, run each 28×28-pixel image through the model as we trained it, and output a larger map, where each pixel now has a probability of that digit being present. There are 10 digits, so we would have 10 output maps. This sliding window approach certainly works, but it's very computationally expensive, as many simplistic implementations of algorithms often are.

Fortunately for us, we can do better by converting the fully connected layer into an equivalent convolutional layer to make the model a fully convolutional network. In a fully convolutional network, there are no fully connected layers, and we're not restricted to using a fixed input size. You'll learn about the relationship between input size and the output of the network when it is fully convolutional in Chapter 10, but the essential operation is to look at the size of the final standard convolutional or pooling layer and replace the fully connected layer that follows with a convolutional layer using a kernel of the same size.

In Figure 9-3, the output of the pooling layer is 12×12×64. Therefore, instead of the 128-element fully connected layer that we saw fixes our input size, we can mathematically get the same calculation by changing the fully connected layer into a 12×12×128 convolutional layer. Convoluting a 12×12 kernel over a 12×12 input produces a single number. Therefore, the output of the 12×12×128 convolutional layer will be a 1×1×128 array, which is functionally the same as the 128 outputs of the fully connected layer that we originally used. Additionally, the convolution operation between a 12×12 kernel and a 12×12 input is to simply multiply the kernel values by the input values, element by element, and sum them. A fully connected layer does this for each of its nodes.

We don't save anything in terms of the number of parameters when using a convolutional layer this way, as we can see from Figure 9-3. The 9,216 elements of the pooling layer output times the 128 nodes of the fully connected layer means we have $9,216 \times 128 = 1,179,648$ weights + 128 bias terms needed for both the fully connected and fully convolutional layers. When moving to the $12 \times 12 \times 128$ convolutional layer, we have $12 \times 12 \times 64 \times 128 = 1,179,648$ weights to learn, the same as before. However, now we also have the freedom to change the input size, as the $12 \times 12 \times 128$ convolutional layer will automatically convolve over any larger input, giving us outputs that represent the application of the network to 28×28 regions of the input with a stride determined by the specific architecture of the network.

Fully convolutional networks stem from the 2014 paper by Jonathan Long et al., "Fully Convolutional Networks for Semantic Segmentation." The term *semantic segmentation* refers to assigning a class label to each pixel of the input image. Currently, the go-to architecture for semantic segmentation is the U-Net (see "U-Net: Convolutional Networks for Biomedical Image Segmentation" by Olaf Ronneberger et al., 2015), which has seen widespread success, especially in medical domains. We'll explore U-Nets in later chapters.

We've discussed the primary CNN layers, those found in Figure 9-3. We could cover many more, but they are generally beyond what we want to present at this level. Batch normalization is one exception, which we'll experiment with in Chapter 13. New layer types are being added all the time in response to active research projects. However, in the end, the core includes the layers we've discussed in this chapter. Let's move on now and see how a trained CNN processes unknown inputs.

How the CNN Operates

In the previous sections, we discussed the architecture and layers of our sample CNN, Figure 9-3. In this section, we'll illustrate the operation of the network to see how it responds to two new inputs, one a 4 and the other a 6. We assume the network is fully trained; we'll train it for real in Chapter 10.

The input image is passed through the model layer by layer, using the trained weights and biases to calculate outputs for each layer:

$$\text{Input} \rightarrow \text{Conv}_0 \rightarrow \text{Conv}_1 \rightarrow \text{Pool} \rightarrow \text{Dense} \rightarrow \text{Softmax}$$

We refer to the layer outputs as the *activations*. The output of the first convolutional layer is a stack of 32 26×26 images, the response of the input image to each of the 32 kernels. This stack then passes to the second convolutional layer to produce 64 24×24 outputs.

Between the two convolutional layers is a ReLU operation that clips the output so that anything that would have been negative is now 0. Doing this adds a nonlinearity to the data as it flows through the network. Without this nonlinearity, the net effect of the two convolutional layers is to act like a single convolutional layer. With the nonlinearity imposed by the ReLU, we enable the two convolutional layers to learn different things about the data.

The second ReLU operation makes the stack of 64 24×24 outputs 0 or positive. Next, a 2×2 max-pooling operation reduces the 64 outputs to 12×12 in size. After this, a standard fully connected layer produces 128 output values as a vector from the 9,216 values in the stack of 12×12 activations. From this, a set of 10 outputs, one for each digit, is calculated via a softmax. These are the output values of the network representing the network's confidence as to which class label should be assigned to the input image.

We can illustrate the activations by displaying the output images: either 26×26 for the first convolutional layer, 24×24 for the second convolutional layer, or 12×12 for the pooling layer. To show the activations from the fully connected layer, we can make an image of 128 bars, where the intensity of each bar represents the vector value. Figure 9-8 shows the activations for our two sample digits.

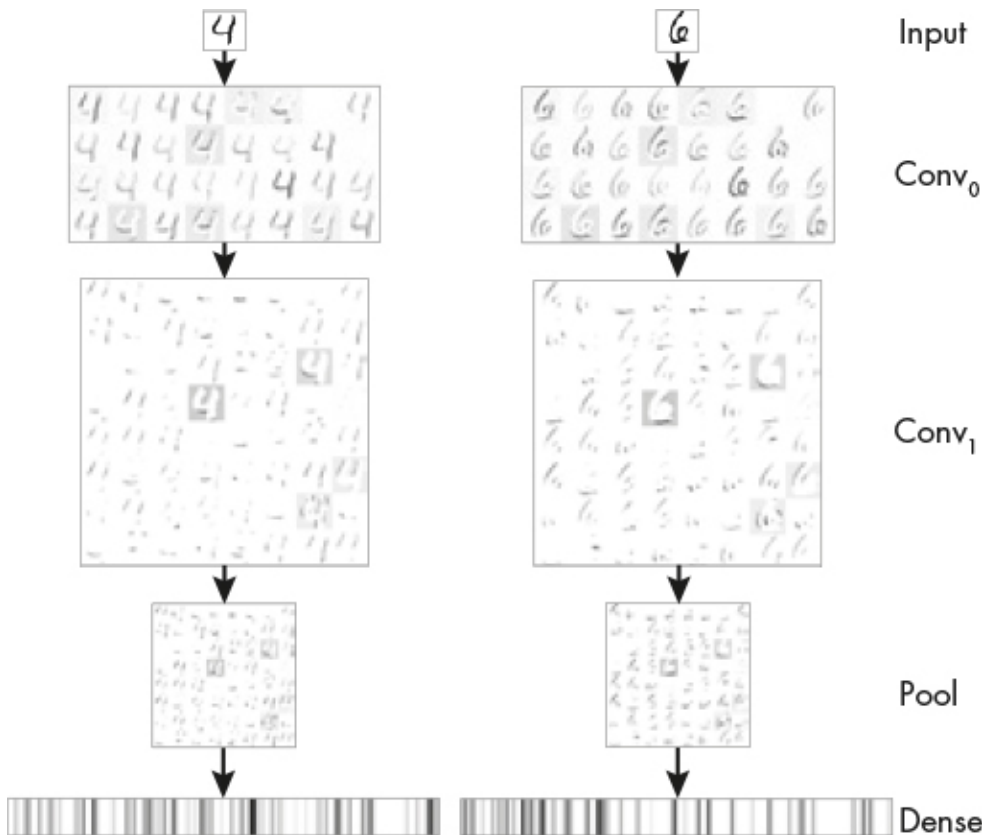


Figure 9-8: Model activations per layer. The output is inverted: Darker implies stronger activation.

Note that the images are inverted so that darker corresponds to stronger activation values. We aren't showing the softmax outputs. These values are as follows:

	0	1	2	3	4	5	6	7	8	9
4	0.00	0.00	0.00	0.00	0.99	0.00	0.00	0.00	0.00	0.00
6	0.00	0.00	0.00	0.00	0.00	0.00	0.99	0.00	0.00	0.00

This indicates that in both cases, the model is very confident of the class label that should be assigned and that it is, in fact, correct.

Looking back at Figure 9-8, we see that the output of the first convolutional layer is simply the response of the single input image (grayscale) and the kernels of the layer. This harkens back to Figure 9-2, where we saw that convolution could be used to highlight aspects of the input image. After the ReLU operation, the responses of the 64 filters of

the second convolutional layer, each a stack of 32 kernels, seems to be picking out different portions, or strokes, in the input images. Think of these as a set of smaller components from which the input is constructed. The second ReLU and pooling operation preserve much of the structure of the second convolutional layer outputs, but reduce the size to one-quarter what it was previously. Finally, the output of the fully connected layer shows the pattern derived from the input image, the new representation that we expect to be easier to classify than the raw image input.

The dense layer outputs of Figure 9-8 are different from each other. This begs the question: What do these outputs look like for several instances of 4 and 6 digits? Is there something in common that we can see, even in these values? We might expect that there is because we know this network has been trained and has achieved a very high accuracy of over 99 percent on the test set. Let's take a look at running 10 "4" and 10 "6" images from the test set through the network and compare the dense-layer activations. This gives us Figure 9-9.

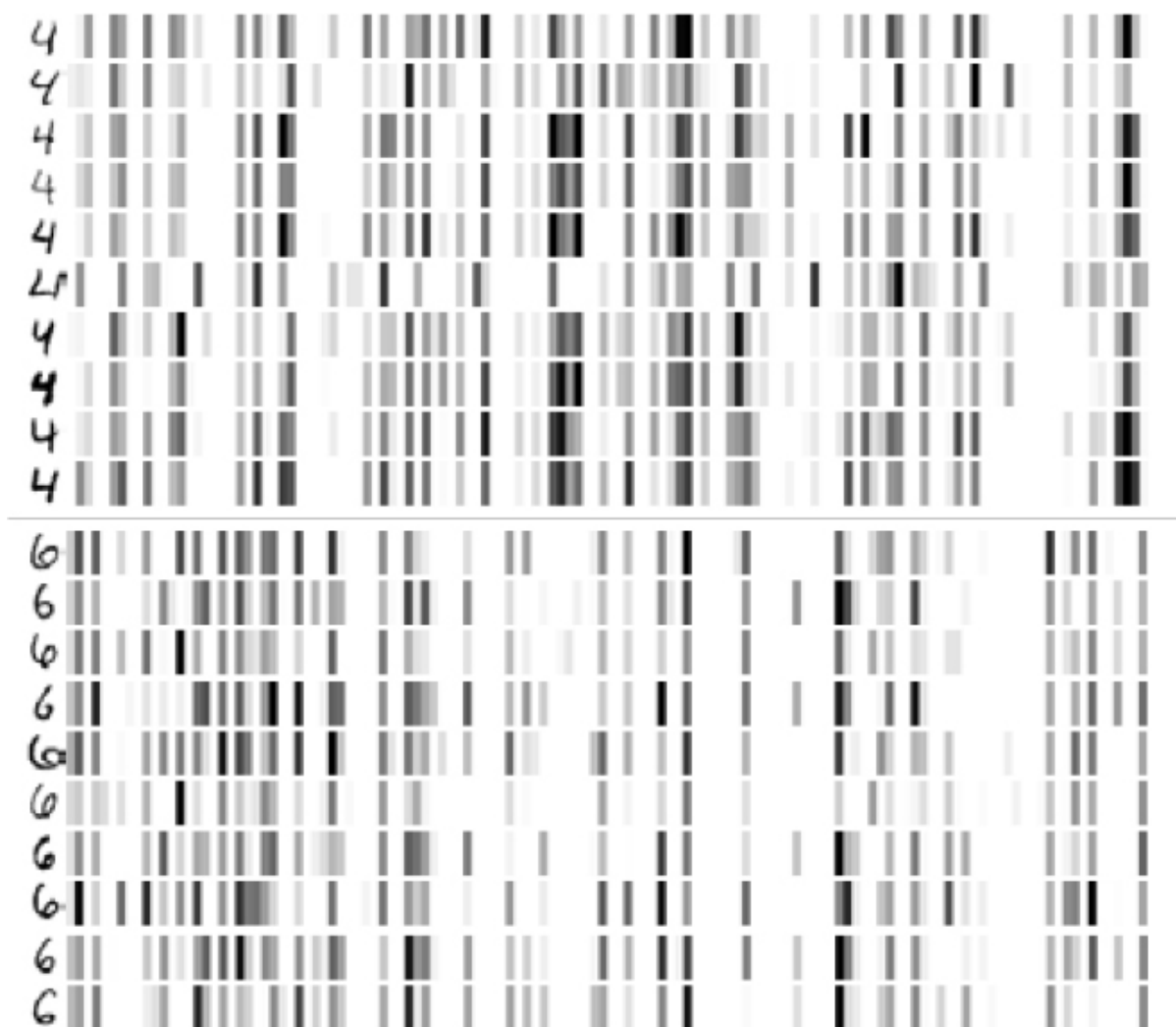


Figure 9-9: Dense-layer activations for 10 instances of the numbers 4 and 6. The output is inverted: Darker implies stronger activation.

On the left, we see the actual input to the model. On the right is the representation of the 128 outputs in the fully connected layer, the one that feeds into the softmax. Each digit has a particular pattern that is common to each one of the digits. However, variations also occur. The middle 4 has a very short stem, and we see that its representation in the fully connected layer is also different from all the other examples. Still, this digit was successfully called a 4 by the model with a certainty of 0.999936.

Figure 9-9 provides evidence that the model learned what we wanted it to in terms of representation of the input. The softmax layer maps the 128 elements of the dense layer to 10, the output nodes from which the softmax probabilities are calculated. This is, in effect, a simple traditional neural network with no hidden layers.

This simpler model succeeds in correctly labeling the images because the new representation of the inputs does a much better job of separating the classes so that even a simple model can make solid predictions. It also succeeds because the training process jointly optimizes both the weights of this top-layer model and the weights of the lower layers that generate the input to the model at the same time, so they reinforce each other. You'll sometimes see this referred to in the literature as *end-to-end training*.

We can demonstrate the claim that the features are better separated by looking at a plot of the dense-layer activations for the MNIST test data. Of course, we can't look at the actual plot, as I have no idea how to visualize a plot in 128 dimensions, but all is not lost. The machine learning community has created a powerful visualization tool called t-SNE, which, fortunately for us, is part of sklearn. This algorithm intelligently maps high-dimensional spaces to lower-dimensional spaces, including 2D.

If we run a thousand randomly selected MNIST test images through the model and then run the resulting 128-dimension dense-layer activations through t-SNE, we can produce a 2D plot where the separation between classes reflects the actual separation in the 128-dimensional space. Figure 9-10 is the result.

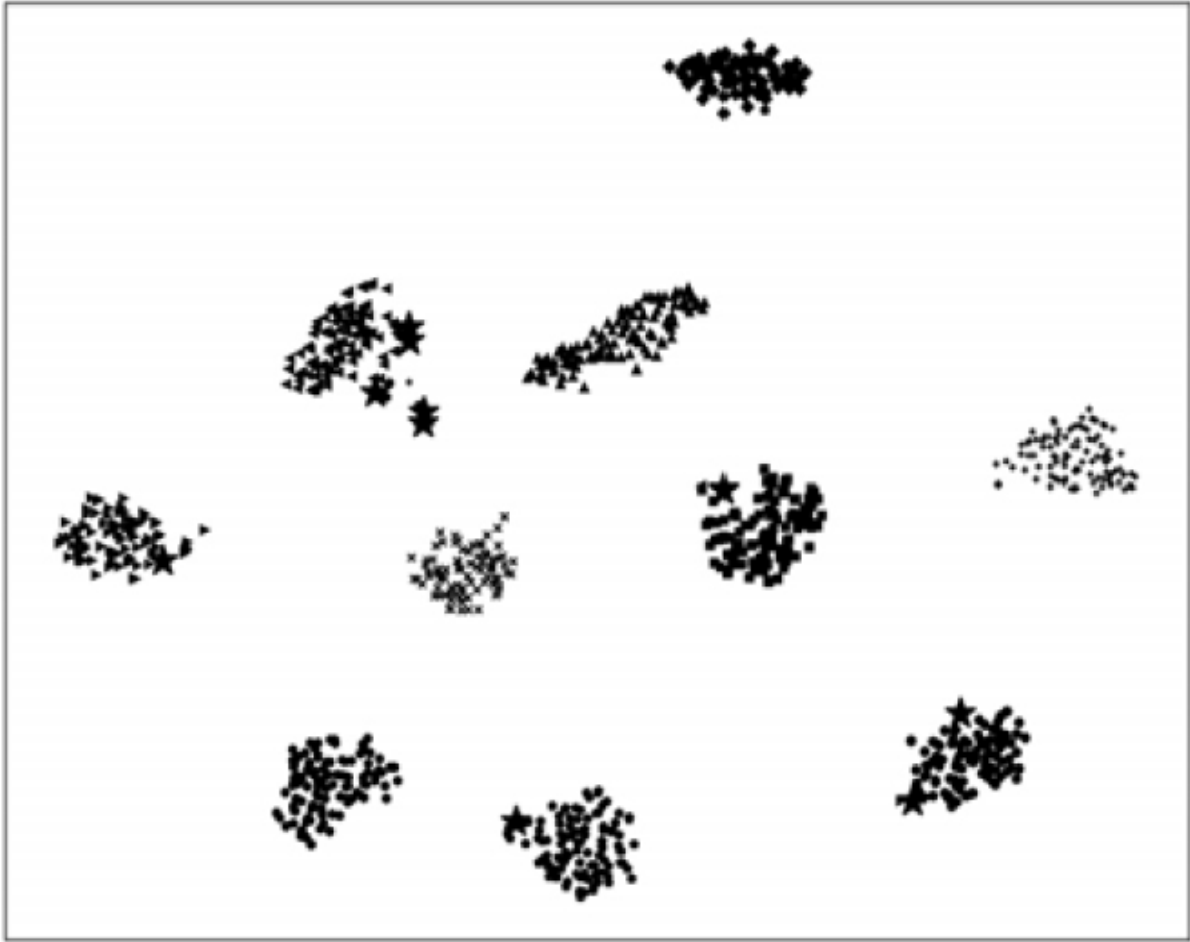


Figure 9-10: A t-SNE plot that shows how well the model separates test samples by class

Here each class uses a different plot symbol. If the model did not correctly classify the sample, it is shown as a larger star. Only a handful of these samples were misclassified. The separation by class type is evident; the model has learned a representation that makes deciding on the correct class label straightforward in most cases. We can readily count 10 blobs in the t-SNE plot.

Summary

This chapter introduced the major components of convolutional neural networks. These are workhorse networks for modern deep learning, especially for vision tasks because of their ability to learn from spatial relationships. We worked through a model to classify MNIST digits and detailed new processing layers, including convolutional layers and pooling

layers. You then learned that the fully connected layers of a CNN are analogs of the traditional neural networks from earlier chapters.

Next, we saw how to modify the fully connected layers to enable operation on larger inputs. Finally, we looked at the activations generated by the network when a sample image was passed through and saw how the convolution and pooling layers worked together to produce a new representation of the input, one that helped to separate the classes in the feature space, thereby enabling high accuracy.

In the next chapter, we'll continue looking at CNNs, but instead of theory, we'll work with actual examples to see how the various parameters of the network, and the hyperparameters used during training, affect model performance. This will help you build intuition about how to work with CNNs in the future.

OceanofPDF.com

10

EXPERIMENTS WITH KERAS AND MNIST



In the preceding chapter, we covered the essential components and functionality of a CNN. In this chapter, we'll work with our test model from Chapter 9. You'll first learn how to implement and train it in Keras. After that, we'll conduct a set of experiments that will build your intuition for how various architectures and learning parameter choices affect the model.

From there, we'll move beyond classification of simple input images and expand the network by converting it into a fully convolutional model capable of processing arbitrary inputs and locating digits wherever they occur in the input.

After fully convolutional networks, we'll wander a little deeper into the pool of deep learning and fulfill a promise made in Chapter 4: We'll explore how well CNNs perform on the scrambled MNIST digit experiment. We saw in Chapter 7 that scrambling the pixels of the digits made it virtually impossible for us to see what the digit was but had little to no effect on how well a traditional neural network was able to interpret the digits. Is the same true for a CNN? We'll find out.

Building CNNs in Keras

The model from Figure 9-3 on page 244 is straightforward to implement in Python with the Keras library. We'll list the code first, explain it, and then run it to see the sort of output it produces. The code naturally falls into three sections. The first loads the MNIST data and configures it for Keras, the second builds the model, and the third trains the model and applies it to the test data.

Loading the MNIST Data

Listing 10-1 has the first part of our code.

```
import tensorflow.keras as keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras import backend as K

batch_size = 128
num_classes = 10
epochs = 12
img_rows, img_cols = 28, 28

❶ (x_train, y_train), (x_test, y_test) = mnist.load_data()

❷ if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

❸ x_train = x_train.astype('float32')
    x_test = x_test.astype('float32')
```

```
x_train /= 255
x_test /= 255

④ y_train = keras.utils.to_categorical(y_train, num_classes)
   y_test = keras.utils.to_categorical(y_test, num_classes)
```

Listing 10-1: Data loading and preprocessing

Keras, part of TensorFlow, is a rather large toolkit consisting of many modules. We import the library first and then specific functions from it. The `mnist` module gives us access to the MNIST data from within Keras; the `sequential` model type is for implementing basic CNNs. We'll use a more flexible approach to constructing Keras models in later chapters. Our CNN will need specific layers, the ones used in Figure 9-3: `Dense`, `Dropout`, `Flatten`, `Conv2D`, and `MaxPool2D`, all of which we import. Keras supports a plethora of other layers; I encourage you to spend some quality time with their documentation pages: <https://keras.io>.

Next, we set the learning parameters, including the number of epochs, classes, and minibatch size. There are 10 classes, and the images are 28×28-pixel grayscale. As in `sklearn`, in Keras, you specify the number of epochs (full passes through the training set), not the number of minibatches that should be processed. Keras automatically processes the entire training set per epoch in sets of the minibatch size (here, 128 samples at a time). Recall that MNIST's training set consists of 60,000 samples, so there are at least $60,000/128 = 468$ minibatches per epoch, using integer division. There will be 469 if Keras uses the remainder, the samples that do not build a complete minibatch. Remember that each minibatch process results in a gradient-descent step: an update of the parameters of the network.

After loading the MNIST train and test data ❶ come a few lines of code that may seem somewhat mysterious at first. Keras is a higher-level toolkit that uses potentially different lower-level backends. In our case, the backend is TensorFlow, which we installed in Chapter 0. Different backends expect the model input in different forms. The `image_data_format` function returns a string indicating where the underlying toolkit expects to see the number of channels or filters for convolutional layers. The

TensorFlow backend returns `channels_last`, meaning it expects an image to be represented as a 3D array of $H \times W \times C$, where H is the image height, W is the image width, and C is the number of channels. For a grayscale image like MNIST, the number of channels is 1. This block of code reformats the input images to match what Keras is expecting to see ❷.

The next block of code converts the byte image values to floating-point numbers in the range $[0, 1]$ ❸. This is the only scaling done to the input data, and this type of scaling is typical of CNNs that work with images.

Finally, the `to_categorical` function is used to map the class labels in `y_test` to one-hot vector representations ❹, which is how Keras wants to see the labels. As we'll see, the model has 10 outputs, so the mapping is to a vector of 10 elements; each element is 0 except for the element whose index corresponds to the label in `y_test`. That element is set to 1. For example, `y_test[333]` is of class 6 (a 6 digit). After the call to `to_categorical`, `y_test[333]` becomes the following:

```
array([0.,0.,0.,0.,0.,0.,1.,0.,0.,0.], dtype=float32)
```

Here all entries are 0 except index 6, which is 1.

Building the Model

With the dataset preprocessed, we can build our model. Listing 10-2 builds the exact model we defined with pictures in Figure 9-3.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
```

```
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])

print("Model parameters = %d" % model.count_params())
print(model.summary())
```

Listing 10-2: Building the MNIST model

Keras defines the model as an instance of the `Sequential` class. The model is built by adding layers to that instance, hence all the calls to the `add` method. The argument to `add` is the new layer. The layers are added from the input side to the output side, so the first layer we need to add is the 2D convolutional layer that uses a 3×3 kernel on the input image. Note, we are not specifying the number of images nor the minibatch size; Keras will handle that for us when the model is put together and trained. Right now, we're defining the architecture.

Using the architecture defined in Figure 9-3, the first layer is a `Conv2D` layer. The first argument is the number of filters—here, 32. The kernel size is given as a tuple, (3, 3). Kernels don't need to be square, hence the kernel width and height. It's possible that the spatial relationship of the parts of your input might be better detected with a non-square kernel. If so, Keras lets you use one. That said, almost all kernels in practical use are square. After the kernel, we define an activation function to apply to the output of the convolutional layer, here a ReLU. The shape of the input to this layer is explicitly defined via `input_shape`, and we saw that earlier for our MNIST model using a TensorFlow backend, the shape is a tuple, (28, 28, 1).

Next, we add the second convolutional layer. This one has 64 filters, also using a 3×3 kernel and a ReLU activation on the output. We do not need to specify the shape here: Keras knows the input shape because it knows the shape of the previous convolutional layer's output.

Max pooling comes next. We explicitly state that the pooling size is 2×2 , with an implied stride of 2. If we wanted to use average pooling

here, we would replace `MaxPooling2D` with `AveragePooling2D`.

After pooling comes our first dropout layer, which uses a 25 percent probability of dropping an output, here the output of the max-pooling layer.

We discussed earlier how Keras separates the operations of a fully connected layer into flatten and dense layers; see “Anatomy of a Convolutional Neural Network” on page 243. This allows more fine-grained control of the architecture. We add a `Flatten` layer to map the pooling output to a vector and then pass this vector to a `Dense` layer to implement the classic fully connected layer. The dense layer has 128 nodes and uses a ReLU for the activation function. If we want dropout on the output of the dense layer, we need to add it explicitly, so we add one with a probability of 50 percent.

The final dense layer has 10 nodes, one for each possible class label. The activation is set to `softmax` to get a softmax output on the inputs to this layer. Since this is the last layer we define, the output of this layer—the softmax probabilities for membership in each of the 10 classes—is the output of the entire model.

To configure the model for training, we need to call the `compile` method. This sets the loss function used during training (`loss`) and the specific optimization algorithm to use (`optimizer`). The `metrics` keyword defines which metrics to report during training. For our example, we are using the categorical cross-entropy loss, which is the multiclass version of the binary cross-entropy loss. Chapter 6 described this loss function; it is the go-to loss function for many CNNs.

We need to discuss the `optimizer` keyword more thoroughly. Chapter 6 presented gradient descent and the more common version, stochastic gradient descent (SGD). As you might expect, the machine learning community has not been content to simply use this algorithm as is; much research has been done to see if it can be improved upon for training neural networks. This has led to the development of multiple variations on gradient descent, many of which Keras supports.

If we want, we can use classic SGD descent here. The example, however, uses a variant called Adam. Adam seeks to change the learning rate (step size) intelligently during training. For practical purposes, we

should consider Adam an improved version of SGD. Keras also supports other optimization approaches that we won't cover here, but you can read about them in the Keras documentation.

After the call to `compile`, our model is defined. The convenience methods `count_params` and `summary` produce output characterizing the model itself. When we run the code, we'll see the sort of output they produce.

Training and Evaluating the Model

Finally, with both data and model defined, we can train and then evaluate the model on the test data. The code for this is in Listing 10-3.

```
history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1,
                    validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
model.save("mnist_cnn_model.keras")
```

Listing 10-3: Training and testing the MNIST model

The `fit` method trains the network by using the supplied training samples (`x_train`) and a one-hot vector version of the associated class label (`y_test`). We also pass in the number of epochs and the minibatch size. Setting `verbose` to 1 will produce loss stats during training. Lastly, we have `validation_data`. For this example, we're being a bit sloppy and passing in all the test data instead of holding some back for final testing. (This is just a simple example, after all.) Normally, we'd hold back some test data to use after the final model has been trained. This ensures that results on this held-out test data represent what we might encounter when using the model in the wild.

Notice that the `fit` method returns something. This is a `History` object, and its `history` property holds a per-epoch summary of the

training and validation loss and accuracy values. We can use these to make summary plots if we wish.

Once the model is trained, we can get a score, similar to the `score` of `sklearn`, by calling the `evaluate` method and passing in the test data. The method returns a list with the loss and accuracy of the model on the supplied data, which we simply print.

We can use the `save` method to write the model itself to disk for future use. Notice the file extension. Keras dumps the model in a custom file format that contains all the weights and biases of the model and the layer structure.

Running this code produces the output in Listing 10-4.

```
Using TensorFlow backend.
```

```
Model parameters = 1199882
```

Layer (type)	Output Shape	Param #
=====		
conv2d_1 (Conv2D)	(None, 26, 26, 32)	320
conv2d_2 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 12, 12, 64)	0
dropout_1 (Dropout)	(None, 12, 12, 64)	0
flatten_1 (Flatten)	(None, 9216)	0
dense_1 (Dense)	(None, 128)	1179776
dropout_2 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 10)	1290
=====		

```
Total params: 1,199,882
```

```
Trainable params: 1,199,882
```

```
Non-trainable params: 0
```

```
Train on 60000 samples, validate on 10000 samples
```

```
Epoch 1/12-loss:0.2800 acc:0.9147 val_loss:0.0624 val_acc:0.9794
```

```
Epoch 2/12-loss:0.1003 acc:0.9695 val_loss:0.0422 val_acc:0.9854
```

```
Epoch 3/12-loss:0.0697 acc:0.9789 val_loss:0.0356 val_acc:0.9880
```

```
Epoch 4/12-loss:0.0573 acc:0.9827 val_loss:0.0282 val_acc:0.9910
```

```
Epoch 5/12-loss:0.0478 acc:0.9854 val_loss:0.0311 val_acc:0.9901
```



```
Epoch 6/12-loss:0.0419 acc:0.9871 val_loss:0.0279 val_acc:0.9908
Epoch 7/12-loss:0.0397 acc:0.9883 val_loss:0.0250 val_acc:0.9914
Epoch 8/12-loss:0.0344 acc:0.9891 val_loss:0.0288 val_acc:0.9910
Epoch 9/12-loss:0.0329 acc:0.9895 val_loss:0.0273 val_acc:0.9916
Epoch 10/12-loss:0.0305 acc:0.9909 val_loss:0.0296 val_acc:0.9904
Epoch 11/12-loss:0.0291 acc:0.9911 val_loss:0.0275 val_acc:0.9920
Epoch 12/12-loss:0.0274 acc:0.9916 val_loss:0.0245 val_acc:0.9916
Test loss: 0.02452171179684301
Test accuracy: 0.9916
```

Listing 10-4: The MNIST training output

I've excluded some informational and warning messages from the lower-level TensorFlow toolkit and condensed the output to make it easier to follow in the text.

At the start of the run, Keras informs us that TensorFlow is our backend. It also shows us the shape of the training data, the now familiar 60,000 samples with a shape of $28 \times 28 \times 1$ ($\times 1$ since the images are grayscale). We have the usual 10,000 test samples as well.

Next comes a report on the model. This report shows the layer type, shape of the output from the layer, and the number of parameters in the layer. For example, the first convolutional layer uses 32 filters and 3×3 kernels, so the output with a 28×28 input will be $26 \times 26 \times 32$. The `None` listed for each layer is in the place where the number of elements in the minibatch normally is. The printout is showing only the relationship between the layers; because nothing in the architecture changes the number of elements in the minibatch, there's no need to explicitly mention the minibatch elements (hence the `None`). The parameters are $3 \times 3 \times 32$ for the filters plus an additional 32 bias terms for the 320 parameters listed.

As mentioned in Chapter 9, the lion's share of the parameters in the model are between the flatten layer and the dense layer. The layer named `dense_2` is the softmax layer mapping the 128 elements of the dense layer to the 10 elements of the softmax: $128 \times 10 + 10 = 1,290$, where the additional 10 are the bias terms. Note that the dropout and

pooling layers have no parameters because there is nothing to learn in those layers.

After the report on the model's structure, we have the verbose output of the training call to `fit`. We asked for 12 epochs (12 full passes through the training data) using a minibatch of 128 samples. The output lists the stats for each pass. We see that the loss goes down as we train, which is expected if the model is learning, and that the accuracy (`acc`) on the training data goes up. The validation data is used during training to test the model, but this data is not used to update the model's weights and biases. The loss on the validation data is also going down with each epoch, but more slowly. What we don't want to see here is the validation loss going up, though it will jump around somewhat, especially if the validation set is not very big. We see an opposite effect with the validation accuracy (`val_acc`); it's going up for each epoch of training. If the model were to start overfitting, this accuracy would go down after a certain point. This is the value of validation data: It tells us when to stop training.

The final two lines of output are the loss and accuracy of the model on the test samples passed to the `evaluate` method. Since the validation and test sets are the same in this example, these lines match the output for epoch 12. The final accuracy of this model is 99.16 percent—certainly a very good accuracy to see.

Plotting the Error

We can use the saved history to plot the loss or error ($1 - \text{accuracy}$) as a function of the training epoch. The plots are similar in shape, so I show only the error plot in Figure 10-1.

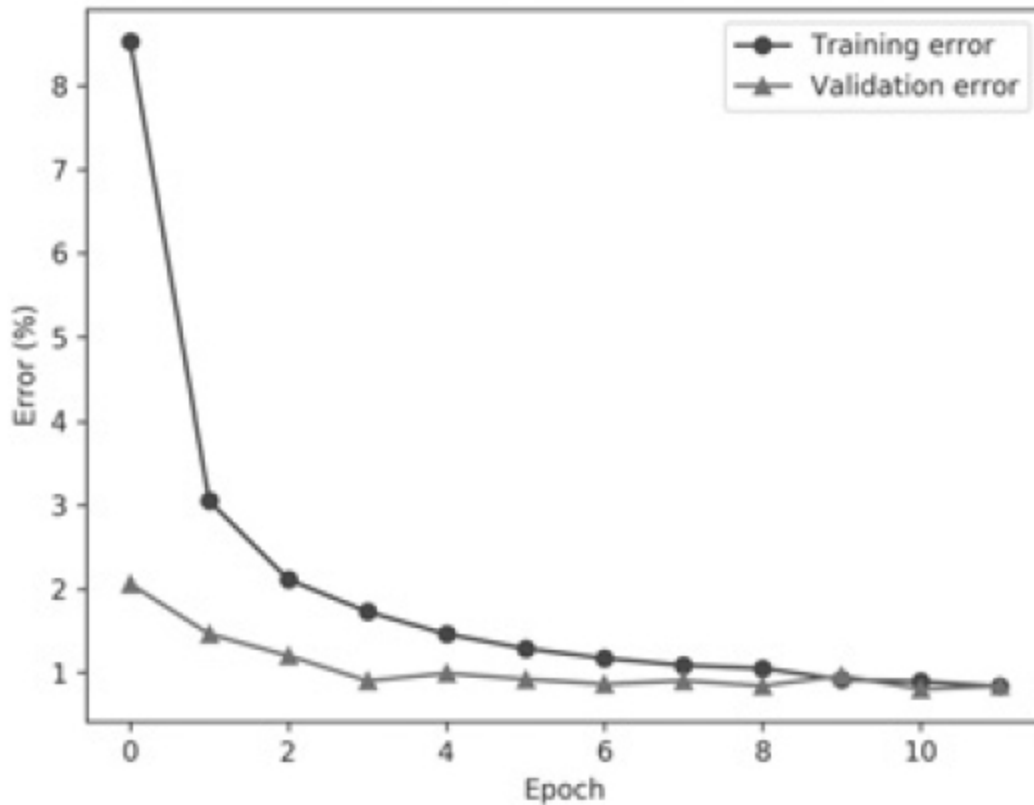


Figure 10-1: The MNIST training and validation errors as a function of epoch

The error on the training data falls off quickly and, as we’ve seen before, tends toward 0 as training continues. In this case, the validation error falls slightly and then levels off at a value similar to the training error. At this point in the book, you might have alarm bells going off in your head when you look at Figure 10-1. The initial training error is *greater than* the initial validation error!

The full cause of this is hard to pin down, but one component is using dropout in the network. Dropout is applicable only during training, and because of the dropping of nodes in layers, dropout is, in effect, training many models at once, which, initially, causes a large error before the model “settles down” and the error drops per epoch. We can see that this might be the case here because if we simply comment out the Dropout layers in Listing 10-4 and retrain, we get a new error plot: Figure 10-2.

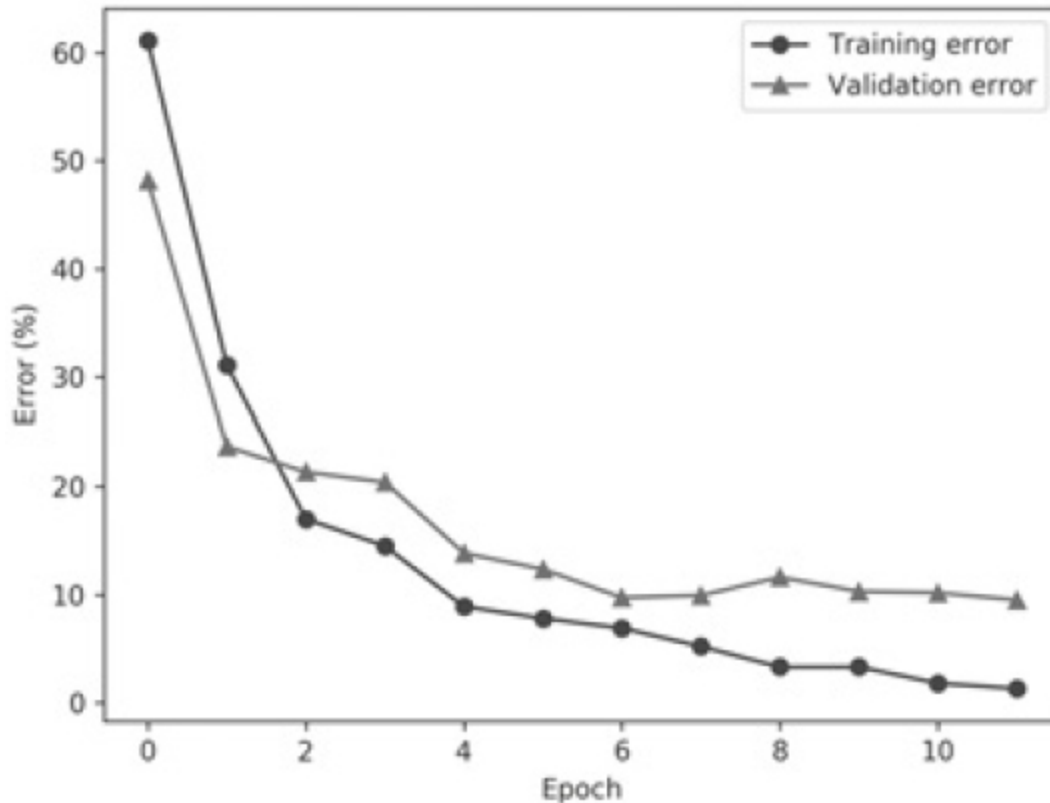


Figure 10-2: The MNIST training and validation errors as a function of epoch when no dropout layers are present

In this plot, we see that the validation error quickly becomes greater than the training error, as we would expect. Additionally, the final validation error is much greater than the final validation error for Figure 10-1, about 10 percent versus 1 percent. This is also something we expect if dropout is actually a sensible thing to use, which it is. Note also that by the 12th epoch, the training-set error is roughly the same regardless of the presence of dropout layers.

Finally, some of what we see in Figures 10-1 and 10-2 is due to the way Keras reports training and validation accuracies. The reported training accuracy (and loss) at the end of an epoch is the average over the epoch, but, of course, this changes as the model learns and tends to increase. However, the validation accuracy reported is for the model as it is at the end of the epoch, so at times it's possible for the training accuracy to be reported as less than the validation accuracy.

Now that you've seen how to build a simple CNN and run it on a dataset, we can start experimenting with CNNs. We could perform an infinite number of experiments; just look at the rate at which new papers on deep learning appear on sites like <https://arxiv.org> or the explosion of attendance at machine learning conferences. Therefore, we need to restrict ourselves to some basic explorations. Hopefully, these will motivate you to explore more on your own.

Basic Experiments

We did a bit of experimentation already when we removed the dropout layer. All our experiments follow the same general pattern: Make a slightly different version of the model, train it, and evaluate it against the test set. We will try three types of experiments. The first type modifies the architecture of the model; removing dropout layers falls into this category. The second type explores the interplay between training-set size, minibatch size, and epochs. The last type alters the optimizer used during training.

In all three cases, to avoid excessive code listings, we'll simply comment on the variation to the code in the previous section with the understanding that the remaining code is the same from experiment to experiment. We'll number the experiments, and you can match the results with the number to find the actual Python source code for the experiment. The code is available from the book's GitHub site.

In the previous section, we used the entire training set of 60,000 samples for training and the whole test set of 10,000 samples both for validation and as the final test set. Here we'll restrict ourselves to using the first 1,000 or 1,024 training samples as the entire training set. Additionally, we'll use the first 1,000 samples of the test set as the validation set and reserve the last 9,000 samples for the final test set we'll use when training is complete. We'll report the accuracy of these 9,000 images that were unseen during training. The results will include the baseline model accuracy and number of parameters, for comparison purposes.

Bear in mind that unless stated otherwise, the accuracies we present represent a single training session for each experiment. You should get slightly different results if you run these experiments yourself, but those slight differences shouldn't outweigh the larger differences in accuracy that will result from changing the model or training process.

Finally, the models in this section are multiclass, so we could examine the confusion matrices to see how the models are making their mistakes. However, it would be exceedingly tedious to do this for each experiment. Instead, we'll use the overall accuracy as our metric, trusting that it is a sufficient measure in this case.

Architecture Experiments

Architecture modifications imply removing or adding new layers or altering the parameters of a layer. We've made architecture modifications and compiled the resulting mean accuracies over 10 runs in Table 10-1.

Table 10-1: The Results from Modifying the Model Architecture

Experiment	Modification	Test accuracy	Parameters
0	Baseline	92.64%	1,199,882
1	Add Conv ₃ , 3×3×64 before Pooling	94.11%	2,076,554
2	Duplicate Conv ₂ , Pooling layer	94.04%	261,962
3	Conv ₁ , 3×3×32 to 5×5×32	93.57%	1,011,978
4	Dense layer to 1,024 nodes	93.37%	9,467,274
6	Second Dense layer with 128 nodes	92.54%	1,216,394

Experiment	Modification	Test accuracy	Parameters
8	Remove Pooling layer	92.27%	4,738,826
5	Conv ₁ , Conv ₂ , halve number of filters	91.65%	596,042
9	No ReLU after Conv layers	90.90%	1,199,882
10	Remove Conv ₂	90.41%	693,962
7	Dense layer to 32 nodes	90.36%	314,090

In Table 10-1, the baseline results and model size are given first, followed by the various experiments from best performing to worst performing. The experiment number corresponds to the model filename, meaning experiment 0 is *mnist_cnn_exp0.py* and experiment 10 is *mnist_cnn_exp10.py*. Let's look at the table and interpret the results.

First, we see that adding a third convolutional layer after the second convolutional layer (experiment 1) improves the performance of the model but also adds 876,672 parameters. Increasing the depth of the network seems to improve the performance of the model but at the expense of increasing the network's size.

However, in experiment 2 we also increase the depth of the network by duplicating the second convolutional layer and the following pooling layer, but because of the second pooling layer, the total number of parameters in the network goes down by 937,920. This is a substantial savings for virtually the same performance. This indicates that depth is good, but so is the judicious use of pooling layers to keep the number of parameters small. For this dataset, experiment 2 is a solid architecture to use.

Next, we see that an adjustment to the kernel size of the first convolutional layer, experiment 3, leads to an improvement relative to

the baseline. There are more parameters in the first convolutional layer (832 versus 320), but because of the edge effects when using an exact convolution, by the time we get to the output of the `Flatten` layer, there are now only 7,744 values versus 9,216 for the baseline model. This means that the large matrix between the `Flatten` and `Dense` layers goes from 1,179,776 down to 991,360 parameters with a net result that the model overall has 187,904 fewer parameters.

This is good: better performance and fewer parameters to learn. Is there a downside to the change of experiment 3? Not really. Instead, we might argue that adjusting the kernel size for the first convolutional layer has made the model more appropriate for the spatial information in the digit images, thereby making the new representation learned by the convolutional and pooling layers that much better at separating the classes. In general, there seems to be a best kernel size for the first convolutional layer, the layer that deals with the input to the model. That kernel size is related to the spatial structure of the inputs: Some sizes will be better at detecting input features that are better for separating the classes. This general rule does not appear to hold for higher convolutional layers, wherein the prevailing wisdom is to use 3×3 kernels for most convolutional layers except the first.

Can we combine experiment 3 and experiment 2? Certainly. We simply make the first convolutional layer of experiment 2 use a 5×5 kernel instead of a 3×3 kernel. If we do this, we get a model with an average accuracy of 93.81 percent over 10 runs that needs only 188,746 parameters. With this trivial change, we've achieved performance midway between experiment 2 and experiment 3 by using only 72 percent of experiment 2's parameter count.

You might be tempted to simply increase the size of the `Dense` layer, the layer that can be thought of as using the new feature representation discovered by the convolutional and pooling layers below it. However, doing so (experiment 4) results in no real improvement in overall accuracy, but with a substantial increase in the number of parameters. We know the cause: The $9,216\times 128$ weight matrix between the `Flatten` and `Dense` layers is now a $9,216\times 1,024$ matrix. For CNNs, we want to

create the best feature representation so that a simpler top layer can be used.

Experiment 6 adds a second `Dense` layer with 128 nodes for no gain in performance over the baseline but at the expense of an additional 16,512 weights to learn. Experiment 8 removes the pooling layer for no improvement and a huge price tag of 3,538,944 additional weights. Clearly, experiment 8 isn't the way to go.

With experiment 5, we can make the model significantly smaller, a reduction of 603,840 parameters, for a small reduction in overall accuracy by simply halving the number of filters learned in each of the convolutional layers: $32 \rightarrow 16$ for `Conv1` and $64 \rightarrow 32$ for `Conv2`. This is a good optimization provided the 1 percent difference in accuracy is acceptable. Looking again at Figure 9-8 on page 259, we can see that the responses are very similar for many filters, especially for the second convolutional layer with 64 filters. This implies that there are redundant filters that aren't adding much to the new feature representation presented to the `Dense` layers. Despite halving the number of filters learned, some filters still learn to capture the important aspects of the input data used to separate the classes.

Experiment 9 removes the ReLU activations after the convolutional layers. A change in model size occurs because of this as ReLU layers have no learnable parameters, but model accuracy decreases by 1.7 percent. Adding nonlinearities after convolutional layers is generally a good idea.

Experiment 10 removes the second convolutional layer, thereby reducing model size relative to the baseline, but also reducing the model's ability to generalize to unseen data (that is, the held-out test set). Similarly bad is experiment 7, which is the smallest of the models because of the small `Dense` layer with only 32 nodes, meaning the model needs to learn 884,736 fewer weights. (Do you see why?)

From all these results, do we have an architecture that is lean but highly effective? We do: It is experiment 2 with a 5×5 kernel for the first convolutional layer. To build this architecture in Keras, we need the code in Listing 10-5.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(5, 5),
                activation='relu',
                input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Listing 10-5: Building the architecture for experiment 2

We've simply duplicated the Conv2D, MaxPooling2D, and Dropout layers and used (5, 5) for the kernel size of the first layer.

If we train this model 10 times by using all 60,000 samples of the MNIST training set, we get a final, average held-out test-set accuracy of 99.42 percent for an error of 0.58 percent. This is using all 10,000 test samples. According to <https://paperswithcode.com>, a website that tracks current bests on various machine learning datasets, the state-of-the-art MNIST error is currently 0.13 percent, so we are not state of the art, but we are better than the 99.16 percent accuracy in Listing 10-4 for the default architecture.

Training-Set Size, Minibatches, and Epochs

These experiments examine the interplay between training-set size, minibatch size, and number of epochs. Our model will be the default model we were using previously, experiment 0, but this time we'll use 1,024 samples for the training set. We'll be using powers of two as the minibatch sizes; this is a convenient size, as all our minibatch sizes divide it evenly.

Recall that Keras, like sklearn, runs through a given number of epochs, or full passes through the training set. Additionally, the minibatch size (`batch_size`) specifies the number of samples used in each iteration, after which the average error (the cross-entropy loss) is used to update the parameters. Therefore, each processed minibatch leads to a single gradient-descent step, and the training-set size divided by the minibatch size is the number of gradient-descent steps taken per epoch.

For our experiments, we'll use the following minibatch sizes:

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024

For a training set of 1,024 samples (approximately 100 for each digit), the number of gradient-descent steps per epoch is the reverse of this list: 1,024 for a batch size of 1, down to 1 for a batch size of 1,024.

Let's generate two plots. The first will show the final average test-set accuracy for the two cases: a fixed number of gradient-descent steps regardless of minibatch size, and a fixed number of epochs irrespective of minibatch size. The second plot will show us the clock time to train the model for each case. The code leading to the plots is in *exp_fixed_steps.py* (fixed number of gradient-descent steps) and *exp_fixed_epochs.py* (fixed number of epochs). The plot points are produced by filtering the output of two shell scripts, *go_fixed_steps* and *go_fixed_epochs*, which run the Python code for each of the 11 minibatch sizes.

The test-set accuracy as a function of minibatch size is given as the mean of five runs in Figure 10-3, and the standard error of the mean given by the error bars. Notice that the x-axis is logarithmic base 2.

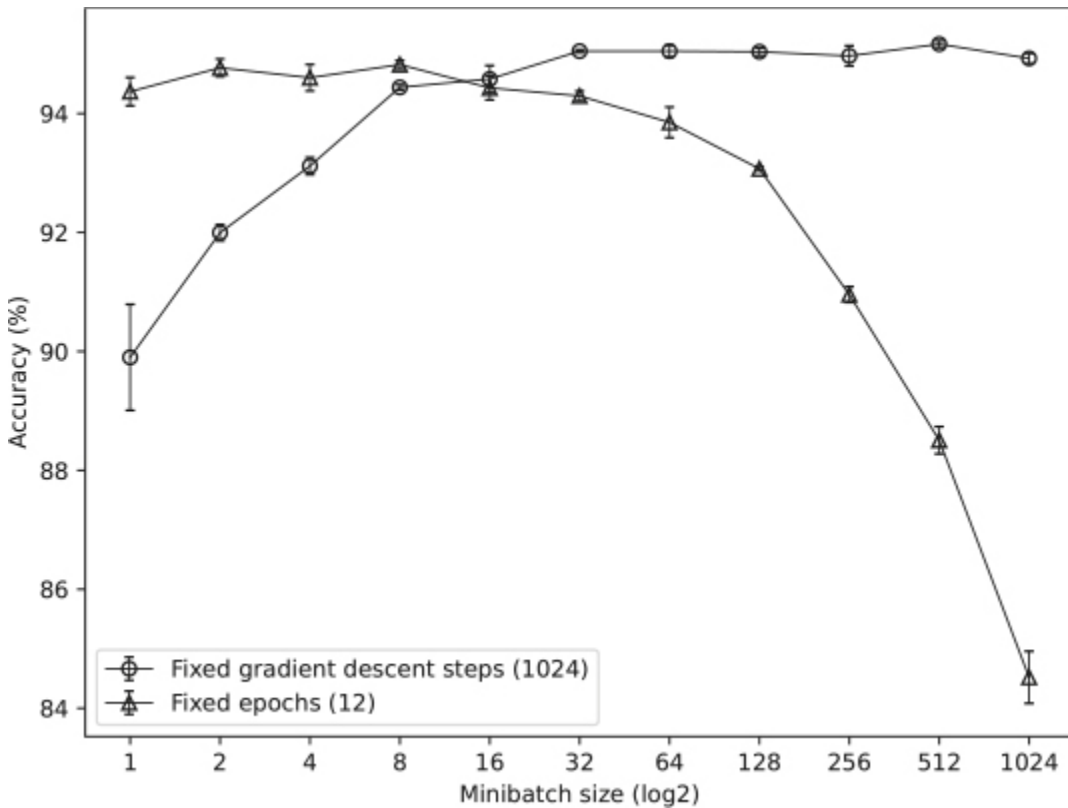


Figure 10-3: The MNIST test-set accuracy as a function of minibatch size and fixed gradient-descent steps or fixed epochs

Let's look at the fixed number of gradient-descent steps regardless of minibatch size (circles). In some toolkits, this is referred to as using a fixed number of iterations, where *iteration* means a single update leading to a gradient-descent step.

The number of gradient-descent steps is fixed at 1,024. This means that we need to change the number of epochs, depending on how many minibatches are in the training set. For the case of a single sample used per update (`batch_size=1`), we get the required 1,024 steps in a single epoch, so we set the number of epochs to 1. For a minibatch size of two, we get 512 steps per epoch, so we set the number of epochs in the code to 2 to get 1,024 steps overall. The pattern continues: To get the 1,024 gradient-descent steps for each minibatch size, we need to set the number of epochs to the minibatch size.

We see that except for the smallest minibatch sizes, when we fix the number of gradient-descent steps, we get consistent overall accuracies. After minibatch sizes of about 16, results don't change too much. The

general rule for CNNs is to use smaller batch sizes. This seems to help with model generalization for datasets that aren't as simple as MNIST, and, as you'll soon see, decreases training time significantly.

The second curve of Figure 10-3 shows the accuracy for a fixed number of epochs (triangles). If we change the minibatch size but do not increase the number of training epochs to make the number of gradient-descent steps remain constant, we'll be using larger and better estimates of the gradient, but we'll also be taking fewer steps. We see that doing this quickly results in a substantial decrease in accuracy.

We should not be surprised. With a minibatch size of one, training for 12 epochs gives us a model that took $12 \times 1,024 = 12,288$ gradient-descent steps. Granted, the gradient estimate was particularly noisy in this case since we use only one training sample to estimate it, but with lots of steps, we arrived at a well-performing model all the same. By the time we get to a minibatch of 1,024 samples, the size of our training set, we see that we took only 12 gradient-descent steps before calling it a day. No wonder the results are so poor.

The second plot of this section is Figure 10-4.

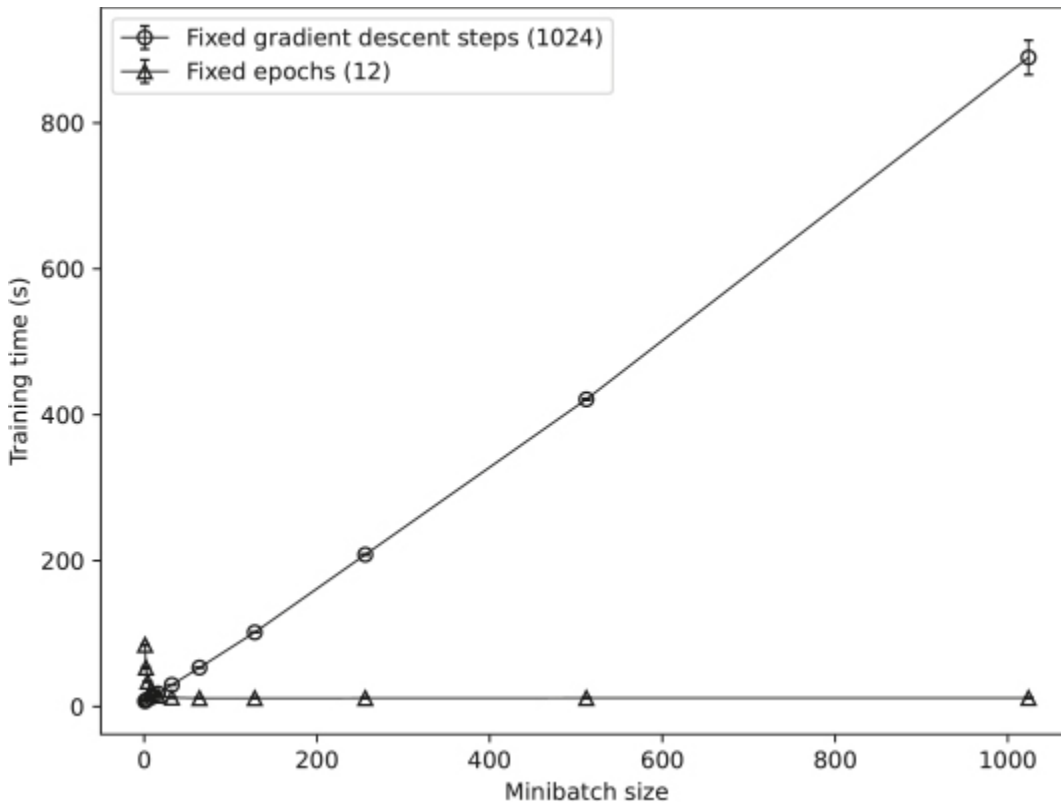


Figure 10-4: The model training time as a function of minibatch size and fixed gradient-descent steps or fixed epochs

As before, let's start with the fixed number of gradient-descent steps (circles). We see immediately that the training time is linearly proportional to the minibatch size. This is reasonable since we just saw that we need to increase the number of epochs to keep the number of gradient-descent steps constant while increasing the minibatch size. Therefore, the amount of data passed through the network is increasing proportionally, so the time required for the forward and backward training passes will increase proportionally as well. From this, we see that large minibatch sizes cost clock time.

For a fixed number of epochs, we see a different story. For tiny minibatch sizes, training time goes up because of the number of forward and backward passes. For a minibatch of 1, we need 12,288 such passes, as we just saw. However, by the time we get to minibatches of even 32 samples at a time, we have only $1,024/32 = 32$ passes per epoch, for a total of 384 for the entire training session. This is far fewer than for the smallest minibatch size, so we might expect the training time for

minibatches this size or larger to be roughly constant in time, as we see in Figure 10-4.

What can we glean from these two plots? The following: To balance runtime and accuracy, we want to use minibatch sizes that are large enough to give us a reasonable estimate of the gradient but small enough that, for a fixed number of model updates (gradient-descent steps), we can train quickly. This argues for minibatch sizes in the 16 to 128 range, in general. Indeed, a review of the deep learning literature sees minibatches in this range almost exclusively for most applications.

For this example, based on Figure 10-3 (circles), minibatch sizes above 16 result in models that are all basically the same in terms of accuracy. But according to Figure 10-4, the training time for a model using a minibatch size of 16 compared to one of 1,024 is a few seconds versus approximately 14 minutes.

Optimizers

So far, all our experiments have used the same gradient-descent algorithm, or optimizer: Adam. Let's take a look at how our MNIST model does if we change the optimizer but leave everything else fixed.

Our model is still experiment 0, the model we've been using all along in this chapter. We'll continue to use 1,000 test samples for validation and 9,000 to determine our final accuracy. However, instead of using the first 1,000 training samples, we'll increase the number to the first 16,384 samples. We fix the minibatch size at 128 and the number of epochs to 12, as before. We'll report the results as mean and standard error over five runs.

Let's test the following optimizers: SGD, RMSprop, Adam, and Nadam. We leave the parameters at their default settings except for SGD, where we set the momentum to 0.9, a typical value, and we set a learning rate of 0.01 for SGD and RMSprop. The code is found in *mnist_cnn_optimizers.py* and is used by the *go_optimizers* script to produce the plot values.

Figure 10-5 shows the test-set accuracy (top) and training time (bottom) for each optimizer.

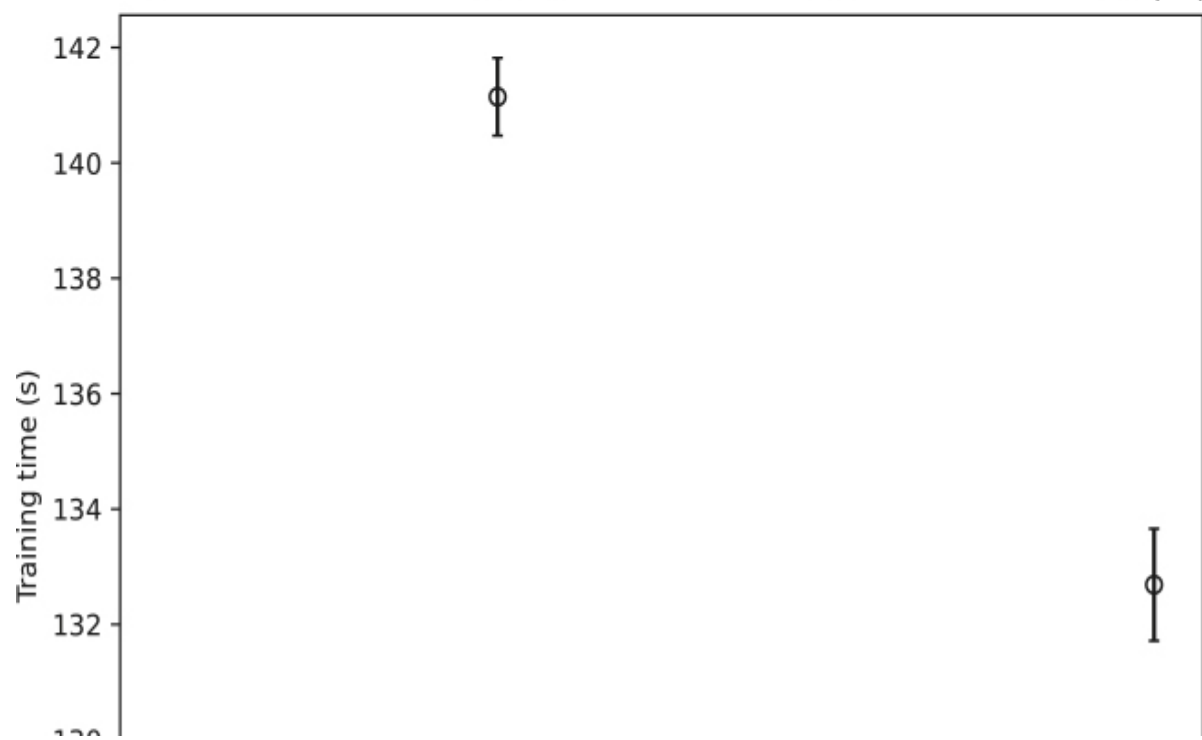
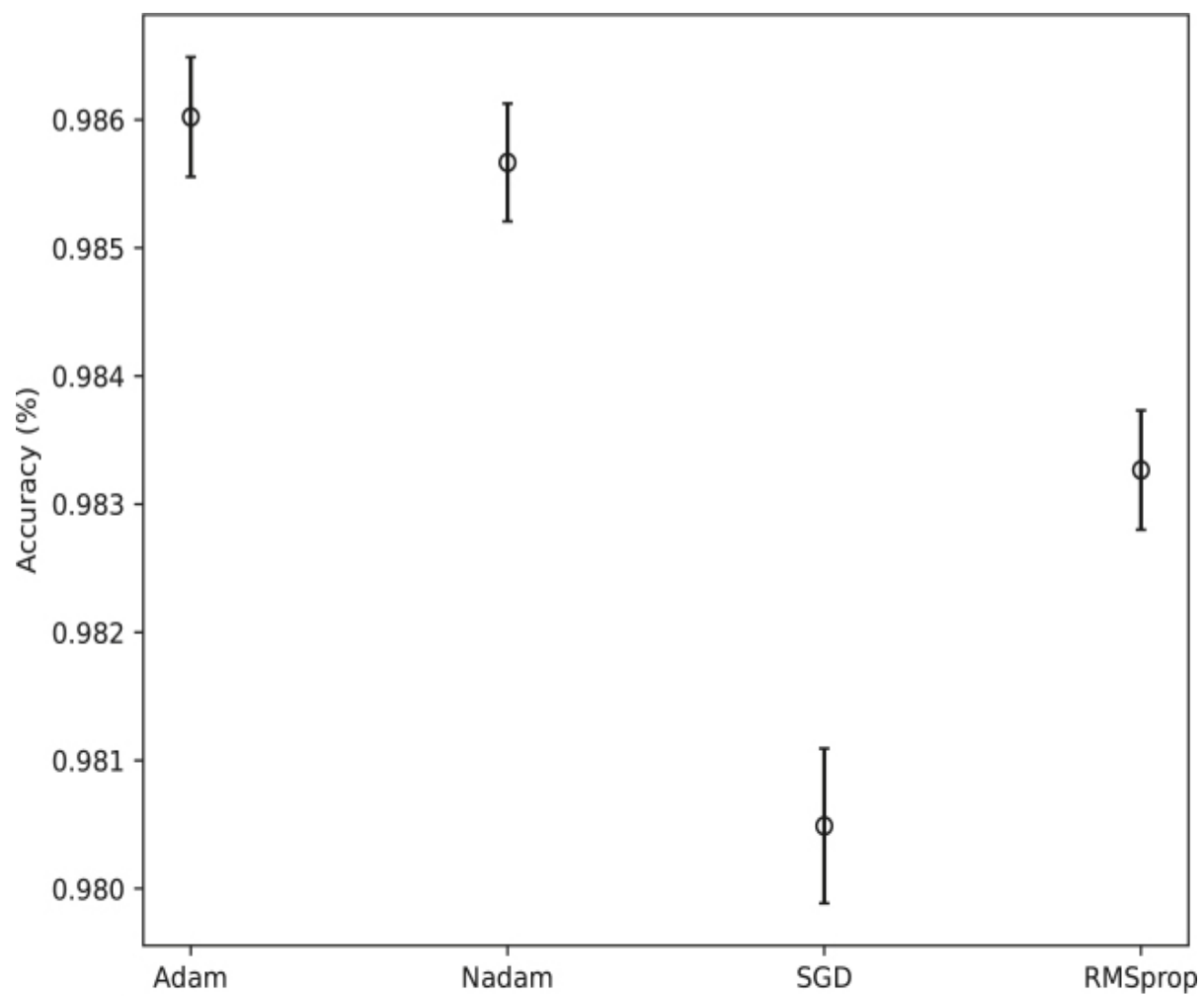


Figure 10-5: The test-set accuracy (top) and training time (bottom) by optimization algorithm

First, notice that there isn't too much difference between the results produced by each optimizer. This is good news. However, by looking at the error bars, it seems clear that Adam and Nadam perform slightly better than SGD or RMSprop. This is borne out in the deep learning literature as well, though each dataset should be looked at independently. Nadam is merely a slight variation on Adam, so we might expect the two to perform similarly.

In terms of training time, the different optimizers are also roughly equivalent, though Nadam is consistently slower than the others. This performance difference might be important for a very large dataset.

This section has been thorough in terms of minute details associated with changing model architectures and training parameters. Hopefully, it's helped you develop intuition about how to configure a CNN and how to train it. Rules of thumb are hard to present in this area, but I have given some general guidance. Exceptions to these rules abound, however, and you have to simply try things, observe the results, and adapt when presented with a new dataset.

Let's move on now from classifying simple inputs to answering the question of how to make a model that can locate targets in arbitrary images.

Fully Convolutional Networks

Chapter 9 introduced fully convolutional networks. Let's take our basic MNIST CNN model and convert it to a fully convolutional version and see how we can use it to locate digits in larger images.

Our basic approach is to train the model by using fully connected layers, as before, and then to create a fully convolutional version and update the weights with those from the fully connected model. We can then apply the new model to arbitrary size inputs to locate digits (hopefully). To follow along, look at *README.txt* in the *mnist_large* directory and run the code in the order listed there.

Building and Training the Model

First, we train our base model on the full MNIST dataset. The only change we'll make is to train for 24 epochs instead of just 12. The output of this process is a file that contains the trained weights and biases. All we then need to do is create the fully convolutional version by changing the fully connected layer and copy the weights and biases from the old model to the new. The code for this is straightforward, as in Listing 10-6.

```
from keras.models import Sequential, load_model
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
❶ weights = load_model('mnist_cnn_base_model.keras').get_weights()

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                activation='relu',
                ❷ input_shape=(None, None, -1)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

❸ model.add(Conv2D(128, (12, 12), activation='relu'))
model.add(Dropout(0.5))

❹ model.add(Conv2D(10, (1, 1), activation='softmax'))

❺ model.layers[0].set_weights([weights[0], weights[1]])
model.layers[1].set_weights([weights[2], weights[3]])
model.layers[4].set_weights([weights[4].reshape([12,12,64,128]), weights[5]])
model.layers[6].set_weights([weights[6].reshape([1,1,128,10]), weights[7]])

model.save('mnist_cnn_fcn_model.keras')
```

Listing 10-6: Creating the trained fully connected model

After importing the necessary Keras modules, we load the trained weights from the fully connected model ❶. Then, we construct the fully convolutional version much as we did for the fully connected version. However, some key differences exist. The first has to do with the input convolutional layer ❷. In the fully connected model, we specified the input image size here, 28×28 pixels with one channel (grayscale). For the fully convolutional case, we do not know the size of the input, so we use `None` instead for the width and height. We do know that the input will be a single channel image, so we leave the 1.

Since the `Dense` layers are the reason we have to use fixed-size inputs, we replace them with equivalent convolutional layers ❸. We replace

```
model.add(Flatten())  
model.add(Dense(128, activation='relu'))
```

with this:

```
model.add(Conv2D(128, (12, 12), activation='relu'))
```

The (12, 12) is the output size of the max-pooling layer, and the 128 is the number of filters to learn that stand in for the 128 nodes we had before. Again, the critical point here is that the output of this convolutional layer is 1×1×128 because convolving a 12×12 input with a 12×12 kernel produces a single output value. The difference is that the convolutional layer is not tied to any fixed input size as the combination of `Flatten` and `Dense` was.

The final softmax layer also needs to be made fully convolutional ❹. There are 10 outputs, 1 per digit, and the activation remains the same. The kernel size, however, is 1×1. The input to this layer is 1×1×128, so the kernel size to cover it is 1×1. Again, if we were to work through the math, we'd see that a 1×1×128 input to a 1×1×10 convolutional layer matches that of a fully connected layer of 128 nodes mapped to 10 nodes in the next layer. The difference here is that if the input is larger than 1×1, we can still convolve over it.

Now that we've constructed the fully convolutional version of our model, we need to copy the weights from the trained fully connected

model to it ⑤. We loaded the trained weights into `weights`. This is a list of NumPy arrays for the weights and biases, layer by layer. So, `weights[0]` refers to the weights of the first `conv2D` layer, and `weights[1]` are the biases. Similarly, `weights[2]` and `weights[3]` are the weights and bias values for the second convolutional layer. We set them in the new fully convolutional model by updating the proper layers via the `set_weights` method. Layers 0 and 1 are the two convolutional layers.

Layer 4 is the new `conv2D` layer that replaced the `Flatten` and `Dense` layers of the original model. Here when we set the weights, we need to reshape them to match the form of a convolutional layer: $12 \times 12 \times 64 \times 128$. This is for a 12×12 kernel mapped over 64 inputs leading to 128 outputs. The 64 is the number of 12×12 outputs from the pooling layer above.

Finally, we set the output layer weights. Again, we need to reshape them to $1 \times 1 \times 128 \times 10$ for the $1 \times 1 \times 128$ input and 10 outputs. The biases for the two new `conv2D` layers are in `weights[5]` and `weights[7]`, so we add them as well.

The fully convolutional model is now defined and completely populated with the weights and biases from the fully connected model. Figure 10-6 shows the mapping between models, with the original architecture on the left and the fully convolutional architecture on the right. The boxes represent layers, with the top set of numbers being the input and the bottom the output. For the fully convolutional model, input height and width are arbitrary and marked with “--”.

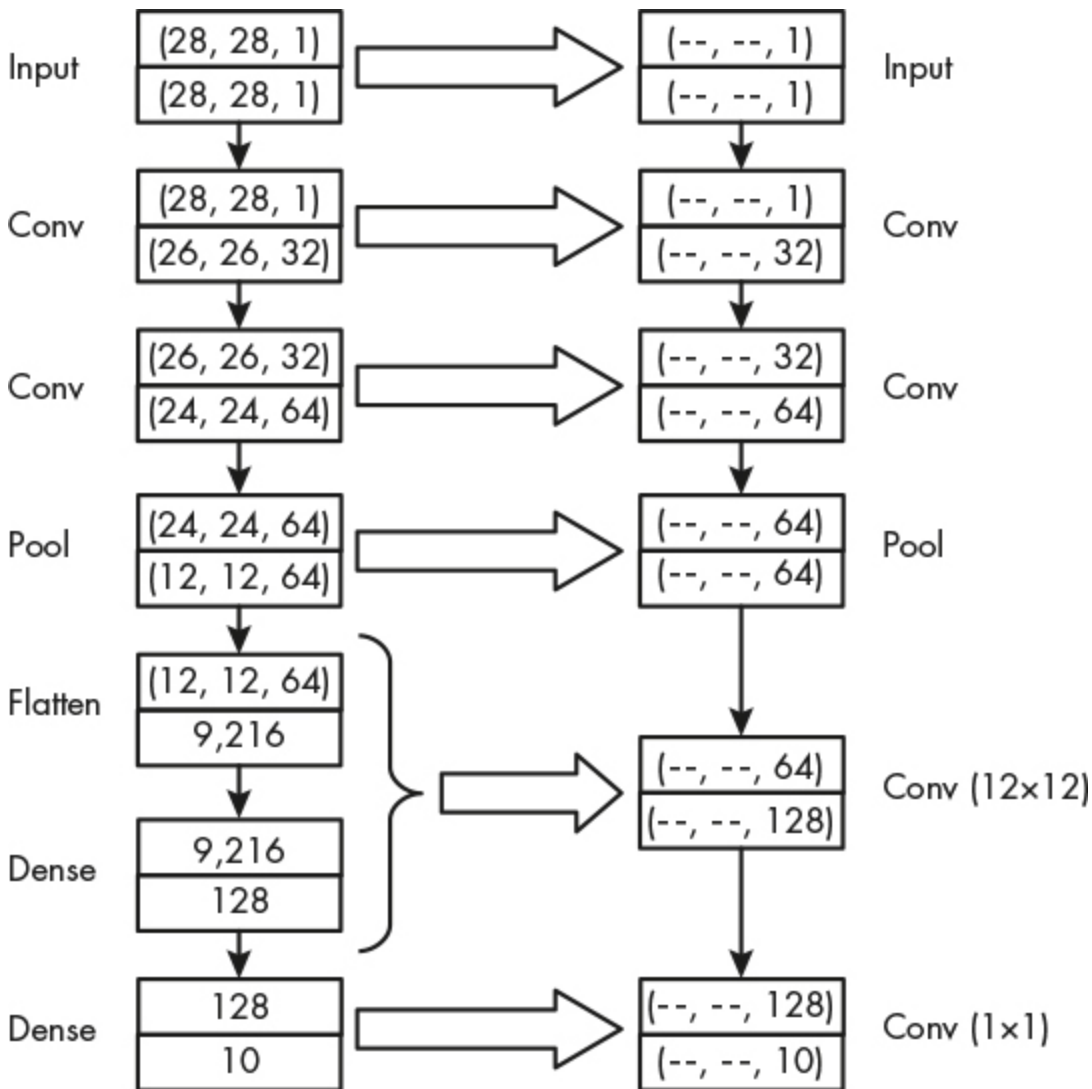


Figure 10-6: Mapping a fully connected model (left) to a fully convolutional model (right)

All that's left to do is write the new fully convolutional model to disk, and it's ready to use. Let's see how.

Making the Test Images

To test the fully convolutional model, we first need images with digits. Unlike our training images, which were small and had a single digit in the center, we want larger test images that contain many digits in arbitrary locations. The MNIST dataset consists of shades of gray on a black background; therefore, our test images should have a black background also. This will make the test images come from the same “domain” as the training images, which, as we've emphasized before, is

critical. Making models adapt to different data domains is an active research area. Search for *domain adaptation*.

Making the test images by using Python and the digits from the MNIST test set is straightforward. We didn't use the test-set images for training, so using them to make our larger test images isn't cheating. Listing 10-7 shows the code.

```
import os
import sys
import numpy as np
import random
from PIL import Image

os.system("rm -rf images; mkdir images")

if (len(sys.argv) > 1):
    N = int(sys.argv[1])
else:
    N = 10

x_test = np.load("data/mnist/mnist_test_images.npy")

for i in range(N):
    ❶ r, c = random.randint(6, 12), random.randint(6, 12)
    g = np.zeros(r*c)
    ❷ for j in range(r*c):
        if (random.random() < 0.15):
            g[j] = 1
    g = g.reshape((r, c))
    g[:,0] = g[0,:] = g[:,-1] = g[-1,:] = 0

    ❸ img = np.zeros((28*r, 28*c), dtype="uint8")
    for x in range(r):
        for y in range(c):
            if (g[x, y] == 1):
                ❹ n = random.randint(0, x_test.shape[0])

                im = x_test[n]
```

```
img[28*x:(28*x+28), 28*y:(28*y+28)] = im
```

```
Image.fromarray(img).save("images/image_%04d.png" % i)
```

Listing 10-7: Building large MNIST test-set images

We're using the MNIST test-set file we created in Chapter 2. We could just as quickly have loaded the test images through Keras, as we did earlier for our basic CNN experiments. The code itself creates an output directory, *images*, and gets the number of images to build from the command line, if given (*N*).

The images are of random sizes ❶. Here *r* and *c* are the number of rows and columns in the large image in terms of the number of 28×28 MNIST digits. To decide where to place our digits so they don't overlap, we create a grid, *g*, with either a 0 or a 1 in each possible digit position (*r***c* of them) ❷. There is a 15 percent chance that any grid position will contain a 1. We then reshape the grid into an actual 2D array and set the border positions of the grid to 0 to ensure that no digits appear on the edge of the image.

The actual output image is then defined ❸ as the number of rows and columns are multiplied by 28, the width and height of the MNIST digit. We loop over each digit position (*x* and *y*), and if the grid value at that row and column is 1, then we select a digit at random and copy it to the current row and column digit position in the output image (*img*) ❹. When every grid position has been examined, the image is written to disk so that we can use it with our fully convolutional network.

Testing the Model

Let's test the model—first on single MNIST digits and then on the randomly generated large digit images. The fully convolutional model should work as well with single MNIST digits as the fully connected model did. The code to test this assertion is in Listing 10-8.

```
import numpy as np
from keras.models import load_model
```

```

x_test = np.load("data/mnist/mnist_test_images.npy")/255.0
y_test = np.load("data/mnist/mnist_test_labels.npy")
model = load_model("mnist_cnn_fcn_model.keras")

N = y_test.shape[0]
nc = nw = 0.0
for i in range(N):
    ❶ p = model.predict(x_test[i][np.newaxis,:,:,np.newaxis])
    c = np.argmax(p)
    if (c == y_test[i]):
        nc += 1
    else:
        nw += 1
print("Single MNIST digits, n=%d, accuracy = %0.2f%%" % (N, 100*nc/N))

```

Listing 10-8: Verifying that the fully convolutional model works with single MNIST digits

We load the MNIST test images and labels, along with the fully convolutional model, and then loop over each test image and ask the model to make a prediction ❶. Note, the image is 2D, but we must pass a 4D array to the `predict` method, hence using `np.newaxis` to create the missing axes. The prediction for the digit is stored in `p` as a vector of per-class probabilities. The label associated with the largest of these probabilities is the label assigned to the input digit by the model `c`. If `c` matches the actual test label, we increment the number of correct predictions (`nc`); otherwise, we increment the number of wrong predictions (`nw`). Once all 10,000 test images have been processed, we can output the overall accuracy, which is 99.35 percent for my training of the fully convolutional model.

Okay, the fully convolutional model is highly accurate, but so what? We passed single-digit images to it as inputs and got a single output value. We had this capability before with the fully connected model. To expose the utility of the fully convolutional model, let's now pass the large MNIST digit images as input. Listing 10-9 shows how we do this in code.

```

import os
import numpy as np
from keras.models import load_model
from PIL import Image

model = load_model("mnist_cnn_fcn_model.keras")

os.system("rm -rf results; mkdir results")
n = len(os.listdir("images"))

for i in range(n):
    f = "images/image_%04d.png" % i
    ❶ im = np.array(Image.open(f))/255.0
    p = model.predict(im[np.newaxis,:,:,:np.newaxis])
    np.save("results/results_%04d.npy" % i, p[0,:,:,:])

```

Listing 10-9: Running the fully convolutional model over large test images

We import the necessary modules and then load the fully convolutional model. We create a new output directory, *results*, and find the number of large digit images (*n*). Next, we loop over each of the large digit images.

After loading the image from disk, being careful to make a NumPy array from it and scaling it by 255 since the training data was also scaled by 255 ❶, we make a prediction and store the model output in *p*. Notice, we make a 4D input to *predict*, just as we did for the single digits earlier, but this time, *im* is larger than 28×28 and contains multiple digits. Because the model is fully convolutional, this isn't an issue; we will not get an error. Instead, *p* is a 4D array with the first dimension of 1, the number of input images, and a final dimension of 10, the number of digits. The middle two dimensions of *p* are a function of the size of the input passed to the *predict* method. Since the input was larger than 28×28 pixels, the entire model convolved over the input image as though the model was a convolutional layer with a kernel of 28×28. Specifically, the output of this convolution has height and width of the following, where *H*, *W* are the height and width of the input image and *h*, *w* are the height and width of the output array from *predict*:

$$h = \frac{H-28}{2} + 1, \quad w = \frac{W-28}{2} + 1$$

The 28 in the formula is the size of the inputs we initially trained on, 28×28-digit images. Where did the mysterious 2 come from in the denominator? This is the stride of the 28×28 kernel over the input image. It is 2 because that is the factor the input image is changed by when it gets down to the fully convolutional output layers. The input was 28×28, but, after the two convolutional layers and the pooling layer, the input is mapped to 12×12 and $\lfloor 28 / 12 \rfloor = 2$.

We stated that the array in ρ is 4D; now we know we get a specific-size output based on convolving a 28×28 region over the input image by using a stride of 2. What do we get at each of the h, w output array positions? The last element of the 4D output has size 10; these are the per-class predictions at the specific h, w output position that corresponds to the 28×28 kernel.

Let's make this abstract description more concrete. The upper-left corner of Figure 10-7 shows one of the large input images that we've inverted to make it black on a white background and added a border so you can see the full size of the image.

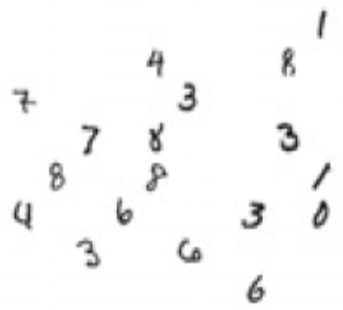
	0	1
2	3	4
5	6	7
8	9	

Figure 10-7: The per-digit heatmap output of the fully convolutional model for the input image on the upper left. The model was trained on the standard MNIST dataset.

The image in the top left of Figure 10-7 is 336 pixels wide and 308 pixels tall, meaning the output from passing this image to the model will be an array that is $1 \times 141 \times 155 \times 10$, exactly what we expect from the equations for the output array dimensions. The output array represents the model's predictions at each of the 28×28 regions of the input image when using a stride of 2. There is one prediction for each digit. For example, if `p` is the 4D output of the `predict` method for the image on the left of Figure 10-7, then `p[0,77,88,:]` will return a 10-element vector representing the per-class probabilities of each digit class for the 28×28 input region of the image that maps to 77×88 . In this case, we get the following:

```
array([0.10930195, 0.12363277, 0.131005   , 0.10506018, 0.05257199,
       0.07958104, 0.0947836  , 0.11399861, 0.08733559, 0.10272926],
      dtype=float32)
```

This tells us that there is no strong likelihood, according to the model, that any particular digit is present at this location. We know this because all the output probabilities are much lower than even the minimum cutoff threshold of 0.5. Think of the output of `predict` as a probability map, typically called a *heatmap*, giving us the probability that a digit is at that location. The model's output can be thought of as 10 heatmaps, one for each digit.

The remaining images of Figure 10-7 show the heatmaps for each of the 10 digits, again inverted so that higher probabilities are darker. The heatmaps were thresholded at 0.98, meaning any probability value less than 0.98 was set to 0. This removes the weak outputs like the ones we just saw. We are interested only in the model's strongest responses per digit.

To make the heatmaps, we double the size of the output from the model and set the output image locations with an offset to account for the position of the convolutional output. This is akin to what we saw in Figure 9-1 on page 240, where the convolution operation returns an output that is smaller than the input when no zero-padding is used.

Specifically, the code that produces the digit heatmaps is in Listing 10-10.

```
import os
import sys
import numpy as np
from PIL import Image

❶ threshold = float(sys.argv[1])
   iname = sys.argv[2]
   rname = sys.argv[3]
   outdir = sys.argv[4]
   os.system("rm -rf %s; mkdir %s" % (outdir, outdir))

❷ img = Image.open(iname)
   c, r = img.size
   hmap = np.zeros((r, c, 10))
   res = np.load(rname)

   x, y, _ = res.shape
   xoff = (r - 2*x) // 2
   yoff = (c - 2*y) // 2

❸ for j in range(10):
    h = np.array(Image.fromarray(res[:, :, j]).resize((2*y, 2*x)))
    hmap[xoff:(xoff+x*2), yoff:(yoff+y*2), j] = h
    np.save("%s/graymaps.npy" % outdir, hmap)
❹ hmap[np.where(hmap < threshold)] = 0.0
   for j in range(10):
       img = np.zeros((r, c), dtype="uint8")
       for x in range(r):
           for y in range(c):
               ❺ img[x, y] = int(255.0*hmap[x, y, j])
       img = 255-img
       Image.fromarray(img).save("%s/graymap_digit_%d.png" % (outdir, j))
```

Listing 10-10: Building heatmap images

We're calling the output images *graymaps* because they're grayscale images representing the response of the model to different locations in the input image. We first pass in the threshold value, the source image name, the response of the model to that source image, and an output directory where the graymaps will be written ❶. This directory is overwritten each time.

Next, the source image is loaded to get its dimensions ❷. These are used to create the output heatmaps (*hmap*). We also load the associated model responses (*res*) and calculate the offsets. Note that *hmap* is the same size as the image. We then fill in each digit graymap of *hmap* with the resized model response ❸ and store the full set of graymaps in the output directory.

To make the output grayscale images like those shown in Figure 10-7, we first threshold the heatmaps, setting any value less than the supplied cutoff to 0 ❹. For each digit, we create an output image and simply scale the remaining heatmap values by 255 since they are probabilities in the range $[0, 1)$ ❺. Then, before writing the image to disk, we invert by subtracting from 255. This makes stronger activations dark and weaker activations lighter. Because of the strong threshold applied (0.98), our output graymaps are effectively binary; this is what we want, to indicate where the model is most certain of a digit being located.

Let's look back at Figure 10-7 and see if we can interpret these responses. The source image has one 0 on the lower right. If we look at the graymap for digit 0, we see a single dark blob in that location. This means the model has indicated a strong response that a 0 digit is at that location. So far, so good.

However, we also see another strong response from the model near the 4 on the left side of the input image. The model has made a mistake. The input has two 4s in it. If we look at the graymaps for digit 4, we see two dark blobs corresponding to these digits, but we also see many other small areas of strong activation near other digits that are not 4s.

The model we trained was over 99 percent accurate on single MNIST test digits, so why are the responses of the fully convolutional

model so noisy? Just look at all the small strong responses for 2s when the input does not contain any 2s. Sometimes the model is doing well, such as for 8s, as the graymap shows strong responses for all the 8s in the input, but then it does poorly for other digits like 7s. And, there are no 5s at all, but the model is returning many hits.

Here is an opportunity for us to expand our thinking and intuition. We trained the model on the standard MNIST digit dataset. All the digits in this dataset are well centered in the images. However, when the model is convolved over the large input image, there will be many times when the input to the model is not a well-centered digit but only part of a digit. The model has never seen partial digits, and since it must give an answer, it sometimes offers answers that are meaningless; the part of a digit it sees may be part of a 6, for example, but the model “thinks” it is a 5.

One possible solution is to teach the model about partial MNIST digits. We can do this by augmenting the standard MNIST dataset with shifted versions of its digits. Imagine a 4 shifted to the lower right so that only part of it is visible. It will still be labeled a 4, so the model will have an opportunity to learn what a shifted 4 digit looks like.

The code to make this shifted dataset is in *make_shifted_mnist_dataset.py*, but we’ll show only the function that makes a shifted copy of an input MNIST digit here. This function is called four times for each training and test image (to create a shifted test dataset). We keep the original centered digit and four randomly shifted copies of it to make a dataset that is five times as large as the original. The random shift function follows, with *im* being the input image supplied as a NumPy array:

```
def shifted(im):
    r, c = im.shape
    x = random.randint(-r//4, r//4)
    y = random.randint(-c//4, c//4)
    img = np.zeros((2*r, 2*c), dtype="uint8")
    xoff = r//2 + x
    yoff = c//2 + y
    img[xoff:(xoff+r), yoff:(yoff+c)] = im
```

```
img = img[r//2:(r//2+r), c//2:(c//2+c)]  
return img
```

Based on the input size, we pick random x and y shifts that can be positive or negative and up to one-quarter of the image size. Varying this limit to, say, one-third or one-half would be worth the experiment. A new image is created (`img`), which is twice the size of the original. The original is then put into the larger image at an offset based on the shift positions, and the center portion of the larger image, matching the input image dimensions, is returned as the offset version of the input.

To use the augmented dataset, we first need to retrain the fully connected MNIST model, then rebuild the fully convolutional model by using the new weights and biases and, finally, run the large test images through the model as before. Doing all this leads to new graymaps (Figure 10-8).

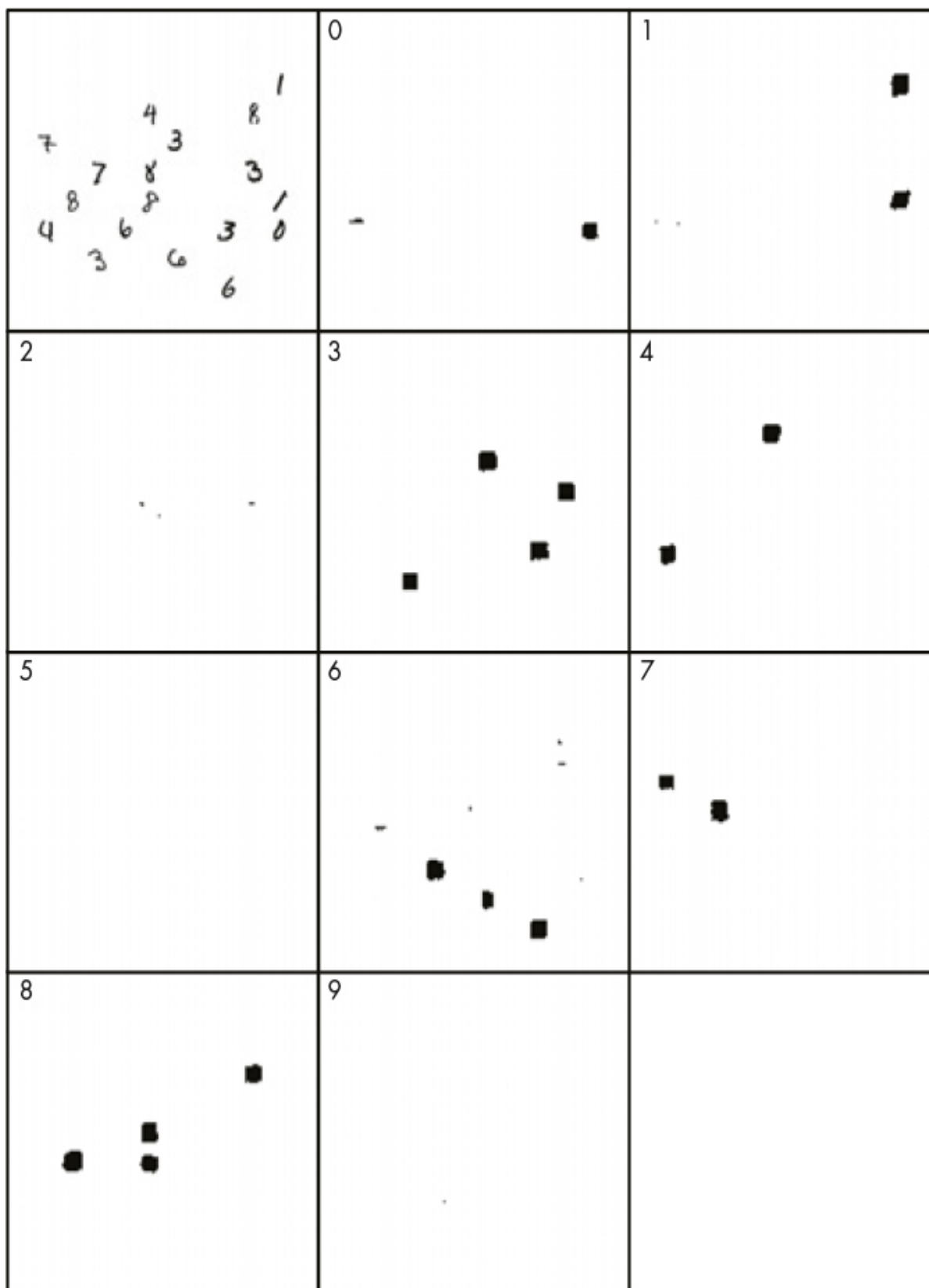


Figure 10-8: The per-digit heatmap output of the fully convolutional model for the upper-

left input image. The model was trained on the MNIST dataset augmented by shifting the digits.

We see a vast improvement, so we have impressive evidence that our intuition was correct: The initial model was unable to deal effectively with partial digits, but when we trained it with partial digits included, the resulting responses were robust over the actual digits and very weak to nonexistent for other digits. We should not be surprised by these results. Our first model did not represent the space of inputs that the model would see when used in the wild. It knew nothing about partial MNIST digits. The second model was trained on a dataset that is a better representation of the space of possible inputs, so it performs significantly better.

Using fully convolutional networks in this way has been superseded by more-advanced techniques that localize objects in images with bounding boxes or by assigning every pixel to a class. For example, the YOLO model is capable of real-time object detection in images via bounding boxes. In Chapter 15, we'll explore models producing bounding boxes and per-pixel labels.

Scrambled MNIST Digits

Chapter 7 showed that scrambling the order of the pixels in an MNIST digit (Figure 4-3 on page 120), provided the remapping of pixels is deterministically applied to each image, poses no problem for a traditional neural network. It is still able to train well and assign class labels just as effectively as with unscrambled digits (see Figure 7-9 on page 202).

Let's see if this still holds with CNNs. We made the scrambled MNIST digit dataset in Chapter 2. All we need to do here is substitute it for the standard MNIST dataset in our baseline CNN model, the one we started the chapter with. If we train this model with the scrambled data ($n = 16,384$), and do so five times to get some error bars, we get Figure 10-9.

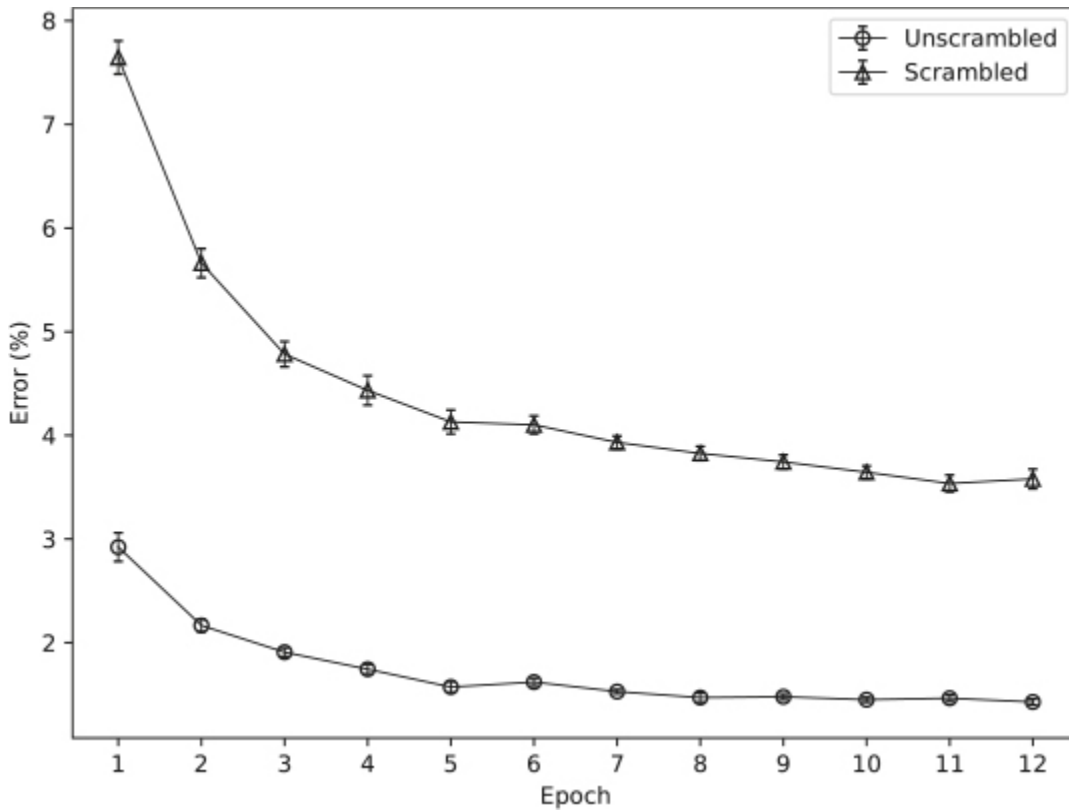


Figure 10-9: The test-set error per epoch for a model trained on unscrambled and scrambled MNIST digits (mean $\pm$ SE, $n = 5$)

Here we see that unlike the traditional neural network, the CNN has some trouble: The test error for the scrambled digits is higher than for the unscrambled digits. Why? Recall, the CNN uses convolution and learns kernels that help create a new representation of the input, one that a simple model, the top layers, can readily use to distinguish between classes.

Convolution generates responses that are spatially dependent. For the scrambled digits, that spatial dependence is mostly eliminated; it is only by considering the digit image as a whole, like a traditional neural network, that a class determination can be made consistently. This means there is little for the lower layers of the CNN to learn. Of course, the CNN is still learning, but the distinction between scrambled and unscrambled is more significant.

Summary

In this chapter, we built our intuition around CNNs by working with the MNIST dataset. We explored the effect of basic architecture changes; learned about the interplay among training-set size, minibatch size, and number of training epochs; and explored the effect of the optimization algorithm.

We saw how to convert a model using fully connected layers into a fully convolutional model. We then applied that model to search for digits in arbitrarily sized input images. You learned that we needed to increase the expressiveness of our dataset to do a better job of representing the distribution of inputs the model sees when used.

Finally, we saw via an experiment with the scrambled MNIST digits that the strength of CNNs—their ability to learn spatial relationships within data—can sometimes be of little help when the spatial relationships are weak or nonexistent.

In the next chapter, we'll continue our exploration of basic CNNs with a new dataset, one of actual images: CIFAR-10.

OceanofPDF.com

11

EXPERIMENTS WITH CIFAR-10



In this chapter, we'll perform a series of experiments with the CIFAR-10 dataset we built in Chapter 2. First, we'll see how two models, one shallow, the other deeper, perform on the full dataset. After that, we'll work with grouped subsets of the entire dataset to see if we can tell the difference between animals and vehicles. Next, we'll answer the question of what's better for the CIFAR-10 dataset: a single multiclass model or a set of binary models, one per class.

We'll close the chapter by introducing transfer learning and fine-tuning. These are important concepts, often confounded, that are widely used in the machine learning community, so you should develop an intuitive feel for how they work.

A CIFAR-10 Refresher

Before we dive into the experiments, let's refamiliarize ourselves with the dataset we're working with. CIFAR-10 is a 10-class dataset from the Canadian Institute for Advanced Research (CIFAR). We built this dataset in Chapter 2 but deferred its use until now. CIFAR-10 consists of 32×32 -pixel RGB images of animals (six classes) and vehicles (four

classes). Take a look at Figure 2-4 on page 51 for some sample images. The training set has 50,000 images, 5,000 from each class, so it is a balanced dataset. The test set consists of 10,000 images, 1,000 from each class. CIFAR-10 is probably the second most widely used standard dataset in machine learning after MNIST. There is also a 100-class version, CIFAR-100, that we won't work with in this book, but you'll see it pop up often in the literature.

As of this writing, the best-performing model on unaugmented CIFAR-10 has achieved a 0.5 percent error on the test set (<https://paperswithcode.com>). The model that did this has 632 million parameters. Our models will be significantly smaller and have a much larger test error. However, this is a true image dataset, unlike MNIST, which is very clean and has a uniform black background for every digit. Because of the variation in natural images, especially in their backgrounds, we might expect models to have a harder time learning the CIFAR-10 classes compared to MNIST.

For reference throughout the chapter, here are the CIFAR-10 classes:

Label	Class
0	airplanes
1	automobiles
2	birds
3	cats
4	deer
5	dogs
6	frogs
7	horses
8	ships
9	trucks

The Full CIFAR-10 Dataset

Let's train two models on the entire CIFAR-10 dataset. The first model is the same one we used in Chapter 10 for the MNIST dataset. We'll refer to this model as our *shallow model* because it has only two convolutional layers. We'll need to adapt it a touch for the 32×32 RGB inputs, but that's straightforward enough to do. The second model, which we'll call our *deep model*, uses multiple convolutional layers before the pooling and fully connected layers.

Additionally, we'll experiment with both stochastic gradient descent and Adam as our optimization algorithms. We'll fix the minibatch size at 64 and train for 60 epochs, for a total of 46,875 gradient-descent steps. For SGD, we'll use a learning rate of 0.01 and a momentum of 0.9. Recall, Adam is adaptive and alters the learning rate on the fly. We can decrease the learning rate for SGD as training progresses, but 0.01 is relatively small, and we have many gradient-descent steps, so we'll just leave it a constant.

The shallow model has 1,626,442 parameters, while the deep model has only 1,139,338. The deep model is “deep” because it has more layers, but because each convolutional layer is using exact convolution, the output decreases by two each time (for a 3×3 kernel). Therefore, the flatten layer after the pooling layer has only 7,744 values compared to 12,544 for the shallow model. The weight matrix between the flatten layer and the dense layer of 128 nodes contains the vast majority of the parameters, $7,744 \times 128 = 991,232$ compared to $12,544 \times 128 = 1,605,632$. Thus, going deeper has actually reduced the number of parameters to learn. This slightly counterintuitive result reminds us of the large expense incurred by fully connected layers and some of the initial motivation for the creation of CNNs.

Building the Models

You'll find the code for the shallow model in *cifar10_cnn.py* (Adam) and *cifar10_cnn_SGD.py* (SGD). We'll work through the code in pieces. The shallow model starts in much the same way as for the MNIST dataset, as Listing 11-1 shows.

```
import tensorflow.keras as keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras import backend as K
import numpy as np

batch_size = 64
num_classes = 10
epochs = 60
img_rows, img_cols = 32, 32

x_train = np.load("cifar10_train_images.npy")
y_train = np.load("cifar10_train_labels.npy")
x_test = np.load("cifar10_test_images.npy")
y_test = np.load("cifar10_test_labels.npy")

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 3, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 3, img_rows, img_cols)
    input_shape = (3, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 3)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 3)
    input_shape = (img_rows, img_cols, 3)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255

x_test /= 255

y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

Listing 11-1: Preparing the CIFAR-10 dataset

We import the necessary modules and load the CIFAR-10 dataset from the NumPy files we created in Chapter 2. Notice that the image dimensions are now 32×32, not 28×28, and that the number of channels is 3 (RGB) instead of 1 (grayscale). As before, we scale the inputs by 255 to map the images to [0, 1] and convert the label numbers to one-hot vectors by using `to_categorical`.

Next, we define the model architecture (Listing 11-2).

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                activation='relu',
                input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])
```

Listing 11-2: Building the shallow CIFAR-10 model

This step is identical to the MNIST version for the shallow model (see Listing 10-1 on page 264). For the deep model, we add more convolutional layers, as Listing 11-3 shows.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                activation='relu',
                input_shape=input_shape))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
```

```
model.add(Conv2D(64, (3, 3), activation='relu'))

model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])
```

Listing 11-3: Building the deep CIFAR-10 model

The extra convolutional layers give the model the opportunity to learn a better representation of the input data, which for CIFAR-10 is more complex than the simple MNIST images. The representation might be better because a deeper network can learn more-abstract representations that encompass larger structures in the inputs.

The code snippets in Listings 11-2 and 11-3 compile the model by using Adam as the optimization algorithm. We also want a version of each that uses SGD. If we replace the reference to Adam() in the compile method with the following, we'll use SGD with the learning rate and momentum values we indicated earlier:

```
optimizer=keras.optimizers.SGD(learning_rate=0.01, momentum=0.9)
```

For completeness, Listing 11-4 shows the rest of the code for both shallow and deep models.

```
print("Model parameters = %d" % model.count_params())
print(model.summary())

history = model.fit(x_train, y_train,
                   batch_size=batch_size,
```

```

        epochs=epochs,
        verbose=1,
        validation_data=(x_test[:1000], y_test[:1000]))

score = model.evaluate(x_test[1000:], y_test[1000:], verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

model.save("cifar10_cnn_model.keras")

```

Listing 11-4: Training and testing the CIFAR-10 models

This code summarizes the model architecture and number of parameters, trains by calling the `fit` method using the first 1,000 test samples for validation, and then evaluates the trained model on the remaining 9,000 test samples by calling the `evaluate` method. We report the test loss and accuracy. Then we write the model to disk (`save`), and store the history showing the per-epoch loss and accuracy during training. We'll use the history files to generate plots showing the loss and error ($1 - \text{accuracy}$) as a function of the training epoch.

Listing 11-4 gives us four files: shallow model + Adam, shallow model + SGD, deep model + Adam, and deep model + SGD. Let's run each to see our final test accuracy and then look at plots of the training process to see what we can learn.

Running the code trains and evaluates the models. This takes about seven hours total on my CPU-only system. The random initialization that Keras uses means that when you run the code yourself, you should see slightly different answers. A single run of the code produced Table 11-1.

Table 11-1: Test-Set Accuracies by Model Size and Optimizer

	Shallow	Deep
Adam	72.1%	74.8%

	Shallow	Deep
SGD	70.7%	73.7%

The table tells us that using Adam gives us more-accurate models compared to SGD. We also see that the deep model outperforms the shallow model regardless of optimizer. Adaptive optimizers like Adam are generally preferred to plain old SGD for this reason. However, I have seen claims that SGD is ultimately just as good or better once the learning rate is set up right and decreased as training proceeds.

Of course, nothing prevents us from starting with an adaptive optimizer and then switching to SGD after a certain number of epochs. The idea here is that the adaptive optimizer “gets close” to a minimum of the loss function, while SGD fine-tunes the process at that point.

Analyzing the Models

Let’s look at how the validation error changes during training. Figure 11-1 plots the error by epoch. The error is easier to understand visually; it should tend toward 0 as accuracy increases. The Adam results are on the top, with the SGD on the bottom.

As expected, regardless of optimizer, the deeper model performs better and has a lower validation-set error during training. Note, the validation-set error is not the final, held-out test-set error reported in Table 11-1, but instead the portion of the test set used during training, the first 1,000 samples in this case.

The curves are noisy because they represent a single training run. Repeated runs averaged together would produce smoother curves.

Both optimizers produce good results and follow the expected pattern of the deeper model producing a lower validation error than the shallow model. Adam edges out SGD after 60 epochs, but not by much.

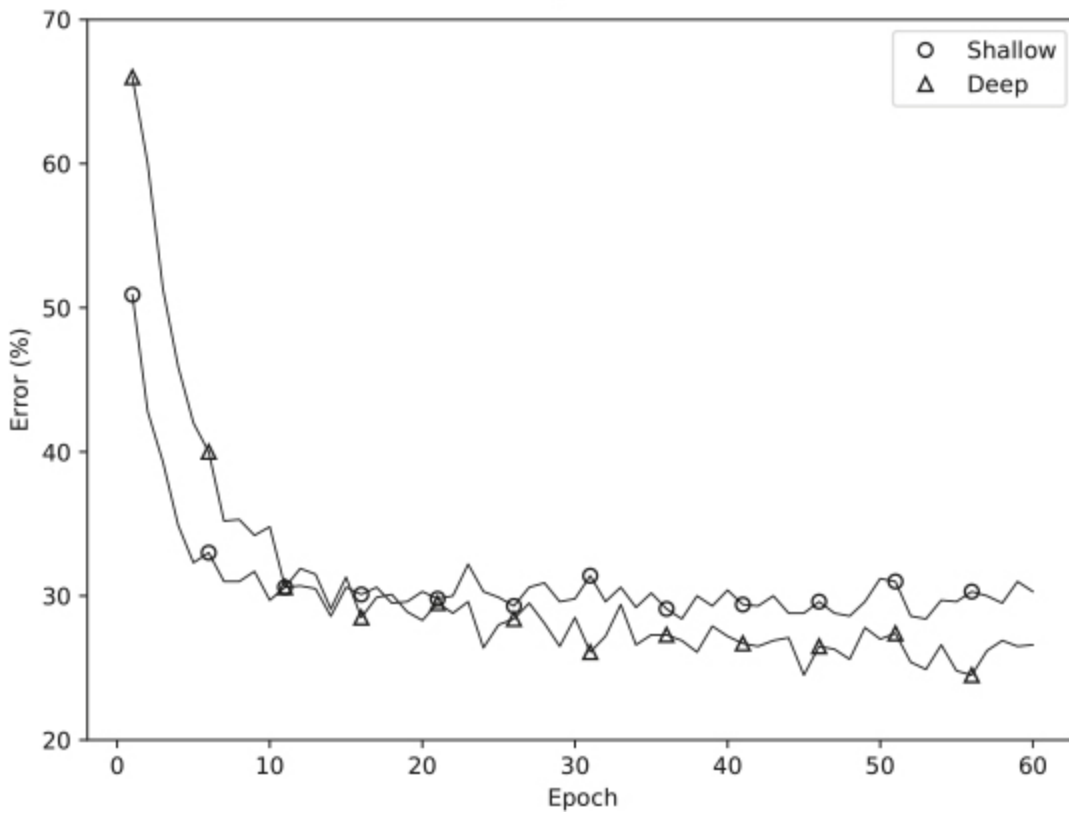
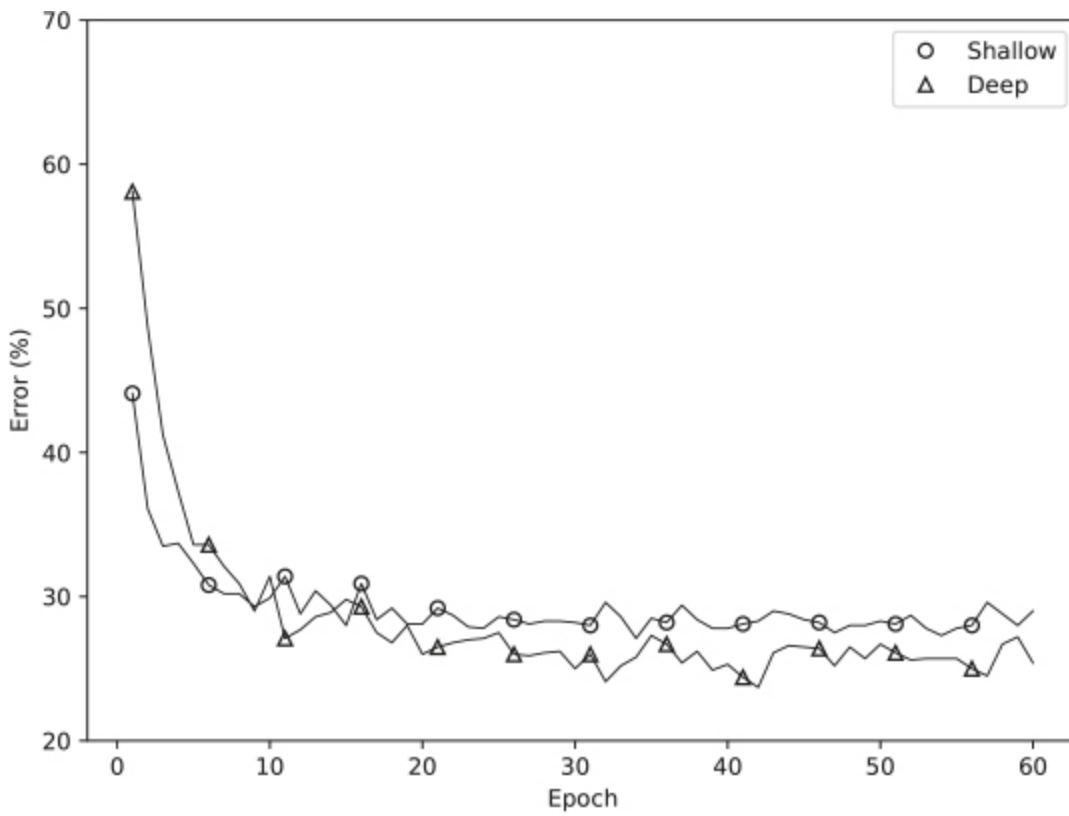


Figure 11-1: The validation-set error for shallow and deep models using Adam (top) and

SGD (bottom)

These experiments tell us that adaptive optimization algorithms and deeper networks (to a point) tend toward better-performing models. While recognizing the danger inherent in attempting to offer advice in this field, it seems safe to say that we should start with adaptive optimization, like Adam, and use a large enough model. To find out what *large enough* means, I suggest starting with a modest model and, after training, making it deeper and seeing if that improves the results. Eventually, the model will be too large for the training set, so at a certain cutoff point, increasing the size of the model no longer helps. In that case, get more training data if possible.

Let's now shift our attention to working with subsets of CIFAR-10.

Animal or Vehicle?

Four of the ten classes in CIFAR-10 are vehicles; the remaining six are animals. Let's build a model to separate the two and see what we can learn from it. We already have the images; all we need to do is recode the labels so that all the vehicles are marked as class 0 and all the animals as class 1. Doing this is straightforward, as Listing 11-5 shows (*cifar10_make_animal_vehicle.py*).

```
import numpy as np
y_train = np.load("cifar10_train_labels.npy")
y_test = np.load("cifar10_test_labels.npy")
for i in range(len(y_train)):
    if (y_train[i] in [0, 1, 8, 9]):
        y_train[i] = 0
    else:
        y_train[i] = 1
for i in range(len(y_test)):
    if (y_test[i] in [0, 1, 8, 9]):
        y_test[i] = 0
    else:
        y_test[i] = 1
```

```
np.save("cifar10_train_animal_vehicle_labels.npy", y_train)
np.save("cifar10_test_animal_vehicle_labels.npy", y_test)
```

Listing 11-5: Adjusting the labels of CIFAR-10 into vehicles (class 0) and animals (class 1)

We load the existing train and test label files, already matched in order with the train and test image files, and build new label vectors mapping the vehicle classes (classes 0, 1, 8, and 9) to 0, and all the others to 1.

The code in the previous section for building and training the model remains the same except for the definition of the model architecture and the particular file we load for the train and test labels. The number of classes (`num_classes`) is set to 2, the minibatch size is 128, and we'll train for 12 epochs. The training set isn't completely balanced: It has 20,000 vehicles and 30,000 animals. However, the imbalance isn't severe, so we should be in good shape. Remember that when one class is scarce, learning it well becomes difficult for the model. We'll stick with Adam as the optimizer and use the first 1,000 test samples for validation and the remaining 9,000 for final testing. We'll use the same shallow architecture used in the previous section; see *cifar10_animals_vehicles.py*.

Training this model on the CIFAR-10 images with the recoded labels gives us a final test accuracy of 94.4 percent. Let's be a little pedantic and calculate all the performance metrics from Chapter 8. To do this, we update the `tally_predictions` function defined in that chapter (Listing 8-1 on page 207) to work with a Keras model. We'll also use `basic_metrics` (Listing 8-2 on page 211) and `advanced_metrics` (Listing 8-3 on page 216). Listing 11-6 shows the updated code for `tally_predictions`.

```
def tally_predictions(model, x, y_test):
    y = np.argmax(y_test, axis=1)
    pp = model.predict(x)
    p = np.argmax(pp, axis=1)
    tp = tn = fp = fn = 0
    for i in range(len(y)):
        if (p[i] == 0) and (y[i] == 0):
            tn += 1
```

```

elif (p[i] == 0) and (y[i] == 1):
    fn += 1
elif (p[i] == 1) and (y[i] == 0):
    fp += 1
else:
    tp += 1
score = float(tp+tn) / float(tp+tn+fp+fn)
return [tp, tn, fp, fn, score]

```

Listing 11-6: Calculating basic metrics for Keras models

We pass in the model, test samples (x), and the one-hot test labels (y_{test}). Unlike the sklearn version of `tally_predictions`, here we first convert the onehot test labels to integers by applying NumPy's `argmax` function across the columns (y), then use the model to predict per-class probabilities (pp). This returns a 2D array, one row for each sample in x , where the columns are the likelihoods assigned per class. Here we have two columns because there are only two classes: vehicle or animal.

Before we can tally the true positives, true negatives, false positives (vehicle classified as animal), and false negatives (animal classified as vehicle), we need to convert the predictions in pp to class labels in p . Once we have assigned a predicted class label, we can calculate the tallies and return them along with the overall score (accuracy). We pass the list returned by `tally_predictions` to `basic_metrics` and then pass the output of both functions to `advanced_metrics`, as in Chapter 8.

The full set of binary classifier metrics gives us Table 11-2.

Table 11-2: Animal vs. Vehicle Metrics

Metric	Result
TP	5,160
FP	287
TN	3,337
FN	216

Metric	Result
TPR (sensitivity, recall)	0.9598
TNR (specificity)	0.9208
PPV (precision)	0.9473
NPV	0.9392
FPR	0.0792
FNR	0.0402
F1	0.9535
MCC	0.8836
κ (kappa)	0.8834
Informedness	0.8806
Markedness	0.8865
Accuracy	0.9441

As argued in Chapter 8, the Matthews correlation coefficient (MCC) is possibly the best single number for characterizing a binary classifier. Here we have an MCC of 0.8836 out of 1.0, indicative of a solid model.

Recall that the sensitivity is the probability that an animal is called an animal by this model, and the specificity is the probability that a vehicle is called a vehicle. The precision is the probability that when the model assigns a label of animal, it is correct, and the negative predictive value (NPV) is the probability of the model being correct when it assigns a label of vehicle. Note also that the false-positive rate (FPR) is $1 - \text{specificity}$, and the false-negative rate (FNR) is $1 - \text{sensitivity}$.

A little more code will calculate the ROC curve and its area:

```
from sklearn.metrics import roc_auc_score, roc_curve
def roc_curve_area(model, x, y):
    pp = model.predict(x)
```

```
p = np.zeros(pp.shape[0], dtype="uint8")
for i in range(pp.shape[0]):
    p[i] = 0 if (pp[i,0] > pp[i,1]) else 1
auc = roc_auc_score(y,p)
roc = roc_curve(y,pp[:,1])
return [auc, roc]
```

Again, we pass in the trained model, the test samples (x), and the animal or vehicle labels (y). We also convert the output probabilities to class predictions, as we did in Listing 11-6. The AUC is 0.9403, and Figure 11-2 shows the ROC curve. This curve is steep and close to the upper-left corner of the plot—all good signs of a well-performing model.

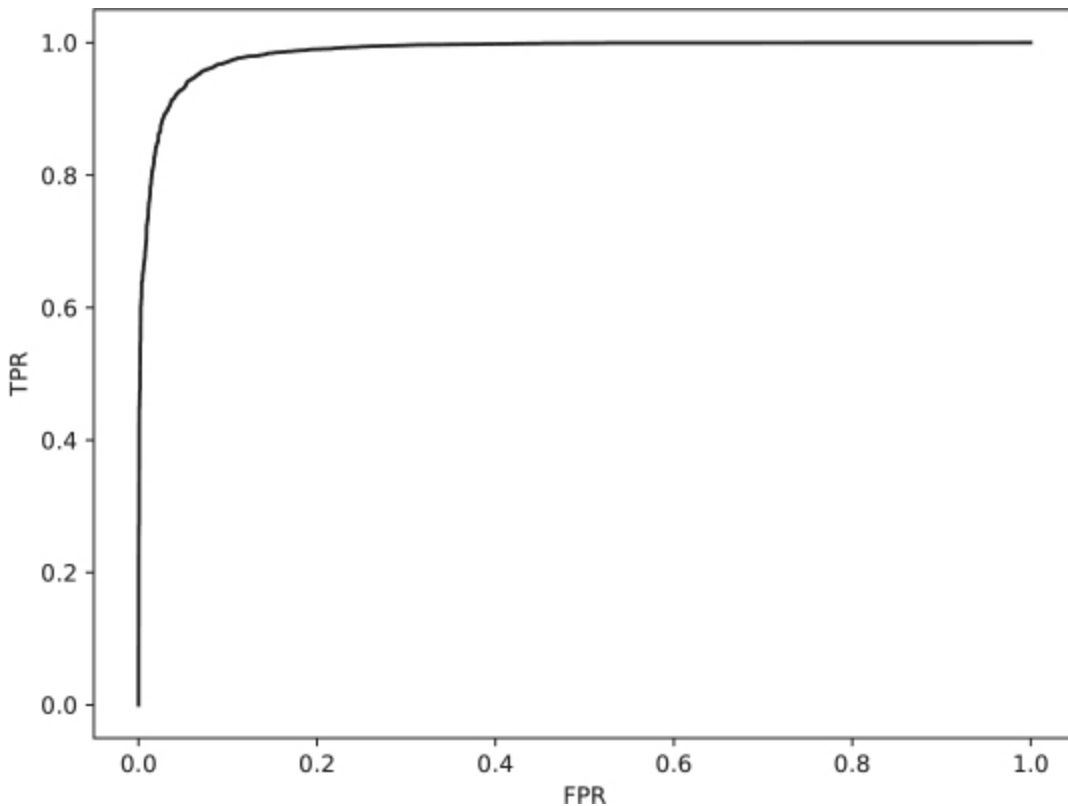


Figure 11-2: The ROC curve for the animal or vehicle model

We grouped animals and vehicles and asked a single model to learn something about the difference between them. Clearly, some characteristics differentiate the two classes, and the model has learned to use them successfully. However, unlike most binary classifiers, we know

finer label assignments for the test data. For example, we know which of the animals are birds or deer or frogs. Likewise, we know which samples are airplanes, ships, or trucks.

When the model makes a mistake, it is either a false positive (calling a vehicle an animal) or a false negative (calling an animal a vehicle). We chose animals to be class 1, so false positives are cases where a vehicle was called an animal. The converse is true for false negatives. We can use the full class labels to tell us how many of the false positives are represented by which vehicle classes, and we can do the same for the false negatives to tell us which animal classes were assigned to the vehicle class. A few lines of code in Listing 11-7 give us what we are after.

```
def false_positives_and_negatives(model, x, y):
    y_label = np.load("../data/cifar10/cifar10_test_labels.npy")[1000:]
    y_test = np.argmax(y, axis=1)
    pp = model.predict(x)
    p = np.argmax(pp, axis=1)
    hp = []; hn = []
    for i in range(len(y_test)):
        if (p[i] == 0) and (y_test[i] == 1):
            hn.append(y_label[i])
        elif (p[i] == 1) and (y_test[i] == 0):
            hp.append(y_label[i])

    hp = np.array(hp)
    hn = np.array(hn)
    a = np.histogram(hp, bins=10, range=[0,9])[0]
    b = np.histogram(hn, bins=10, range=[0,9])[0]
    print("vehicles as animals: %s" % np.array2string(a))
    print("animals as vehicles: %s" % np.array2string(b))
```

Listing 11-7: Using the fine class labels to determine which classes account for false positives and false negatives

First, we load actual labels (`y_label`) from disk. Then, we apply the model to the test set (`x`) and track the assigned class label (`p`) to compare with the true class label (`y_test`). Note that the ordering of `y_test` and

`y_label` matches because when we defined the animal or vehicle labels, we were careful to not alter the arrangement.

The `for` loop tracks false positives (`hp`) and false negatives (`hn`), keeping the actual label in each case. Here's where we take advantage of knowing the fine-grained class assignments.

Once we have the actual labels for all FP and FN cases, we use `histogram` to do the tallying for us. There are 10 actual class labels, so we tell `histogram` that we want to use 10 bins. We also need to specify the range for the bins (`range=[0,9]`). We want only the counts themselves, so we need to keep only the first array returned by `histogram`, hence the `[0]` at the end of the call. Finally, we print the arrays to get Table 11-3.

Table 11-3: Model Mistakes Using True Class Labels

Class	Vehicles as animals	Animals as vehicles
(0) airplanes	133	...
(1) automobiles	44	...
(2) birds	...	75
(3) cats	...	39
(4) deer	...	37
(5) dogs	...	17
(6) frogs	...	23
(7) horses	...	25
(8) ships	51	...
(9) trucks	59	...

This means that of the vehicles the model called “animal,” 133 of them were of class 0, airplane. The vehicle class least likely to be identified as an animal is class 1, automobile. Ships and trucks were similarly likely to be mistaken for an animal. Going the other way, we

see that class 2, birds, were most likely to be mistaken for vehicles and class 5, dogs, were least likely to be misclassified.

What can we make of this? The most commonly misclassified vehicle is an airplane, while the most commonly misclassified animal is a bird. This makes sense: A picture of an airplane and a picture of a bird flying do look similar. I'll leave it to you to make connections among the other categories.

Binary or Multiclass?

Conventional wisdom in machine learning is that a multiclass model will generally outperform multiple binary models. While this is almost certainly true for large datasets, large models, and situations with many classes (like the ImageNet dataset of 1,000 classes), how does it pan out for small models like the ones we're working with in this chapter? Let's find out.

The CIFAR-10 dataset has 5,000 instances of each class and 10 classes. This means we can train 10 binary models where the target class (class 1) is one of the 10 classes, and the other class is everything else. This is known as a *one-versus-rest* approach. To classify an unknown sample, we run it through each of the 10 classifiers and assign the label of the model returning the most confident answer. The datasets are all imbalanced (5,000 class 1 instances to 45,000 class 0) but, as you'll see, there is still enough data to learn the difference between classes.

We need some code to train 10 one-versus-rest models. We'll use the shallow architecture we've used before, with a minibatch size of 128, and we'll train for 12 epochs. Before we can train, however, we need to reassign the class labels for the train and test sets so that all instances of the target class are a 1 and everything else is a 0. To build the per-class labels, we'll use Listing 11-8.

```
import sys
import numpy as np
❶ class1 = eval "["+sys.argv[1]+"")
y_train = np.load("cifar10_train_labels.npy")
```

```
y_test = np.load("cifar10_test_labels.npy")
for i in range(len(y_train)):
    if (y_train[i] in class1):
        y_train[i] = 1
    else:
        y_train[i] = 0
for i in range(len(y_test)):
    if (y_test[i] in class1):
        y_test[i] = 1
    else:
        y_test[i] = 0
np.save(sys.argv[2], y_train)
np.save(sys.argv[3], y_test)
```

Listing 11-8: Building the per-class labels

This code makes use of the command line. To call it, use something like this:

```
$ python3 make_label_files.py 1 train_1.npy test_1.npy
```

The first argument is the desired target class label (here, 1 for automobiles), and the next two arguments are the names in which to store the new label assignments for the train and test images. The code itself loops over the actual train and test labels, and if the label is the target class, the corresponding output label is 1; otherwise, it is 0.

This code is more flexible than mapping a single class. By using `eval` ❶, we can pass in a comma-separated string of all the CIFAR-10 labels we want to treat as the target class. For example, to use this code to make labels for the animal versus vehicle example of the previous section, we'd make the first argument `2,3,4,5,6,7`.

Once we have new labels for each of the 10 classes, we can use them to train 10 binary models. Our multiclass model is the shallow architecture trained on the full CIFAR-10 dataset for 12 epochs. To train all the models, use the *train_binary_multiclass_models* script. This script calls *cifar10_cnn_arbitrary.py* to train a model by using a specified binary dataset.

To test the models, we need to first load them all from disk along with the test-set data. Then we need to run all the data through the multiclass model and each of the individual class models keeping the predictions. From the predictions, we can assign class labels and build confusion matrices to see how well each approach does. First, let's load the test set and the models:

```
x_test = np.load("cifar10_test_images.npy")/255.0
y_test = np.load("cifar10_test_labels.npy")
mm = load_model("cifar10_cnn_model.keras")
m = []
for i in range(10):
    m.append(load_model("cifar10_cnn_%d_model.keras" % i))
```

Notice that we're scaling the test set by 255, as we did with the training data. We'll keep the multiclass model in `mm` and load the 10 single-class models into the list, `m`.

Next, we apply the models to each test-set sample:

```
mp = np.argmax(mm.predict(x_test), axis=1)
p = np.zeros((10, 10000), dtype="float32")
for i in range(10):
    p[i,:] = m[i].predict(x_test)[:,:1]
bp = np.argmax(p, axis=0)
```

Calling `predict` with the 10,000 test samples returns a $10,000 \times 10$ matrix for the multiclass model or $10,000 \times 2$ for the individual models. Each row corresponds to a test sample, and each column is the model's output for each class. For the multiclass case, we set `mp` to the maximum value across the columns (`axis=1`) to get a vector of 10,000 values, each of which is the predicted class label.

We loop over the individual models and call `predict`, keeping only the class 1 probabilities. These are placed into `p`, where the rows are the individual model outputs for that class label, and the columns are the specific class 1 prediction probabilities for each of the 10,000 test samples. If we return the maximum value across the rows by using `argmax`

and `axis=0`, we'll get the class label of the model that had the highest predicted probability for each test sample. This is what is in `bp`.

With our predictions in hand, we can generate the confusion matrices:

```
cm = np.zeros((10,10), dtype="uint16")
cb = np.zeros((10,10), dtype="uint16")

for i in range(10000):
    cm[y_test[i], mp[i]] += 1
    cb[y_test[i], bp[i]] += 1

np.save("cifar10_multiclass_conf_mat.npy", cm)
np.save("cifar10_binary_conf_mat.npy", cb)
```

Here rows represent the true class label, and columns represent the model's predicted label. We also store the confusion matrices for future use.

We can display the confusion matrices with Listing 11-9.

```
cifar10_one_vs_many.py
print("One-vs-rest confusion matrix (rows true, cols predicted):")
print("%s" % np.array2string(100*(cb/1000.0), precision=1))
print()
print("Multiclass confusion matrix:")
print("%s" % np.array2string(100*(cm/1000.0), precision=1))
```

Listing 11-9: Displaying the confusion matrices

We divide the counts in `cb` and `cm` by 1,000 because each class is represented by that many samples in the test set. This converts the confusion matrix entries to a fraction and then a percent when multiplied by 100. All the processing is in *cifar10_one_vs_many.py*.

So, how did we do? The multiple one-versus-rest classifiers produced Table 11-4.

Table 11-4: The Multiple One-vs.-Rest Confusion Matrix

Class	0	1	2	3	4	5	6	7	8	9
0	72.5	1.6	5.5	2.7	2.0	1.3	2.5	1.8	6.3	3.8
1	1.7	82.4	0.6	0.8	0.4	0.8	1.4	0.7	2.6	8.6
2	5.5	0.4	54.2	5.4	7.9	8.7	9.6	5.9	1.4	1.0
3	1.2	1.0	5.3	48.2	6.7	16.9	12.8	4.8	1.6	1.5
4	2.0	0.4	4.3	5.7	67.3	2.2	9.5	7.1	1.0	0.5
5	1.2	0.5	3.8	11.6	4.5	68.0	3.6	4.9	1.2	0.7
6	0.4	0.2	2.8	2.9	2.2	1.5	87.8	0.8	0.7	0.7
7	1.1	0.2	1.8	2.6	4.7	3.7	1.4	82.3	0.5	1.7
8	4.7	3.2	1.4	1.6	0.4	1.2	1.4	1.0	82.0	3.1
9	2.2	5.5	0.7	1.5	0.4	0.4	0.9	1.9	2.2	84.3

The multiclass classifier came up with Table 11-5.

Table 11-5: The Multiclass Confusion Matrix

Class	0	1	2	3	4	5	6	7	8	9
0	73.4	1.6	4.7	2.7	1.5	0.6	1.2	1.7	8.7	3.9
1	2.1	79.2	0.5	1.8	0.5	0.5	1.1	1.0	3.1	10.2
2	7.3	0.9	51.8	6.0	9.9	11.0	6.1	4.9	1.5	0.6
3	1.2	0.7	7.9	50.7	6.5	21.4	5.9	3.0	1.5	1.2
4	2.5	0.4	8.1	7.7	62.3	4.2	5.5	7.7	1.2	0.4
5	0.7	0.2	4.8	14.7	3.9	66.5	2.1	5.5	1.1	0.5
6	0.5	0.7	5.1	6.9	3.5	3.1	79.2	0.5	0.2	0.3
7	0.9	0.1	2.7	3.4	5.9	7.4	0.5	78.0	0.1	1.0

Class	0	1	2	3	4	5	6	7	8	9
8	6.9	3.5	1.3	1.4	0.5	1.1	0.7	0.4	81.4	2.8
9	2.8	6.8	0.8	1.5	0.6	1.6	0.7	3.2	3.1	78.9

The diagonals are the correct class assignments. Ideally, the matrix would be only diagonal elements. All other elements are mistakes, cases where the model or models chose the wrong label. Since each class is equally represented in the test set, we can calculate an overall accuracy for both models by using the unweighted average of the diagonals. If we do this, we get the following:

One-versus-rest 72.9 percent

Multiclass 70.1 percent

The one-versus-rest classifiers have the edge in this case. Of course, we needed to do 10 times the work to get the one-versus-rest confusion matrix; 10 classifiers were used instead of just 1. The multiclass model was about 1 percent better on class 0 (airplane) and 3 percent better on class 3 (cats) than the one-versus-rest models, but it was worse on every other class, especially class 6 (frogs) by 9 percent.

Did we arrive at a definitive answer regarding one-versus-rest or multiclass models? No, nor could we in general. However, we did, objectively, get better performance by using the multiple models.

One argument given against using multiple models, besides the extra computation necessary, is that using a single model for multiple classes provides the model with the opportunity to see examples that are similar to a particular class but are not instances of that class. These hard negatives serve to regularize the model by forcing it to (indirectly) pay attention to features that are dissimilar between classes instead of features that might be strongly associated with a class but are also present in other classes. We first encountered hard negatives in Chapter 1.

Using the Augmented CIFAR-10 Dataset

In Chapter 2, we constructed an augmented version of the CIFAR-10 dataset with 500,000 samples instead of the original 50,000 samples. In this section, we explore whether the augmentation was worthwhile by comparing a model trained on the original 50,000-sample training set with one trained on the 500,000-sample augmented training set. The original 10,000-sample test is used for both cases, including for validation because we won't rely on the validation-set performance to decide when to stop training the model, thereby avoiding any philosophical issues. We'll stick with the base architecture we've used consistently throughout the chapter so that the only difference between the two models is the dataset itself.

The augmented CIFAR-10 dataset cropped the images to avoid artifacts introduced by rotation. It's best when building datasets to select potential training images that are large enough to rotate and then crop to remove edge effects, but we didn't have that option with CIFAR-10 as the images were already 32×32 pixels. The net result is that the augmented training images are 28×28 pixels, meaning we must similarly center-crop the original training and test sets to use them with the models. Cropping happens in the code when the training set is loaded from disk.

We'll train the model for 12 epochs by using the augmented dataset. However, we must be careful if we want to make a fair comparison with the original dataset. The augmented dataset is 10 times as large, meaning we must increase the number of epochs for the model using the smaller dataset by a factor of 10 to take the same number of gradient-descent steps. Therefore, the smaller dataset model will train for 120 epochs. The relevant code is in *cifar10_cnn_crop.py* and *cifar10_cnn_augmented.py*. Both models took a little more than an hour to train on my test machine. We'll examine the results momentarily, but before we do, a few comments on working with a 500,000-sample dataset are in order.

Previously, we loaded the dataset as byte data, then converted it to 32-bit floating-point data by dividing by 255 to map the data from [0,

255] to [0, 1]. The augmented dataset has 500,000 28×28-pixel color images in it, meaning the raw dataset occupies the following bytes of memory:

$$500,000 \times 28 \times 28 \times 3 = 1,176,000,000$$

Mapping directly to 32-bit floating-point numbers increases the storage requirement from just over 1GB to 4.7GB of RAM. That's not too much for many modern systems, but we need not be so wasteful. We must represent the images in the range [0, 1], but pixel values are stored in increments of $1/255 = 0.004$ when rounded to three decimals. Storing the scaled image data as a 32-bit float is overkill. As it happens, 32-bit floats are not the smallest floating-point numbers around.

The large datasets used by deep learning models have resulted in increased use of number formats that were mostly ignored in the past, to say nothing of a plethora of new formats designed explicitly for neural networks. One of the neglected formats is the IEEE 754 *binary16* format, a 2-byte floating-point format known to NumPy as `float16`. The smallest increment representable by a 16-bit float is 0.001, which is less than the 0.004 necessary to represent the image data after scaling to [0, 1]. In other words, we can use a 16-bit floating-point array to hold the training data. Since each pixel value now occupies 2 bytes instead of 4, we need only 2.35GB of RAM for the training data, and there is no loss of precision. If you look at the code in *cifar10_cnn_crop.py* and *cifar10_cnn_augmented.py*, you'll notice that both store the training data as `float16`:

```
x_train = x_train.astype('float16')/255
```

NumPy supports 16-bit floats for storage only and converts values to 32-bit floats before using them in calculations, but that is exactly what we want.

Table 11-6 presents the confusion matrix using the original held-out test set for models trained on the original dataset.

Table 11-6: The Confusion Matrix for the Original Model

Class	0	1	2	3	4	5	6	7	8	9
0	748	21	44	26	17	9	7	16	72	40
1	21	828	8	10	2	2	9	3	31	86
2	64	14	567	87	78	70	64	33	11	12
3	24	13	56	528	63	195	51	42	16	12
4	22	2	59	81	660	33	62	63	10	8
5	13	5	39	185	41	649	18	41	8	1
6	8	5	47	71	34	22	799	5	6	3
7	17	5	33	50	58	67	8	748	2	12
8	67	37	12	23	8	5	4	6	812	26
9	26	82	7	25	8	4	8	15	33	792

Table 11-7 presents the confusion matrix for models trained on the augmented dataset.

Table 11-7: The Confusion Matrix for the Augmented Model

Class	0	1	2	3	4	5	6	7	8	9
0	710	25	55	23	11	8	10	6	115	37
1	6	898	2	9	0	0	9	2	22	52
2	48	4	637	55	90	38	79	22	18	9
3	12	6	61	582	69	130	88	19	17	16
4	14	2	52	59	723	24	64	43	17	2
5	6	6	34	178	55	640	34	31	10	6
6	2	2	35	40	33	15	865	3	3	2
7	11	3	36	45	83	66	9	731	2	14

Class	0	1	2	3	4	5	6	7	8	9
8	30	30	12	11	4	4	6	3	881	19
9	24	86	4	17	3	1	6	4	41	814

Both matrices are somewhat noisy, implying that neither model is sterling. Even without careful reading, the bottom matrix may seem less visually cluttered, which we would expect for a model that makes fewer mistakes.

Calculating the overall accuracy (sum the diagonal and divide by the sum of all elements) and the corresponding MCC tells the story:

Dataset	Accuracy	MCC
Original	71.31%	0.6813
Augmented	74.81%	0.7205

Training with the augmented dataset gives us a 3.5 percent increase in overall accuracy matched by a large increase in the MCC. Clearly, augmentation helps regularize the model so that it's more likely to correctly label unknown inputs (that is, those in the test set). Data augmentation is vindicated (for this simple example).

Augmenting the dataset offers us another possible avenue of attack. With 500,000 training samples, we're now able to train a deeper model, one that might not train well with the smaller training set. Listing 11-10 shows the base model.

```

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
                input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))

```

```
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Listing 11-10: The base model

The model uses two convolutional layers with ReLUs between them before pooling. Let's make the network deeper by adding additional convolutional layers as in Listing 11-11.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
                input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Dropout(0.25))
model.add(Conv2D(96, (3, 3), activation='relu'))
model.add(Dropout(0.25))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Listing 11-11: The deeper CIFAR-10 model

The deeper model keeps the second convolutional layer with 64 filters, but adds two additional convolutional and ReLU layers of 96 and 128 filters before the pooling layer. The deeper network has 1,825,322 parameters compared to 1,200,458 in the base model. We'll train the deeper model for 12 epochs as well. The code is in *cifar10_cnn_augmented_deep.py*, with the resulting confusion matrix after some five hours of training in Table 11-8.

Table 11-8: The Confusion Matrix for the Deeper CIFAR-10 Model

Class	0	1	2	3	4	5	6	7	8	9
0	848	10	36	16	18	3	11	8	32	18

Class	0	1	2	3	4	5	6	7	8	9
1	21	882	1	2	4	1	8	1	12	68
2	57	1	670	41	84	47	77	14	3	6
3	24	1	44	543	79	175	91	23	11	9
4	5	0	38	31	835	24	36	24	5	2
5	9	0	29	89	43	787	16	17	5	5
6	6	0	18	23	18	12	917	4	1	1
7	12	0	21	23	70	57	2	804	2	9
8	67	11	7	12	3	2	4	2	862	30
9	22	22	6	7	3	1	8	2	11	918

The confusion matrix looks better than those in Table 11-7, a fact that is borne out by comparing the accuracy and MCC with the earlier results:

Dataset	Accuracy	MCC
Original	71.31%	0.6813
Augmented	74.81%	0.7205
<i>Deeper</i>	<i>80.66%</i>	<i>0.7858</i>

The combination of an augmented training set and a deeper model has improved performance by nearly 10 percent relative to the baseline model. All of this validates data augmentation as a viable regularizer to improve model performance. We'll return to the augmented CIFAR-10 dataset in Chapter 13.

Summary

12

A CASE STUDY: CLASSIFYING AUDIO SAMPLES



Let's bring together everything that you've learned so far in the book. We'll be looking at a single case study. Here's our scenario: We are data scientists, and our boss has tasked us with building a classifier for audio samples stored as *.wav* files. We'll begin with the data itself. We first want to build some basic intuition for how it's structured. From there, we'll build augmented datasets for training models.

The first dataset uses the sound samples themselves, a 1D dataset. You'll see that this approach isn't as successful as we'd like it to be. We'll then turn the audio data into images to allow us to explore 2D CNNs. This change of representation will lead to a big improvement in model performance. Finally, we'll use an ensemble of models to leverage the relative strengths and weaknesses of the individual models to boost overall performance still more.

Building the Dataset

There are 10 classes in our dataset, which consists of 400 samples total, 40 samples per class, each 5 seconds long. We'll assume we cannot get any more data because it's time-consuming and expensive to record the samples and label them. We must work with the data we are given and no more.

Throughout this book, I've consistently preached about the necessity of having a good dataset. We'll assume that the dataset we've been handed is complete in the sense that our system will encounter only types of sound samples in the dataset; there will be no unknown class or classes. Additionally, we'll assume that the balanced nature of the dataset is real, and all classes are indeed equally likely.

The audio dataset we'll use is called ESC-10. For a complete description, see "ESC: Dataset for Environmental Sound Classification" by Karol J. Piczak (2015). The dataset is available at <https://github.com/karoldvl/ESC-50>, but it needs to be extracted from the larger ESC-50 dataset, which doesn't have a license we can use. The ESC-10 subset does.

Let's do some preprocessing to extract the ESC-10 *.wav* files from the larger ESC-50 dataset. Download the single ZIP file version of the dataset from the preceding URL to the *src/chapter_14* directory and expand it. This will create a directory called *ESC-50-master*. Then, use the code in Listing 12-1 to build the ESC-10 dataset from it.

```
extract_esc10.py
import sys
import os
import shutil

classes = {
    "rain":0,
    "rooster":1,
    "crying_baby":2,
    "sea_waves":3,
    "clock_tick":4,
    "sneezing":5,
    "dog":6,
```

```

    "crackling_fire":7,
    "helicopter":8,
    "chainsaw":9,
}

with open("ESC-50-master/meta/esc50.csv") as f:
    lines = [i[:-1] for i in f.readlines()]
lines = lines[1:]

os.system("rm -rf ../data/audio/ESC-10")
os.system("mkdir ../data/audio/ESC-10")
os.system("mkdir ../data/audio/ESC-10/audio")

meta = []
for line in lines:
    t = line.split(",")
    if (t[-3] == 'True'):
        meta.append("../data/audio/ESC-10/audio/%s %d" % (t[0],classes[t[3]]))
        src = "ESC-50-master/audio/"+t[0]
        dst = "../data/audio/ESC-10/audio/"+t[0]
        shutil.copy(src, dst)

with open("../data/audio/ESC-10/filelist.txt","w") as f:
    for m in meta:
        f.write(m+"\n")

```

Listing 12-1: Building the ESC-10 dataset

The code uses the ESC-50 metadata to identify the sound samples that belong to the 10 classes of the ESC-10 dataset and then copies them to the *ESC-10/audio* directory. It also writes a list of the audio files to *filelist.txt*. After running this code, we'll use only the ESC-10 files.

If all is well, we should now have 400 five-second *.wav* files, 40 from each of the 10 classes: rain, rooster, crying baby, sea waves, clock tick, sneezing, dog, crackling fire, helicopter, and chainsaw. We'll politely refrain from asking our boss exactly why she wants to discriminate between these particular classes of sound.

Augmenting the Dataset

Our first instinct should be that our dataset is too small. After all, we have only 40 examples of each sound, and some of those will need to be held back for testing, leaving even fewer per class for training.

We could resort to k -fold validation, but in this case, we'll instead opt for data augmentation. So, how do we augment audio data?

Recall, the goal of data augmentation is to create new data samples that could plausibly come from the classes in the dataset. With images, we can make obvious changes like shifting, flipping left and right, and so on. With continuous vectors, we've seen how to use PCA to augment the data in Chapter 2. To augment the audio files, we need to think of steps we can take to produce new files that still sound like the original class. Four thoughts come to mind.

First, we can shift the sample in time, much as we can shift an image to the left or right a few pixels. Second, we can simulate a noisy environment by adding a small amount of random noise to the sound itself. Third, we can shift the pitch of the sound, and make it higher or lower by a small amount; this is known as *pitch shifting*. Finally, we can lengthen or compress the sound in time. This is known as *time shifting*.

Doing all this sounds complicated, especially if we haven't worked with audio data before. In practice, being presented with unfamiliar data is a very real possibility; we don't all get to choose what we need to work with.

Fortunately for us, we're working in Python, and the Python community is vast and talented. It turns out that adding one library, *librosa*, to our system will allow us to easily perform time stretching and pitch shifting. Let's install the library:

```
$ sudo pip3 install librosa
```

With the necessary library installed, we can augment the ESC-10 dataset with the code in Listing 12-2 (*make_augmented_audio.py*).

```
import os
import random
```

```

import numpy as np
from scipy.io.wavfile import read, write
import librosa as rosa

N = 8
os.system("rm -rf ../data/audio/ESC-10/augmented; mkdir ../data/audio/ESC-10/augmented")
os.system("mkdir ../data/audio/ESC-10/augmented/train ../data/audio/ESC-10/augmented/test")

src_list = [i[:-1] for i in open("../data/audio/ESC-10/filelist.txt")]
z = [[] for i in range(10)]
for s in src_list:
    _,c = s.split()
    z[int(c)].append(s)

train = []
test = []
for i in range(10):
    p = z[i]
    random.shuffle(p)
    test += p[:8]
    train += p[8:]

random.shuffle(train)
random.shuffle(test)
augment_audio(train, "train")
augment_audio(test, "test")

```

Listing 12-2: Augmenting the ESC-10 dataset, part 1

This code loads the necessary modules, including the `librosa` module, which we'll import as `rosa`, and two functions from the SciPy `wavfile` module that let us read and write NumPy arrays as *.wav* files.

We set the number of samples per class that we'll hold back for testing ($N = 8$) and create the output directory where the augmented sound files will reside (`augmented`). Then we read the file list we created

with Listing 12-1. Next, we create a nested list (*z*) to hold the names of the audio files associated with each of the 10 classes.

Using the list of files per class, we pull it apart and create `train` and `test` file lists. Notice that we randomly shuffle the list of files per class and the final `train` and `test` lists. This code follows the convention we discussed in Chapter 1 of separating train and test first, then augmenting.

We can augment the train and test files by calling `augment_audio`. This function is in Listing 12-3.

```
def augment_audio(src_list, typ):
    flist = []
    for i, s in enumerate(src_list):
        f, c = s.split()
        wav = read(f) # (sample rate, data, type) ❶
        base = os.path.abspath("../data/audio/ESC-10/augmented/%s/%s"
                                % (typ, os.path.basename(f)[:4]))
        fname = base+".wav"
        write(fname, wav[0], wav[1]) ❷
        flist.append("%s %s" % (fname, c))
        for j in range(19):
            d = augment(wav)
            fname = base+("_%04d.wav" % j)
            write(fname, wav[0], d.astype(wav[1].dtype)) ❸
            flist.append("%s %s" % (fname, c))
    random.shuffle(flist)
    with open("../data/audio/ESC-10/augmented_%s_filelist.txt" % typ, "w") as f:
        for z in flist:
            f.write("%s\n" % z)
```

Listing 12-3: Augmenting the ESC-10 dataset, part 2

The function loops over all the filenames in the given list (`src_list`), which will be either train or test. The filename is separated from the class label, and then the file is read from disk ❶. As indicated in the comment, `wav` is a list of three elements. The first is the sampling rate in Hz (cycles per second). This is how often the analog waveform was

digitized to produce the *.wav* file. For ESC-10, the sampling rate is always 44,100 Hz, which is the standard rate for a compact disc. The second element is a NumPy array containing the actual digitized sound samples. These are the values we'll augment to produce new data files.

After setting up some output pathnames, we write the original sound sample to the augmented directory ❷. Then, we start a loop to generate 19 more augmented versions of the current sound sample. The augmented dataset, as a whole, will be 20 times larger, for a total of 8,000 sound files, 6,400 for training and 1,600 for testing. Note, the sound samples for an augmented source file are assigned to *d*. The new sound file is written to disk by using the sample rate of 44,100 Hz and the augmented data matching the data type of the source ❸.

As we create the augmented sound files, we also keep track of the filename and class and write them to a new file list. Here *typ* is a string indicating train or test.

This function calls yet another function, *augment*. This is the function that generates an augmented version of a single sound file by randomly applying a subset of the four augmentation strategies mentioned previously: shifting, noise, pitch shifting, or time shifting. Some or all of these might be used for any call to *augment*. Listing 12-4 shows the *augment* function itself.

```
def augment(wav):
    sr = wav[0]
    d = wav[1].astype("float32")
    ❶ if (random.random() < 0.5):
        s = int(sr/4.0*(np.random.random()-0.5))
        d = np.roll(d, s)
        if (s < 0):
            d[s:] = 0
        else:
            d[:s] = 0
    ❷ if (random.random() < 0.5):
        d += 0.1*(d.max()-d.min())*np.random.random(d.shape[0])
    ❸ if (random.random() < 0.5):
        pf = 20.0*(np.random.random()-0.5)
```

```

        d = rosa.effects.pitch_shift(d, sr=sr, n_steps=pf)
    ❷ if (random.random() < 0.5):
        rate = 1.0 + (np.random.random()-0.5)
        d = rosa.effects.time_stretch(d, rate=rate)
        if (d.shape[0] > wav[1].shape[0]):
            d = d[:wav[1].shape[0]]
        else:
            w = np.zeros(wav[1].shape[0], dtype="float32")
            w[:d.shape[0]] = d
            d = w.copy()
    return d

```

Listing 12-4: Augmenting the ESC-10 dataset, part 3

This function separates the samples (`d`) from the sample rate (`sr`) and makes sure the samples are floating-point numbers. For ESC-10, the source samples are all of type `int16` (signed 16-bit integers). Next come four `if` statements. Each asks for a single random float, and if that float is less than 0.5, we execute the body of the `if`. This means that we apply each possible augmentation with a probability of 50 percent.

The first `if` shifts the sound samples in time ❶ by rolling the NumPy array, a vector, by a number of samples, `s`. This value amounts to at most an eighth of a second, `sr/4.0`. Note that the shift can be positive or negative. The quantity `sr/4.0` is the number of samples in a quarter of a second. However, the random float is in the range `[-0.5, +0.5]`, so the ultimate shift is at most an eighth of a second. If the shift is negative, we need to zero samples at the end of the data; otherwise, we zero samples at the start.

Random noise is added by literally adding a random value of up to one-tenth of the range of the audio signal back in ❷. When played, this adds hiss, as you might hear on an old cassette tape.

Next comes shifting the pitch of the sample with `librosa`. The pitch shift is expressed in musical steps, or fractions thereof. We randomly pick a float in the range `[-10, +10]` (`pf`) and pass it along with the data (`d`) and sampling rate (`sr`) to the `librosa pitch_shift` effect function ❸.

The last augmentation uses `librosa`'s function to stretch or compress time (`time_stretch`) ④. We adjust using an amount of time (`rate`) that is in the range $[-0.5, +0.5]$. If time was stretched, we need to chop off the extra samples to ensure that the sample length remains constant. If time was compressed, we need to add zero samples at the end. Lastly, we return the new, augmented samples.

Running the code in Listing 12-2 takes about 11 minutes and creates a new *augmented* data directory in the *data/audio/ESC-10* directory with subdirectories *train* and *test*. These are the raw sound files that we'll work with going forward. I encourage you to listen to some of them to understand what the augmentations have done. The filenames should differentiate the originals from the augmentations.

Preprocessing the Data

Are we ready to start building models? Not yet. Our experience told us that the dataset was too small, and we augmented accordingly. However, we haven't yet turned the raw data into something we can pass to a model.

A first thought is to use the raw sound samples. These are already vectors representing the audio signal, with the time between the samples set by the sampling rate of 44,100 Hz. But we don't want to use them as they are. The samples are all exactly five seconds long. At 44,100 samples per second, that means each sample is a vector of $44,100 \times 5 = 220,500$ samples. That's too long for us to work with effectively.

With a bit more thought, we might be able to convince ourselves that distinguishing between a crying baby and a barking dog might not need such a high sampling rate. What if instead of keeping all the samples, we kept only every 100th sample? Moreover, do we really need five seconds' worth of data to identify the sounds? What if we kept only the first two seconds?

Let's keep only the first two seconds of each sound file; that's 88,200 samples. And let's keep only every 100th sample, so each sound file now becomes a vector of 882 elements. That's hardly more than an unraveled MNIST digit image, and we know we can work with those.

Listing 12-5 has the code to build the actual initial version of the dataset we'll use to build the models.

```
import os
import random
import numpy as np

from scipy.io.wavfile import read

sr = 44100 # Hz
N = 2*sr   # number of samples to keep
w = 100    # every 100 (0.01 s)

afiles = [i[:-1] for i in open("../data/audio/ESC-
10/augmented_train_filelist.txt")]
trn = np.zeros((len(afiles), N/w, 1), dtype="int16")
lbl = np.zeros(len(afiles), dtype="uint8")

for i,t in enumerate(afiles):
    f, c = t.split() ❶
    trn[i,:,0] = read(f)[1][:N:w]
    lbl[i] = int(c)

np.save("../data/audio/ESC-10/esc10_raw_train_audio.npy", trn)
np.save("../data/audio/ESC-10/esc10_raw_train_labels.npy", lbl)

afiles = [i[:-1] for i in open("../data/audio/ESC-
10/augmented_test_filelist.txt")]
tst = np.zeros((len(afiles), N/w, 1), dtype="int16")
lbl = np.zeros(len(afiles), dtype="uint8")

for i, t in enumerate(afiles):
    f, c = t.split()
    tst[i,:,0] = read(f)[1][:N:w]
    lbl[i] = int(c)

np.save("../data/audio/ESC-10/esc10_raw_test_audio.npy", tst)
np.save("../data/audio/ESC-10/esc10_raw_test_labels.npy", lbl)
```

Listing 12-5: Building the reduced samples dataset

This code builds train and test NumPy files containing the raw data. The data is from the augmented sound files we built in Listing 12-2. The file list contains the file location and class label ❶. We load each file in the list and put it into an array, either the train or test array.

We have a 1D feature vector and a number of train or test files, so we might expect we need a 2D array to store our data, either 6,400×882 for the training set or 1,600×882 for the test set. However, we know we'll ultimately be working with Keras, and we know that Keras wants a dimension for the number of channels, so we define the arrays to be 6,400×882×1 and 1,600×882×1 instead. The most substantial line in this code is the following:

```
trn[i,:,0] = read(f)[1][:N:w]
```

It reads the current sound file, keeps only the sound samples ([1]), and from the sound samples keeps only the first two seconds' worth at every 100th sample, [:N:w]. Spend a little time with this code. If you're confused, I suggest experimenting with NumPy at the interactive Python prompt to understand what it's doing.

In the end, we have train and test files for the 882-element vectors and associated labels. We'll build our first models with these. Figure 12-1 shows the resulting vector for a crying baby.

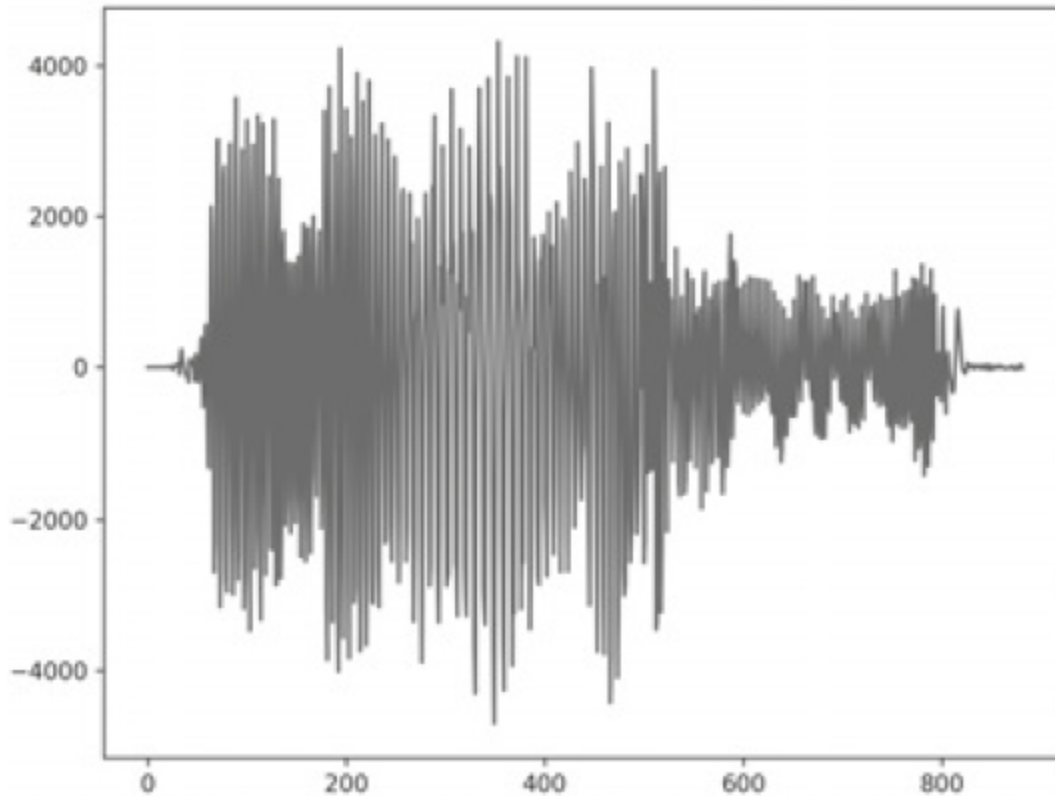


Figure 12-1: The feature vector for a crying baby

The x-axis is the sample number (think *time*), and the y-axis is the sample value.

Classifying the Audio Features

We have our training and test sets. Let's build some models and see how they do. Since we have feature vectors, we can start quickly with classical models. After those, we can build some 1D convolutional networks and see if they perform any better.

With Classical Models

We can test the same suite of classical models we used in Chapter 4 with the breast cancer dataset. Listing 12-6 has the setup code.

```
esc10_audio_classical.py
import numpy as np
from sklearn.neighbors import NearestCentroid
```

```

from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import LinearSVC

x_train = np.load("../data/audio/ESC-10/esc10_raw_train_audio.npy")[:, :, 0]
y_train = np.load("../data/audio/ESC-10/esc10_raw_train_labels.npy")

x_test = np.load("../data/audio/ESC-10/esc10_raw_test_audio.npy")[:, :, 0]
y_test = np.load("../data/audio/ESC-10/esc10_raw_test_labels.npy")

❶ x_train = (x_train.astype('float32') + 32768) / 65536
   x_test = (x_test.astype('float32') + 32768) / 65536

train(x_train, y_train, x_test, y_test)

```

Listing 12-6: Classifying the audio features with classical models, part 1

We import the necessary model types, load the dataset, scale it, and then call a `train` function that we'll introduce shortly.

Scaling is crucial here. Consider the y-axis range for Figure 12-1. It goes from about $-4,000$ to $4,000$. We need to scale the data so that the range is smaller and the values are closer to being centered around 0. Recall, for the MNIST and CIFAR-10 datasets, we divided by the maximum value to scale to $[0, 1]$.

The sound samples are 16-bit signed integers. This means the full range of values they can take on covers $[-32,768, +32,767]$. If we make the samples floats, add 32,768, and then divide by 65,536 (twice the lower value) ❶, we'll get samples in the range $[0, 1)$, which is what we want.

Training and evaluating the classical models is straightforward, as Listing 12-7 shows.

```

def run(x_train, y_train, x_test, y_test, clf):
    clf.fit(x_train, y_train)
    score = 100.0*clf.score(x_test, y_test)
    print("score = %0.2f%%" % score)

```

```

def train(x_train, y_train, x_test, y_test):
    print("Nearest Centroid          : ", end='')
    run(x_train, y_train, x_test, y_test, NearestCentroid())
    print("k-NN classifier (k=3)      : ", end='')
    run(x_train, y_train, x_test, y_test, KNeighborsClassifier(n_neighbors=3))
    print("k-NN classifier (k=7)      : ", end='')
    run(x_train, y_train, x_test, y_test, KNeighborsClassifier(n_neighbors=7))
    print("Naive Bayes (Gaussian)       : ", end='')
    run(x_train, y_train, x_test, y_test, GaussianNB())
    print("Random Forest (trees= 5) : ", end='')
    run(x_train, y_train, x_test, y_test,
        RandomForestClassifier(n_estimators=5))
    print("Random Forest (trees= 50) : ", end='')
    run(x_train, y_train, x_test, y_test,
        RandomForestClassifier(n_estimators=50))
    print("Random Forest (trees=500) : ", end='')
    run(x_train, y_train, x_test, y_test,
        RandomForestClassifier(n_estimators=500))
    print("Random Forest (trees=1000): ", end='')
    run(x_train, y_train, x_test, y_test,
        RandomForestClassifier(n_estimators=1000))

    print("LinearSVM (C=0.01)          : ", end='')
    run(x_train, y_train, x_test, y_test, LinearSVC(C=0.01))
    print("LinearSVM (C=0.1)           : ", end='')
    run(x_train, y_train, x_test, y_test, LinearSVC(C=0.1))
    print("LinearSVM (C=1.0)           : ", end='')
    run(x_train, y_train, x_test, y_test, LinearSVC(C=1.0))
    print("LinearSVM (C=10.0)          : ", end='')
    run(x_train, y_train, x_test, y_test, LinearSVC(C=10.0))

```

Listing 12-7: Classifying the audio features with classical models, part 2

The `train` function creates the particular model instances and then calls `run`. We saw this same code structure in Chapter 4. The `run` function uses `fit` to train the model and `score` to score the model on the test set. For the time being, we'll evaluate the models based solely on their

overall accuracy (the score). Running this code takes about eight minutes to produce output like this:

Nearest Centroid	: score = 11.9%
k-NN classifier (k=3)	: score = 12.1%
k-NN classifier (k=7)	: score = 10.5%
Naive Bayes (Gaussian)	: score = 28.1%
Random Forest (trees= 5)	: score = 22.6%
Random Forest (trees= 50)	: score = 30.8%
Random Forest (trees=500)	: score = 32.8%
Random Forest (trees=1000)	: score = 34.4%
LinearSVM (C=0.01)	: score = 16.5%
LinearSVM (C=0.1)	: score = 17.5%
LinearSVM (C=1.0)	: score = 13.4%
LinearSVM (C=10.0)	: score = 10.2%

We can quickly see that the classical models have performed terribly. Many are essentially guessing the class label. There are 10 classes, so random chance guessing should have an accuracy of around 10 percent. The best-performing classical model is a random forest with 1,000 trees, but even that is performing at only 34.44 percent—far too low an overall accuracy to make the model one we’d care to use in most cases. The dataset is not a simple one, at least not for old-school approaches. Somewhat surprisingly, the Gaussian naive Bayes model is right 28 percent of the time. Recall that the Gaussian naive Bayes expects the samples to be independent of one another. Here the independence assumption between the sound samples for a particular test input is not valid. The feature vector, in this case, represents a signal evolving in time, not a collection of independent features.

The models that failed the most are nearest centroid, k -NN, and the linear SVMs. We have a reasonably high-dimensional input of 882 elements, but only 6,400 of them in the training set. That is likely too few samples for the nearest-neighbor classifiers to make use of; the feature space is too sparsely populated. Once again, the curse of dimensionality is rearing its ugly head. The linear SVM fails because the features seem not to be linearly separable. We did not try an RBF (Gaussian kernel) SVM, but we’ll leave that as an exercise for the reader.

If you do try it, remember that there are now two hyperparameters to tune: C and γ .

With a Traditional Neural Network

We haven't yet tried a traditional neural network. We'll use the `sklearn.MLPClassifier` class as we did before; Listing 12-8 has the code.

```
esc10_audio_mlp.py
from sklearn.neural_network import MLPClassifier
import numpy as np

num_classes = 10

x_train = np.load("../data/audio/ESC-10/esc10_raw_train_audio.npy")[:, :, 0]
y_train = np.load("../data/audio/ESC-10/esc10_raw_train_labels.npy")
x_test = np.load("../data/audio/ESC-10/esc10_raw_test_audio.npy")[:, :, 0]
y_test = np.load("../data/audio/ESC-10/esc10_raw_test_labels.npy")

x_train = (x_train.astype('float32') + 32768) / 65536
x_test = (x_test.astype('float32') + 32768) / 65536

model = MLPClassifier(hidden_layer_sizes=(512, 128),
                      max_iter=200,
                      solver='lbfgs')
model.fit(x_train, y_train)
score = 100.0*model.score(x_test, y_test)
print("score = %0.2f%%" % score)
```

Listing 12-8: Using a traditional neural network

After loading the necessary modules, we load the data itself and scale it as we did for the classical models. Next, we build the model by using two hidden layers of 512 and 128 nodes, respectively, then train by calling `fit`.

The `MLPClassifier` constructor has a new keyword, `solver`, which is set to `lbfgs`. This is new. *Solver* is sklearn-speak for an optimizer. If not specified, it defaults to Adam. Here I've set the solver to use the L-

BFGS algorithm instead. The training set is small, 6,400 vectors, and that isn't enough to get a traditional neural network to learn anything when using SGD or Adam. In those cases, the final test-set accuracy is 10 percent, pure random guessing.

The label `lbfgs` stands for *limited-memory BFGS*, and *BFGS* stands for *Broyden–Fletcher–Goldfarb–Shanno*. Standard machine learning optimizers, as we know, use the gradient, which is the multivariable version of the derivative; think of it as the slope of a curve at a point. Optimization algorithms based on the gradient alone are known as *first-order algorithms*, as discussed in Chapter 6. Both SGD and Adam are first-order algorithms.

Ideally, optimization makes use of what we might call the *gradient of the gradient*—that is, information about how the gradient itself is changing in the vicinity of a point. This is the second derivative, so algorithms using this information are known as *second-order* optimization algorithms. The BFGS algorithm falls into this class. The “limited” part approximates the second-order information making the algorithm useful for high-dimensional spaces, like the 882-element vectors we’re working with in this example. Therefore, we might expect the L-BFGS algorithm to do a better job than SGD or Adam. Unfortunately, our expectation, while true, isn’t anything to brag about; even with L-BFGS, the best we achieve is 19.1 percent accuracy, placing the traditional neural network near the bottom of the pack of models tested so far.

With a Convolutional Neural Network

Classical models and the traditional neural network don’t cut it. We shouldn’t be too surprised, but giving them a try was easy. Let’s move on and apply a 1D convolutional neural network to this dataset to see whether it performs any better.

We haven’t worked with 1D CNNs yet. Besides the structure of the input data, the only difference is that we replace calls to `conv2d` and `MaxPooling2D` with calls to `conv1d` and `MaxPooling1D`.

Listing 12-9 shows the code for the first model we’ll try.

```
import tensorflow.keras as keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv1D, MaxPooling1D
import numpy as np

batch_size = 32
num_classes = 10
epochs = 16
nsamp = (882, 1)
x_train = np.load("../data/audio/ESC-10/esc10_raw_train_audio.npy")
y_train = np.load("../data/audio/ESC-10/esc10_raw_train_labels.npy")
x_test = np.load("../data/audio/ESC-10/esc10_raw_test_audio.npy")
y_test = np.load("../data/audio/ESC-10/esc10_raw_test_labels.npy")
x_train = (x_train.astype('float32') + 32768) / 65536
x_test = (x_test.astype('float32') + 32768) / 65536
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
model = Sequential()
model.add(Conv1D(32, kernel_size=3, activation='relu',
                input_shape=nsamp))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])
history = model.fit(x_train, y_train,
                   batch_size=batch_size,
                   epochs=epochs,
                   verbose=1,
                   validation_data=(x_test, y_test))
```

```
score = model.evaluate(x_test, y_test, verbose=0)
print('Test accuracy:', score[1])
```

Listing 12-9: A 1D CNN in Keras

This model loads and preprocesses the dataset as before. This architecture, which we'll call the *shallow* architecture, has a single convolutional layer of 32 filters with a kernel size of 3. We'll vary this kernel size in the same way we tried different 2D kernel sizes for the MNIST models. Following the `Conv1D` layer is a single max-pooling layer with a pool kernel size of 3. `Dropout` and `Flatten` layers come next, before a single `Dense` layer of 512 nodes with dropout. A softmax layer completes the architecture.

We'll train for 16 epochs by using a batch size of 32. We'll keep the training history so we can examine the losses and validation performance as a function of epoch. We have 1,600 test samples, and we use all of them to monitor (but not alter) training and for performance testing. Finally, we'll vary the `Conv1D` kernel size from 3 to 33 in an attempt to find one that works well with the training data.

Let's define four other architectures. We'll refer to them as *medium*, *deep0*, *deep1*, and *deep2*. With no prior experience working with this data, trying multiple architectures makes sense. At present, there's no way to know the best architecture for a new dataset ahead of time. All we have is our previous experience.

Listing 12-10 lists the specific architectures, separated by comments.

```
# medium
model = Sequential()
model.add(Conv1D(32, kernel_size=3, activation='relu',
                 input_shape=nsamp))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
```

```

model.add(Dense(num_classes, activation='softmax'))

# deep0
model = Sequential()
model.add(Conv1D(32, kernel_size=3, activation='relu',
                 input_shape=nsamp))

model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

# deep1
model = Sequential()
model.add(Conv1D(32, kernel_size=3, activation='relu',
                 input_shape=nsamp))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(512, activation='relu'))

```

```

model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

# deep2
model = Sequential()
model.add(Conv1D(32, kernel_size=3, activation='relu',
                 input_shape=nsamp))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Conv1D(64, kernel_size=3, activation='relu'))

model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(Conv1D(64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=3))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

```

Listing 12-10: Various 1D CNN architectures

The files *esc10_audio_cnn_shallow.py* through *esc10_audio_cnn_deep2.py* contain the code. Each file expects an integer argument on the command line that it uses as the kernel size for the first convolutional layer. Combining this flexibility with the statements in the *train_all_1d_cnn* script trains and tests each combination of

architecture and kernel size to produce the results in Table 12-1. The best-performing model for each architecture is bolded.

Table 12-1: Test-Set Accuracies by Convolutional Kernel Size and Model Architecture

Kernel size	Shallow	Medium	Deep0	Deep1	Deep2
3	41.63	38.75	47.06	10.00	10.00
5	38.13	40.44	46.82	10.00	10.00
7	38.19	38.69	44.69	50.50	50.44
9	40.13	38.25	42.94	49.19	50.19
11	38.00	39.44	43.19	46.88	10.00
13	37.00	37.06	45.44	49.38	49.63
15	35.25	40.38	41.13	10.00	10.00
33	32.81	36.81	43.06	46.50	10.00

Looking at Table 12-1, we see a general trend of accuracy improving as the model depth increases. However, at the deep2 model, things start to fall apart. Some of the models fail to converge, showing an accuracy equivalent to random guessing. The deep1 model is the best performing for larger kernel sizes. When looking across by kernel size, there is no clear winner among the five architectures, but deep1's highest overall accuracy for a kernel width of 7 implies this might be a good combination.

We trained the deep1 with kernel width 7 architecture for only 16 epochs. Will accuracy improve if we train for more? Let's train the deep1 model for 60 epochs and plot the training and validation loss and error to see how they converge (or don't). Doing this produces Figure 12-2, which shows the training and validation loss and error as a function of epoch.

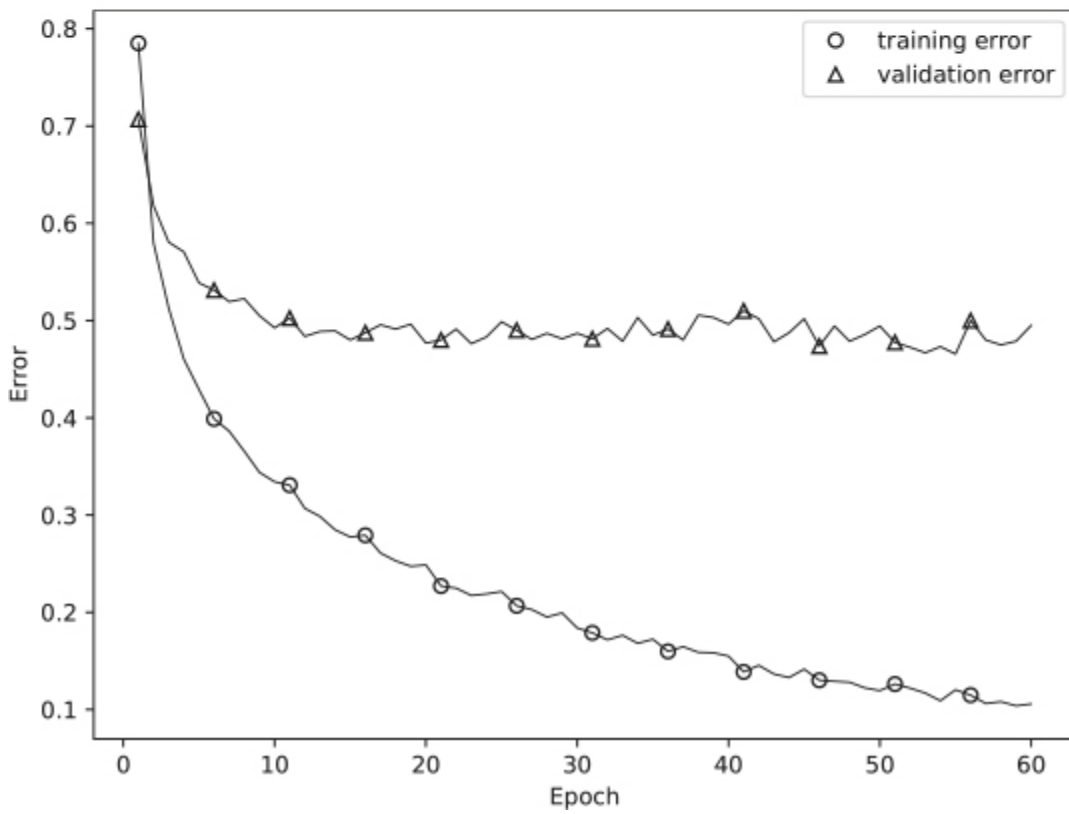
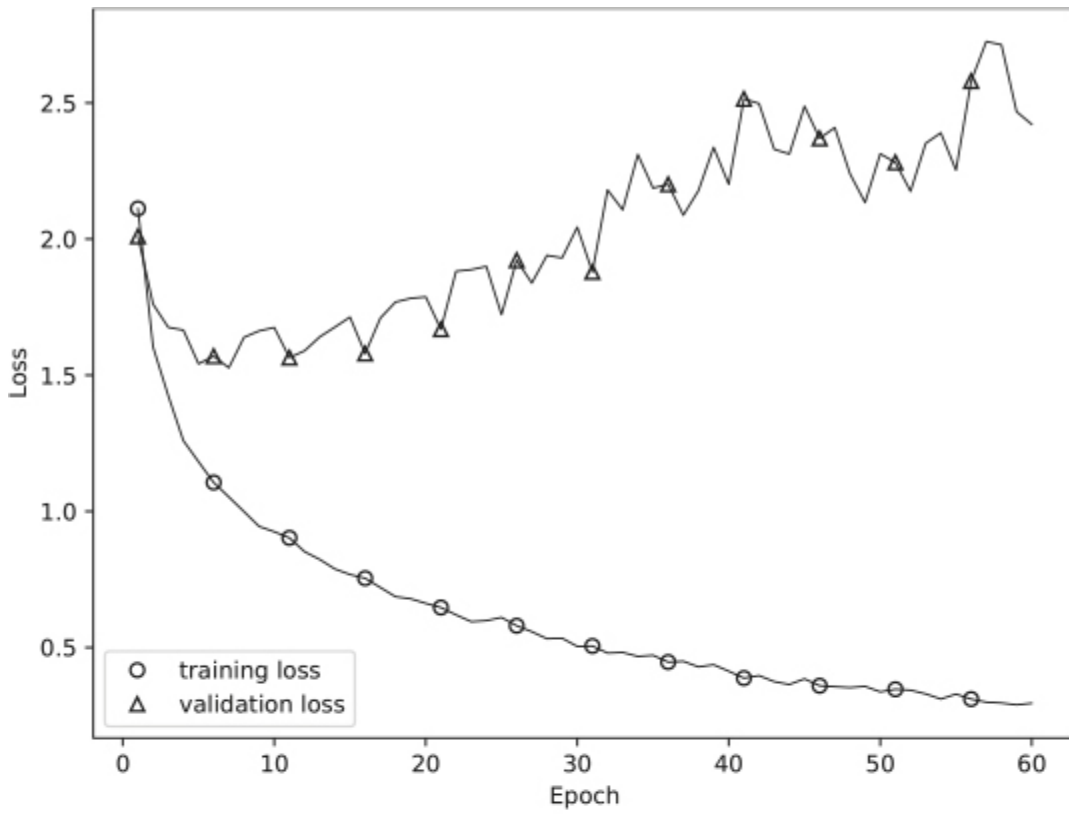


Figure 12-2: The training and validation loss (top) and error (bottom) for the deep1

Immediately, we should pick up on the explosion of the loss for the validation set. The validation loss is continually decreasing until after about epoch 15 or so; then it goes up and becomes oscillatory. This is a clear example of overfitting. The likely source of this overfitting is our limited training set size, only 6,400 samples, even after data augmentation. The validation error remains more or less constant after initially decreasing. The conclusion is that we cannot expect to do much better than an overall accuracy of about 50 percent for this dataset using 1D vectors.

If we want to improve, we need to be more expressive with our dataset. Fortunately for us, we have another preprocessing trick up our sleeves.

Spectrograms

Let's return to our augmented set of audio files. To build the dataset, we took the sound samples, keeping only two seconds' worth and only every 100th sample. The best we could do is an accuracy of a little more than 50 percent.

However, if we work with a small set of sound samples from an input audio file, say 200 milliseconds' worth, we can use the vector of samples to calculate the Fourier transform. The *Fourier transform* of a signal measured at regular intervals tells us the frequencies that went into building the signal. Any signal can be thought of as the sum of many sine and cosine waves. If the signal is composed of only a few waves, like the sound you might get from an instrument such as the ocarina, then the Fourier transform will have essentially a few peaks at those frequencies. If the signal is complex, like speech or music, then the Fourier transform will have many frequencies, leading to many peaks.

The Fourier transform itself is complex-valued: Each element has both a real and an imaginary component. You can write it as $a + bi$, where a and b are real numbers and $i = \sqrt{-1}$. If we use the absolute value of these quantities, we'll get a real number representing the energy of a particular frequency. This is called the *power spectrum* of the

signal. A simple tone might have energy in only a few frequencies, while something like a cymbal crash or white noise will have energy more or less evenly distributed among all frequencies. Figure 12-3 shows two power spectra.

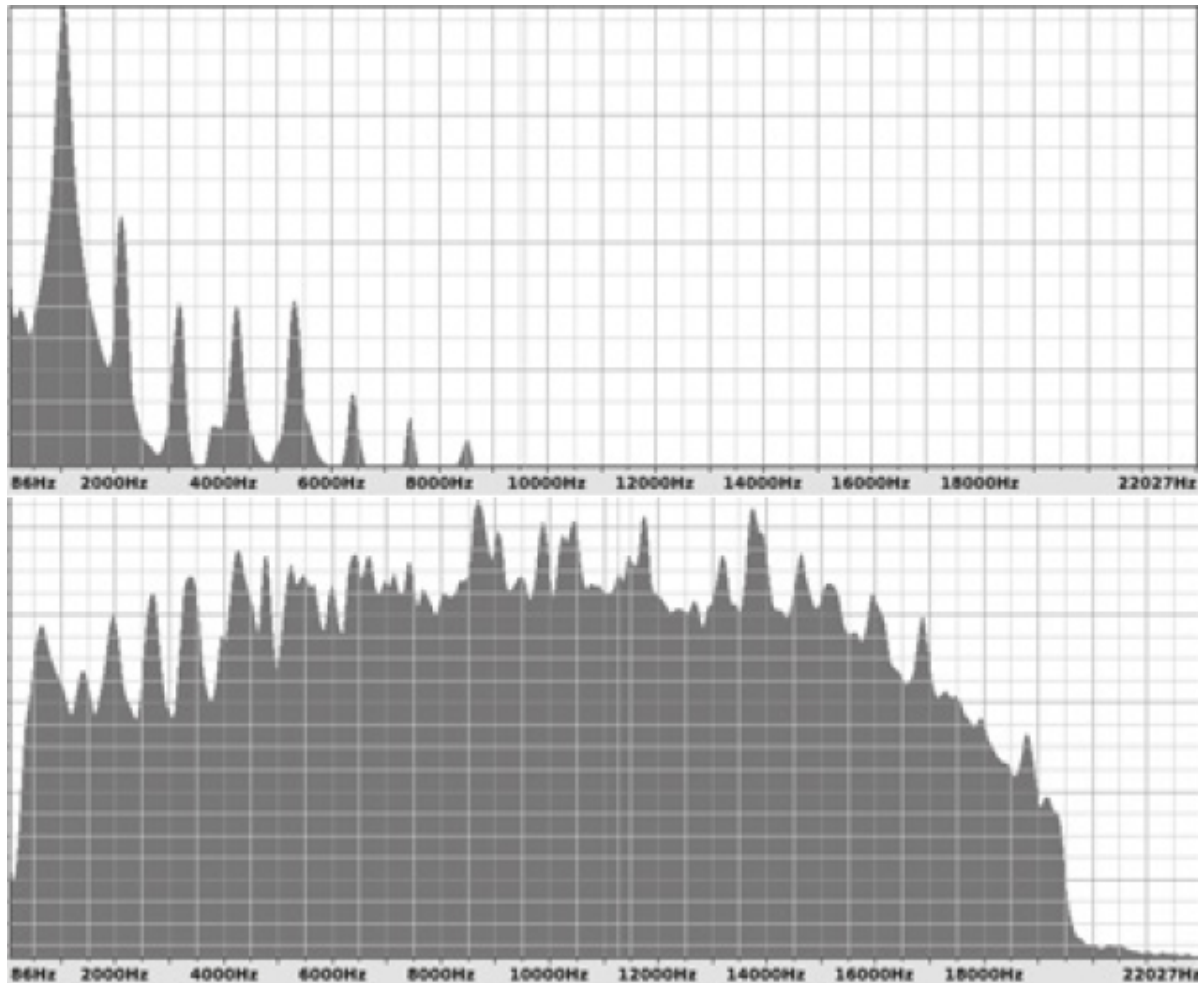


Figure 12-3: The power spectrum of an ocarina (top) and a cymbal (bottom)

On the top is the spectrum of an ocarina, and on the bottom is a cymbal crash. As expected, the ocarina has energy in only a few frequencies, while the cymbal uses all the frequencies. The important point for us is that *visually* the spectra are quite different from each other. (The spectra were made with Audacity, an excellent open source audio-processing tool.)

We could use these power spectra as feature vectors, but they represent only the spectra of tiny slices of time. The sound samples are

five seconds long. Instead of using a spectrum, we will use a *spectrogram*. This is an image made up of columns that represent individual spectra. The x-axis represents time, and the y-axis represents frequency. The color of a pixel is proportional to the energy in that frequency at that time.

In other words, a spectrogram is what we get if we orient the power spectra vertically and use color to represent intensity at a given frequency. With this approach, we can turn an entire sound sample into an image. For example, Figure 12-4 shows the spectrogram of a crying baby. Compare this to the feature vector of Figure 12-1.

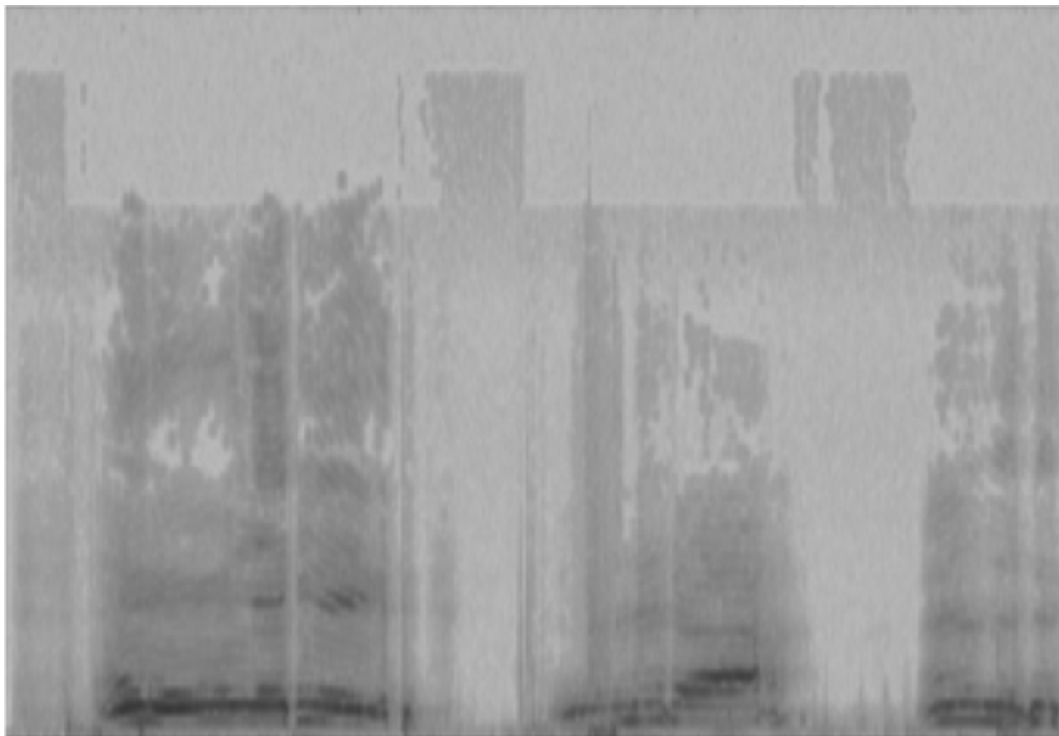


Figure 12-4: The spectrogram of a crying baby

To create spectrograms of the augmented audio files, we need a new tool and a bit of code. The tool we need is called `sox`. It's not a Python library, but a command line tool. Odds are that it is already installed if you're using our canonical Ubuntu Linux distribution. If not, you can install it:

```
$ sudo apt-get install sox
```

We'll use `sox` from inside a Python script to produce the spectrogram images we want. Each sound file becomes a new spectrogram image.

The source code to process the training images is in Listing 12-11 (see *make_augmented_spectrograms.py*).

```
import os
import numpy as np
from PIL import Image

rows = 100
cols = 160
flist = [i[:-1] for i in open("../data/audio/ESC-
10/augmented_train_filelist.txt")] ❶
N = len(flist)
img = np.zeros((N, rows, cols, 3), dtype="uint8")
lbl = np.zeros(N, dtype="uint8")
p = []

for i,f in enumerate(flist):
    src, c = f.split()
    os.system("sox %s -n spectrogram" % src) ❷
    im = np.array(Image.open("spectrogram.png").convert("RGB"))
    im = im[42:542,58:858,:] ❸
    im = Image.fromarray(im).resize((cols,rows))
    img[i,:,:,:] = np.array(im)
    lbl[i] = int(c)
    p.append(os.path.abspath(src))

os.system("rm -rf spectrogram.png")
p = np.array(p)
idx = np.argsort(np.random.random(N)) ❹
img = img[idx]
lbl = lbl[idx]
p = p[idx]
np.save("../data/audio/ESC-10/esc10_spect_train_images.npy", img)
np.save("../data/audio/ESC-10/esc10_spect_train_labels.npy", lbl)
np.save("../data/audio/ESC-10/esc10_spect_train_paths.npy", p)
```

Listing 12-11: Building the spectrograms

We start by defining the size of the spectrogram. This is the input to our model, and we don't want it to be too big because we're limited in the size of the inputs we can process. We'll settle for 100×160 pixels. We then load the training file list ❶ and create NumPy arrays to hold the spectrogram images and associated labels. The list `p` will hold the pathname of the source for each spectrogram in case we want to get back to the original sound file at some point. In general, it's a good idea to preserve information to get back to the source of derived datasets.

Then we loop over the file list. We get the filename and class label and then call `sox`, passing in the source sound filename ❷. The `sox` application is sophisticated. The syntax here turns the given sound file into a spectrogram image with the name *spectrogram.png*. We immediately load the output spectrogram into `im`, making sure it's an RGB file with no transparency layer (hence the call to `convert("RGB")`).

The spectrogram created by `sox` has a border with frequency and time information. We want only the spectrogram image portion, so we subset the image ❸. We determined the indices we're using empirically. It's possible, but somewhat unlikely, that a newer version of `sox` will require tweaking these to avoid including any border pixels.

Next, we resize the spectrogram so that it fits in our 100×160-pixel array. This is downsampling, true, but hopefully enough characteristic information is still present to allow a model to learn the difference between classes. We keep the downsampled spectrogram and the associated class label and sound filepath.

When we've generated all the spectrograms, the loop ends, and we remove the final extraneous spectrogram PNG file. We convert the list of sound filepaths to a NumPy array so we can store it in the same manner as the images and labels. Finally, we randomize the order of the images as a precaution against any implicit sorting that might group classes ❹. This is so that minibatches extracted sequentially are representative of the mix of classes as a whole. To conclude, we write the images, labels, and pathnames to disk. We repeat this entire process for the test set.

Are we able to visually tell the difference between the spectrograms of different classes? If we can do that easily, we have a good shot of getting a model to tell the difference too. Figure 12-5 shows 10 spectrograms of the same class in each row.

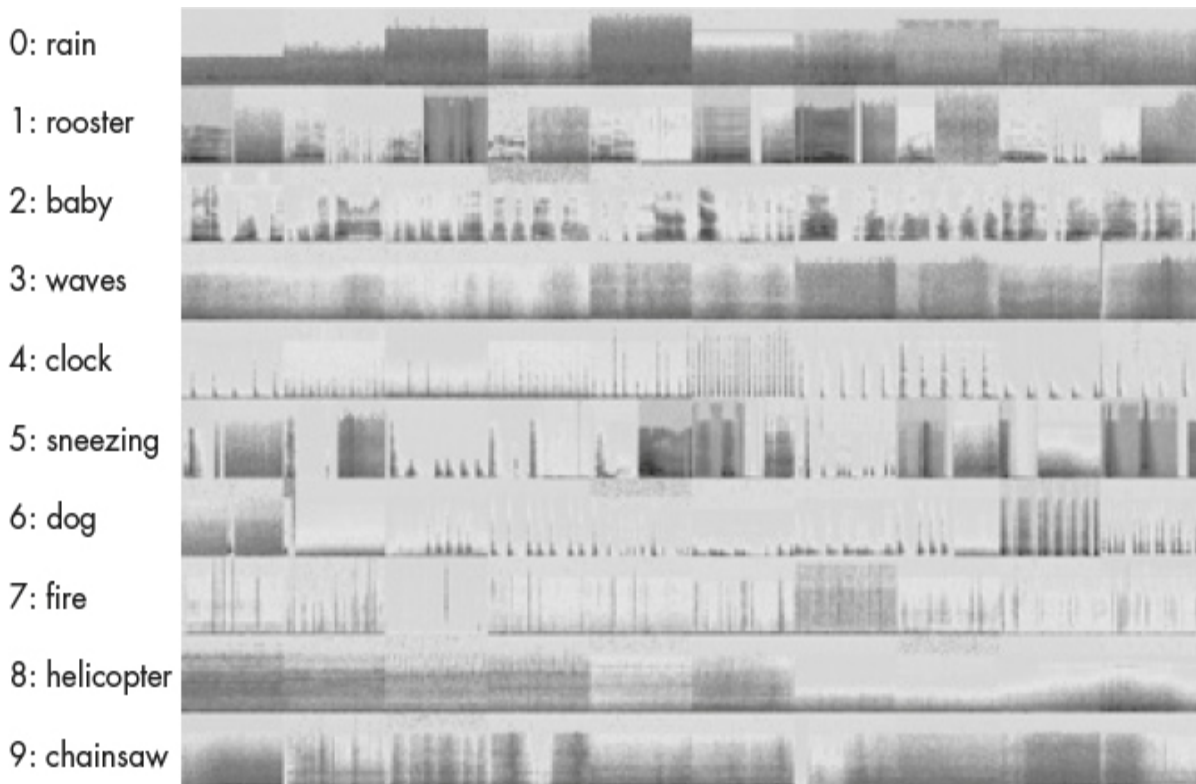


Figure 12-5: The sample spectrograms for each class in ESC-10. Each row shows 10 examples from the same class.

Visually, we can usually tell the spectra apart, which is encouraging. With our spectrograms in hand, we're ready to try some 2D CNNs to see if they do better than the 1D CNNs.

Classifying Spectrograms

To work with the spectrogram dataset, we need 2D CNNs. A possible starting point is to convert the shallow 1D CNN architecture to 2D by changing `Conv1D` to `Conv2D`, and `MaxPooling1D` to `MaxPooling2D`. However, if we do this, the resulting model has 30.7 million parameters, which is many more than we want to work with. Instead, let's opt for a deeper

architecture that has fewer parameters and then explore the effect of different kernel sizes for the first convolutional layer. The code is in Listing 12-12.

```
esc10_cnn_deep.py
import sys
import pickle
import tensorflow.keras as keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
import numpy as np

# kernel size -- 3 or 7
z = int(sys.argv[1])
batch_size = 16
num_classes = 10
epochs = 16
img_rows, img_cols = 100, 160
input_shape = (img_rows, img_cols, 3)

x_train = np.load("../data/audio/ESC-10/esc10_spect_train_images.npy")
y_train = np.load("../data/audio/ESC-10/esc10_spect_train_labels.npy")
x_test = np.load("../data/audio/ESC-10/esc10_spect_test_images.npy")
y_test = np.load("../data/audio/ESC-10/esc10_spect_test_labels.npy")

x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

model = Sequential()
model.add(Conv2D(32, kernel_size=(z, z),
                activation='relu',
                input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
```

```

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])

history = model.fit(x_train, y_train,
                   batch_size=batch_size,
                   epochs=epochs,
                   verbose=0,
                   validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('%dx%d: accuracy: %0.3f' % (z,z,100.0*score[1]))

```

Listing 12-12: Classifying spectrograms

We're using a minibatch size of 16 for 16 epochs along with the Adam optimizer. The model architecture has two convolutional layers, a max-pooling layer with dropout, another convolutional layer, and a second max-pooling layer with dropout. There is a single dense layer of 128 nodes before the softmax output.

We'll test two kernel sizes for the first convolutional layer: 3×3 and 7×7. The kernel size is expected on the command line.

All the initial 1D convolutional runs used a single training of the model for evaluation. Because of random initialization, we'll get slightly different results from training to training, even if nothing else changes. For the 2D CNNs, let's train each model six times and present the overall accuracy as a mean ± standard error of the mean. Doing just this gives us the following overall accuracies:

Kernel size	Score
3×3	75.54 ± 0.92%
7×7	76.18 ± 0.85%

The standard error ranges overlap, indicating this experiment finds no meaningful difference between using a 3×3 initial convolutional layer kernel size or a 7×7. Therefore, we'll stick with 3×3 going forward.

As an aside, just because the results from six runs of each kernel size aren't statistically significantly different doesn't mean there is, truly, no difference between 3×3 and 7×7 kernels. There might be a true difference, but the effect size is small enough that six trials isn't able to reveal it. What might we get with 100 trials of each kernel size? If those results are statistically significantly different, with a p-value well below 0.05, then we might come to believe one kernel size is superior to the other, but my money is on such a difference having a small Cohen's effect size (d), making the difference real but meaningless in practice. The "it's real but effectively meaningless in practice" scenario is more common than we might think.

Figure 12-6 shows the training and validation loss and error for one run of the 2D CNN trained on the spectrograms. As we saw in the 1D CNN case, after only a few epochs, the validation error starts to increase, but not as dramatically.

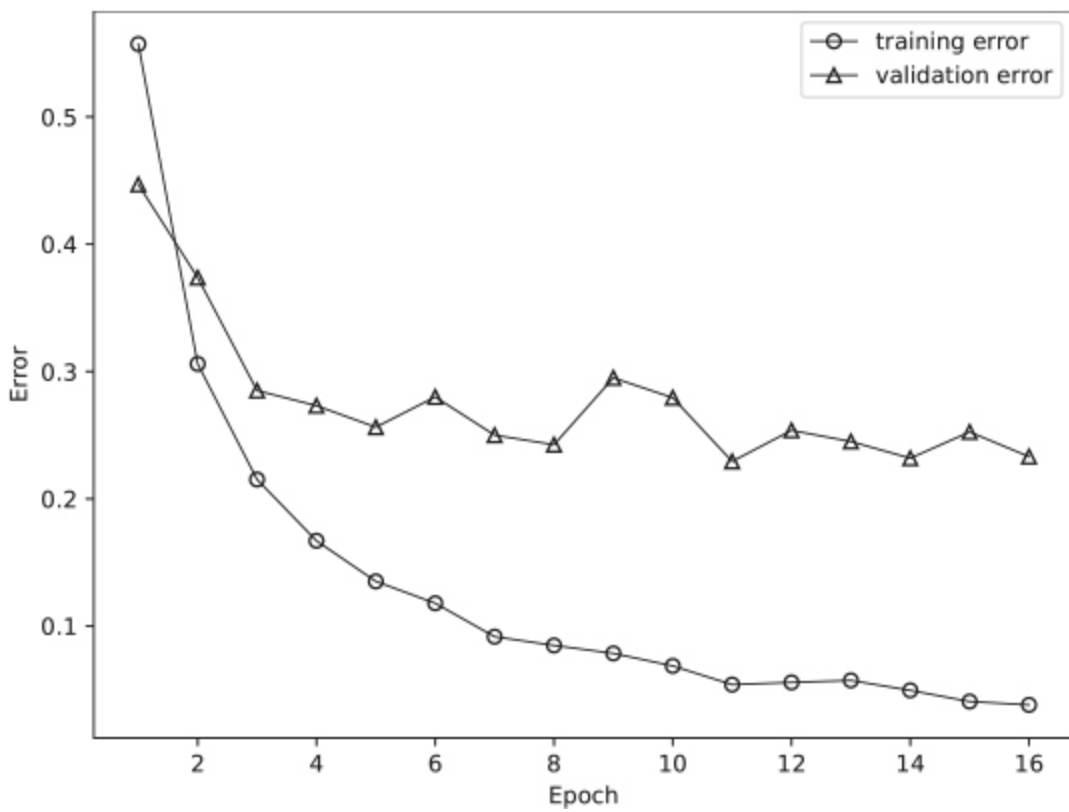
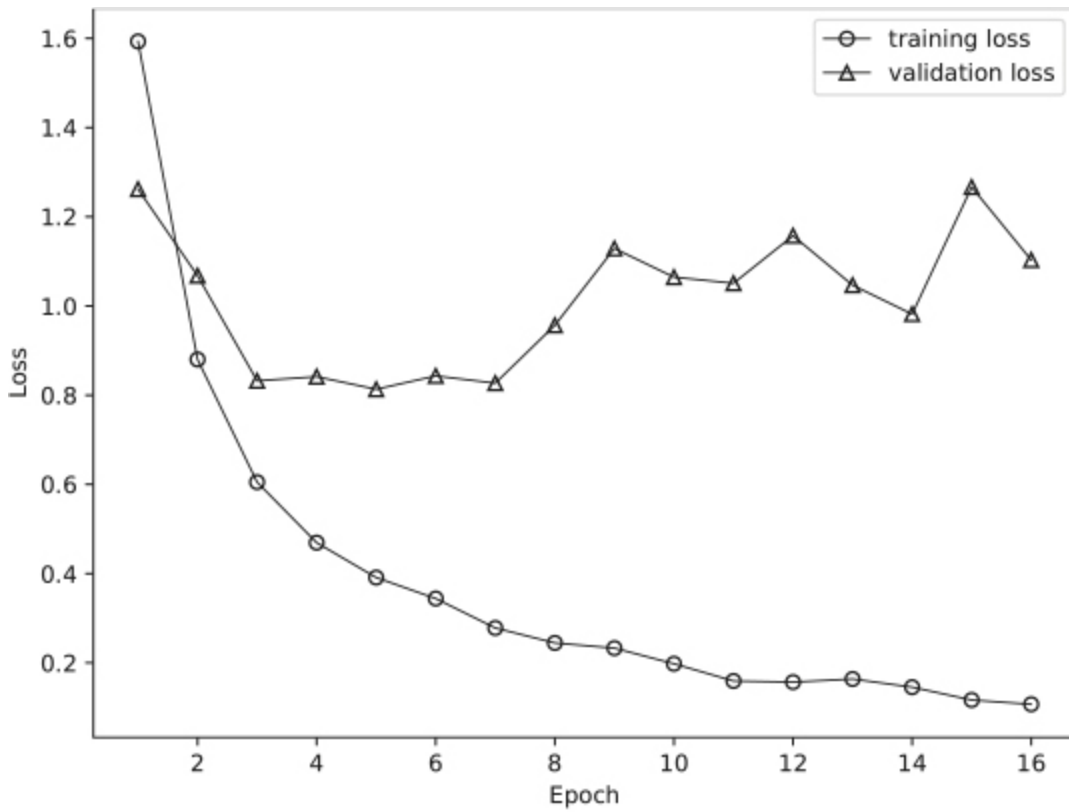


Figure 12-6: The training and validation loss (top) and error (bottom) for the 2D CNN

The 2D CNN performs significantly better than the 1D CNN: 75 percent accuracy versus only 50 percent. This level of accuracy is still not particularly useful for many applications, but for others, it might be completely acceptable. Nevertheless, we'd like to do better if we can.

Let's take a quick look at the confusion matrix generated by one of the models using our chosen architecture. We've seen previously how to calculate the matrix; we'll show it here for discussion and for comparison with the confusion matrices we'll make in the next section. To generate it, run *stats.py* passing in a trained model (a *.keras* file).

Table 12-2 shows the matrix; as always, rows are the true class label, and columns are the model-assigned label. The elements have been scaled by the number of examples per class so that the entries in the matrix represent percentages. For example, 93.8 percent of class 1 examples were correctly labeled by the model.

Table 12-2: The Confusion Matrix for the Spectrogram Model

Class	0	1	2	3	4	5	6	7	8	9
0	67.5	0.0	0.0	27.5	0.0	0.0	0.0	0.0	0.6	4.4
1	0.0	93.8	5.0	0.0	0.0	0.6	0.6	0.0	0.0	0.0
2	0.0	9.4	79.4	0.0	0.6	0.0	0.0	0.0	0.0	10.6
3	13.8	0.0	0.0	68.8	0.0	0.0	0.0	1.2	3.1	13.1
4	0.6	0.0	0.0	0.0	83.1	0.0	0.0	16.2	0.0	0.0
5	0.0	1.9	4.4	0.0	13.1	60.6	16.2	3.8	0.0	0.0
6	0.0	3.8	2.5	0.0	8.1	1.2	82.5	1.9	0.0	0.0
7	0.0	0.0	0.0	0.0	2.5	0.0	0.0	89.4	8.1	0.0
8	11.2	0.0	0.0	2.5	1.2	0.0	0.0	6.9	77.5	0.6
9	16.9	0.0	0.6	5.6	0.0	0.0	0.0	0.0	8.1	68.8

The two worst-performing classes are sneezing (5) and rain (0). Rain is most often confused with waves (3), which seems reasonable. Sneezing is roughly equally confused with a clock (4) or dog barking (6), with the edge going to the dogs. I can understand dogs, maybe, but confusing a sneeze for a ticking clock is a bit odd. Consider it an example of the mysterious ways of deep learning models. The two best-performing classes are a rooster (1) and a crackling fire (7).

Does this mean we're stuck at 76 percent accuracy? No, we have one more trick to try. We've been training and evaluating the performance of single models, but nothing is stopping us from training multiple models and combining their results. This is *ensembling*. We presented ensembles briefly in Chapter 3 and again in Chapter 6 when discussing dropout. Now let's use the idea directly to see if we can improve our sound sample classifier.

Ensembles

The core idea of an ensemble is to take the output of multiple models trained on the same, or extremely similar, dataset(s) and combine them. It embodies the “wisdom of the crowds” concept: One model might be better at certain classes or types of inputs for a particular class than another, so it follows that if they work together, they might arrive at a final result better than either one could do on its own.

We'll use the same machine learning architecture we used in the previous section. Our different models will be separate trainings of this architecture, using the spectrograms as input. This is a weaker form of ensembling. Typically, the models in the ensemble are quite different from one another, either different architectures of neural networks, or completely different types of models like random forests and k -nearest neighbors. Of course, as we now understand, random forests are themselves ensembles of decision trees. The variation between models here is due to the random initialization of the networks and the different parts of the loss landscape the network finds itself in when training stops. Our approach works like this:

1. Train multiple models ($n = 6$) by using the spectrogram dataset.
2. Combine the softmax output of these models on the test set in some manner.
3. Use the resulting output from the combination to predict the assigned class label.

We hope that the set of class labels assigned after combining the individual model outputs is superior to the set assigned by the model architecture used alone. Intuitively, we feel that this approach should buy us something. It makes sense.

However, a question immediately arises: How do we best combine the outputs of the individual networks? We have total freedom in the answer to that question. What we are looking for is an $f()$ such that:

$$y_{\text{predict}} = f(y_0, y_1, y_2, \dots, y_n)$$

Here $y_i, i = 0, 1, \dots, n$ are the outputs of the n models in the ensemble, and $f()$ is a function, operation, or algorithm that best combines them into a single new prediction, y_{predict} .

Some combination approaches come readily to mind: We could average the outputs and select the largest, keep maximum per-class output across the ensemble and then choose the largest of those, or use voting to decide which class label should be assigned.

We need the predictions of each model on the test set. To get those, run *train_six_cnn*, which is nothing more than a script that executes *esc10_cnn_deep.py* six times by using an additional command line argument to save the test-set predictions in a NumPy file:

```
python3 esc10_cnn_deep.py 3 prob_run0.npy 2>/dev/null
python3 esc10_cnn_deep.py 3 prob_run1.npy 2>/dev/null
python3 esc10_cnn_deep.py 3 prob_run2.npy 2>/dev/null
python3 esc10_cnn_deep.py 3 prob_run3.npy 2>/dev/null
python3 esc10_cnn_deep.py 3 prob_run4.npy 2>/dev/null
python3 esc10_cnn_deep.py 3 prob_run5.npy 2>/dev/null
```

Appending `2>/dev/null` to the command line directs TensorFlow output to the null device to prevent the output from cluttering the screen.

The script takes a little more than four hours to run on my test machine. The per-model accuracy on the test set is displayed when each training run finishes. My run produced the following individual model accuracies:

```
model 0: 76.688%
model 1: 76.375%
model 2: 78.875%
model 3: 78.000%
model 4: 75.937%
model 5: 72.188%
```

These accuracies are in line with the results from earlier in the chapter, but, clearly, not all these models were created (trained) equal. Model 2 is the most accurate at 78.9 percent, while model 5 is rather poor, producing an accuracy of only 72.2 percent. Recall that the only difference between model 2 and model 5 is the random weight initialization—something worth remembering when a paper claims state-of-the-art accuracies yet provides no uncertainty estimates.

The file *ensemble.py* loads the six test-set prediction files, *prob_run0.npy* through *prob_run5.npy*, along with the true test-set class labels and calculates the average predictions, the maximum predictions, and voting among the six models. Listing 12-13 contains the code.

```
import sys
import numpy as np

def Accuracy(y, p):
    cm = np.zeros((10, 10))
    for i in range(len(y)): cm[y[i], p[i]] += 1
    return 100.0*np.diag(cm).sum()/cm.sum()

p0 = np.load("prob_run0.npy"); p1 = np.load("prob_run1.npy")
p2 = np.load("prob_run2.npy"); p3 = np.load("prob_run3.npy")
p4 = np.load("prob_run4.npy"); p5 = np.load("prob_run5.npy")
```

```

y = np.load("../data/audio/ESC-10/esc10_spect_test_labels.npy")

prob = (p0+p1+p2+p3+p4+p5)/6.0
p = np.argmax(prob, axis=1)
print("Accuracy (average) = %0.2f%%" % Accuracy(y,p))

p = np.zeros(len(y), dtype="uint8")
for i in range(len(y)):
    t = np.array([p0[i], p1[i], p2[i], p3[i], p4[i], p5[i]])
    p[i] = np.argmax(t.reshape(60)) % 10
print("Accuracy (maximum) = %0.2f%%" % Accuracy(y, p))

t = np.zeros((6, len(y)), dtype="uint32")
t[0,:] = np.argmax(p0, axis=1); t[1,:] = np.argmax(p1, axis=1)
t[2,:] = np.argmax(p2, axis=1); t[3,:] = np.argmax(p3, axis=1)
t[4,:] = np.argmax(p4, axis=1); t[5,:] = np.argmax(p5, axis=1)
p = np.zeros(len(y), dtype="uint8")
for i in range(len(y)):
    q = np.bincount(t[:,i])
    p[i] = np.argmax(q)
print("Accuracy (voting) = %0.2f%%" % Accuracy(y, p))

```

Listing 12-13: Ensembling the test-set predictions

The `Accuracy` function returns the overall accuracy by first creating the confusion matrix and then calculating the accuracy from it. The next code paragraph loads the six sets of test-set predictions. Remember, the predictions are softmax outputs, a set of 1,600 vectors of 10 elements stored in a 1,600×10 matrix. Finally, `y` holds the true class labels.

The final three code paragraphs calculate the average, maximum, and voting predicted class labels before printing the accuracy. The average is straightforward: Sum the predictions and divide by 6 before using `np.argmax` to assign a class label. The maximum approach uses a loop to first form a vector of 60 elements for each prediction by appending the six softmax vectors. Then, `np.argmax` returns the index of the maximum, which modulo 10 (for the 10 classes) returns a class label corresponding to the largest value in the extended vector. Finally, voting

uses `np.bincount` with `np.argmax` to select the class label that appears most often among the six models.

My run of *ensemble.py* produced the following:

```
Accuracy (average) = 78.88%
```

```
Accuracy (maximum) = 78.81%
```

```
Accuracy (voting) = 78.50%
```

All of these are an improvement over an individual model, with averaging the softmax predictions providing the best performance on the test set, 78.9 percent.

In most cases, we'd stop here, claim victory on this assignment, and present our boss with the ensemble model. However, we're not that easily satisfied. Notice that simple averaging produced the best result from the ensemble approach. Why should we use simple averaging when we know that the six models did not perform equally on the test set? Some were significantly better than the others. Might it be that simple averaging gives the poorer models too much say in the outcome?

Let's mix things up by using a *weighted average* of the models instead. That means, effectively, replacing

```
prob = (p0+p1+p2+p3+p4+p5)/6.0
```

with something like the following for scalar weights w_0 through w_5 that reflect the importance or belief we place on each individual model's quality:

```
prob = w0*p0 + w1*p1 + w2*p2 + w3*p3 + w4*p4 + w5*p5
```

Great! Weighted averaging is a small tweak to the existing approach, but we have a small problem: How do we pick the weights?

A first thought might be to build them from the individual model accuracies. After all, we have them, and they represent a relative measure of each model's quality. To turn them into weights, we divide each accuracy by the sum of all six, then use those weights to calculate the class label. This helps a tiny bit and produces an ensemble accuracy of 79.1 percent.

It's intuitively reasonable to make the weights based on the performance of the individual models, but that isn't likely to be ideal. It's an ad hoc substitute for doing what machine learning is always doing: learning the necessary parameter values. In other words, let's learn what the best set of weights is to maximize the ensemble's performance on the test set in the hopes that the weighting reflects true learning and will work equally well when the ensemble is used in the wild.

The next question is, of course, how do we learn the weights? Whenever I am presented with a situation where I'm looking for the best set of something, I think "optimization"—in this case, swarm intelligence optimization. As a topic, swarm intelligence and related evolutionary algorithms are far beyond what we can do justice to here. The short description is that *swarm intelligence* uses a collection of agents that move, somewhat intelligently, through the search space. Here the search space is 6D, and each dimension corresponds to one of the weights we want to find. The best set of weights then becomes a best position in this 6D space. The swarm will locate this point (we hope), and when it does, we'll have the weights that give us the best possible ensemble.

The swarm search code is in *swarm_ensemble.py*. We won't walk through it, but the problem-specific code begins on line 1,370. If you want to learn more about swarm techniques, I recommend my book *The Art of Randomness: Randomized Algorithms in the Real World* (No Starch Press, 2024). Running *swarm_ensemble.py* tells us how to use it:

```
swarm_ensemble <npart> <niter> <alg>
```

```
<npart> - number of particles  
<niter> - number of iterations  
<alg>   - PSO, JAYA, DE, RO, GWO, GA
```

Three arguments are required. The code is already configured to load the *prob_run0.npy* files along with the true test-set labels. The first argument is the size of the swarm. Good values are in the vicinity of 20. The second argument is the number of iterations, meaning the number

of times the swarm updates its position in the search space. Empirically, for this experiment, a few hundred iterations is sufficient.

The final argument is the specific swarm algorithm to use. There are hundreds in the literature, but only six are available here: bare-bones particle swarm optimization (PSO), Jaya, differential evolution (DE), random optimization (RO), Grey Wolf Optimizer (GWO), and a genetic algorithm variant (GA). I recommend experimenting with all of them. Random optimization is the simplest algorithm and generally isn't the best choice. My experimenting indicates that Jaya, differential evolution, and bare-bones PSO work well. For example, a run using Jaya produced the following:

```
> python3 swarm_ensemble.py 20 500 Jaya
```

```
0001: accuracy = 80.00000000
0100: accuracy = 81.00000000
0200: accuracy = 81.00000000
0300: accuracy = 81.00000000
0400: accuracy = 81.06250000
0500: accuracy = 81.06250000
```

```
Accuracy: 81.062500000
```

```
Weights:
```

```
0: 0.120140320
1: 0.139839471
2: 0.244902847
3: 0.281249365
4: 0.212761744
5: 0.001106253
```

```
(11 best updates, 10020 function calls, time: 11.262 seconds)
```

The output shows the evolution of the ensemble accuracy as the weight vector is improved by the swarm as it searches. The final position leads to an ensemble accuracy of just over 81 percent, which is about 2 percent better than simple averaging.

The weights per model are listed, indicating the relative importance placed on each model's predictions. Models 2 and 3 are given the greatest weight. These were the best-performing models individually, so this weighting makes sense. Model 5 is given a tiny weight, thereby greatly minimizing its contribution to the ensemble. Model 5 was the worst-performing individual model, so, again the search results are sensible; a bad model was in a sense detected and removed from the ensemble.

This exercise demonstrates that ensemble techniques are helpful, but at a cost: We needed to train multiple models and then use each of them to classify new samples. Because of this, at least for large models, ensembling is not used as often as it might be.

Summary

This chapter presented a case study, a new dataset, and the steps we need to take to work through building a useful model. We started by working with the dataset as given to us, as raw sound samples, which we were able to augment successfully. We noticed that we had a feature vector and attempted to use classical models. From there, we moved on to 1D CNNs. Neither of these approaches was particularly successful.

Fortunately for us, our dataset allowed for a new representation, one that illustrated more effectively what composed the data and, especially important for us, introduced spatial elements so that we could work with 2D convolutional networks. With these networks, we improved quite a bit on the best 1D results, but we were still not at a level that was likely to be useful.

We then moved to ensembles of classifiers. With these, we discovered a modest improvement by using simple approaches to combining the base model outputs (for example, voting).

We can show the progression of models and their overall accuracies to see how our case study evolved. Table 12-3 shows the power of

modern deep learning and the utility of combining it with well-proven classical approaches like ensembles.

Table 12-3: Case Study Model Performance

Model	Data source	Accuracy
Gaussian naive Bayes	1D sound sample	28.1%
Random forest (1,000 trees)	1D sound sample	34.4%
1D CNN	1D sound sample	50.5%
2D CNN	Spectrogram	75.5%
Ensemble (simple average)	Spectrogram	78.9%
Ensemble (weighted average)	Spectrogram	81.1%

The next chapter continues our exploration of convolutional networks by introducing us to advanced CNN architectures, specific model architectures that have become staples, especially for computer vision tasks.

PART V

ADVANCED NETWORKS AND GENERATIVE AI

The deep learning revolution didn't end with convolutional networks but continues to this day. This part explores advanced CNN architectures of the sort that have become standard tools for image-classification tasks. It also introduces fine-tuning and transfer learning, various approaches to detection as opposed to mere classification, self-supervised learning from unlabeled data, and powerful generative AI architectures such as generative adversarial networks and large language models.

OceanofPDF.com

13

ADVANCED CNN ARCHITECTURES



The previous chapters used basic, hand-designed CNN architectures based primarily on intuition. While this was once the only option, a collection of go-to architectures has emerged over time. This chapter implements three that are amenable to our no-GPU requirement. But first, you'll be introduced to the Keras functional API, the no-training-wheels approach to implementing neural networks.

We begin with the VGG family of models, the first successfully trained deep CNNs. VGG models are parameter heavy, especially their deeper incarnations, but we'll get by in this chapter with a smaller version.

Next comes the residual network, or ResNet, probably the most widely used family of models in practice. The smallest of the ResNet family is all we need.

The third model, MobileNet, was designed to be “mobile” (that is, to run on smaller devices like smartphones). MobileNet is significantly lighter weight compared to VGG and ResNet but, as you'll learn, it is nonetheless reasonably effective.

We'll code each model by using the Keras functional API instead of the simpler sequential API of previous chapters. Then, we'll experiment

with CIFAR-10 as our dataset. The primary goal is to introduce the architectures; we'll see some of them again in Chapter 14 when we fine-tune pretrained models. Training the models from scratch on a small dataset like CIFAR-10 won't give us state-of-the-art results, though we'll close the chapter by ensembling the three models to boost performance.

The Keras Functional API

To understand the Keras functional API, you must first understand how Python implements callable objects. Fortunately, doing so is straightforward, as Listing 13-1 demonstrates.

```
class MyClass:
    def __init__(self, thing1, thing2):
        self.thing1 = thing1
        self.thing2 = thing2
    def __call__(self, *args, **kwargs):
        print("I know about:", self.thing1, self.thing2)
        print("arguments    :", args)
        print("keywords     :", kwargs)
```

Listing 13-1: Python's callable interface

Listing 13-1 defines `MyClass` consisting of a constructor (`__init__`) and the `__call__` method. The latter is a special name Python uses when an object is accessed like a function (that is, when given arguments). Consider the following example, which creates an instance of `MyClass` named `obj`:

```
>>> from MyClass import *
>>> obj = MyClass([1,2,3], "cat in the hat")
>>> obj(1,2,3, thing1="cat", thing2="hat")
I know about: [1, 2, 3] cat in the hat
arguments    : (1, 2, 3)
keywords     : {'thing1': 'cat', 'thing2': 'hat'}
```

Notice that the two arguments required by the constructor are supplied. Then, `obj` is used as if it were a function with both standard and keyword arguments passed to it.

The Keras functional API uses this approach. Typical practice merges instance creation with a call to the object. The same approach with `MyClass` looks like this:

```
>>> MyClass([1,2,3], "cat in the hat")(1,2,3, thing1="cat", thing2="hat")
I know about: [1, 2, 3] cat in the hat
arguments    : (1, 2, 3)
keywords     : {'thing1': 'cat', 'thing2': 'hat'}
```

Here `MyClass` creates the object instance, which is immediately called. Take a few moments to convince yourself that the example makes sense. We'll see this type of construct many times in this chapter and those that follow.

To become familiar with the functional API, let's walk through an example. The file *lenet5.py* implements a version of the LeNet-5 architecture from the late 1990s by using the new functional API and then again using the older sequential API. The architecture is similar to the original LeNet model but not an exact copy, and we're using it with CIFAR-10, not MNIST.

Listing 13-2 shows how to build the model by using the sequential API.

```
model = Sequential()
model.add(Conv2D(6, (3, 3), input_shape=input_shape, activation='relu'))
if (useBN): model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(16, (3, 3), activation='relu'))
if (useBN): model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(120, (3, 3), activation='relu'))
if (useBN): model.add(BatchNormalization())
model.add(Flatten())
model.add(Dense(84))
if (useBN): model.add(BatchNormalization())
```

```
model.add(ReLU())
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Listing 13-2: LeNet-5 using the sequential API

We create the `model` as an instance of the `Sequential` class, then use the `add` method to build the network layer by layer. In this case, the network consists of three convolutional layers with 3×3 kernels, ReLU activations, and optional batch normalization layers (I'll describe these in detail in the next section). Two max-pooling layers and a fully connected layer with dropout round out the model. Review the full source in *lenet5.py* for details.

Let's break down Listing 13-3, which presents the same network using the functional API.

```
inp = Input(input_shape)
_ = Conv2D(6, (3, 3))(inp)
if (useBN): _ = BatchNormalization()(_)
_ = ReLU(_)
_ = MaxPooling2D((2, 2))(_)
_ = Conv2D(16, (3, 3))(_)
if (useBN): _ = BatchNormalization()(_)
_ = ReLU(_)
_ = MaxPooling2D((2, 2))(_)
_ = Conv2D(120, (3, 3))(_)
if (useBN): _ = BatchNormalization()(_)
_ = ReLU(_)
_ = Flatten(_)
_ = Dense(84)(_)
if (useBN): _ = BatchNormalization()(_)
_ = ReLU(_)
_ = Dropout(0.5)(_)
_ = Dense(num_classes)(_)
outp = Softmax(_)
model = Model(inputs=inp, outputs=outp)
```

Listing 13-3: LeNet-5 using the functional API

First, note the underscores; many Python developers use the underscore (`_`) as a scratch variable. This might have something to do with the fact that in interactive mode, Python automatically assigns the underscore the value of the last expression. For example:

```
>>> 1 + 2
3
>>> _ + 4
7
```

This automatic assignment to the underscore can come in handy.

Now that you know that we can use the underscore as a temporary variable, let's continue by parsing the first line of the functional API example:

```
inp = Input(input_shape)
```

The first line defines the model's input by using the `Input` class. Here `input_shape` is a tuple, `(32, 32, 3)`, which corresponds to the 32×32 RGB images we want the model to process. What, then, is `inp`? If we break after the assignment to `inp`, we can interrogate to find out:

```
>>> inp
<KerasTensor: shape=(None, 32, 32, 3) dtype=float32 (created by layer 'input_1')>
```

`Input` has created a `KerasTensor`, which is ultimately a wrapper around a TensorFlow tensor. The output of each layer is a `KerasTensor`. Let's continue with the next line, the call to `Conv2D`:

```
_ = Conv2D(6, (3, 3))(inp)
```

Here we see the underscore, though many developers use another variable name, like `x`. I like underscores, so we'll stick with them. The line follows the “define an object that supplies required constructor arguments then immediately use it” format we saw earlier with `MyClass`. The arguments to `conv2d` are the same ones we might pass when using the sequential API. I'm passing the number of filters (6) and the kernel size `((3, 3))` and taking the defaults for all the others (stride of 1, exact convolution, and so on).

The single argument to the resulting object is `inp` with the output assigned to the underscore:

```
>>> _  
<KerasTensor: shape=(None, 30, 30, 6) dtype=float32 (created by layer 'conv2d')>
```

The output of the convolutional layer is a six-channel 30×30 tensor. Remember, for a 3×3 kernel, exact convolution will lose a spatial dimension on each side to make the 32×32 input a 30×30 output. Notice also that the new value of the underscore is the output of `conv2d`. To continue the model, then, we need to give this output as the input to the next layer, `ReLU` (ignoring the batch normalization layer):

```
_ = Conv2D(6, (3, 3))(inp)  
_ = ReLU(_)()
```

Now the underscore is the output of the newly added `ReLU` layer since we used its previous value, the output of the convolutional layer, as the input. Continuing to use the output of one layer as the input to the next builds the model. We finish with the dense layers leading to a softmax layer, where I am again ignoring the optional batch normalization layer:

```
_ = Flatten()(_)  
_ = Dense(84)(_)  
_ = ReLU(_)()  
_ = Dropout(0.5)(_)  
_ = Dense(num_classes)(_)  
outp = Softmax()(_)  
model = Model(inputs=inp, outputs=outp)
```

I have two comments about finishing the model. First, the `Flatten` layer must turn the output of the final convolutional layer into a vector, as required by the dense layers. Second, the output of the softmax layer, `outp`, is used along with `inp`, the model input, to define the model itself courtesy of `Model`. At this point, `model` is equivalent to the `model` defined via the sequential API, meaning compiling and calling `fit` to train works as before. Again, review the code in *lenet5.py*.

Why use the more verbose functional API if the sequential API can build the LeNet model? One word: flexibility. The sequential API is limited in the types of neural networks it can create. The functional API, however, is far more versatile and sufficient to implement all the advanced neural networks we'll experiment with throughout the remainder of the book. Indeed, you can construct virtually all the neural network architectures you'll encounter in the literature via the Keras functional API.

Execute *lenet5.py* without arguments to learn what it expects. Then, try experimenting with the various combinations of sequential versus functional models with and without batch normalization layers. Should you expect any difference in model performance when using one API or another? The *lenet5.py* file uses CIFAR-10, which we'll continue to use for the remainder of the chapter. Let's begin our survey of advanced CNN architectures, starting with VGG.

VGG

The VGG family of models was among the first deep convolutional neural networks to appear in the literature. *VGG* stands for *Visual Geometry Group*, a research group at the University of Oxford in England; see the VGG paper “Very Deep Convolutional Networks for Large-Scale Image Recognition” by Karen Simonyan and Andrew Zisserman (2014).

We denote each model within the VGG family by the number of *learnable* layers, or layers with weights and biases updated during the training process. The standard VGG model is VGG16, with 16 such layers. The model we'll work with is VGG8, a scaled-down version of VGG16 to accommodate the smaller CIFAR-10 dataset in terms of the number of training samples and the size of the input images. VGG16 was designed to work with the ImageNet dataset of 224×224-pixel RGB images.

In 2014, when the VGG paper was published, CNNs were bare-bones. The original VGG models consist of nothing more than convolutional layers and pooling layers. Batch normalization and other

normalization layers with learnable parameters were not yet public. Batch normalization was presented at ICML 2015, a month after the International Conference on Learning Representations (ICLR) 2015. The VGG paper does refer to a local response normalization (LRN) layer in one model version, but LRN layers don't have learnable parameters and must be configured by hand. Keras no longer supports LRN layers. We'll configure our VGG8 model to use batch normalization as an option.

This section proceeds as follows: You'll learn what batch normalization is and why it's used, explore mixing dropout and convolutional layers, and build a VGG8 model and test it against CIFAR-10.

Standardizing with Batch Normalization

In Chapter 1, you learned about standardizing machine learning vectors to make the per-feature mean 0 and the per-feature standard deviation 1. This is all a batch normalization layer does. Batch normalization subtracts the mean from its input and divides the result by the standard deviation. Naturally, this begs the question: What mean, and what standard deviation?

Batch normalization layers maintain a running average of the mean and standard deviation across minibatches. We use these values during training and inference to standardize the layer inputs. Batch normalization layers maintain additional parameters, γ and β , used to scale the standardized inputs.

For example, let's assume that the input to a layer consists of a minibatch worth of activations, $x_0, x_1, x_2, \dots$, where x_i is a tensor representing the output of a previous layer and i the index into the minibatch. Batch normalization then standardizes each x_i , where μ and σ are the running averages of the per-minibatch means and standard deviations:

$$\hat{x}_i = \frac{x_i - \mu}{\sigma}$$

Notice that μ and σ aren't scalars but vectors to match the structure of x_i . If batch normalization is applied to the output of a convolutional layer, the elements of μ and σ are applied across the channel dimension (the number of feature maps). Batch normalization with a dense layer results in μ and σ , where each element of the vector operates on the corresponding element of the dense-layer output. In other words, batch normalization applied to a dense layer is the same as standardizing a feature vector in classical machine learning.

After standardization, $\hat{x}_i$ has, over the minibatch, a mean of 0 and a standard deviation of 1. Here's where γ and β come into the picture. Batch normalization applies additional scaling and shifting by using vectors γ and β , to produce the layer output, y_i , for minibatch input x_i :

$$y_i = \gamma \hat{x}_i + \beta$$

Both γ and β are learned during training via backpropagation and gradient descent. Note that γ and β are vectors like μ and σ and are applied in the same manner with convolutional and dense-layer outputs.

We typically use batch normalization layers before the activation function. Convolutional, dense, and batch normalization layers include bias terms. Because of this, people often remove the bias term from the preceding convolutional or dense layer. Batch normalization makes including the bias term redundant because it scales the output of those layers so they have a 0 mean, thereby eliminating the bias offset. Batch normalization's β value becomes the learned bias term.

Batch normalization offers several benefits that explain its popularity. First, standardizing each layer's output keeps the activations from becoming too large (exploding) or too small (vanishing). Second, standardization shifts the inputs to each layer so that they're consistent throughout training. During training, the distribution of the inputs to a layer, as characterized by the mean and standard deviation, can shift as each layer's weights are adjusted. This is the covariate shift problem and was the primary motivation for the introduction of batch normalization. As the experiments later in the chapter will illustrate, batch normalization is generally a good idea for most models.

Applying Dropout After Convolutional Layers

The original VGG models used dropout on the output of the dense layers, as is standard practice; however, you can place dropout layers after convolutional layers as well.

We have options when applying dropout to the output of a convolutional layer. One option is standard dropout, which randomly zeroes individual feature map values. A second option uses *spatial dropout*, which zeroes at the feature-map level.

For example, imagine a convolutional layer that produces a $32 \times 32 \times 16$ output, meaning a stack of 16 feature maps, each 32×32 pixels (the spatial component).

With a dropout rate of 0.5 (50 percent), standard dropout will randomly zero, on average, half the values in the $32 \times 32 \times 16$ stack. On the other hand, a spatial dropout rate of 50 percent means, on average, that 8 of the 16 feature maps will be set to 0. Setting a feature map to 0 means setting all pixels to 0.

The argument in favor of spatial dropout refers to the pixels of a feature map being highly correlated (think of the pixels in an image), so dropping an entire feature map will (hopefully) introduce stronger regularization than standard dropout that zeroes only some of the pixel values in each feature map. As deep learning is an empirical enterprise, exploring both dropout approaches is worth our time. Therefore, our VGG8 implementation will support both standard and spatial dropout after the convolutional layers as an extension of the classic VGG structure.

Building the VGG8 Model

The time has come to build the VGG8 model. As the name indicates, the model will have eight layers with learnable weights and biases. If we elect to use batch normalization, then, technically, the number should go up because batch normalization layers have learnable parameters, but we'll ignore this and stick with VGG8 as the model name.

The layers of a deep network are like LEGO bricks. Individual layers are bricks that can be stacked together to form blocks that are

repeated to construct the entire network. VGG uses two such blocks: a convolutional block and a dense block. Advanced networks are often built from repeated blocks; each block is a specific collection of low-level layers. This will become clearer as we walk through the implementation.

We'll start with the code that builds the model and then work down from there to the content of the blocks. Listing 13-4 shows how to build the VGG8 model.

vgg8.py

```
inp = Input(input_shape)
_ = ConvBlock(inp, 64, dropout=dropout, pct=pct, useBN=useBN)
_ = ConvBlock(_, 128, dropout=dropout, pct=pct, useBN=useBN)
_ = ConvBlock(_, 256, dropout=dropout, pct=pct, useBN=useBN)
_ = Flatten()(_)
_ = DenseBlock(_, 2048, useBN=useBN)
_ = DenseBlock(_, 2048, useBN=useBN)
_ = Dense(num_classes)(_)
outp = Softmax()(_)
```

Listing 13-4: Building VGG8 in code

The `input_shape` tuple is (32, 32, 3) because CIFAR-10 consists of 32×32-pixel color images. The final dimension is the number of channels, here three for the image's red, green, and blue channels. The `Input` layer begins building the model. Its output (`inp`) is given a (slightly) more meaningful name because `inp` is needed after the model is constructed.

Next come three calls to `ConvBlock`, a function I'll detail momentarily, followed by a call to `Flatten` and two calls to `DenseBlock`, another function yet to be described. The model is completed by a call to `Dense` combined with a call to `softmax` to add the layers producing the 10-element output of the model. Notice that `softmax`'s output is kept in `outp`.

Listing 13-4 contains four instances of the functional API in action: `Input`, `Flatten`, `Dense`, and `Softmax`. The `ConvBlock` and `DenseBlock` functions act as blocks, groupings of Keras layers that work together. This mixing of

individual layers and functions encapsulating multiple layers will be a common theme throughout this chapter and the remainder of the book.

Listing 13-4 shows the overall structure of a VGG model: a series of convolutional blocks followed by a series of dense blocks. This structure is essentially identical to what we've used previously, as the VGG models are the first evolution of the simpler CNNs that came before.

The calls to `ConvBlock` have an argument that is at first 64, then 128, and finally 256; these refer to the number of filters learned by the block (the number of output channels). Similarly, `DenseBlock`'s 2048 argument refers to the number of nodes in the layer. Let's understand the blocks beginning with `DenseBlock`, as in Listing 13-5.

```
def DenseBlock(_, nodes, useBN=False):
    _ = Dense(nodes)(_)
    if (useBN):
        _ = BatchNormalization()(_)
    _ = ReLU()(_)
    _ = Dropout(0.5)(_)
    return _
```

Listing 13-5: The dense-layer block

This block is straightforward and what we might expect the top part of a CNN to be like. The `Dense` layer of `node` nodes is followed by an optional `BatchNormalization` layer, a nonlinear layer (`ReLU`), and a `Dropout` layer with a fixed percentage of 50 percent. The return value is the output of the `Dropout` layer, to make `DenseBlock` fit within the framework of Listing 13-4.

Let's move on to the more interesting `ConvBlock` (Listing 13-6).

```
def ConvBlock(_, filters, dropout=0, pct=0.25, useBN=False):
    if (useBN):
        _ = Conv2D(filters, (3, 3), padding='same')(_)
        _ = BatchNormalization()(_)
        _ = ReLU()(_)
        _ = Conv2D(filters, (3, 3), padding='same')(_)
        _ = BatchNormalization()(_)
```

```

    _ = ReLU()(_)
else:
    _ = Conv2D(filters, (3, 3), padding='same')(_)
    _ = ReLU()(_)
    if (dropout == 1):
        _ = Dropout(pct)(_)
    elif (dropout == 2):
        _ = SpatialDropout2D(pct)(_)
    _ = Conv2D(filters, (3, 3), padding='same')(_)
    _ = ReLU()(_)
    if (dropout == 1):
        _ = Dropout(pct)(_)
    elif (dropout == 2):
        _ = SpatialDropout2D(pct)(_)
return MaxPooling2D((2, 2))(_)

```

Listing 13-6: The convolutional block

Which low-level layers are added to the model depends on the arguments to `ConvBlock`. At a minimum, if `dropout` is 0 (not 1 or 2) and `useBN` is `False`, the function adds two sets of 3×3 convolutional layers followed by ReLU and ends with a maximum pooling layer, which is what the original VGG paper implements:

```

_ = Conv2D(filters, (3, 3), padding='same')(_)
_ = ReLU()(_)
_ = Conv2D(filters, (3, 3), padding='same')(_)
_ = ReLU()(_)
_ = MaxPooling2D((2, 2))(_)

```

Mixing dropout and batch normalization is often frowned upon based on empirical evidence, so `ConvBlock` ignores any possible dropout if `useBN` is `True`. In that case, the two convolutional-ReLU pairs are augmented with `BatchNormalization` layers. Notice that the normalization layers come before the activation (ReLU). This standard approach makes intuitive sense: We want to get the layer output into a consistent state (mean and standard deviation) before applying the nonlinearity.

If batch normalization is not desired, and `dropout` is 1 or 2, dropout is used. If dropout is 1, standard `Dropout` is applied after the nonlinearity; otherwise, `SpatialDropout2D` is used. We typically use dropout, regardless of the kind, after the nonlinearity. In all cases, `ConvBlock` exits with a maximum pooling layer using a step size of 2 in each direction.

The remainder of the code in *vgg8.py* loads the CIFAR-10 dataset, scales it, then builds and trains the VGG8 model before storing the results in an output directory. I won't list the code here; you can review it in the file. The models we'll implement later in the chapter, ResNet-18 and MobileNet, are nearly identical to *vgg8.py*, differing primarily in the way the model is constructed.

Look again at Listing 13-4. It mixes Keras functional API calls and ordinary Python function calls. This works fine, though it's somewhat unsatisfying from a software development perspective because it treats Keras layers and blocks constructed from multiple layers differently.

It would be nice, and aesthetically pleasing, to use the functional API approach with the blocks as well. Fortunately, Python is up to the challenge. Indeed, it isn't difficult to separate the configuration of a block from its implementation. Doing so mirrors Listing 13-1, though it does require us to have some familiarity with object-oriented programming.

The file *vgg8_functional.py* is virtually identical to *vgg8.py*, differing only in the definition of `ConvBlock` and `DenseBlock`. Both are now classes, not functions. The class constructor (`__init__`) configures the block using what were the arguments to the function, and the `__call__` method accepts the previous layer output and uses it to add the proper block layers.

For example, Listing 13-7 shows the new definition of `DenseBlock`.

```
class DenseBlock:
    def __init__(self, nodes, useBN=False):
        self.nodes = nodes
        self.useBN = useBN
    def __call__(self, _):
        _ = Dense(self.nodes)(_)
```

```
if (self.useBN):
    _ = BatchNormalization()(_)
    _ = ReLU()(_)
    _ = Dropout(0.5)(_)
return _
```

Listing 13-7: Implementing the dense block as a class

The `DenseBlock` class configures an instance when created and then, when called, builds the necessary low-level Keras layers. A similar code-factoring exercise, which I leave to you, makes a class out of `ConvBlock`. With both `DenseBlock` and `ConvBlock` now mimicking the Keras functional API, building the VGG8 model becomes less visually jarring, as Listing 13-8 demonstrates.

```
inp = Input(input_shape)
_ = ConvBlock( 64, dropout=dropout, pct=pct, useBN=useBN)(inp)
_ = ConvBlock(128, dropout=dropout, pct=pct, useBN=useBN)(_)
_ = ConvBlock(256, dropout=dropout, pct=pct, useBN=useBN)(_)
_ = Flatten()(_)
_ = DenseBlock(2048, useBN=useBN)(_)
_ = DenseBlock(2048, useBN=useBN)(_)
_ = Dense(num_classes)(_)
outp = Softmax()(_)
```

Listing 13-8: Building VGG8 by using a purely functional API approach

The overall structure of VGG8 remains the same, but the blocks act like other Keras layers in that they accept, after instantiation, the output of the previous layer definition and return the updated model. Each model from here on will be adaptable to this approach if desired.

Ordinary Python functions or classes aside, we have what we need to construct the model. Let's build it and put it to the test.

Testing the VGG8 Model

Running `vgg8.py` from the command line without arguments tells us how to use the code:

```
vgg8 <minibatch> <epochs> <dropout_type> <dropout_pct> <useBN> <outdir>
```

```
<minibatch>    -  minibatch size (e.g. 128)

<epochs>       -  number of training epochs (e.g. 16)
<dropout_type> -  0=none, 1=dropout/dropout, 2=spatial/dropout
<dropout_pct>  -  conv layer dropout fraction (e.g. 0.5)
<useBN>        -  0=no, 1=yes (ignores dropout settings yes)
<outdir>       -  output file directory (overwritten)
```

You may get warning messages from TensorFlow about CPU instructions when running *vgg8.py*; these are safe to ignore.

We're familiar with the minibatch size and epochs arguments. The next three configure the architecture of the VGG8 model. First, we tell the code whether we want dropout on the convolutional layers and, if so, which kind. Standard dropout is always applied to the dense layers.

The `dropout_pct` argument sets the dropout fraction. It must be given, even if `dropout_type` is 0. As this is the first time we've set the dropout fraction directly, let's explore dropout further in terms of what the percentage means and how it's used during training.

Regardless of the type of dropout, the fraction specifies the probability that an output value will be set to 0, either an individual output value (that is, of the layer's output tensor) or the entire activation map if using spatial dropout with a convolutional layer. On average, for each sample passed through the model during a minibatch, the specified dropout fraction, p , of the activations will be set to 0. At inference time, when the model is used, dropout is not applied and all neurons are active.

Setting a p fraction of the output activations to 0 reduces the overall output of the layer. During training, we want this overall output to be as if all neurons were active because that's the case during inference. Therefore, when dropout is applied, the remaining outputs are scaled by $1/(1 - p)$.

For example, if the dropout fraction for a layer is $p = 0.3$, then during training, 30 percent of the layer outputs are set to 0, and the

remaining outputs are multiplied by s :

$$s = \frac{1}{1-p} = \frac{1}{1-0.3} = \frac{1}{0.7} \approx 1.4286$$

Notice that $s \rightarrow 1$ as $p \rightarrow 0$. This makes sense: As the probability of dropping an output value decreases, we should expect the scale factor necessary to maintain roughly the same overall layer output as we'll get at inference time to approach 1.

To use *vgg8.py*, we specify a kind of dropout and the dropout fraction. Next comes a Boolean telling *vgg8.py* whether to use batch normalization. We're warned that using batch normalization makes any specified dropout moot. We still need to supply values because *vgg8.py* uses simple command line parsing, but those values no longer have meaning if we select batch normalization. The final command line argument tells the code where to dump its results.

Let's give *vgg8.py* a go with no batch normalization and no dropout on the convolutional layers. This is the original VGG paper configuration:

```
>>> python3 vgg8.py 128 10 0 0 0 tmp
```

The command line says to use a minibatch size of 128, train for 10 epochs, use no dropout and no batch normalization, and dump all output in the *tmp* directory. The resulting model has 13,752,906 learnable parameters and no non-learnable parameters. My run of the command took about 48 minutes and produced this:

```
Test set accuracy: 0.7877, MCC: 0.7644
[[815 10 40 19 22 3 6 13 51 21]
 [ 8 878 2 6 3 5 11 2 31 54]
 [ 51 1 682 53 77 36 39 43 15 3]
 [ 17 5 48 631 60 95 55 63 16 10]
 [ 8 1 48 47 774 19 38 60 4 1]
 [ 4 0 37 179 42 641 16 69 6 6]
 [ 7 4 51 58 39 9 813 8 7 4]
 [ 13 3 9 36 29 18 1 881 5 5]
```

```
[ 28  8 11 11  4  4  3  4 904 23]
[ 12 66  9 13  7  1  4 10 20 858]]
```

The model's final test-set overall accuracy was 78.8 percent, but the resulting confusion matrix (recall, CIFAR-10 has 10 classes) is still rather noisy, meaning we have many off-diagonal entries. In Chapter 11, we trained two small CNNs on the full CIFAR-10 dataset, achieving a best overall accuracy of 74.8 percent; see Table 11-1 on page 300. With that result in mind, the plain VGG8 model is already doing better with a 4 percentage point improvement in overall accuracy—hedging our bets a bit because both results are based on a single training session.

The code generates more output than what is dumped to the screen. The *tmp* directory contains the following:

```
accuracy_mcc.txt
confusion_matrix.npy
model.keras
predictions.npy
results.pkl
```

The first is a text file that contains the test-set accuracy line printed to the screen. The second is the confusion matrix as a NumPy array, here 10×10 for the 10 classes. Next comes the trained model in Keras format. The *predictions.npy* file contains the test-set softmax predictions, meaning a NumPy array of 10,000 rows and 10 columns. Applying `np.argmax` across the columns of this array returns the predicted label.

The *results.pkl* file is a Python pickle file containing four items: training loss, validation loss, training error, and validation error, all per epoch. For this experiment, *validation* refers to the entire CIFAR-10 test set. We aren't using the validation results to decide when to stop training, so we can use the whole test set this way. Suppose we were to decide on model training or architecture based on the validation results. In that case, we can't justify using the whole test set for validation and would instead be obligated to use only a portion so that a held-out test set is available for final model analysis.

Let's train the VGG8 model again, but this time with batch normalization. The command line becomes the following:

```
>>> python3 vgg8.py 128 10 0 0 1 tmp
```

After 59 minutes on my test machine, the code produced this output:

```
Test set accuracy: 0.8212, MCC: 0.8026
[[816  4  60  26  9  2  2  3  74  4]
 [ 8 891  7  8  0  2  3  0  46 35]
 [ 23  0 875  32 33 18  5  5  9  0]
 [ 6  2 105 722 30 102  4 11 15  3]
 [ 5  0  84  43 802 23  9 24 10  0]
 [ 5  0  62 116 17 778  4 10  6  2]
 [ 2  3 104 110 47  34 690  1  7  2]
 [ 7  2  46  34 26  31  1 847  2  4]
 [22  6  12  8  0  1  2  2 942  5]
 [20 46  7 16  2  3  0  5  52 849]]
```

Based on this example, batch normalization appears to live up to its hype; the overall accuracy improved from 78.8 percent to 82.1 percent.

Of course, we should train multiple versions of each model and average the results to convince ourselves that batch normalization is helpful. The *vgg8_runs* script does just this, not only for batch normalization but for standard and spatial dropout as well. Each model type is trained five times. Execute the script with the following, then go on a weekend getaway:

```
> sh vgg8_runs
```

When you return, the *runs* directory will contain all the results. Run *vgg8_analysis.py* without arguments to generate summary plots, confusion matrices (averaged over all runs), and statistical tests. For example, my run of *vgg8_runs* and *vgg8_analysis.py* produced Table 13-1.

Table 13-1: VGG8 Results ($n = 5$)

	Accuracy	MCC
Batch normalization	0.8098 ± 0.0035	0.7905 ± 0.0036
No dropout	0.7823 ± 0.0048	0.7585 ± 0.0054

	Accuracy	MCC
Standard dropout	0.7679 ± 0.0093	0.7435 ± 0.0099
Spatial dropout	0.7210 ± 0.0035	0.6910 ± 0.0038

These results declare batch normalization the clear winner over both forms of dropout and no dropout. A t-test between batch normalization and no dropout validates the claim with a p -value of less than 0.002. The nonparametric Mann–Whitney U test confirms the claim with a p -value of 0.008. Recall that nonparametric tests don’t assume normally distributed data and are generally harder to satisfy. Therefore, we have a fairly robust level of confidence in the claim that batch normalization is the way to go with VGG8.

Applying either form of dropout to the convolutional layers results in worse performance, which might appear counterintuitive at first. After all, dropout is supposed to help, not hurt. However, a few factors are at play that might explain dropout’s poor showing. First, each model was trained for only 10 epochs, but adding dropout tends to slow model convergence, meaning more gradient-descent steps are necessary. Second, CIFAR-10 is a smaller dataset (50,000 training samples), and the VGG8 model has over 13 million parameters to learn. Compare that to the 180,000 in the LeNet-style model used earlier in the chapter. If a model is overfitting, or fitting well without dropout, then adding dropout might do more harm than good. The results in Table 13-1 likely reflect a combination of both issues.

Ultimately, VGG models are bigger, deeper versions of the CNNs we’ve used up to this point in the book. Let’s move on to our second advanced model, which throws something new into the mix.

ResNet

The 2016 paper “Deep Residual Learning for Image Recognition” by Kaiming He et al. introduced the world to residual networks (ResNets). *ResNets* are deep CNNs arranged in blocks, like VGG models. However,

each residual block includes a *skip connection* that bypasses the block so that the input to the block is added to the block's output to become the input of the next block. Figure 13-1 illustrates a residual block.

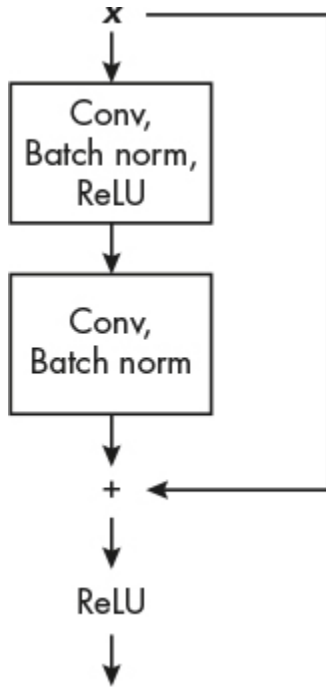


Figure 13-1: A residual block with a skip connection

The input to the block, x , is passed through the block, schematically represented as two subblocks of a convolutional layer, a batch normalization layer, and a ReLU. Notice, however, that the output of the second subblock is added to the original block input before the ReLU layer.

Think of the process like so: The goal of a layer or block is to learn a function, a transformation of the input to produce an output, $y = f(x)$. The input, x , is a tensor of a shape, and so is the output, y . This explains why I'm using $f()$, as f is a function that accepts a tensor input and returns a tensor output.

Intuitively, we know this is what a layer does. The goal of training is to find $f()$ so that:

$$x \xrightarrow{f} y$$

A residual block, because of its skip connection, learns a different function:

$$\mathbf{y} = \mathbf{x} + \mathbf{g}(\mathbf{x}) \quad (13.1)$$

This is still $\mathbf{y} = \mathbf{f}(\mathbf{x})$, but we're enforcing an overall form on $\mathbf{f}(\mathbf{x})$.

Adding the block output to the block input means that we're asking the block to learn a *residual*, a function that, when its output is added to the block input, gives us the transformation we want. There are several reasons to do this, the most important being that forcing blocks to learn residuals because of the skip connection addresses the vanishing gradient problem.

Mitigating the Vanishing Gradient Problem

To understand the vanishing gradient problem, imagine a small, three-layer neural network where each layer has a single node. The input is x (a scalar), and the output is y with activation function σ (a sigmoid).

In that case, we can write the output y for input x as the following for weights w_i and biases b_i :

$$y = \sigma(w_3\sigma(w_2\sigma(w_1x + b_1) + b_2) + b_3) \quad (13.2)$$

Take a moment to convince yourself that Equation 13.2 is valid.

To train the network via gradient descent requires knowing how the output, y , changes with changes in the parameters of the model. Let's consider w_1 's effect on the output. We want to know $\partial y / \partial w_1$. The calculation of $\partial y / \partial w_1$ is easier to follow if we split Equation 13.2 into pieces like so:

$$\begin{aligned} y &= \sigma(z_3) \\ z_3 &= w_3\sigma(z_2) + b_3 \\ z_2 &= w_2\sigma(z_1) + b_2 \\ z_1 &= w_1x + b_1 \end{aligned}$$

Working from top to bottom, we get this:

$$\begin{aligned}\frac{\partial y}{\partial w_1} &= \sigma'(z_3) \frac{\partial z_3}{\partial w_1} \\ \frac{\partial z_3}{\partial w_1} &= w_3 \sigma'(z_2) \frac{\partial z_2}{\partial w_1} \\ \frac{\partial z_2}{\partial w_1} &= w_2 \sigma'(z_1) \frac{\partial z_1}{\partial w_1} \\ \frac{\partial z_1}{\partial w_1} &= x\end{aligned}$$

When put together, this gives the following:

$$\frac{\partial y}{\partial w_1} = \sigma'(z_3) w_3 \sigma'(z_2) w_2 \sigma'(z_1) x \quad (13.3)$$

Here σ' is the derivative of the sigmoid activation function.

Because the derivative of e^x is e^x , a bit of manipulation proves this:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Finally, notice that the maximum value of $\sigma'(z)$ occurs when $z = 0$ with $\sigma(0) = 0.5$, implying that the largest value possible for the derivative of the sigmoid is as follows:

$$\sigma'(0) = \sigma(0)(1 - \sigma(0)) = 0.5(1 - 0.5) = 0.5(0.5) = 0.25$$

In other words, the derivative of the sigmoid is always less than 1. Equation 13.3 contains the product of three derivatives of the sigmoid, all less than 1. The gradient for lower layers of a model becomes smaller and smaller as model depth increases because the product of values less than 1 approaches 0. This is the vanishing gradient problem, and it slows learning in deep networks, particularly for the lower layers.

Note that I used the sigmoid activation function, which is an issue. Older networks used sigmoid or hyperbolic tangent activation functions, both of which suffer from the vanishing gradient problem in practice, even though the maximum of the derivative of the hyperbolic tangent is 1. The S shape of the graph of either function means that inputs far from the origin are in a region where the derivative is small, also contributing to the vanishing gradient problem.

We can try to compensate for this issue in various ways. One is to use ReLU activation functions because the derivative of the ReLU is either 0 or 1, always. However, ReLUs have issues at times too, because of the *dying ReLU problem*: Neurons always output 0 because their inputs are negative. Skip connections allow gradients to flow unhindered through the network during backpropagation. To understand this, consider what the derivative of Equation 13.1 entails. The skip connection path, represented by $\mathbf{x}$, has a derivative of $\mathbf{1}$ (a tensor of all 1s), while the derivative of the block can be represented by $\mathbf{g}'(\mathbf{x})$ (with other terms). Regardless of what happens to the gradient in the block, the skip connection passes the incoming gradient to the next lower layer unchanged, thereby mitigating the vanishing gradient problem and allowing CNNs with over 100 layers to be successfully trained.

Another benefit of the skip connection is that layers can learn identity functions without too much effort. If the proper transformation for a layer in a model is to do nothing to the input, an identity function, then the form in Equation 13.1 means that the residual function is $\mathbf{g}(\mathbf{x}) = \mathbf{0}$ (a tensor of all 0s).

Forcing blocks to learn residuals means they learn adjustments to the inputs to arrive at proper outputs instead of learning an arbitrary function from scratch. The practical success of ResNets demonstrates that this is a valuable approach.

Exploring ResNet Configurations

ResNets come in multiple configurations. The version we'll experiment with in this chapter is ResNet-18 because it contains 18 learnable layers (convolutional or dense). Note that the batch normalization layers have learnable parameters, but the naming convention ignores this fact.

ResNet-18 is the smallest ResNet model discussed in the original He paper. We're using it because CIFAR-10 is a small dataset. The paper also explores ResNet-34, ResNet-50, ResNet-101, and ResNet-152. ResNet-50 is usually sufficient for many computer vision tasks (but exceptions abound).

Figure 13-2 presents ResNet-18. We'll implement this in the next section. Some decoding is in order.

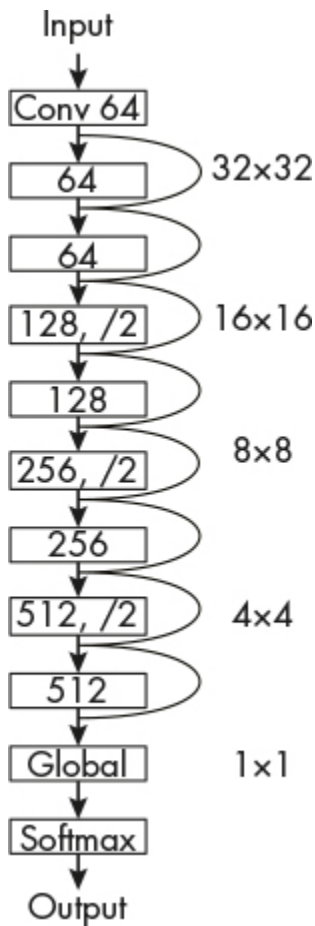


Figure 13-2: The ResNet-18 architecture

The first layer is denoted Conv 64. It's a standard 2D convolutional layer using a 3×3 kernel to produce 64 activation maps of the $32 \times 32 \times 3$ input image. Every convolutional layer in a ResNet uses a 3×3 kernel.

Residual blocks are marked with a number indicating the number of filters learned by the block and include a line around the block to mark the skip connection. Therefore, the first two residual blocks each learn 64 filters for each of the two convolutional layers in the block (review Figure 13-1). The 32×32 on the residual block's right side indicates the activation map's spatial dimensions. In other words, the first two residual blocks output $32 \times 32 \times 64$ tensors.

Turning our attention to the remaining residual blocks, we see that two are learning 128, then 256, and finally 512 filters. Some of these blocks are marked with /2, corresponding to a reduction in the spatial dimension of the output by 2. ResNets don't use pooling layers but reduce the spatial dimension by changing the stride used by the convolution kernels. By default, the stride is 1 in each direction, but the /2 blocks change the stride to 2 in each direction, thereby ignoring half the input values when applying the kernels learned by the layer.

The final two layers are labeled Global and Softmax. We know about the softmax layer, which we'll represent in code as a `Dense` layer with 10 nodes for CIFAR-10's classes followed by a `softmax` layer. The Global layer refers to a spatial average pooling layer, one that is applied across the spatial dimensions.

For example, suppose the input to the global pooling layer is $4 \times 4 \times 512$, as Figure 13-2 indicates. In that case, the spatial pooling layer will calculate the average value of each 4×4 channel and use that in the output, which becomes a $1 \times 1 \times 512$ tensor. Each input channel is reduced to its average value.

Now that you have a conceptual understanding of the ResNet architecture, let's build it in code. Along the way, we'll discover a hidden issue that isn't immediately evident from Figure 13-2.

Building the ResNet-18 Model

I suggest reviewing the contents of *resnet18.py* before reading on to build the model. The primary difference between *vgg8.py* and *resnet18.py* is the model's architecture.

We can translate Figure 13-2 level by level, as in Listing 13-9.

```
inp = Input(input_shape)
_ = Conv2D(64, (3, 3), strides=(1, 1), padding='same')(inp)
_ = ResidualBlock(_, 64, downsample=False, useBN=useBN)
_ = ResidualBlock(_, 64, downsample=False, useBN=useBN)
_ = ResidualBlock(_, 128, downsample=True, useBN=useBN)
_ = ResidualBlock(_, 128, downsample=False, useBN=useBN)
_ = ResidualBlock(_, 256, downsample=True, useBN=useBN)
```

```
_ = ResidualBlock(_, 256, downsample=False, useBN=useBN)
_ = ResidualBlock(_, 512, downsample=True, useBN=useBN)
_ = ResidualBlock(_, 512, downsample=False, useBN=useBN)
_ = GlobalAveragePooling2D()(_)
_ = Dense(num_classes)(_)
outp = Softmax()(_)
```

Listing 13-9: The residual block in code

This is similar to the way we implemented VGG8. We use low-level Keras layers (Input, Conv2D, GlobalAveragePooling2D, Dense, and Softmax) along with multiple calls to ResidualBlock:

```
def ResidualBlock(x, filters, downsample=False, useBN=False):
    if (downsample):
        strides = (2, 2)
        inp = Conv2D(filters, (1, 1), strides=strides, padding='same')(x)
    else:
        strides = (1, 1)
        inp = x

    _ = Conv2D(filters, (3, 3), strides=strides, padding='same')(x)
    if (useBN):
        _ = BatchNormalization()(_)
    _ = ReLU()(_)

    _ = Conv2D(filters, (3, 3), strides=(1, 1), padding='same')(_)
    if (useBN):
        _ = BatchNormalization()(_)

    _ = Add()([_ , inp])
    return ReLU()(_)
```

Listing 13-10: Building a residual block

While the original ResNet paper used batch normalization, we made it optional to provide an additional experimental degree of freedom. The `downsample` argument selects between strides of (1, 1) or (2, 2). The spatial dimensions are halved if a stride of (2, 2) is used.

The `ResidualBlock` function consists of four code paragraphs. Ignore the first for a moment and focus on the second. It implements the first convolutional layer by using `filters` filters and (1, 1) or (2, 2) strides as selected by `downsample`. If selected, batch normalization is used before the ReLU.

The third code paragraph is the second convolutional layer of the block. Notice that it always uses a stride of (1, 1) because the output of the first convolutional layer has already reduced the spatial dimension by a factor of two if `downsample` is `True`. Also note that batch normalization is applied to the output of `conv2d`, if selected, and that output is added, element by element, to `residual` before the `ReLU` layer ends the function by returning its output.

Now consider the first code paragraph, which implements downsampling. If `downsample` is `False`, `inp` is made to refer to the input, `x`. Otherwise, we must spatially downsample the input so that it matches the dimensions of the main branch, the part that used (2, 2) strides. Therefore, a `conv2d` layer is used with 1×1 kernels to learn the proper number of output `filters` and reduce the spatial dimension of the input tensor, `x`.

Listing 13-9 sets `downsample` to `True` every time the number of channels (filters) is increased. In that case, the `conv2d` layer of the first code paragraph in Listing 13-10 does double duty. It downsamples spatially using (2, 2) strides and adjusts to the proper number of output filters.

Consider the following scenario. The input tensor to the residual block has M channels (say, 64), and it's now time to learn N (for example, 128) channels. We're also downsampling, so the stride is (2, 2), but that's orthogonal to moving from M input channels to N output channels. We can think of the 1×1 kernels as 3D $1 \times 1 \times M$ tensors, and we need to learn N of them. The convolution operation says to multiply the overlap between the kernel values ($1 \times 1 \times M$) and the input tensor, element by element, then sum. For a $1 \times 1 \times M$ kernel, this multiplies each spatial position, whether the stride is (1, 1) or (2, 2), by the corresponding kernel value before summing:

$$y_{ij} = k_1 x_{ij1} + k_2 x_{ij2} + \cdots + k_M x_{ijM} \quad (13.4)$$

Here y_{ij} is the output value for spatial location (i, j) using one of the N learned $1 \times 1 \times M$ kernels, and x_{ijm} is the input tensor value as location (i, j, m) , where the third dimension is the channel number. Notice that I'm indexing from 1 instead of 0 to avoid clumsy notation involving $M - 1$.

Because the kernel's spatial dimensions are 1×1 , no structural information in the input tensor, x , is used. The convolution output is a linear transformation of the M input channels. The `conv2d` layer learns N kernels with dimensions $1 \times 1 \times M$ to produce an output tensor with N channels.

Testing the ResNet-18 Model

Let's test the ResNet-18 architecture. You'll find the implementation in *resnet18.py*. Running without arguments shows us what's expected:

```
> python3 resnet18.py
resnet18 <minibatch> <epochs> <useBN> <outdir>

<minibatch> - minibatch size (e.g. 128)
<epochs>    - number of training epochs (e.g. 16)
<useBN>     - 0=no, 1=yes
<outdir>    - output file directory (overwritten)
```

The required arguments are familiar. Notice that *resnet18.py* makes batch normalization optional. The shell script *resnet_runs* trains and tests five models each, with and without batch normalization. If you run the script, it's time for another weekend getaway. Run *resnet18_analysis.py* when the script finishes to get the results. Table 13-2 summarizes my run's results.

Table 13-2: ResNet-18 Results ($n = 5$)

Batch normalization	Accuracy	MCC
Yes	0.7879 ± 0.0103	0.7664 ± 0.0107
No	0.7354 ± 0.0050	0.7075 ± 0.0056

That batch normalization matters is evident from the table and supported by the t-test and Mann–Whitney U test results reported by *resnet18_analysis.py*. Notice, also, that ResNet-18 (78.8 percent) doesn't match the performance of VGG8 (81.0 percent). However, the difference is not statistically significant, but potentially suggestive with a t-test p-value of 0.08.

Residual blocks use (2, 2) strides when downsampling instead of pooling layers. This begs the question: What happens if we use pooling layers and not striding to downsample? The file *resnet18_pooling.py* answers the question by replacing downsampling via striding with either max or average pooling layers.

I ran one training session with max pooling for a test-set accuracy and MCC of 0.8238 and 0.8053, respectively. Switching to average pooling produced a model with accuracy 0.8296 and MCC 0.8112. Both results exceed the best performance of my five runs of standard ResNet-18 (all with batch normalization on), which produced accuracy 0.8172 and MCC 0.7977. Recall that max pooling selects the largest value in the 2×2 sliding window (stride 2), while average pooling uses the mean of those four values.

Should we read anything into these results? How likely is it, based on one run each, that the difference seen is meaningful? We know that one run isn't sufficient; therefore, I trained five models using average pooling and compared them to the five standard ResNet-18 models using batch normalization. I analyzed by hand like so:

```
>>> import numpy as np
>>> from scipy.stats import ttest_ind, mannwhitneyu
>>> base = np.array([0.7541, 0.8172, 0.7897, 0.7964, 0.7823])
>>> avg = np.array([0.8296, 0.7956, 0.8293, 0.8277, 0.8418])
>>> base.mean(), base.std(ddof=1) / np.sqrt(5)
(0.7879400000000001, 0.010265894992644338)
>>> avg.mean(), avg.std(ddof=1) / np.sqrt(5)
(0.8248, 0.007724441727400115)
>>> ttest_ind(avg, base)
Ttest_indResult(statistic=2.869063755122692, pvalue=0.020858903352199008)
```

```
>>> mannwhitneyu(avg, base)
```

```
MannwhitneyuResult(statistic=23.0, pvalue=0.031746031746031744)
```

The base ResNet-18 mean accuracy is 78.8 percent to the average pooling accuracy of 82.5 percent. Further, the standard error of the mean is smaller, with the average pooling models indicating more-uniform performance from training session to training session.

I read the results as suggestive that there might be a performance advantage to replacing (2, 2) strides with average pooling when downsampling, for this dataset, because both the t-test and Mann-Whitney U test produced p-values below the commonly accepted threshold of 0.05. Naturally, confidence will improve (or vanish) if the exercise is repeated, say, five more times. Remember, 0.05 isn't a magic number; it was merely a suggestion made nearly a century ago.

The price paid by the change to average pooling layers is time: The average pooling layer model takes about twice as long to train. I selected average pooling because it had the highest accuracy on my first run. Do the results hold up for max pooling? I leave that as an exercise for you.

Assuming average pooling is slightly better (test other datasets!), why might that be the case? Downsampling by striding ignores every other input to the residual block, both in what passes through the block and in what passes through the skip connection. In effect, striding asks a single convolutional layer to learn values based on half the data.

The average pooling approach, on the other hand, uses all the data during the convolution and downsampling process, including values ignored by a (2, 2) stride. For this one experiment, using all the available information seems beneficial.

We have one more CNN model to implement. Let's do that now.

MobileNet

MobileNet is a lightweight architecture intended for small devices like mobile phones. It introduces us to depthwise convolutions, a new kind of convolutional layer that dramatically reduces the number of model parameters.

We begin this section by describing depthwise convolutional layers and how they differ from standard ones. Then, we build the MobileNet model in code. Finally, we test with CIFAR-10 and compare the results to VGG8 and ResNet-18.

Implementing Depthwise Convolutions

To better understand depthwise convolutions, it helps to look back to standard convolution. Chapter 9 discussed the operation of a convolutional layer as a series of 2D kernels passed over each channel before summing the result across the channels to produce a single output feature map. We can more compactly describe this process as a 3D convolution operation.

For example, assume the input to a standard convolutional layer is a $D \times D \times M$ tensor, that is, width and height D and M channels. We want the output to be $D \times D \times N$, also $D \times D$ spatially but with N channels. This implies padding in the spatial dimensions. Further, we'll assume that the kernels are $K \times K \times M$, now 3D tensors, and we must learn N of them, one for each of the N output feature maps.

Convolution becomes, for each spatial position, multiplying the $K \times K \times M$ kernel by the portion of the $D \times D \times M$ input tensor currently being processed before summing the products to produce a single output feature map value. Moving over the spatial dimensions produces the entire output feature map. This process is repeated N times to produce the full output of the layer.

On the other hand, a *depthwise convolutional* layer performs 2D convolution over each input channel independently. With padding, a $D \times D \times M$ input tensor becomes a $D \times D \times M$ output tensor. The layer learns M kernels, each $K \times K$.

In a MobileNet, depthwise convolutional layers are followed by *pointwise convolutional* layers, which are simply standard 2D convolutional layers using a 1×1 kernel (see Equation 13.4). If the pointwise layer produces an N -channel output, the net effect of the pair—the depthwise convolution followed by the pointwise convolution—is

to map a $D \times D \times M$ input tensor to a $D \times D \times N$ output tensor, just like a standard 2D convolutional layer.

Why not use a standard 2D convolutional layer if this is the case? Here's where the cleverness of pairing depthwise and pointwise convolutions comes into the picture: the part that saves substantially on the number of model parameters. Of course, little in life is free, so this cleverness comes with a price, as you'll learn.

How many multiplications are needed to implement a standard 2D convolutional layer mapping a $D \times D \times M$ input to a $D \times D \times N$ output using $K \times K$ kernels? We must know this to understand the savings from pairing depthwise and pointwise convolutions.

Considering 3D convolutions means each spatial position involves $K \times K \times M$ multiplications. And, because we have padding, we must do this $D \times D$ times to convolve the 3D kernel with the input. Finally, all of this produces a single output feature map, and we want N of those. Putting it all together tells us that standard convolution involves the following multiplications to map the $D \times D \times M$ input to a $D \times D \times N$ output:

$$K^2MD^2N$$

Let's do the same for a depthwise convolution followed by a pointwise convolution. Depthwise convolution applies the $K \times K$ kernel to each of the M channels. That's K^2D^2M multiplications. The pointwise convolution uses a 1×1 kernel for D^2M multiplications for each of the N outputs: NMD^2 . Therefore, MobileNet's convolutional blocks use a total of the following multiplications:

$$K^2D^2M + NMD^2$$

Forming the ratio between the depthwise + pointwise cost and the standard convolutional layer cost will tell us if we're saving any multiplications:

$$\begin{aligned} \frac{K^2D^2M + NMD^2}{K^2MD^2N} &= \frac{K^2D^2M}{K^2MD^2N} + \frac{NMD^2}{K^2MD^2N} \\ &= \frac{1}{N} + \frac{1}{K^2} \end{aligned}$$

The MobileNet approach saves on multiplications because the ratio is always less than 1. As N gets bigger, the ratio approaches $1/K^2$. Fix $K = 3$, and, for large N , the savings become approximately $1/9$, meaning pairing depthwise convolutions with pointwise convolutions reduces the number of parameters by nearly a factor of 9. That's quite a savings.

Let's make the numbers explicit. Assume we have an input tensor of $32 \times 32 \times 3$, which matches CIFAR-10. We'll use 3×3 kernels with padding, and we want an output tensor of $32 \times 32 \times 64$. A standard convolutional layer uses the following:

$$3^3(3)(32)^2(64) = 1,769,472 \text{ multiplications}$$

A depthwise + pointwise pair needs only the following, which you get if you multiply 1,769,472 by $1/64 + 1/9$, the scale factor between the two approaches:

$$3^2(3)(32)^2 + 64(3)(32)^2 = 224,256 \text{ multiplications}$$

Change N to 1,024 output channels, and you go from 28,311,552 multiplications to only 3,173,376.

But there is a catch, as almost all CNNs would trade standard convolutional layers for pairs of depthwise and pointwise convolutions if there weren't. To understand this catch, consider the difference between a full $K \times K \times M$ kernel versus M separate $K \times K$ kernels followed by the $1 \times 1 \times N$ kernel of the pointwise layer.

In full convolution, each output feature map value is influenced by every element in the input tensor stack covered by the $K \times K \times M$ kernel. A lot of cross-channel influence is happening. Depthwise convolutions ignore any correlations between channels. The final pointwise convolution does correlate across channels, but only after the depthwise per-channel convolutions, meaning information present across channels in the input isn't used directly but in diluted form. This is the price paid for the reduction in multiplications: a loss of potentially meaningful cross-channel information.

Consider, for a moment, an RGB input. The information across the channels for any spatial pixel position matters because it specifies the actual color of the pixel. That information is diluted when depthwise convolutions are used before the per-pixel pointwise convolution that works across channels. We don't typically know the association between the channels of tensors flowing through deeper convolutional layers in a model, but the entire point of end-to-end training is to encourage the model to use such information. The MobileNet architecture discards much of that information in the name of implementation efficiency.

Building the MobileNet Model

The file *mobilenet.py* contains the code to build the model; see Listing 13-11.

```
inp = Input(input_shape)
_ = Conv2D(32, (3, 3), strides=(1, 1), padding='same', use_bias=False)(inp)
_ = BatchNormalization()(_)
_ = ReLU()(_)
_ = DepthwiseBlock(_, 64)
_ = DepthwiseBlock(_, 128, strides=(2, 2))
_ = DepthwiseBlock(_, 128)
_ = DepthwiseBlock(_, 256, strides=(2, 2))
_ = DepthwiseBlock(_, 256)
_ = DepthwiseBlock(_, 512, strides=(2, 2))
for i in range(5): _ = DepthwiseBlock(_, 512)
_ = DepthwiseBlock(_, 1024, strides=(2, 2))
_ = DepthwiseBlock(_, 1024)
_ = GlobalAveragePooling2D()(_)
_ = Dense(num_classes)(_)
outp = Softmax()(_)
```

Listing 13-11: Building the MobileNet model

The code mixes low-level Keras classes with calls to `DepthwiseBlock`. The input is first passed through a standard convolutional layer with (1, 1) strides. The original MobileNet paper (“MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications” by

Andrew G. Howard et al., 2017) targeted the ImageNet dataset and used a stride of (2, 2) to downsample immediately. MobileNet always uses batch normalization and a ReLU after a convolution.

Next come many calls to `DepthwiseBlock` to create output tensors with 64, 128, 256, 512, and 1,024 channels. The previous section's analysis demonstrated that more channels lead to improved performance savings. Notice the loop building five depthwise blocks in a row, each with 512 channels. As with ResNet, downsampling uses striding, not pooling. A final global pooling layer leads directly to the softmax output layers. Listing 13-12 shows `DepthwiseBlock`.

```
def DepthwiseBlock(x, filters, strides=(1, 1)):
    _ = DepthwiseConv2D((3, 3), padding="same", strides=strides, use_bias=False)
    (x)
    _ = BatchNormalization()(_)
    _ = ReLU()(_)
    _ = Conv2D(filters, (1, 1), padding="same", use_bias=False, strides=(1, 1))
    (_)
    _ = BatchNormalization()(_)
    return ReLU()(_)
```

Listing 13-12: A depthwise block

It's quite compact, nothing more than a call to `DepthwiseConv2D`, a Keras class, and a call to `conv2d` with a 1×1 kernel. Calls to `BatchNormalization` and `ReLU` are in between. Both convolutional layers dispense with biases because they're followed by batch normalization layers.

MobileNet's goal is to minimize parameters. Notice that `DepthwiseConv2D` does not use `filters`, as the output of a depthwise convolutional layer has the same number of channels as the input. It's `conv2d` that pays attention to the desired number of output channels (`filters`). Notice also that downsampling happens in `DepthwiseConv2D`.

With the architecture in place, let's test it and compare the results with VGG8 and ResNet-18.

Testing the MobileNet Model

Run *mobilenet.py* to learn what it expects on the command line:

```
mobilenet <minibatch> <epochs> <outdir>
```

```
<minibatch> - minibatch size (e.g. 128)
<epochs>    - number of training epochs (e.g. 16)
<outdir>    - output file directory (overwritten)
```

The options are few: minibatch size and number of epochs. The *mobilenet_runs* script mimics the scripts from earlier in the chapter to produce five models using a minibatch size of 128 and 10 epochs. MobileNet is smaller than the other models, so it will train in a reasonable amount of time, less than four hours on my test machine.

Table 13-3 presents the MobileNet mean accuracy and MCC along with those for VGG8 and ResNet-18.

Table 13-3: MobileNet Results ($n = 5$)

Model	Accuracy	MCC
VGG8	0.8098 ± 0.0035	0.7903 ± 0.0036
ResNet-18	0.7879 ± 0.0103	0.7664 ± 0.0107
MobileNet	0.6838 ± 0.0109	0.6523 ± 0.0115

MobileNet's showing is significantly weaker than the other two models, likely because of the difference between standard convolutions and the depthwise + pointwise convolutions of MobileNet.

Standard convolutions capture cross-channel information that is crucial for small input images (such as the CIFAR-10 images, which are only 32×32 pixels) when assigning a class label. MobileNet's approach, on the other hand, loses this information, or at least dilutes it significantly, likely hindering performance for small images because critical information is lost. MobileNet's performance is competitive compared to VGG and ResNet-18 on the ImageNet dataset, which uses 224×224-pixel input images, thereby lending some credence to the

notion that CIFAR-10's small images aren't well suited to MobileNet's approach.

Building an Ensemble

Each of the three CNN architectures we developed in this chapter has strengths and weaknesses, so it makes sense to ensemble them in some fashion. We'll build two ensembles: The first uses all three models, and the second only VGG8 and ResNet-18. MobileNet was the weakest model, so it's worth exploring whether it helps or hurts as a member of the ensemble.

While we built ensembles of models by using an elaborate swarm optimization approach in Chapter 12, here we'll keep it simple and instead average the models' softmax outputs before selecting the largest value's index as the class label. The word *average* is ambiguous; we have options.

We'll first try the arithmetic mean: Run the test sample through each model, then sum the resulting softmax vectors and divide by the number of models used.

Alternatively, we might use a weighted arithmetic mean. The simple arithmetic mean gives equal weight to the model predictions, but the models aren't equally effective. Therefore, we might want to weigh the softmax vectors by the overall accuracy of each model and then divide by the sum of the accuracies. Or we might use the MCC instead, believing it to be a better indicator of model performance. We'll try both types of weighting.

Finally, we might have reason to use the geometric mean of the softmax values instead of the arithmetic mean. The softmax outputs are like probabilities; when the models agree, we want that to be emphasized. The geometric mean is high when the models agree, and lower than the arithmetic mean when they aren't in accord. This seems like an advantageous approach when ensembling multiple models.

The code we want is in *ensemble.py*. It's a series of repeated calculations. Listing 13-13 shows the steps for the geometric mean; we

calculate the various arithmetic means similarly.

```
ensemble.py
ytest = np.load("../data/cifar10/cifar10_test_labels.npy").squeeze()
vprob = np.load("runs/vgg_b128_e10_batchnorm_run2/predictions.npy")
rprob = np.load("runs/resnet_b128_e10_bn_run2/predictions.npy")
mprob = np.load("runs/mobilenet_b128_e10_run2/predictions.npy")

gprob = (vprob*rprob*mprob)**(1/3.0)
gprob0 = np.sqrt(vprob*rprob)

gpred = np.argmax(gprob, axis=1)
gpred0 = np.argmax(gprob0, axis=1)
```

Listing 13-13: Ensembling via the geometric mean of the softmax vectors

The code first loads the CIFAR-10 test-set labels and the VGG8, ResNet-18, and MobileNet softmax output values for run 2. I decided to use run 2 results for no specific reason; I picked this at random.

The remaining code calculates the geometric mean of all three models, or VGG8 and ResNet-18. Recall that the geometric mean of n numbers is the n th root of their product (assuming all are > 0). Means of the softmax vectors at the ready, we find the assigned class labels by selecting the largest value (for example, `gpred` from `gprob`).

Running *ensemble.py* produces a series of mean confusion matrices, which you can review, and the results are summarized in Table 13-4. Let's take a moment to understand them.

Table 13-4: Ensembling the Models

Model	Accuracy	MCC
Ensemble (VR, geometric)	0.8561	0.8406
Ensemble (VR, MCC weighted)	0.8513	0.8352

Model	Accuracy	MCC
Ensemble (VR, ACC weighted)	0.8512	0.8351
Ensemble (VR, average)	0.8497	0.8334
Ensemble (all, geometric)	0.8501	0.8341
Ensemble (all, MCC weighted)	0.8452	0.8285
Ensemble (all, ACC weighted)	0.8450	0.8283
Ensemble (all, average)	0.8417	0.8247
VGG8	0.8152	0.7961
ResNet-18	0.7897	0.7677
MobileNet	0.6935	0.6624

The bottom section of Table 13-4 presents the individual model results. These are single models, run 2 of my output from the various scripts used in this chapter. VGG8 is the top performer in this case, as we have come to expect.

The middle section of the table shows the results when averaging all three models. The geometric mean works the best, which aligns with my earlier comments about why we might prefer the geometric mean to any of the arithmetic means. In the table, *average* refers to the simple mean giving equal weight to each model's output.

Finally, the top section shows the results for VGG8 and ResNet-18 without MobileNet. This combination of models is the best, with the geometric mean again proving to be the most useful by achieving an accuracy of 85.6 percent, a full 4 percent higher than VGG8 alone.

However, it is worth remembering that training models is a stochastic process and your runs might return different results, including results where MobileNet contributes effectively.

Summary

This chapter introduced three advanced CNN architectures along with the Keras functional API, which we'll use from this point forward.

The VGG family of models came first. Essentially, they are a collection of deep CNN models that are direct extensions of the CNNs we used in previous chapters. We implemented a reduced version, VGG8, and used it with unaugmented CIFAR-10. You also learned more about batch normalization and the interaction between convolutional layers and dropout, both standard and spatial.

Next, we implemented ResNet-18, the first CNN to add something new to our toolbox: skip connections. Skip connections address the vanishing gradient problem, which vexed early deep learning models and models using sigmoid and hyperbolic tangent activation functions.

Our final model, MobileNet, is a lightweight alternative to the “heavier” VGG and ResNet models. MobileNet replaces standard convolutional layers with pairs of depthwise and pointwise convolutional layers. This change results in a dramatic reduction in the number of convolutional-layer parameters but comes at a cost by throwing away context information, especially between the channels of the layer inputs.

We closed the chapter by ensembling the models to boost overall performance. You learned that, at least for these models and the CIFAR-10 dataset, using the geometric mean of VGG8 and ResNet-18 resulted in a significant improvement in accuracy compared to either model alone.

We trained models from scratch in this chapter. The next chapter investigates the fine-tuning of pretrained models, a commonly used approach.

14

FINE-TUNING AND TRANSFER LEARNING



Fine-tuning and transfer learning, two approaches widely used in deep learning, involve using models trained on one dataset with another dataset, which might be too small to successfully train a model from scratch. This chapter takes advantage of the Keras `applications` module to explore fine-tuning and transfer learning by using models pretrained on the ImageNet dataset.

The terms “transfer learning” and “fine-tuning” are used somewhat loosely at times by deep learning practitioners. For us, *transfer learning* means using a pretrained convolutional network, minus the topmost dense layers, to transform inputs (images) into output vectors (known as *embeddings*) that we then use for a downstream task.

Fine-tuning, on the other hand, uses the pretrained model as a starting point for training on another dataset. Often, some of the pretrained model is frozen, so training updates only the weights and biases associated with specific layers, usually the topmost dense layers. With fine-tuning, we hope that the features learned by the pretrained model are useful in learning the new dataset. Typically, the pretrained model uses a large labeled dataset, like the over one million images in

ImageNet (N), while the fine-tuning dataset is often significantly smaller (n). In other words, $N \gg n$ is often true.

This chapter experiments with fine-tuning and transfer learning by using VGG16 and MobileNet models pretrained on ImageNet along with the CIFAR-10 dataset. We begin with fine-tuning followed by transfer learning. Along the way, you'll learn what makes our transfer learning experiments considerably more useful.

Fine-Tuning a Pretrained Model

Fine-tuning a pretrained model follows a simple workflow:

1. Begin with a pretrained model, usually one trained on data similar to the fine-tuning dataset.
2. Remove the topmost layers. This step isn't required but is often done, especially if the topmost layers are dense layers that might be too large for the fine-tuning dataset.
3. Add new topmost layers suitable for the fine-tuning dataset—for example, for the number of classes and size of the fine-tuning dataset.
4. Freeze most, some, or all of the pretrained layers so training updates only the weights and biases of the new topmost layers.
5. Train the entire model on the fine-tuned dataset and test as usual against a held-out test set.

Consider these steps to be guidelines; adapt as needed for the task at hand. This workflow admits to many variations, some of which we'll explore in this section.

With fine-tuning, we hope that the features learned by the convolutional layers of the pretrained network are helpful in classifying the fine-tuning dataset, either as they are (if frozen) or as a good starting point, giving gradient descent something to work with. Think of it as an even better model initialization.

The remainder of this section walks through the fine-tuning of VGG16 and MobileNet ImageNet pretrained models using CIFAR-10. This illustrates the process and exposes us to the Keras collection of pretrained models. Then, we dive into the details of fine-tuning with VGG16 and MobileNet to learn which mix of frozen and unfrozen layers works the best (to the level explored) and how to get at said layers and their output in code.

VGG16 and MobileNet with CIFAR-10

Chapter 13 introduced VGG8, a scaled-down version of the more common VGG16 or VGG19 models. As it happens, VGG16 has become a widely used pretrained architecture for fine-tuning. To begin to understand the architecture, see Figure 14-1.

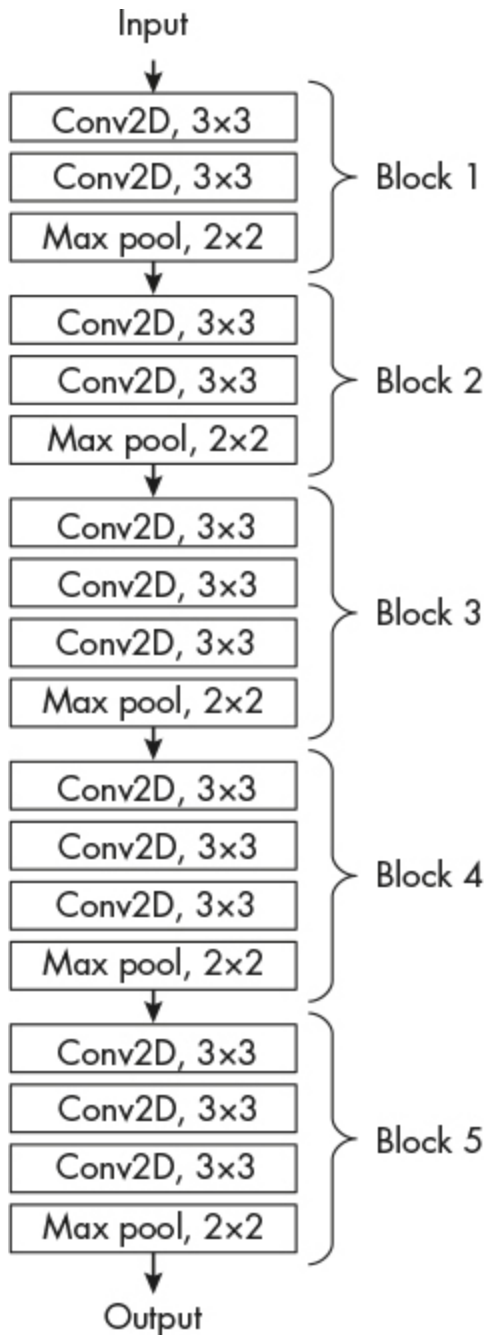


Figure 14-1: The VGG16 architecture minus the topmost layers

This diagram separates the convolutional portion of VGG16 into five blocks. Each block consists of two or three convolutional layers with 3×3 kernels and ReLU activation functions followed by a 2×2 max-pooling layer. We'll experiment with freezing by block, though we can also work with individual layers if desired.

The input to the model is a 32×32 RGB CIFAR-10 training image. The output tensor depends on the input. For CIFAR-10, the output is a $1 \times 1 \times 512$ array, which we'll flatten to a 512-element vector. In other words, the VGG16 architecture transforms a $32 \times 32 \times 3$ input array, the RGB image, into a 512-element representation by using the weights and biases learned from the pretraining exercise on the ImageNet dataset.

Note that CIFAR-10 is 32×32 pixels, and ImageNet is 224×224 pixels, but there is no issue because of the convolutional nature of the layers. The learned 3×3 kernels of each convolutional layer's filters are just as happy to convolve over a 32×32 input as they are over a 224×224 input. The dense layers at the top, which Figure 14-1 removes, restrict the full VGG16 model to 224×224 -pixel inputs.

However, we still need to complete the model by adding new dense layers. The input to the dense layers is a 512-element vector. I arbitrarily picked adding two 1,024-node dense layers, each with ReLU activation functions, dropout at $p = 0.5$, and batch normalization. Arguments have been made for and against using dropout and batch normalization together, but in this case, it seems to help to have both. I encourage you to test my claim.

Using the Applications Module

Keras includes VGG16 in its list of pretrained models, which are accessed via the `applications` module. The code we need is in *`fine-tune_vgg16.py`*. It isn't complex, but it's a bit longer than some of the other examples we've considered. Please review it in its entirety. I'll illustrate only portions of the code here.

We're already familiar with the CIFAR-10 dataset, so I'll gloss over loading it, but we do need to consider scaling it. Fortunately, the `applications` module includes not only the models, but also `preprocess_input` functions for each architecture. Assuming that we've already loaded the CIFAR-10 train and test images as `x_train` and `x_test`, respectively, Listing 14-1 imports from Keras and preprocesses the data.

```
from tensorflow.keras.applications.vgg16 import VGG16, preprocess_input
x_train = preprocess_input(x_train)
```

```
x_test = preprocess_input(x_test)
```

Listing 14-1: Importing the model and preprocessing the data for VGG16

Constructing the base VGG16 model (Figure 14-1) depends on whether we want to use the pretrained ImageNet weights and biases or want to train from scratch. The experiments use both approaches, as reflected by the code in Listing 14-2.

fine-tune_vgg16.py

```
inp = Input(shape=(32, 32, 3))
if (weights == 'imagenet'):
    base = VGG16(input_tensor=inp, include_top=False, weights='imagenet')
    idx = [0, 4, 7, 11, 15, 19][freeze]
    if (idx > 0):
        for layer in base.layers[:idx]:
            layer.trainable = False
else:
    base = VGG16(input_tensor=inp, include_top=False, weights=None)
```

Listing 14-2: Configuring the VGG16 base model

First, define a suitable `Input` layer for 32×32 RGB images (three channels). Next, if using the ImageNet weights, define `base` to be an instance of `vgg16`, passing in the newly defined input (`inp`), using no top layers, and specifying ImageNet weights. The `else` clause illustrates using no ImageNet weights (standard initialization).

If using ImageNet weights, freeze all layers through the desired block, `freeze`, as read from the command line. The value in `idx` corresponds to the end of the block in Figure 14-1 plus one for the input layer. Frozen layers will not be updated during gradient descent.

Adding the top layers follows, as Listing 14-3 illustrates.

```
_ = Flatten()(base.output)
_ = Dense(1024)(_)
_ = BatchNormalization()(_)
_ = ReLU()(_)
_ = Dropout(0.5)(_)
_ = Dense(1024)(_)
```

```
_ = BatchNormalization()(_)
_ = ReLU()(_)
_ = Dropout(0.5)(_)
_ = Dense(num_classes)(_)
outp = Softmax()(_)
model = Model(inputs=inp, outputs=outp)
```

Listing 14-3: Completing the fine-tuning VGG16 model

The model is now complete. It includes the base VGG16 model and the new topmost layers. The remainder of *fine-tune_vgg16.py* compiles the model by using Adam as the optimizer, trains the model by using the fraction of the CIFAR-10 training set as specified on the command line, and stores the model along with information about the training process by using the same code as in Chapter 13.

The code expects the following on the command line:

```
>>> python3 fine-tune_vgg16.py
fine-tune_vgg16 <minibatch> <epochs> <weights> <freeze> <outdir> [<fraction>]

<minibatch> - minibatch size (e.g. 128)
<epochs>    - number of fine-tuning epochs (e.g. 12)
<weights>   - 'none' | 'imagenet' (use pretrained weights)
<freeze>    - freeze blocks through <freeze>, e.g. 5 for all
<outdir>    - output file directory (overwritten)
<fraction>  - fraction of full training set to use (def=1.0)
```

The arguments are as we expect with two new ones: *weights* and *freeze*. The former is a string specifying ImageNet pretrained weights or initialization from scratch. The latter determines how many of the VGG16 blocks to freeze. If *freeze* is 5, all VGG16 layers are frozen, and training updates only the topmost dense layers. If it's 0, no layers are frozen, which treats the ImageNet pretrained weights and biases as a specific set of “random” initialization values.

The number of unfrozen layers determines the number of trainable weights and biases, which affects the speed at which the model trains. Some patience is required at times, but we remain able to run experiments without resorting to a GPU-enabled machine.

Tuning MobileNet

The file *fine-tune_mobilenet.py* implements a MobileNet version. It's nearly identical to the VGG16 version, so I'll let you review the code; however, asking for the command line options reveals a key difference:

```
>>> python3 fine-tune_mobilenet.py
fine-tune_mobilenet <minibatch> <epochs> <weights> <unfreeze> <outdir>

<minibatch> - minibatch size (e.g. 128)
<epochs>    - number of fine-tuning epochs (e.g. 12)
<weights>   - 'none' | 'imagenet' (use pretrained weights)
<unfreeze>  - unfreeze the top <unfreeze> depthwise blocks
               0=freeze all blocks, -1=freeze nothing
<outdir>    - output file directory (overwritten)
```

First, the code provides no option to alter the fraction of the CIFAR-10 dataset used during training. Feel free to add that if you wish. Second, instead of specifying the number of frozen blocks beginning with the lowest (closest to the input), there's an option to *unfreeze* depthwise blocks starting with the top of the MobileNet portion of the model. Therefore, a larger *unfreeze* value means allowing fine-tuning of more and more of the MobileNet convolutional layers, with -1 implying freeze no layers.

Testing the Fine-Tuned Models

Let's take *fine-tune_vgg16.py* out for a test drive with this command:

```
>>> python3 fine-tune_vgg16.py 128 12 imagenet 5 results/run0
```

The command line initializes the model with ImageNet weights and then trains for 12 epochs with a minibatch size of 128 using all the CIFAR-10 training set. The 5 means to freeze all the VGG16 layers, so only the two new dense layers at the top of the model are updated. In other words, we'll treat the VGG16 portion of the model as a feature generator that transforms the CIFAR-10 images into a new representation, a 512-element vector. Keep this perspective in the back of your mind; we'll return to it later in the chapter.

The code dumps the now familiar model summary and then trains for 12 epochs. The model has 16.3 million parameters, but only 1.6 million are free to train. A minibatch size of 128 implies 391 minibatches per epoch ($n = 50,000$), for a total of $391 \times 12 = 4,692$ gradient-descent steps updating only the topmost dense layers.

When training is complete, we're given a summary of the results with the test-set confusion matrix. My run produced the following:

```
Test set accuracy: 0.6994, MCC: 0.6665
array([[779, 27, 21, 14, 17, 9, 8, 13, 74, 38],
       [ 28, 812, 6, 5, 5, 10, 6, 3, 21, 104],
       [ 78, 10, 509, 70, 144, 52, 101, 18, 9, 9],
       [ 25, 24, 39, 514, 54, 172, 90, 43, 11, 28],
       [ 26, 4, 42, 50, 657, 34, 85, 80, 15, 7],
       [ 9, 7, 31, 141, 45, 661, 41, 43, 6, 16],
       [ 9, 12, 29, 60, 50, 34, 794, 4, 3, 5],
       [ 20, 9, 16, 34, 82, 67, 13, 723, 9, 27],
       [ 79, 63, 11, 10, 11, 5, 5, 8, 774, 34],
       [ 42, 103, 3, 20, 4, 9, 3, 21, 24, 771]], dtype=uint16)
```

The model was about 70 percent accurate on the full CIFAR-10 test set, which isn't too bad, but the confusion matrix is quite noisy, implying many erroneous classifications. We froze the VGG16 convolutional weights so they could not adapt to the CIFAR-10 images. Recall that higher convolutional layers learn representations related to the content of the input images, not just the low-level edges, colors, and textures learned by the first few convolutional layers. There's good reason to believe that the higher convolutional layers are not well suited to the new dataset. We probably want to allow some of the pretrained layers to adapt.

Experiments

Two thoughts come to mind regarding the previous section's run of *fine-tune_vgg16.py*: Training a model once isn't usually sufficient, and what happens if not all the VGG16 pretrained layers are frozen?

Let's train models at least twice to address the first thought and report the average between them. I prefer at least five or six examples for any architecture to account for bad initializations, but CPU training is rather slow.

Addressing the second thought is straightforward: Change the 5 on the command line to a 4, then a 3, and so on to freeze successively less and less of the pretrained VGG16 model.

I wrote a small shell script to run *`fine-tune_vgg16.py`* twice for each number of frozen blocks, from 0 through 5. I then ran it two more times, replacing `imagenet` with `none` to train the entire VGG16 model from scratch. I used the full CIFAR-10 training set in all cases and reported performance on the test set. I put all the output in the *`results`* directory with the following naming convention, varying between `run0` and `run1` for each level of frozen blocks:

```
results/fine-tune_vgg16_128_12_0_run0
```

Then I left it for a day or two. When the dust settled, I ran *`fine-tune_vgg16_plot.py`* to produce Figure 14-2.

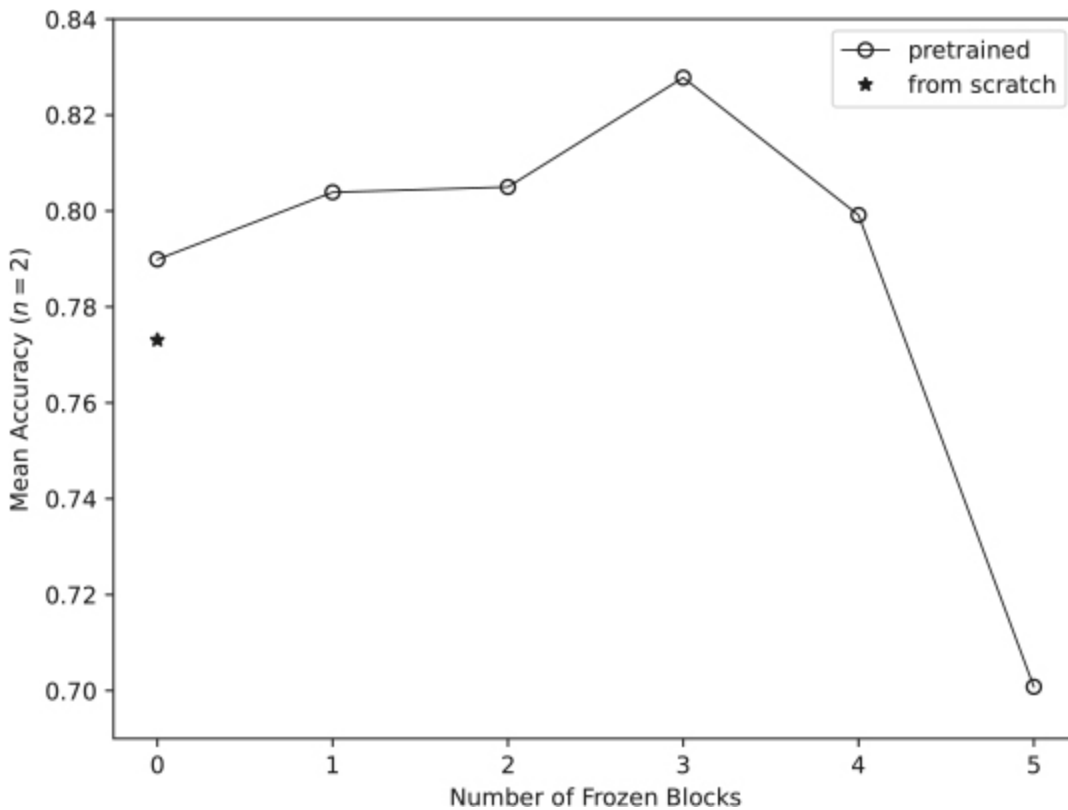


Figure 14-2: The test-set performance for VGG16 fine-tuned on CIFAR-10 by the number of frozen blocks

The open circles represent the average accuracy for two runs with that many frozen pretrained blocks. It's evident that freezing all five blocks is not ideal, thereby validating the intuition from the previous section. The best-performing model achieves an accuracy of over 82 percent when the first three blocks are frozen, but the last two—those most tailored to the ImageNet dataset—are free to adapt themselves along with the new dense layers at the top.

The circle at the far left of the plot shows the performance when no blocks are frozen, thereby treating the ImageNet pretrained weights and biases as a particular initialization. The solid star represents the mean performance from the two `none` training runs. With only two samples per scenario, we don't have error bars, let alone the ability to perform statistical tests. However, that the ImageNet pretrained features perform better is consistent with our expectations. After all, ImageNet contains natural images, as does CIFAR-10.

What about *fine-tune_mobilenet.py*? Like VGG16, the MobileNet architecture consists of repeated blocks combining a depthwise convolutional layer and a 1×1 convolutional layer (see Chapter 13). The code first freezes all convolutional layers and then unfreezes the topmost n blocks as given on the command line. Therefore, an argument of 0 means freeze everything, an argument of 3 means unfreeze the three topmost blocks, and an argument of -1 means freeze nothing. For example, the following trains a model for 12 epochs by using ImageNet pretrained weights, 128 samples per minibatch, with the top four blocks of the convolutional portion unfrozen:

```
> python3 fine-tune_mobilenet.py 128 12 imagenet 4 run0
```

As with *fine-tune_vgg16.py*, the model concludes with two 1,024-node dense layers. There are 13 depthwise convolutional blocks. Repeating the VGG16 exercise and adjusting the number of unfrozen blocks produces Figure 14-3.

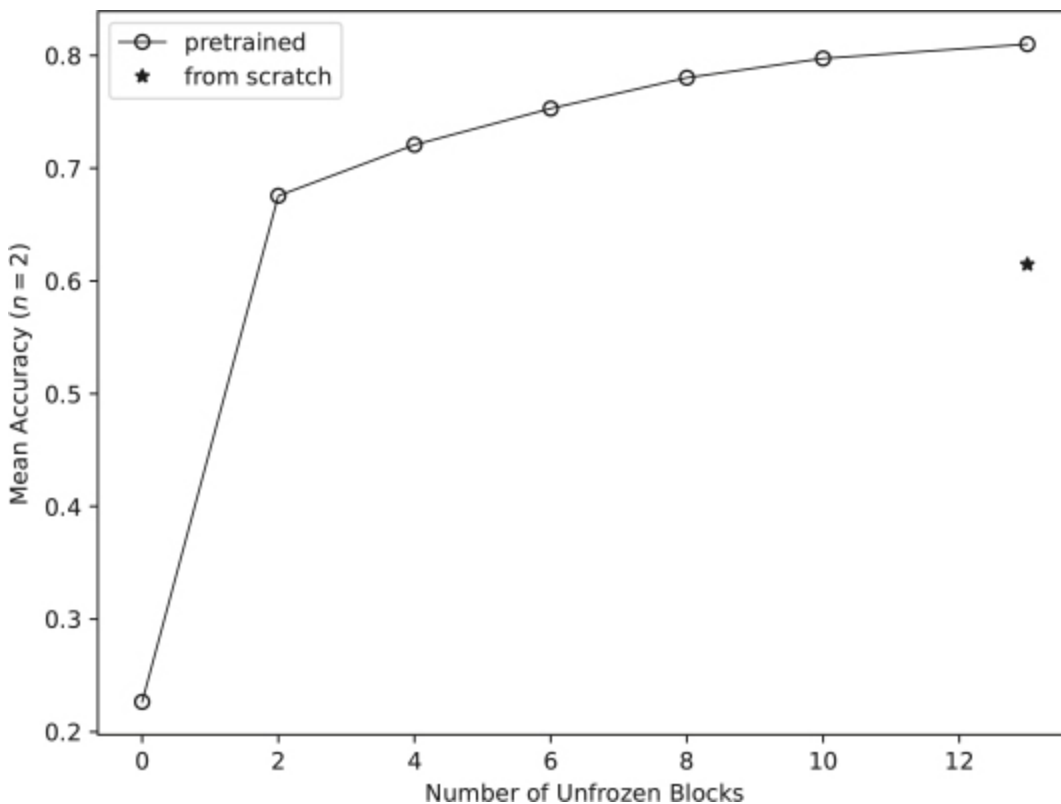


Figure 14-3: The test-set performance for MobileNet fine-tuned on CIFAR-10 by the number of unfrozen blocks

The sense of the x-axis is reversed: Moving from left to right implies that more of the pretrained portion of the model is free to adjust itself during training. Performance improves as the number of free blocks increases, with best performance when all blocks are unfrozen. Notice that the from-scratch models are about 20 percent lower in accuracy than fine-tuning the pretrained model. As with the VGG16 test, all 50,000 training samples were used during training.

Did fine-tuning buy us anything in the end? The fine-tuned MobileNet model achieved about 81 percent average accuracy on the test set. The fine-tuned VGG16 models did better, averaging nearly 83 percent. The VGG8 model from Chapter 13 was around 81 percent accurate when trained from scratch, making fine-tuning MobileNet essentially a wash. Fine-tuning VGG16 offered a slight improvement over VGG8, likely not statistically significant, and didn't match a simple ensemble's nearly 86 percent accuracy (see Table 13-4 on page 376). In this case, fine-tuning was helpful but not essential; we achieved nearly the same performance from a smaller model.

Note, however, that we didn't exhaust all options for building the best possible models. No adjustments were made to most of the hyperparameters, like minibatch size, ideal number of epochs, weight decay, momentum, other optimization algorithms besides Adam, data augmentation, and so on. I suspect, however, that the relative difference between fine-tuned larger models and the smaller VGG8 model suited to the dataset size will remain much the same. The full CIFAR-10 dataset contains a lot of information. But what if we didn't have 50,000 training samples? Will fine-tuning produce a useful model when using 500, or even 50, training samples per class? Let's find out.

A Study in Fine-Tuning

I claimed earlier that a primary motivation for fine-tuning is to take advantage of features learned by models trained on large, labeled datasets when working with small training sets. Let's put that claim to the test by using 1, 3, 5, and 10 percent of the full 50,000 CIFAR-10 training samples to fine-tune VGG16.

We'll fix the minibatch size and training epochs to 128 and 12, respectively. This means that the 1 percent models will use far fewer gradient-descent steps than the 10 percent models, but for the moment, we're more interested in the relative performance difference between fine-tuned VGG16 using ImageNet initialization, random initialization, and the smaller VGG8 model introduced in Chapter 13. For good measure, we'll also test both Adam and SGD as optimizers. The community has reported that Adam sometimes fails to perform well with small datasets, so testing a second optimizer seems reasonable.

First, run the following two commands:

```
> python3 fine-tune_vgg16_fraction.py
> python3 fine-tune_vgg16_fraction_plot.py
```

This produces Figure 14-4.

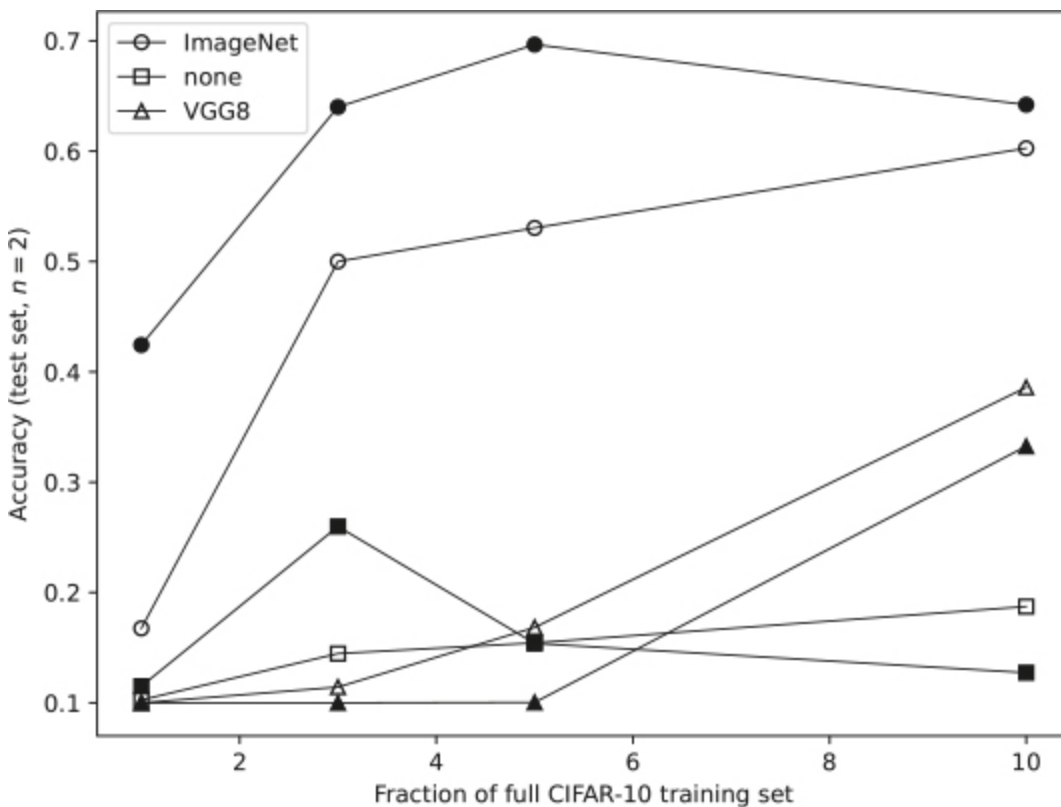


Figure 14-4: Fine-tuning VGG16 with fractions of the full CIFAR-10 training set

The code in the first command calls *fine-tune_vgg16.py*, *fine-tune_vgg16_sgd.py*, and *vgg8.py* repeatedly, passing the fraction of the

CIFAR-10 training set to use for fine-tuning as the last command line parameter. Each model is trained twice with the average accuracy on the full 10,000 sample test set reported in Figure 14-4.

The plot contains six curves; open symbols indicate Adam, and solid symbols indicate SGD. Circles report average performance for VGG16 initialized with ImageNet freezing the first three blocks, which performed best with the full CIFAR-10 training set. Squares represent VGG16 with random initialization and no blocks frozen. Triangles signify VGG8's performance.

It's apparent that ImageNet pretraining works the best, regardless of the fine-tuning dataset. Moreover, SGD outperforms Adam. The SGD learning rate was fixed at $\eta = 0.005$, half the Keras default value.

Interpreting the square and triangle plots requires more nuance, especially with the smallest fine-tuning training sets. At 1 percent, about 50 examples of each class, both models—regardless of the optimizer—fail, performing at essentially chance level.

Random initialization is persistently poor up to 10 percent, or about 500 examples per class. I interpret the bump for SGD and random initialization at 3 percent to illustrate the scatter around the mean. In other words, if we train many models with random initialization, Adam, and SGD, we'd find large error bars associated with the square plot points.

VGG8's experience is different. For each dataset, Adam performs better than SGD, the more typical training experience. Interestingly, the two curves share the same shape, but a constant performance gain is relatively consistent when using Adam. However, the gap closes somewhat for the 10 percent dataset, so using 20, 30, 50 percent, or more of the full training set might see the gap vanish. Why not give it a try?

Let's focus on the 1 percent case. The code fixes NumPy's pseudorandom number seed so each run produces the same sampled dataset. The class distribution for the 1 percent case is the following, with a maximum of 60 deer (class 4) and a minimum of 34 dogs (class 5):

```
>>> np.bincount(ytrain)
array([55, 38, 49, 46, 60, 34, 56, 56, 53, 53])
```

Training VGG8 or VGG16 from scratch with a 1 percent training set fails, as we might expect. VGG8 uses more nodes in its dense layers than VGG16, giving both models roughly the same number of parameters, 13.8 million versus 16.3 million, respectively. We can't expect such a small training set to condition such models.

However, VGG16 using pretrained ImageNet weights does learn at 1 percent. The Adam version barely learns, but the SGD version performs with an overall accuracy of about 43 percent, four times chance level. This is encouraging. Recall that the minibatch size is large (128) and the number of training epochs low (12), implying few gradient-descent steps using a perhaps reasonable estimate of the gradient direction (from the 128 samples in the minibatch).

Let's see if we can do better still. If we can, we'll have a strong example of the utility of fine-tuning. First, let's use a smaller minibatch size to take more gradient-descent steps per epoch. We need to balance things. A very small minibatch maximizes the number of steps at the expense of a poorer estimate of the true gradient direction. We'll use 32 training samples per minibatch, increasing the number of gradient-descent steps by a factor of four. We'll also train longer: 60 epochs versus 12.

The shell script *fine-tune_1_percent* calls *fine-tune_vgg16_sgd.py* five times to train five models by using 1 percent of the CIFAR-10 training set. The *augment* version of the script does the same but adds a new command line argument, *augment*, which augments each training sample nine times. Augmentation includes flipping the image left to right (50 percent probability) followed by either a random rotation of $[-3, 3]$ degrees (33 percent probability) or a random shift $[-3, 3]$ in x and y (66 percent probability). The rotation introduces a small edge effect because the images remain 32×32 pixels, but we can live with that. The shifts are *rolls*, meaning pixels that would fall off one edge are instead moved to the other edge.

Will training with ImageNet weights and biases, a 10× augmented dataset using a smaller minibatch size, more epochs, and SGD result in models outperforming the results in Figure 14-4? I executed both shell scripts, followed by *`fine-tune_vgg16_1_percent_results.py`*. Table 14-1 shows the results parsed from the text output by the code.

Table 14-1: Fine-Tuning Overall Accuracy with 1% of CIFAR-10

	Unaugmented	Augmented
Average model	0.61158 ± 0.00180	0.63286 ± 0.00372
Ensemble model	0.64140	0.65770

The average model row gives us the mean and standard error over the five individual models by unaugmented or augmented datasets. The unaugmented dataset is the same one from Figure 14-4 but trained for 60 epochs using a 32-sample minibatch. The augmented dataset, as described, used the original samples plus nine augmented versions of each. More training with a smaller minibatch size improved the average model’s performance considerably, from about 43 percent overall accuracy to 61 percent. Data augmentation increased this even more to a mean of 63 percent—all from about 50 training samples per class. I would suggest that these results justify fine-tuning.

We have five individual models, so why not combine them as we did in Chapter 13, using the geometric mean of the per-sample softmax outputs? That’s the *Ensemble model* line in Table 14-1. We expect better performance from an ensemble, and we get it—even though, in this case, every model uses the same architecture. The augmented models give us a final accuracy of 65.8 percent from a mere 50 or so actual training samples per class. You’ll find the relevant code in *`fine-tune_vgg16_1_percent_results.py`*.

The augmented ensemble’s confusion matrix is shown here:

[[786	16	44	15	34	2	6	5	50	42]
[	36	765	6	4	6	2	15	3	47 116]

```
[103  3 534  53 116  46  98  32  9  6]
[ 39  7 121 362  92 126 157  53 23 20]
[ 34  1  97  32 651  26  89  60  3  7]
[ 15  6 109 154  78 476  94  60  5  3]
[ 17  3  71  27  57  9 805  4  3  4]
[ 21  3  46  47  93  42  31 691  3 23]
[159 22  11  4  25  1  1  5 740 32]
[ 46 110  2  11  12  1 10  11 30 767]]
```

We have 1,000 samples of each class in the CIFAR-10 test set, meaning the sum across each row of the confusion matrix is exactly 1,000. Class 6 (frogs) performs best at 80.5 percent accuracy. The worst is class 3 (cats) at 36.2 percent. Cats are most often confused with class 6 (frogs), class 5 (dogs), and class 2 (birds), in that order.

Learning frogs from relatively few examples makes sense to me. After all, a frog's appearance is distinctive, doubly so among the other CIFAR-10 classes. Confusing cats with dogs also seems reasonable, though confusing cats with frogs and birds is a tad strange at first blush. The CIFAR-10 images are only 32×32 pixels, so the poor performance on cats is likely due to the small image size; we simply don't have enough pixels to capture the defining characteristics of a cat versus the other animals.

Fine-tuning works and is a valuable technique that we should use when we can. However, the embeddings produced by models like VGG16 pretrained on ImageNet are themselves of value beyond fine-tuning to another dataset. Let's broaden our view to explore pretrained models as feature generators.

Creating New Features with Transfer Learning

Classical machine learning failed most image-classification tasks because the raw pixels were too redundant for the small models to extract meaningful distinguishing characteristics. CNNs changed the game by introducing joint learning of new feature representations and the classifier.

Pretrained convolutional portions of networks, like VGG16 pretrained on ImageNet, act like feature generators to transform inputs (for us, images) into new representations that we can use for other purposes. This is our definition of *transfer learning*. In this section, we explore some of the implications.

Extracting CIFAR-10 Features with VGG16

Let's transform the CIFAR-10 dataset by using pretrained VGG16. Conceptually, we're mapping the raw images into a new embedding space, the flattened output of the convolutional portion of VGG16. We'll do this twice: first using the full VGG16 model and then using the output of block 3 (layer 10).

You learned earlier in the chapter that freezing the first three blocks produced the best fine-tuned model, which implies that the output of the first three blocks is a good representation of the CIFAR-10 inputs. Listing 14-4 illustrates the transformation process.

```
import numpy as np
import tensorflow.keras as keras
from tensorflow.keras.applications.vgg16 import VGG16, preprocess_input
from tensorflow.keras import Model, Input
from tensorflow.keras.layers import Flatten

x_train = np.load("../data/cifar10/cifar10_train_images.npy")
x_test = np.load("../data/cifar10/cifar10_test_images.npy")
x_train = preprocess_input(x_train)
x_test = preprocess_input(x_test)

inp = Input(shape=(32, 32, 3))
base = VGG16(input_tensor=inp, include_top=False, weights='imagenet')
outp = Flatten()(base.output)
model = Model(inputs=inp, outputs=outp)

xtrn = model.predict(x_train, verbose=0)
xtst = model.predict(x_test, verbose=0)
```



```
np.save("features_cifar10_train.npy", xtrn)
np.save("features_cifar10_test.npy", xtst)
```

Listing 14-4: Passing CIFAR-10 through the pretrained VGG16 model

The input images are read from disk and then passed to Keras's `preprocess_input` to condition them for the VGG16 model, changing the channel order from RGB to BGR and subtracting the mean ImageNet color value per channel.

Performing the mapping requires a small model. The second code paragraph after the imports builds the model by adding a suitable input layer, initializing VGG16 with ImageNet weights, followed by a `Flatten` layer to transform the 3D output of the final max-pooling layer, a $1 \times 1 \times 512$ tensor, into a single 512-element vector. We'll use this vector as the new representation of the input image. The final code paragraph in Listing 14-4 runs the train and test data through the model before dumping the transformed dataset to disk. You'll find the code in *features_cifar10_vgg16.py*.

Extracting the output of block 3 requires the creation of an ancillary model, a commonly used technique and one that you should be familiar with. The complete code is in *features_cifar10_vgg16_block.py*. Listing 14-5 teaches us the trick.

```
inp = Input(shape=(32, 32, 3))
base = VGG16(input_tensor=inp, include_top=False, weights='imagenet')

layer = base.layers[10]
model = Model(inputs=base.input, outputs=layer.output)

xtrn = model.predict(x_train, verbose=0).reshape((50000, 4*4*256))
xtst = model.predict(x_test, verbose=0).reshape((10000, 4*4*256))
```

Listing 14-5: Extracting the output of an arbitrary layer

First, create an `Input` layer and VGG16 base model, as before. Then, access the desired layer, whose output we want to capture with `base.layers[10]`. Calling the `summary` method on `base` lets us count which layer we need—here, the output of the `MaxPooling` layer of block 3.

Now comes the trick: Create a new `Model` by using the base model's input and the output of layer 10 as the output. This is `model`, and we use it via `predict` to generate new feature vectors by reshaping the $4 \times 4 \times 256$ output tensor into a 4,096-element vector.

Visualizing Features

Our fine-tuning experiments led us to believe that we might expect the features extracted from the ImageNet pretrained VGG16 model to be more useful as a new representation of CIFAR-10 than the raw images. Let's use t-SNE to visualize the raw images as 3,072-element vectors, the 512-element features extracted from the top of VGG16, and the 4,096-element features from layer 10. Consider Figure 14-5, which is the output of *features_tsne_plots.py*.

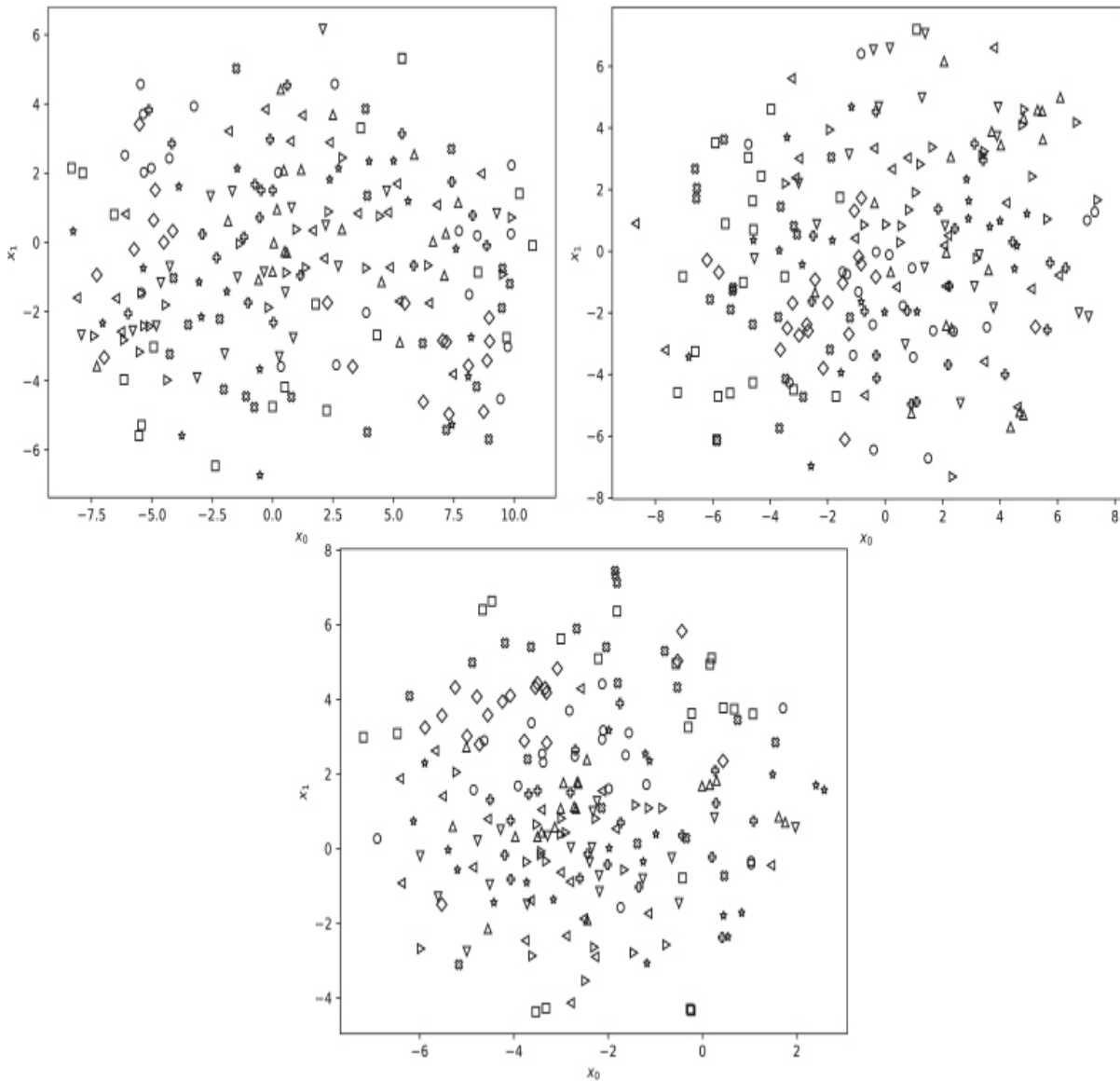


Figure 14-5: The t-SNE plots of sample features: raw (top left), embedded (top right), layer 10 (bottom)

Each plot shows 200 randomly selected test-set samples, the same for each plot—20 samples per class, hence the 10 different plot symbols. Which symbol corresponds to which class isn't important; instead, we'll focus on the organization of the points in the plots. We have the raw features (images), embedded features (output of VGG16), and layer 10 features.

If the features are clustered in their respective 3,072-, 512-, or 4,096-dimensional space, we expect the t-SNE plot to capture these clusters. Visually, I can't discern any meaningful clustering in the plots.

If we look hard enough, we might begin to believe we see some organization, especially for the embedded features (middle) or the layer 10 features (right), but we know better than to trust our eyes too much.

Is there something we can do to gain confidence that the features generated by VGG16 are intrinsically more helpful than the raw images? Yes, we can measure the purity of the clusterings.

Measuring Purity

Purity counts the number of samples (class labels) in each cluster that match the most common label in that cluster. This is the dominant label. For example, if the cluster contains 19 samples and 16 of them are of the same class (say, class 3), then that class (class 3) is the dominant class for that cluster. Purity is the sum of the number of samples in the dominant cluster for each cluster, divided by the total number of samples. A clustering that assigns every sample to clusters where that label is the only label in the cluster will have a purity of 1. Therefore, clusterings producing purities closer to 1 are more desirable. If the features are well grouped initially, we expect a machine learning model to learn more easily than with features that are not well grouped initially.

Listing 14-6 calculates the purity for `rr`, `ee`, and `bb`, 2D arrays holding the 200 points in Figure 14-5 for the raw, embedded, and layer 10 features, respectively.

```
features_tsne_plots.py
from sklearn.cluster import KMeans
krr = KMeans(n_clusters=10).fit_predict(rr)
kee = KMeans(n_clusters=10).fit_predict(ee)
kbb = KMeans(n_clusters=10).fit_predict(bb)

srr, see, sbb = [], [], []
for i in range(10):
    idx = np.where(krr == i)[0]; srr.append(ysamples[idx])
    idx = np.where(kee == i)[0]; see.append(ysamples[idx])
    idx = np.where(kbb == i)[0]; sbb.append(ysamples[idx])
```

```

srrr = scee = scbb = 0
for i in range(10):
    srrr += np.bincount(srr[i], minlength=10).max()
    scee += np.bincount(see[i], minlength=10).max()
    scbb += np.bincount(sbb[i], minlength=10).max()

N = len(ee)
print("Purity:")
print("    raw image features: %0.4f" % (srrr / N))
print("    embedded features : %0.4f" % (scee / N))
print("    layer 10 features : %0.4f" % (scbb / N))

```

Listing 14-6: Calculating the clustering purity of the features in t-SNE space

In the first code paragraph, sklearn's `KMeans` class learns 10 cluster centers, ideally 1 for each of the 10 classes in CIFAR-10. The second code paragraph captures the class label (from `ysamples`) for each sample in each cluster. The third code paragraph uses NumPy's `np.bincount` to determine the number of samples with the dominant label for each cluster. Finally, the `print` calls show the purity of each clustering by dividing the sum across clusters by the total number of samples:

```

Purity:
    raw image features: 0.2800
    embedded features : 0.3400
    layer 10 features : 0.3450

```

The layer 10 features have the highest purity. Therefore, we might claim that the layer 10 features are more organized along the CIFAR-10 classes and should lead to better-performing models than the full VGG16 features or the images themselves. The fine-tuning experiments earlier in the chapter confirm this intuition.

However, the code uses a fixed pseudorandom number seed, and we have only 20 samples per class in the t-SNE plots. We should wonder how representative the output is. What's the distribution over repeated samples from the CIFAR-10 dataset? To find out, I ran a slightly doctored version of the code 20 times by using 100 samples per class and tracked the purity in each case.

The raw image purity was 0.2306 ± 0.0028 (mean $\pm$ SE). The full VGG16 features led to a purity of 0.3271 ± 0.0046 , and the layer 10 features led to a purity of 0.3416 ± 0.0048 . The difference in means between the raw features and the VGG16 features (full or layer 10) is clearly statistically significant. For the two VGG16 features, we get $p = 0.034$ from the t-test and $p = 0.040$ from the Mann–Whitney U test. Both lean toward statistical significance when using the traditional threshold of $p < 0.05$, but the evidence isn't overwhelming. Cohen's $d = 0.69$ indicates a medium-large effect size.

Putting all three results together implies a true difference in purity between the full VGG16 features and the layer 10 features, one that is practically large in its effect. Therefore, in terms of performance, I expect to observe the following when using these features for downstream classification tasks:

$$M_{\text{raw}} < M_{\text{VGG16}} < M_{\text{layer 10}}$$

Here M is a model of some kind. Let's test my claim.

Training Classical Models with CIFAR-10 Features

Suppose the features extracted from the pretrained VGG16 model are better representations of the raw CIFAR-10 images. In that case, we should be able to create reasonably good models from them, even if using classical machine learning techniques. This section puts that belief to the test.

The code we need is in *features_classic.py*. It uses sklearn to train old-school models by using the raw, VGG16, and layer 10 features. The *features_classic.py* file mirrors the code we used in Chapter 4, so I won't list it here. If you review it, you'll notice the full CIFAR-10 dataset is reduced to 20,000 training and 5,000 test samples. You'll also notice that the data is standardized to have a per-feature mean of 0 and a standard deviation of 1 before training the MLP.

Table 14-2 presents the output I found after executing *features_classic.py* with `raw`, `vgg16`, and `layer10` as the command line arguments.

Table 14-2: Classical Models on CIFAR-10 Features

	Raw	VGG16	Layer 10
Gaussian naive Bayes	0.2976	0.4740	0.5876
Nearest centroid	0.2758	0.5278	0.5830
k-NN ($k = 3$)	0.3068	0.5244	0.6508
k-NN ($k = 7$)	0.3212	0.5540	0.6590
Random forest ($n = 100$)	0.4428	0.6098	0.6702
Random forest ($n = 400$)	0.4610	0.6250	0.6876
MLP (512, 256)	0.4810	0.6414	0.8002

The Raw column represents the state of the world before the advent of deep learning. The RGB images were unraveled into vectors and passed to the models. The best model was less than 50 percent accurate. While that's five times better than random guessing, such a model is not likely helpful in practice.

Using the pretrained VGG16 features improves matters for all model types. The layer 10 features offer further improvements. Notice, in particular, the 80 percent accuracy achieved by the MLP. There's not much difference between using the layer 10 features with an MLP than the fine-tuning experiment earlier in the chapter, where we froze the pretrained CNN weights through layer 10 and allowed the remaining VGG16 CNN layers and the dense layers added on top to adjust

themselves while training. Recall that the model's accuracy was around 82 percent.

Run your eyes down the columns of Table 14-2, and you'll see that my claim from the previous section is justified for all model types. In all cases, $M_{\text{raw}} < M_{\text{VGG16}} < M_{\text{layer 10}}$ holds.

Intrigued by the MLP results, I ran *features_mlp.py*, which uses all 50,000 layer 10 features for training and all 10,000 for testing, 10 times. The resulting mean accuracy was 0.8154 ± 0.0011 , essentially identical to what we found when fine-tuning: yet another demonstration of the utility of pretrained models and the features they generate.

The CIFAR-10 features aren't restricted to the mere training of classifiers; they have other uses. Let's explore two of them: anomaly detection and image retrieval.

Detecting Anomalies with CIFAR-10 Features

An *anomaly* is something that doesn't belong or is outside the norm. For many systems, detecting anomalies is critical. For example, is this credit card transaction anomalous for this person? Does this bottle on the assembly line have a crack in it? Did the camera trap capture a picture of an ivory-billed woodpecker?

We have CIFAR-10 features, meaning we have 10 classes represented. To detect anomalies with the CIFAR-10 features, we'll first call all animals except frogs "normal." Then we'll build a one-class support vector machine, a classic but limited approach to anomaly detection, and feed it all the animal features as normal examples. Finally, we'll pass frog and airplane features through the one-class SVM to see how many it flags as anomalous. We'll do this twice, first for the raw features and then again for the layer 10 features.

A one-class SVM attempts to learn the boundaries (support vectors) representing normalcy. Fortunately for us, sklearn includes a one-class SVM class, `OneClassSVM`. The `predict` method returns +1 if the input isn't deemed anomalous and -1 if it is. You'll find all the code in *features_anomaly.py*, with the relevant bits in Listing 14-7.

```
from sklearn.svm import OneClassSVM

def KeepAnimals(x, y):
    animal, frog, airplane = [], [], []
    for i in range(len(y)):
        if (y[i] in [2, 3, 4, 5, 7]):
            animal.append(x[i])
        elif (y[i] == 6):
            frog.append(x[i])
        elif (y[i] == 0):
            airplane.append(x[i])
    return np.array(animal), np.array(frog), np.array(airplane)

xtrn0, xtstf0, xtsta0 = KeepAnimals(xtrn0, ytrn)
xtst0, _, _ = KeepAnimals(xtst0, ytst)

np.random.seed(1066)
idx = np.argsort(np.random.random(len(xtrn0)))
xtrn0 = xtrn0[idx][:10000]
np.random.seed()

clf0 = OneClassSVM(nu=0.03)
clf0.fit(xtrn0)

frog0 = clf0.predict(xtstf0)
airp0 = clf0.predict(xtsta0)
anim0 = clf0.predict(xtst0)

bfr0 = np.bincount(frog0+1, minlength=3)
bai0 = np.bincount(airp0+1, minlength=3)
ban0 = np.bincount(anim0+1, minlength=3)

print("Raw features:")
print("    animal anomaly   : %4d (%0.5f)" % (ban0[0], ban0[0] / ban0.sum()))
print("    frog anomaly      : %4d (%0.5f)" % (bfr0[0], bfr0[0] / bfr0.sum()))
print("    airplane anomaly: %4d (%0.5f)" % (bai0[0], bai0[0] / bai0.sum()))
```

Listing 14-7: Training and using a one-class SVM

Listing 14-7 skips dataset loading and shows code that manipulates the raw features. If you review *features_anomaly.py*, you'll notice that code is repeated for the layer 10 features. The `KeepAnimals` function takes a dataset of features and labels and returns a collection of animals, frogs, and airplanes as NumPy arrays. One-class SVMs are unsupervised in the sense that the training data is assumed to be free of anomalies, so labels aren't required.

Training the model (`clf0`) follows sklearn's standard approach, but notice that `fit` accepts only the feature vectors. The `nu` parameter controls the SVM's sensitivity to outliers. Setting it low, like 0.03, means that the model will be very sensitive to outliers, though, as you'll learn, this doesn't necessarily mean it will capture anomalies well.

The remainder of the code in Listing 14-7 gathers the predictions for animals, frogs, and airplanes, then reports the number deemed anomalous (predict -1) as a count and fraction of the total number of samples. Running the code produces the following:

Raw features:

```
animal anomaly : 178 (0.03560)
frog anomaly   :  90 (0.01800)
airplane anomaly: 321 (0.06420)
```

Layer 10 features:

```
animal anomaly : 179 (0.03580)
frog anomaly   : 139 (0.02780)
airplane anomaly: 332 (0.06640)
```

Let's begin with the raw features: the images themselves. The SVM declared 3.6 percent of the animal test samples to be anomalies. Ideally, this number would be 0. Frogs, which we consider anomalies (poor things), were so labeled a mere 1.8 percent of the time. This means that 98.2 percent of the frogs were *not* flagged as anomalies. Airplanes were slightly better, with 6.4 percent of anomalies flagged. Overall, however, the one-class SVM trained on raw features was far from sterling.

The SVM trained on layer 10 features is similarly pathetic but slightly less so, especially when detecting frogs as anomalies. Both

models were essentially the same with animals and airplanes, but the layer 10 model was noticeably better with frogs. Granted, the model was still terrible, but we might begin to think that there is something in the layer 10 features that captures enough “frog-ness” to let the model perform slightly better.

Advanced anomaly detection uses deep learning, but implementation is beyond our scope. However, this simple experiment indicates that the CIFAR-10 features have some use in this area.

We have one example remaining: image retrieval. Let’s see if we can use the CIFAR-10 features to locate images similar to a given target image.

Retrieving Images with CIFAR-10 Features

Image retrieval aims to locate images matching a given target image. For example, if we have an image of person wearing glasses, an image-retrieval system should return a collection of pictures of people wearing glasses. To do this effectively, we need a way of comparing images to measure their similarity in terms of content. Deep learning provides the key.

Intuitively, we shouldn’t wonder that deep learning has simplified image retrieval. Convolutional features capture essential information about the input images, so we should be able to use them to locate similar images, where *similar* has more to do with the content of the image than simple appearance features like colors and textures.

All we need to do is find a metric that meaningfully compares our feature vectors representing CIFAR-10 images. The natural metric to consider is the Euclidean distance we first encountered in Chapter 3. If two feature vectors are near each other in feature space, it stands to reason that the vectors might encode similar information.

Euclidean distance measures how far apart two feature vectors are in feature space. Recall, an n -dimensional vector is a point in an n -dimensional space. However, the absolute distance between two feature vectors isn’t always what we want. Instead, we want to know that two feature vectors are pointing in roughly the same direction. To interpret

a feature vector as a vector, imagine an arrow beginning at the origin and ending at the n -dimensional point. If the angle between two such vectors is small, the vectors are pointing in roughly the same direction, implying that the cosine of the angle will be close to 1 since $\cos(0) = 1$.

The cosine of the angle between two vectors, $\mathbf{a}$ and $\mathbf{b}$, is shown here:

$$\cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|}$$

Here $\mathbf{a} \cdot \mathbf{b}$ is the dot product, and $\|\mathbf{a}\|$ and $\|\mathbf{b}\|$ are the vector magnitudes.

Metrics are often such that the closer the metric is to 0, the more similar the vectors. The Euclidean distance works this way. To use the cosine of the angle as a metric, we must subtract the cosine from 1, that is, $1 - \cos(\theta)$. We now have two metrics at our disposal. The cosine metric is interpretable. A value of 0 means the vectors point in the same direction, 1 means they are orthogonal (essentially unrelated), and 2 means they point in opposite directions.

Similar Images

The most straightforward approach to locating images similar to a given target image is to scan through all available images by using the selected metric, Euclidean or cosine, and keep the smallest n metric values as the n most similar images.

The file *features_image_retrieval.py* picks n random images of a given class from the CIFAR-10 test set and uses the given metric to locate the m closest matches in the training set. To run the code, supply the necessary arguments:

```
> python3 features_image_retrieval.py
```

```
features_image_retrieval vgg16|layer10 <classname> <ntrain> <metric>
<outdir>
```

```
vgg16|layer10 - feature source
```

```
<classname> - airplane, automobile, bird, cat, deer, dog, frog, horse, ship,
```

truck

<n _{test} >	- number of randomly selected test images to use
<n _{train} >	- number of training set similar images per test image
<metric>	- euclidean cosine
<outdir>	- output directory (overwritten)

Many of the command line arguments are familiar. First, indicate the feature source, either the output of pretrained VGG16 (*vgg16*) or layer 10. Next, choose a class name. The code picks <n_{test}> target images at random from the given class. It then locates the <n_{train}> images with the smallest metric value.

Let's run some examples and contemplate the results before examining the code. The shell script, *go_features_image*, does everything for us. It uses *features_image_retrieval.py* to build montage images for each class, selecting 5 test images and retrieving 10. First, target images are selected and matches located, then

features_image_retrieval_montage.py bundles them together for all metric and feature source combinations. Finally, a single montage image is produced for each class. Run everything with this command:

```
> sh go_features_image
```

All output is in the *results/image* directory. Figure 14-6 presents the results for frog images.



Figure 14-6: Frog image-retrieval results: Euclidean and cosine for VGG16 features (top) and Euclidean and cosine for layer 10 features (bottom)

The top row uses VGG16 features with the Euclidean metric on the left and cosine on the right. The bottom row does the same with layer 10 features. The same 5 target images are selected in each case (the images in the leftmost column of the four examples) along with the 10 best matches using that combination of features and metric. The closer the retrieved image is to the target, the more similar it is according to the metric. Note that the images are best viewed in color and are available on the book's GitHub site.

Figure 14-7 retrieves airplane images.



Figure 14-7: Airplane image-retrieval results: Euclidean and cosine for VGG16 features (top) and Euclidean and cosine for layer 10 features (bottom)

Let's begin with the Figure 14-6 frog images. We have four results. Subjectively, it seems that the layer 10 features (bottom row) selected more-relevant images than the VGG16 features (top row). Moreover, the selected layer 10 images are more faithful to the target in terms of pose. This interpretation matches what we saw with the experiments earlier in the chapter: The layer 10 features are more useful than the VGG16 features.

Now consider the difference between the Euclidean metric (left) and the cosine metric (right). Again, subjectively, I judge the cosine results to be more similar to the target than the Euclidean results. Putting both observations together, we might prefer the results presented on the lower right of Figure 14-6, where layer 10 features were combined with the cosine metric. Figure 14-7 follows the same overall pattern: The layer 10 features have captured more about the structure of the images,

and the cosine metric has used that information to select better matches.

Let's review the essential parts of the code in *features_image_retrieval.py*, as in Listing 14-8.

```
features_image_retrieval.py
def cosine(a, b):
    num = np.dot(a, b)
    den = np.sqrt(np.dot(a, a)) * np.sqrt(np.dot(b, b))
    return 1.0 - num/den

def euclidean(a, b):
    return np.sqrt((a-b)**2).sum())

idx = np.where(ytest == target)[0]
np.random.seed(359)
k = np.argsort(np.random.random(len(idx)))
np.random.seed()
idx = idx[k][:ntest]
tst_img = xtst_img[idx] # selected images of target class
tst_emb = xtst_emb[idx] # corresponding embeddings

results = [None]*ntest
classes = [None]*ntest
for i in range(ntest):
    dist = []
    for j in range(len(ytrain)):
        dist.append(metric(tst_emb[i], xtrn_emb[j]))
    idx = np.argsort(dist)[:ntrain]
    results[i] = xtrn_img[idx]
    classes[i] = ytrain[idx]

os.system("rm -rf %s; mkdir %s" % (outdir, outdir))
for i in range(ntest):
    im = Image.fromarray(tst_img[i])
    im.save(outdir+"/image_%d.png" % i)
    for j in range(ntrain):
        im = Image.fromarray(results[i][j])
```



```
im.save(outdir+"/image_%d_%d.png" % (i, j))
np.save(outdir+"/classes.npy", np.array(classes))
```

Listing 14-8: The image-retrieval code

The `cosine` and `euclidean` functions implement the metrics. They are direct translations of the mathematical expressions.

The second code paragraph selects the desired number of target images. Notice that the seed is fixed; remove it to choose other images for each class. Note also that I've skipped the code to load the raw images and embedded feature vectors.

The third code paragraph uses the metric, assigned to the variable `metric` based on the following command line, to locate the `ntrain` closest training images (`np.argsort`):

```
metric = cosine if (sys.argv[5].lower()=='cosine') else euclidean
```

Recall that Python functions are first-class objects and may be assigned to variables.

The final code paragraph dumps the selected target and matching images to the output directory. The actual class of the selected match image is stored in `classes.npy`. Use this file to measure the purity of the matching process to calculate a score for each run. If you do, you'll find that it matches the intuition gleaned from Figures 14-6 and 14-7: Layer 10 features with cosine distance produce matches most likely from the target image class.

Dissimilar Images

What happens if instead of the most similar images, we select the least similar images that have the largest metric value with the target image? A simple modification to the code is all we need. Change the following

```
idx = np.argsort(dist)[:ntrain]
```

to this:

```
idx = np.argsort(dist)[::-1][:ntrain]
```

Then rerun `go_features_image`. Figure 14-8 shows the results for frogs.

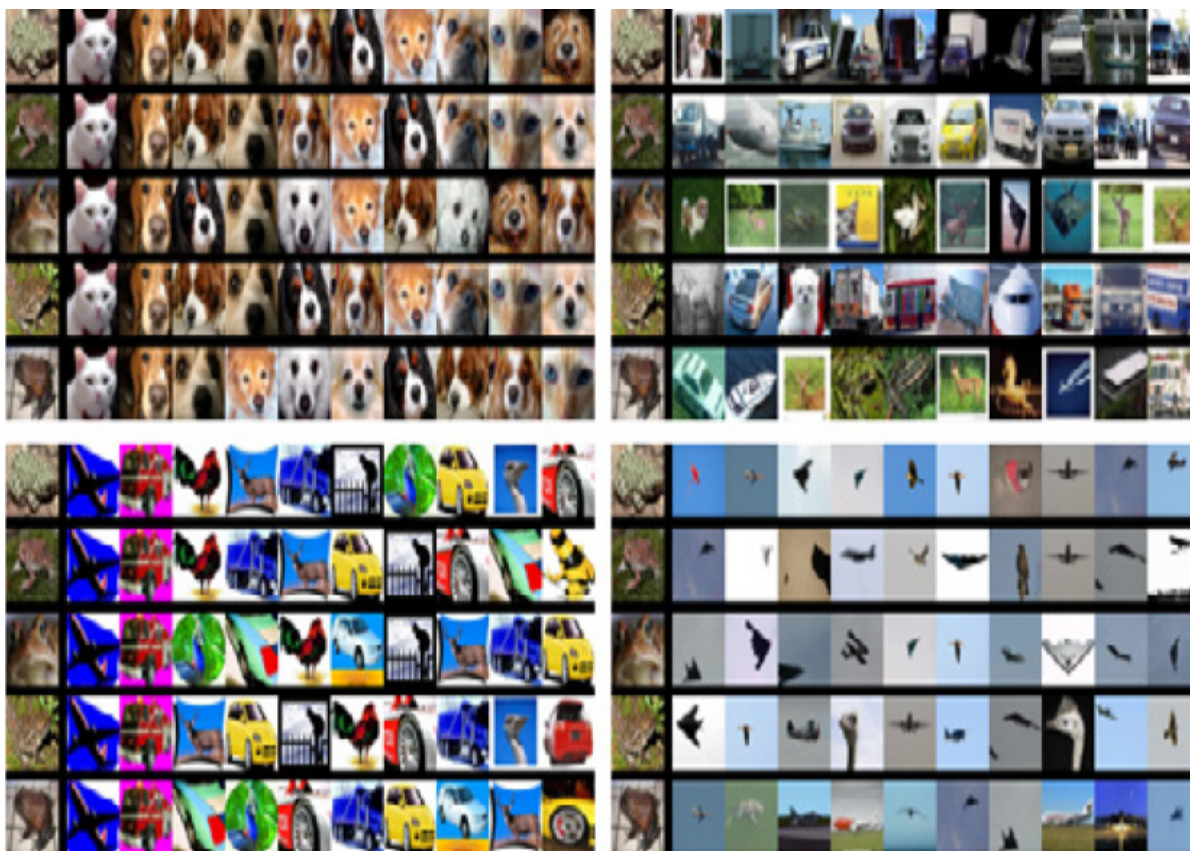


Figure 14-8: The least frog-like images: Euclidean and cosine for VGG16 features (top) and Euclidean and cosine for layer 10 features (bottom)

Figure 14-9 shows the results for airplanes.

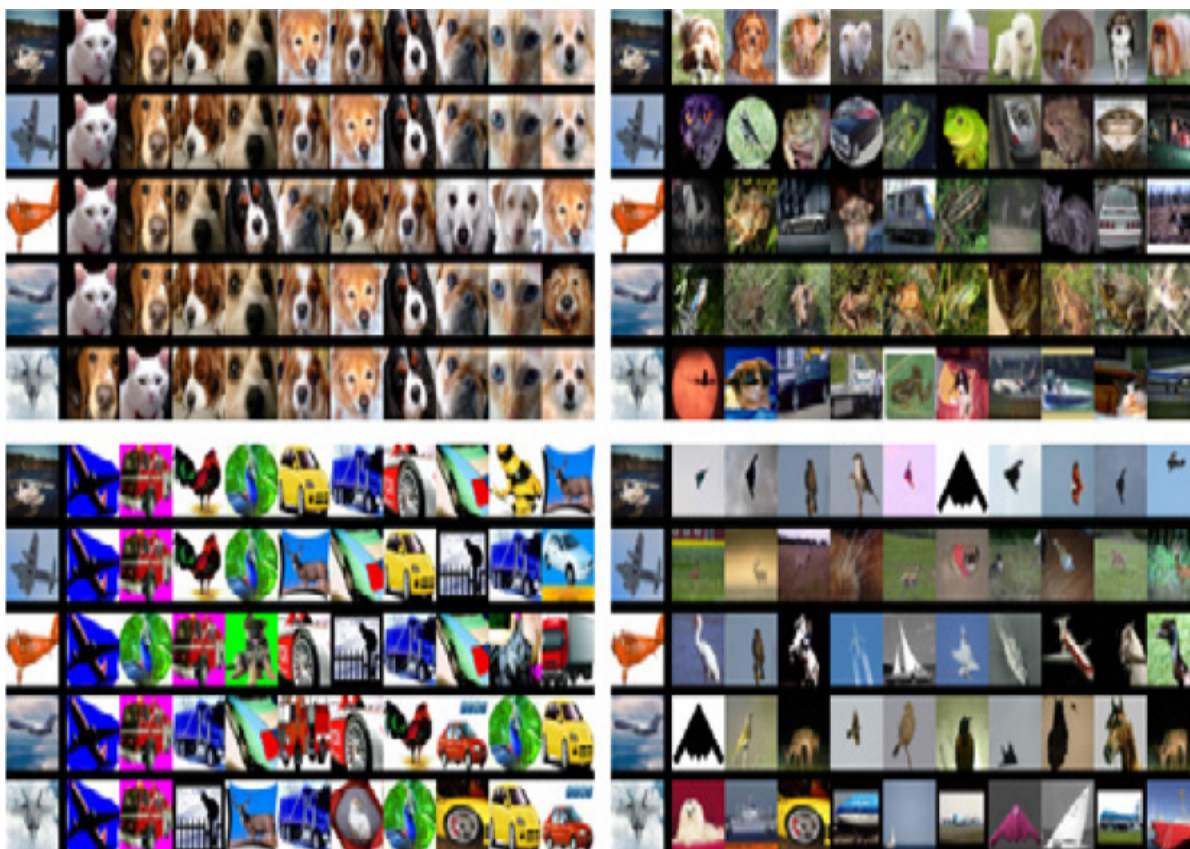


Figure 14-9: The least airplane-like images: Euclidean and cosine for VGG16 features (top) and Euclidean and cosine for layer 10 features (bottom)

The selected images are unlike the target image, as intended. First, notice that the Euclidean metric, the leftmost column of both figures, has selected roughly the same set of images for both the VGG16 and layer 10 feature sets regardless of the target image class. For the VGG16 features, the chosen images are dog and cat faces. This implies that in the 512-dimensional space of the VGG16 features, the points farthest from frog and airplane are dog and cat faces. I think this is merely a coincidence?

Consider now the cosine metric images, the rightmost columns, beginning with the frogs. The images selected by the VGG16 features have no apparent relationship to one another beyond a certain overall feel, like shades of green or a white border. I read this as an expression of what the VGG16 features have captured about the CIFAR-10 images—texture, color, and edges—all low-level features passed through the higher-level CNN layers conditioned on ImageNet classes.

Now look at the images in the lower-right of Figure 14-8. The cosine metric also selected these images when using layer 10 features. The selected images are almost all birds or airplanes that cover a tiny fraction of the image. The feature vectors representing these images are, because the cosine metric was used, pointing in nearly the opposite direction in the 4,096-dimensional space of layer 10 features than the target frog images.

I read this in the following way: The organization of the feature space, the concepts represented by it, is primarily directional. The direction a feature vector points is more important than the length of that vector. The Euclidean metric implicitly pays attention to the length, while the cosine metric is all about direction regardless of the length (scaling) of the vector. Many of these comments also apply to the airplanes of Figure 14-9.

Summary

Fine-tuning and transfer learning are the themes of this chapter. Fine-tuning takes a model trained on one dataset and adapts it to another, similar dataset. Transfer learning takes the output of a pretrained model as a new representation of the input and uses that embedding for another purpose.

We experimented with both approaches by using VGG16 pretrained on ImageNet and adapted to the CIFAR-10 dataset. You learned that fine-tuning involves selecting which parts of the pretrained model to leave alone and which to change. You also learned that the new representations extracted from a pretrained model have utility in training nonneural network models, anomaly detection with a one-class SVM, and image retrieval.

Up to now, we've focused on classification—for example, labeling an image as a picture of a dog or a cat. The following chapter takes the next step, moving from classification to detection and, finally, semantic segmentation.

15

FROM CLASSIFICATION TO LOCALIZATION



While classification says, “There’s a dog in this image,” detection adds, “And here’s where you’ll find it.” Semantic segmentation takes detection to the next level and assigns a class label to every pixel. Finally, multilabel classification states, “There’s a dog, a cat, and a parrot somewhere in this image.”

With the exception of the fully convolutional networks of Chapter 10, all the experiments we’ve conducted to this point involve only classification. Fully convolutional networks locate targets in the image, but imprecisely. The output of a fully convolutional model is a heatmap marking places where the object is likely located. The heatmap must be further processed to pinpoint the object’s actual location and extent.

Detection locates objects in images, assigning a class label and a bounding box surrounding the object. With the ubiquity of deep learning in our present culture, you’ve likely already seen such detection outputs.

Semantic segmentation creates an image that’s the same size as the input image, and every pixel is assigned to a class. Such an output

segments the image into distinct objects, assuming no overlap, making it possible not only to know that there's a dog in the image, or that the dog is within this rectangle, but also to extract every pixel associated with the dog.

We begin with detection and then proceed to semantic segmentation. The chapter's models are small but illustrative, keeping with our theme of CPU-only examples.

The chapter closes with an experiment in multilabel classification, an approach that identifies the presence of multiple class instances within an input image without localization. It's a bit of a stand-alone topic but follows naturally from our explorations of bounding boxes also in this chapter.

Detection Experiments with MNIST

Detection models classify and localize at the same time. You're familiar with the classification part; it's the normal CNN architecture we've used throughout the book. Localization implies, in this case, *bounding boxes*, rectangles that enclose the detected object. Detection implies learning a function that produces the coordinates of the bounding box as a model output along with the softmax we expect to assign a class label.

Producing a continuous output value implies regression; therefore, we intend to learn a regression model that accepts the input image and produces bounding-box coordinates. Ultimately, we want a single model that both classifies and localizes via regression.

To do that, we need a new kind of architecture with *two* outputs, or *heads*. The first, the *classification head*, classifies and produces a softmax vector. The *regression head* regresses to create a four-element bounding-box vector, (x, y, h, w) , for the top-left corner, height, and width, all in normalized coordinates. (Recall, images are indexed from the top-left corner.)

In a CNN, the convolutional layers learn new feature representations. We'll use these representations to help each model

head learn what we want it to learn. Conceptually, the model we want to build looks like Figure 15-1.

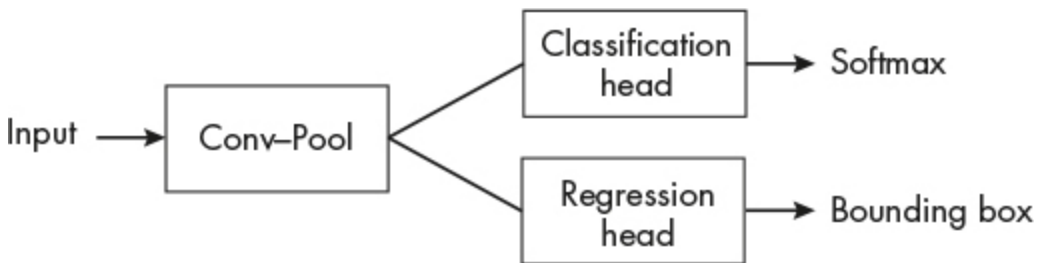


Figure 15-1: A block diagram of the dual-headed detection model

The model’s input, convolution, and pooling layers on the left are familiar and, collectively, increasingly referred to in the literature as the model’s *backbone*. Their job is to learn a new representation of the input image that’s useful to both heads simultaneously. The first head learns the class labels, while the second learns to produce bounding boxes. Both heads consist of standard dense layers. The classification head uses a softmax output. The regression head uses a sigmoid activation function to map its outputs to $[0, 1]$, representing the normalized coordinates of the bounding box.

We propose to work with MNIST digits, so we know the classification head requires 10 nodes, one for each possible class. Since the regression head is to learn bounding boxes, a vector of four elements, it uses a dense layer with four nodes.

If both heads are to produce values in the range $[0, 1]$ —the first a “probability” of class membership and the second a normalized coordinate value—why aren’t both heads using softmax outputs? The reason has to do with the heads’ different goals. The 10 outputs from the classification head’s dense layer are interrelated in that we want the softmax to map those values to $[0, 1]$ so we can meaningfully compare them to select the most likely class label—that is, treat them as a kind of probability distribution. The four outputs from the regression head are independent; each represents a coordinate (or length) in the normalized units of the input image. Therefore, we want them constrained to $[0, 1]$, which the sigmoid does, but we do not want to alter their values relative to one another.

Here's our plan of attack:

1. Use the existing MNIST dataset to construct two new datasets with a single 28×28-pixel digit somewhere within a 128×128 image. The first dataset will use a blank background, while the second will use a background image randomly extracted from a satellite dataset.
2. Construct a dual-head model following the form in Figure 15-1.
3. Train the model on each dataset and evaluate its performance. This requires a new kind of metric suitable for quantifying detection accuracy.

Let's begin.

Building the Dataset

Learning a bounding box for a small dataset such as MNIST (which consists of 28×28-pixel grayscale images with a single handwritten digit centered in the image) is illogical, but what if that 28×28-pixel image was embedded somewhere within a larger image—say, one that is 128×128 pixels? Building such a dataset is the goal of this section.

The algorithm is straightforward: Load the MNIST digits, randomly place them within a larger, blank 128×128-pixel image, and save the resulting images to disk. There are 60,000 training images, but we'll use only the first 6,000. We'll keep all 10,000 test images.

The only twist to the algorithm is adding a randomly selected background image from the satellite dataset. You'll find the UC Merced Land Use dataset at

<http://weegee.vision.ucmerced.edu/datasets/landuse.shtml>. This machine-learning classic contains 21 classes of land cover extracted from satellite data. For us, it's a convenient source of background imagery.

Download the dataset and expand it to the *UCMerced_LandUse* directory within the *data* directory that already houses our other datasets. The imagery is in the public domain, so we're free to use it for any purpose, including selecting random 128×128-pixel grayscale patches from the 256×256 RGB originals.

Review the file *build_bbox_dataset.py*, which contains all the code for this section. First, the code locates all land-use images (*fnames*), then loads the MNIST train and test datasets, keeping the first 6,000 training images.

Next, blank 128×128-pixel images (NumPy arrays) are created, and randomly selected MNIST images are placed within, allowing a buffer around the edge of the image. This creates *xtrain*. The selected digit label is a one-hot vector in *ytrainc*, and the bounding box surrounding the digit is in *ytrainb* as a four-element vector. Notice that we now have two sets of labels, the expected class label and the new bounding-box label. The model uses both during training.

This process repeats for all 10,000 test images. Finally, both train and test images, *xtrain* and *xtest*, respectively, update with randomly selected 128×128 land-use image patches merged with the nonzero pixels of the original image; see *SelectRandomPatch*. The satellite image intensity is divided by 3 to give our model a fighting chance of noticing the digit against the background. When finished, the updated images are stored on disk.

The final result is a collection of four files each for train and test: two image files for the two background conditions, a set of one-hot vector class labels, and a set of associated bounding boxes.

Building the Model

Figure 15-1 is our guide for building the model in code. The file we want is *bbox.py*, and it follows the general format of the *vgg8.py* and *resnet18.py* files from Chapter 13.

Extending Figure 15-1 one level deeper in detail gives us Figure 15-2.

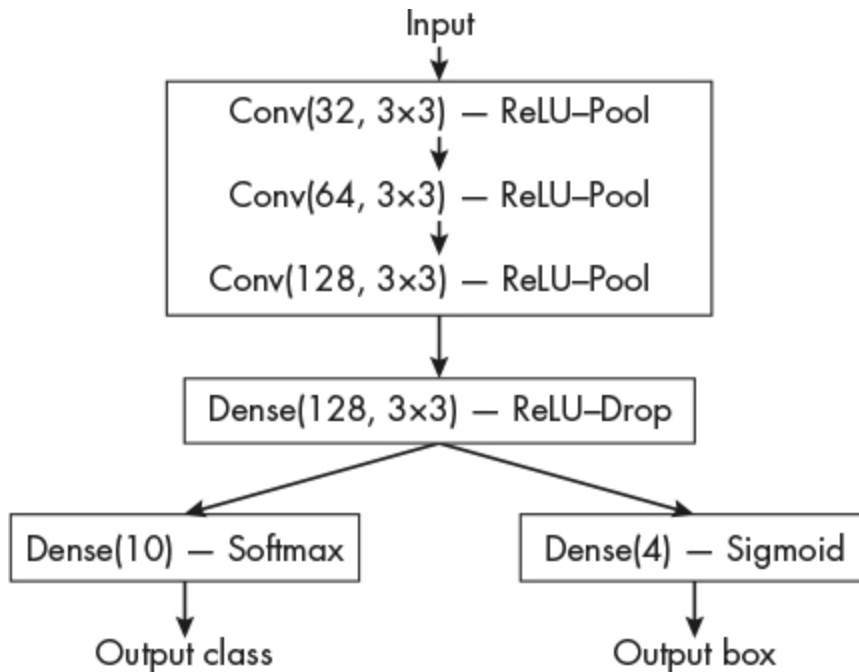


Figure 15-2: The detection model architecture

Structurally, the model is quite simple, as we might expect when dealing with MNIST digits. The first block learns a new feature representation, which is passed to the first dense layer. The output of the first dense layer is then passed to each head to learn how to extract class likelihood (left) and bounding-box coordinates (right).

Listing 15-1 shows that Figure 15-2 is directly implemented in *bbox.py*.

```

inp = Input(shape=(128, 128, 1))
_ = Conv2D( 32, (3, 3))(inp); _ = ReLU(_)(_); _ = MaxPooling2D((2, 2))(_)
_ = Conv2D( 64, (3, 3))(_); _ = ReLU(_)(_); _ = MaxPooling2D((2, 2))(_)
_ = Conv2D(128, (3, 3))(_); _ = ReLU(_)(_); _ = MaxPooling2D((2, 2))(_)
_ = Flatten(_)(_)
_ = Dense(128)(_)
_ = ReLU(_)(_)
_ = Dropout(0.5)(_)
cout = Dense(10, activation='softmax', name='cout')(_)
bout = Dense(4, activation='sigmoid', name='bout')(_)
model = Model(inputs=inp, outputs=[cout, bout])

```

Listing 15-1: A dual-head model to detect MNIST digits

The input layer is passed through the first three convolutional layers, each with a ReLU activation and max pooling. So far, this is nothing new; we've used similar feature-learning layers in previous chapters. The following three lines are likewise familiar: Flatten the 3D output of the last maxpooling layer, pass it through the 128-node dense layer, and apply ReLU activation and dropout.

The final two lines in Listing 15-1 implement the heads. Both `cout` and `bout` receive the output of the dropout layer. The `cout` layer applies a softmax to the 10 outputs to indicate the likely input class. The `bout` layer applies a sigmoid to each of its four outputs, the bounding-box upper-left corner, height, and width. The final `model` has a single input (`inp`) and has both heads as outputs.

Notice that `cout` and `bout` are named. This model has two outputs, and, as you'll learn momentarily, two loss functions. Listing 15-2 shows how to name the layers so we can reference them properly when compiling and training the model.

```
model.compile(optimizer='Adam',
              loss={
                  'cout': 'categorical_crossentropy',
                  'bout': 'mean_squared_error'},
              metrics={'cout': 'accuracy'})

model.fit(xtrain, {'cout': ytrainc, 'bout': ytrainb},
        batch_size=batch_size,
        epochs=epochs,
        validation_data=(xtest[:1000],
                        {'cout': ytestc[:1000], 'bout': ytestb[:1000]}))
```

Listing 15-2: Compiling and training the model

The `compile` call specifies Adam as the optimizer and uses a dictionary with two elements to specify the loss functions, one for each head. The classification head uses categorical cross-entropy as expected, while the regression head uses the mean squared error to measure the deviation between the true bounding box and the network's output. For a single

input sample, then, where i runs over classes or bounding-box elements, we have the following equations:

$$\mathcal{L}_{\text{cross}} = - \sum_i^{10} \hat{y}_i \log y_i \quad \text{and} \quad \mathcal{L}_{\text{MSE}} = \frac{1}{4} \sum_i^4 (y - \hat{y})^2$$

For $\mathcal{L}_{\text{cross}}$, the true class is y , a 10-element one-hot encoded vector with softmax prediction $\hat{y}$. For $\mathcal{L}_{\text{MSE}}$, y is a 4-element vector, a bounding box, with $\hat{y}$ the prediction. The total model loss is the sum of $\mathcal{L}_{\text{cross}}$ and $\mathcal{L}_{\text{MSE}}$.

The `metrics` keyword tells Keras to output the classification accuracy during training. The MSE loss is automatically output along with the classification loss.

The `fit` call trains the model by using input images (`xtrain`) and two sets of associated labels: `ytrainc`, each row of which is a 10-element vector, and `ytrainb`, where each row is a bounding box. We'll indicate the minibatch size and training epochs on the command line as previously. Finally, we'll use the first 1,000 test samples as validation data during training. We do not intend to alter training based on the accuracy of the validation set, so we can be a bit sloppy and use all 10,000 samples when interpreting the model's performance.

The remainder of the code in *bbox.py* gathers the predictions and writes them to disk along with a confusion matrix for the classifier head. The only difference here compared to previous models is that `predict` returns a list of predictions, one set for each head:

```
pred = model.predict(xtest, verbose=0)
```

Here `pred[0]` is a 10,000×10 NumPy array of softmax predictions over the test set, and `pred[1]` is a 10,000×4 array of associated bounding-box predictions.

Let's give *bbox.py* a try.

Running the Model

The file *bbox.py* expects only a few arguments on the command line:

```
> python3 bbox.py
```

```
bbox <batch> <epochs> blank|land <outdir>
```

```
<batch>    - minibatch size (e.g. 32)
<epochs>   - epochs (e.g. 10)
blank|land - image source
<outdir>   - output directory (overwritten)
```

The argument list should be familiar to you except for the image source. If `blank`, it uses the digits dataset with a black background; otherwise, it uses the land-use background dataset. For example, the following trains the model for 60 epochs:

```
> python3 bbox.py 32 60 blank results/mnist_bbox_32_60_blank
```

In this case, this produces the following:

Test set accuracy: 0.9288, MCC: 0.9210

```
[ [ 888  16   9   1   0   4  49   2  11   0]
  [   1 1122   5   0   1   1   2   0   1   2]
  [   7   1 972  15   2   8   8   7   9   3]
  [   3   0  10 942   0  33   0  13   7   2]
  [   1  22   1   0 905   0  13  10   1  29]
  [   5   2   4   7   3 850  14   1   5   1]
  [   2   9   2   0   3   5 937   0   0   0]
  [   4   4  22  20   9   1   0 950   0  18]
  [  10   1   4   9   4  13  51   0 876   6]
  [  10  12   7  19  28   9   2  52  24 846]]
```

The trained model is 93 percent accurate on the test set, which isn't too bad given we have only 6,000 training images. In these images, the digit is somewhere within a larger image, not centered in a small image as is the case with standard MNIST.

A training run with the land-use background produced the following:

Test set accuracy: 0.9407, MCC: 0.9341

```
[ [ 920    5    8    0    0    9   25    2    9    2]
  [   4 1115    3    0    2    2    1    5    2    1]
  [   6    0  957   22    2    9    3   14   13    6]
  [   0    0    8  953    2   16    0   10   18    3]
  [   2    4    1    0  931    0    6    2    2   34]
  [   3    4    7   17    2  839    8    1   10    1]
  [   9    7    0    0    5   12  919    0    6    0]
  [   0    1   14   17    5    0    0  970    0   21]
  [   4    0    2    7    7   15   23    0  885   31]
  [   8    3    5    6   20    4    1   25   19  918]]
```

This model learned better than the blank background model, but we're not fooled; we know that stochastic initialization plays a role, and that we should train multiple models and average the results to convince ourselves that the model is better with one input dataset than the other. However, this time, I leave that thoroughness to you.

Did the model learn the bounding boxes? To answer that question, we need to work a little harder than printing simple confusion matrices. We need the files dumped by *bbox.py* in the output directory:

```
accuracy_mcc.txt
bounding_box.npy
confusion_matrix.npy
model.keras
softmax.npy
```

We also need the code in *bbox_results.py* and a new metric: the intersection over union (IoU).

Building the Intersection-over-Union Metric

Set theory's most fundamental binary operations are union and intersection. Consider two sets:

$$A = \{1, 2, 3, 4\}$$

$$B = \{3, 4, 5, 6\}$$

Both sets have four elements, some of which are shared between them. The *union* of two sets is the full collection of the elements in both, while the shared elements form the *intersection*:

$$A \cup B = \{1, 2, 3, 4, 5, 6\}$$

$$A \cap B = \{3, 4\}$$

Here $\cup$ means union, and $\cap$ means intersection. Recall that if two sets have no elements in common, the intersection is the empty set, $\emptyset$ or $\{\}$.

Let's use intersection and union to build an IoU metric. The model produced bounding-box estimates. For example, here are the true bounding-box coordinates and those estimated by the model for the first blank background test sample, with the true box appearing first:

```
>>> y[0]
array([0.3828125, 0.4609375, 0.21875 , 0.21875 ], dtype=float32)
>>> yp[0]
array([0.434071 , 0.46423987, 0.2185496 , 0.2202593 ], dtype=float32)
```

The vectors represent the upper-left corner of the box, (*row, column*), followed by the height and width, all in normalized coordinates. But is this a good fit? How might we quantify it? This is where IoU enters the picture.

We're working with rectangular bounding boxes, meaning we can derive an algorithm to mathematically determine the intersection and union. Later in the chapter, we won't have this luxury. Figure 15-3 shows the relationship between two overlapping bounding boxes.

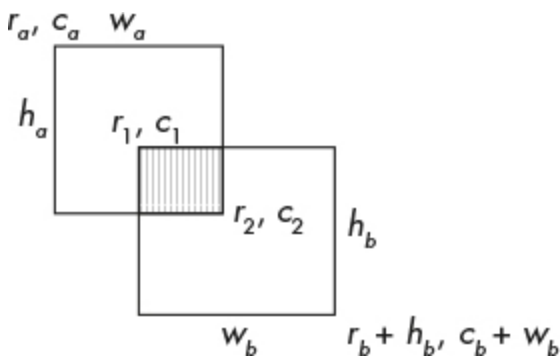


Figure 15-3: Determining the IoU for two bounding boxes

The first bounding box is anchored at (r_a, c_a) . The second has lower-right-corner coordinates $(r_b + h_b, c_b + w_b)$. The overlap between them has upper-left and lower-right coordinates (r_1, c_1) and (r_2, c_2) . We must determine these. Some inspection and contemplation tell us the following:

$$\begin{aligned}(r_1, c_1) &= (\max(r_a, r_b), \max(c_a, c_b)) \\ (r_2, c_2) &= (\min(r_a + h_a, r_b + h_b), \min(c_a + w_a, c_b + w_b))\end{aligned}$$

This implies that the area of the intersection between the two bounding boxes is as follows:

$$I_{\text{area}} = \max(0, r_2 - r_1) \times \max(0, c_2 - c_1)$$

Here the max function allows for no overlap, which happens when one or both of the lengths are negative. If the bounding boxes have no overlap, the IoU is 0.

To find the union of the two bounding boxes, sum their respective areas and subtract the area of the intersection to avoid double-counting the overlap region:

$$U_{\text{area}} = h_a(w_a) + h_b(w_b) - I_{\text{area}}$$

This gives us the IoU:

$$\text{IoU} = \frac{I_{\text{area}}}{U_{\text{area}}}$$

The IoU ranges from 0 (no overlap) to 1 (perfect overlap). We'll report the per-class mean IoU over the test set and the overall mean to give us a single-number representation of how well the model learned bounding boxes. This score, along with the overall accuracy at digit classification, informs us about the model's ability to detect digits.

The file *bbox_results.py* contains code to calculate the per-class and mean IoU. We'll run the code in the next section, but it's worth considering the implementation first. Listing 15-3 illustrates the `iou` function.

```

def IoU(soft, bbox, yc, yb, correctOnly=False):
    def CalcIoU(a, b):
        ra, ca, ha, wa = a
        rb, cb, hb, wb = b
        r1, c1 = max(ra, rb), max(ca, cb)
        r2, c2 = min(ra+ha, rb+hb), min(ca+wa, cb+wb)
        i = max(0, r2-r1) * max(0, c2-c1)
        u = ha*wa + hb*wb - i
        iou = (i/u) if (u>0) else 0
        return iou
    iou, n = np.zeros(10), np.zeros(10)
    for i in range(len(soft)):
        pc, tc = np.argmax(soft[i]), np.argmax(yc[i])
        iu = CalcIoU(bbox[i], yb[i])
        if (correctOnly):
            if (pc == tc):
                iou[tc] += iu
                n[tc] += 1
        else:
            iou[tc] += iu
            n[tc] += 1
    return iou / n

```

Listing 15-3: Calculating IoU over the test set

The `IoU` function accepts softmax predictions (`soft`), bounding-box predictions (`bbox`), true class labels (`yc`), and true bounding-box coordinates (`yb`). The `correctOnly` keyword decides what to calculate and return.

The body of `IoU` loops over each test sample to calculate the per-sample IoU and add that to the class total in `iou`. The per-class mean, `iou/n`, is returned.

The inner `calcIoU` function uses the previously mentioned formulas to determine the intersection (`i`) and union (`u`) for the predicted bounding box (`a`) and the true bounding box (`b`).

Review the remainder of *bbox_results.py*. It contains code to load results and manipulate output images.

Evaluating the Bounding Boxes

Reporting mean IoU along with a confusion matrix is effective, but we have images and bounding boxes, so let's use them. The file *bbox_results.py* generates images showing the classified digit, the predicted class, the true bounding box, and the predicted bounding box. Run it like so:

```
> python3 bbox_results.py results/mnist_bbox_32_60_blank blank images
```

The command line evaluates the model's performance on the blank background digits. The `images` argument generates the prediction images. The text output in this case was as follows:

IoU for all test samples:

```
0: 0.473523
1: 0.539877
2: 0.516175
3: 0.484983
4: 0.543795
5: 0.512930
6: 0.594875
7: 0.511954
8: 0.494774
9: 0.492151
mean = 0.516504
```

I'm showing only the IoU scores per class for all test samples. The results for correctly classified samples are also displayed and are slightly higher, as we might expect, but not dramatically so. A mean IoU of 52 percent isn't sterling, but it is convincing enough to believe that the model did learn an approximate function mapping digit locations to bounding boxes.

Figure 15-4 presents three image outputs showing the digit, the assigned class as a small digit in the upper-left corner, the original

bounding box, and the predicted bounding box.



Figure 15-4: Two correctly classified test samples (left, middle) and one error (right)

The left and middle examples are correctly classified and well localized. The right example is a misclassification and a correspondingly poor, though not entirely wrong, bounding box.

Running *bbox_results.py* on the land-use dataset output (replace `blank` with `land`) produces the following:

IoU for all test samples:

```
0: 0.455503
1: 0.555674
2: 0.456176
3: 0.468004
4: 0.501238
5: 0.465800
6: 0.527479
7: 0.493099
8: 0.508545
9: 0.429388
mean = 0.486091
```

The mean IoU is lower, as we might expect because we're forcing the model to consider more cluttered backgrounds.

Figure 15-5 mirrors Figure 15-4 to show two correct classifications (left, middle) and one error (right).

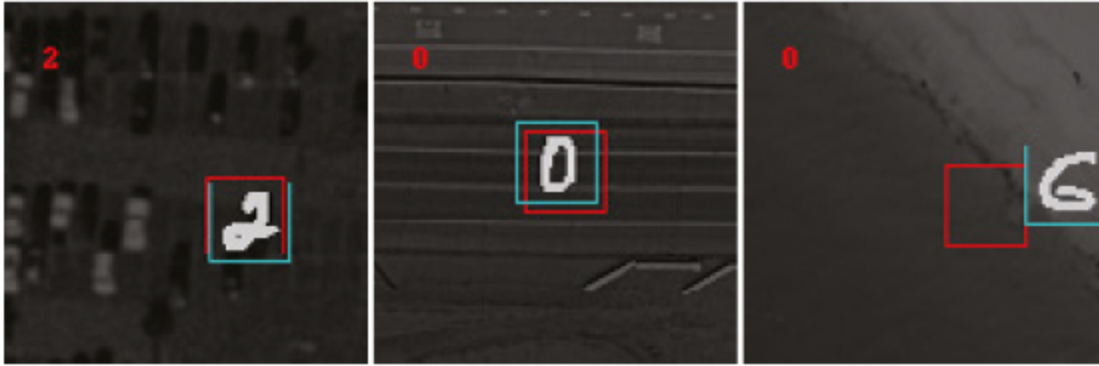


Figure 15-5: Classified land-use test samples

As with the blank background images, wrong classifications are associated with wrong bounding-box locations.

Balancing the Loss Function

Our model learned the classes and bounding boxes, but if you were watching during the training process, you might have noticed TensorFlow output status lines like this:

```
loss: 0.3479 - cout_loss: 0.3380 - bout_loss: 0.0099
```

The text shows the loss on the minibatch just processed when I manually interrupted training. The total loss is the sum of the `cout` loss (softmax loss) and the `bout` loss (bounding box loss). Training seeks to minimize the overall loss. The bounding-box loss is about two orders of magnitude less than the softmax loss. This means that the softmax loss dominates the training process, causing the model to emphasize learning proper class labels and not bounding boxes. We can fix this by altering the weights assigned by Keras to the different loss functions. To do this, we must update the `compile` call by adding the bolded line:

```
model.compile(optimizer='Adam',
              loss={'cout': 'categorical_crossentropy', 'bout':
'mean_squared_error'},
              loss_weights={'cout': 1.0, 'bout': 50.0},
              metrics={'cout': 'accuracy'})
```

The `loss_weights` keyword expects a dictionary of multipliers for each of the loss functions, allowing us to make the bounding-box loss 50 times larger relative to the class-label loss. Stopping training at roughly the same point (during epoch 5) produces the following, where the total loss is now, to within four-digit rounding, $\text{cout} + 50 \times \text{bout}$:

```
loss: 0.4791 - cout_loss: 0.3187 - bout_loss: 0.0032
```

I retrained the model by using the new loss weights, then ran *bbox_results.py*. The mean IoU for the blank background dataset increased from 51.7 percent to 73.7 percent. The land-use dataset also improved, going from 48.6 percent to 70.2 percent. The improvement in overlap between the true and predicted bounding boxes is visually evident.

Figure 15-6 presents the same three test images as in Figure 15-4 after training the model with loss weights.



Figure 15-6: Three test images after training with loss weights

The bounding boxes are tighter, and the previously wrong rightmost classification is now correct, an effect evident over the entire test set. The previous model's overall accuracy was 92.9 percent, but the loss-weight-adjusted model's accuracy has increased to 95.2 percent.

Exploring How the Model Works

Our dual-headed model learns digits and bounding boxes, but how? We shouldn't be satisfied with success in building and training; we should

also seek to understand what the model is doing, how it operates, and how it learns what it needs to learn.

We're familiar with the concept that the convolutional layers of the model are learning a new representation of the input images better suited to the needs of the model's topmost layers. Previously, the topmost layers were focused on a single task: classification. With two heads and a shared 128-node dense layer before them, the model must now learn a 128-element feature vector that encodes a new representation of the digit's identity and the digit's location in the input image.

In Figure 9-8 on page 259, we examined the final dense-layer output vector that encoded information about the MNIST digit passed to the model. Then, in Figure 9-9 on page 260, we observed common patterns in this vector for 10 instances of the digits 4 and 6. We could discern differences in the patterns by digit type, even though there were variations on the main theme. A similar effect is at work with the output of the 128-node dense layer in our dual-head model, but this time, the patterns encode information related to the digit class *and* the digit's position and size. The representations are abstract and likely tricky, if not impossible, for humans to learn, but that doesn't mean the model's dual heads can't, as the experiments of this section demonstrate.

Even though the model is learning class, position, and scaling from the training data, the model will still learn effectively if we use a dataset containing digits that are not all the same size. If you run the following three commands, you'll build a dataset that includes digits scaled anywhere from 40 percent to 160 percent of normal size, train a model using that dataset, and then evaluate its performance on the test set:

```
> python3 build_bbox_scaled_dataset.py
> python3 bbox_scaled.py 32 60 blank results/mnist_bbox_scaled_blank
> python3 bbox_scaled_results.py results/mnist_bbox_scaled_blank blank images
```

My run's mean IoU was 62.1 percent, lower than the fixed-digit-size case but still reasonable. I suspect the new degree of freedom, the scale factor, might require a larger training set. Feel free to build a dataset with more than 6,000 training images and try again. A land-use version

of the dataset was also created, achieving a mean IoU of 61.8 percent. Figure 15-7 presents three test-sample outputs.

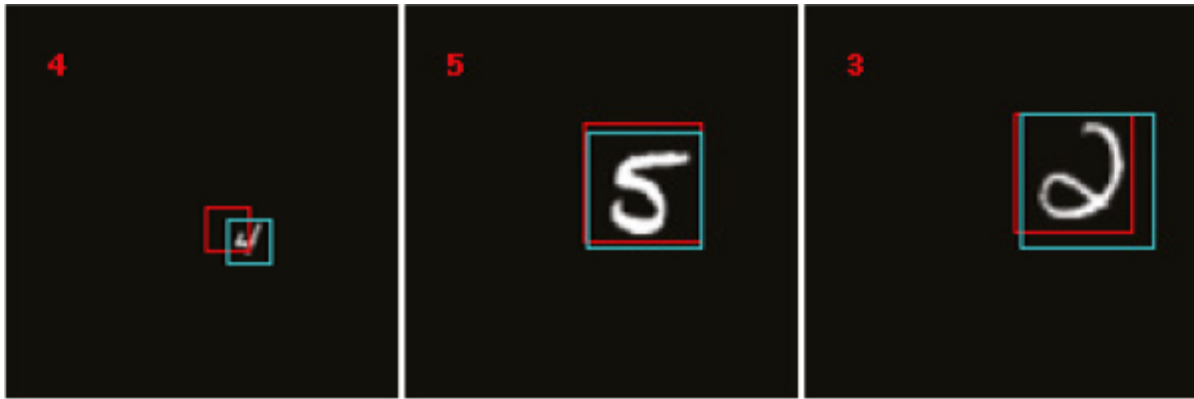


Figure 15-7: Classified scaled test samples

The left and middle samples are correctly classified, and the rightmost sample is an error.

Running an Advanced Detection Model

Our detection model is limited to a single-digit instance; advanced detection models can localize many instances of the known classes. One such advanced model is You Only Look Once (YOLO). The architecture is sophisticated but can be summarized as follows:

- YOLO is a single model that divides the input image into an $S \times S$ grid.
- Grid cells are responsible for classifying and bounding objects with centers within the cell.
- Anchor boxes, predefined and of specific aspect ratios, are available to each grid.
- YOLOv3 uses three anchor boxes per cell, implying that, at most, three objects can be detected within the cell, though the number is usually smaller.
- Suppression keeps the most likely bounding box + class probability combination for objects detected by multiple boxes within a cell.

For more details, I suggest reviewing the original YOLO paper, “You Only Look Once: Unified, Real-Time Object Detection” by Joseph Redmon et al. (2016), available at <https://arxiv.org/pdf/1506.02640.pdf>.

Training YOLO is beyond our scope, but you can run it by downloading the repository, building the executable (Linux only), and acquiring the pretrained weights:

```
> git clone https://github.com/pjreddie/darknet
> cd darknet
> make
> wget https://pjreddie.com/media/files/yolov3.weights
```

Run the executable on any RGB image you desire like so, where *dog.jpg* is an included example file:

```
> ./darknet detect cfg/yolov3.cfg yolov3.weights data/dog.jpg
```

The file *predictions.jpg* holds the predictions. A GPU is not required for inference. If the `darknet` command fails on your system, try using absolute filepaths.

I ran the detector on the images in the *data/images* directory, three examples of which are in Figure 15-8.



Figure 15-8: Three YOLOv3 predictions

The model knows 80 classes, but stroller and trilobite are not among them. The model ignores the stroller in the second image but declares the trilobite (*Arctinurus boltoni*) to be a zebra, a reasonable mistake given the trilobite’s “striped” body.

Let’s move beyond detection with bounding boxes and explore assigning a class label to every pixel via semantic segmentation.

Semantic Segmentation with U-Net

Semantic segmentation takes detection to the limit by assigning a class label to every pixel of the input image. The most commonly used architecture for semantic segmentation goes by the name *U-net* because the architecture resembles the letter *U*: The input image is progressively reduced in size until a bottleneck layer is reached, and then a series of upscaling layers restore the image to its original dimensions. Skip connections are also introduced for the same reason as in a residual network: to provide a path for the gradient during training and to use features learned during the reduction step when implementing the upscaling step.

Supervised training requires labeled data. For classification, labels are little more than integers representing the input class. As we now appreciate, detection requires bounding boxes to specify the objects’ locations in the input images. A semantic segmentation dataset requires labels for *every* pixel in the image, though it is common to label the desired objects and mark everything else as “background.”

Therefore, we need a semantically labeled dataset to explore semantic segmentation with U-nets. Fortunately, the open source M2NIST dataset fits the bill. The original is available at <https://www.kaggle.com/datasets/farhanbubble/multimnistm2nist>. The CC-BY-SA 4.0 license allows us to use the dataset and present a version on the book’s GitHub site as a 3.2MB ZIP file. The *m2nist.zip* file is all you need, but if you decide to work with the original file, run *repackage.py* to save disk space.

The dataset name implies that it contains MNIST digits. There are 5,000 images, each 64×84 pixels and containing one to three standard

MNIST digits without overlapping. This much isn't particularly novel. Novelty comes from the labels.

Each image has an associated $64 \times 84 \times 11$ binary label array. Each of the 11 label channels is a mask with a 1 where a pixel of that class appears in the image. For example, if the image contains a two and a nine, then channel 2 and channel 9 will contain 1s where the respective digits are located in the image. All other pixels across all channels will be 0. The last channel, channel 10, is the background mask to mark pixels that do not contain part of a digit. Figure 15-9 serves as an illustration.



Figure 15-9: A sample M2NIST image (left) and the corresponding label masks for each digit and the background

Let's construct three datasets as we did for the bounding-box experiments. Run the following:

```
> python3 build_unet_dataset.py
> python3 build_ucmerced_background_dataset.py
```

The first command creates a 4,000-image training set and a 1,000-image test set with a blank background. The second command replaces the blank background with randomly selected land-use image patches to create two new datasets. The first new dataset, with *land* in the filename, sets the pixels occupied by the digits to half the digit intensity. The second new dataset, with *subtle* in the name, adds 20 to whatever intensity is already present in the background image at the locations where the digits exist.

Figure 15-10 presents three versions of a test image.



Figure 15-10: A sample M2NIST image: blank (left), land use (middle), and subtle (right)

The masks are identical across all three, as are the backgrounds of the land-use and subtle images.

Let's design a simple U-net model to classify the M2NIST datasets. Figure 15-11 depicts what we want the model to do to the spatial dimensions of each layer's tensor output. The U shape is evident by following the bold arrows.

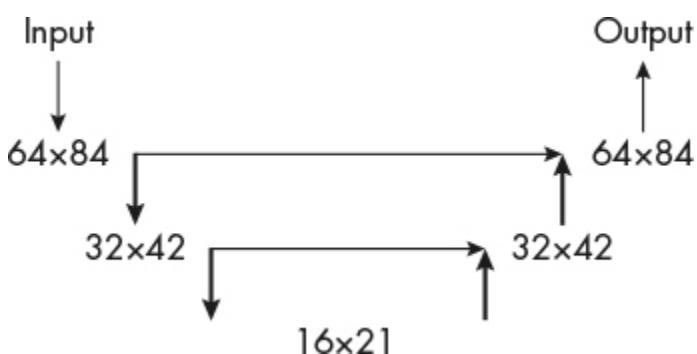


Figure 15-11: The M2NIST U-net architecture (conceptual)

Each bold downward arrow represents a 2×2 max-pooling operation. The bold upward arrows are instances of `UpSampling2D`, a new Keras layer. As the name suggests, the layer uses nearest neighbors to upsample the height and width of the input tensor to double the spatial dimensions. In other words, it undoes the max-pooling operation, but the layer does not invent information and has no learnable parameters. Finally, the dotted arrows are skip connections that concatenate the earlier layer's output with the incoming tensor. This concatenation increases the number of channels in the tensor but leaves the spatial extent unchanged.

The input tensor is $64 \times 84 \times 1$, the input grayscale image. The output tensor is $64 \times 84 \times 11$, where the channels are the softmax vector for that

pixel. The largest softmax value across the channels tells us the pixel's predicted label.

We can downsample only twice because the bottleneck tensor is 16×21 spatially, and 21 is not an even number. However, two downsampling steps are all we need, so our architecture will do what we want it to do.

Let's implement Figure 15-11 in code and experiment with our three semantic segmentation datasets.

Implementing the U-Net Model

The U-net model is in *unet.py*. As this model is more advanced than many of our previous models, take some time to review the code before continuing.

Figure 15-11 is our guide. We'll implement each convolutional and pooling (downsampling) layer as a block, keeping the appropriate layer output to use as a skip connection when building the upsampling portion of the network. Convolutional blocks come in pairs via calls to `ConvBlock`. Listing 15-4 adds two 3×3 convolutional layers, each with `filters` filters, optional batch normalization, and a ReLU nonlinearity.

```
def ConvBlock(inp, filters, useBN=False):
    _ = Conv2D(filters, (3, 3), padding='same')(inp)
    if (useBN):
        _ = BatchNormalization()(_)
    _ = ReLU()(_)
    _ = Conv2D(filters, (3, 3), padding='same')(_)
    if (useBN):
        _ = BatchNormalization()(_)
    return ReLU()(_)
```

Listing 15-4: A convolutional block

The downsampling portion of the U-net with the 256-filter bottleneck becomes Listing 15-5.

```
inp = Input(input_shape)
c1 = ConvBlock(inp, 64, useBN=useBN)
```

```
_ = MaxPooling2D((2, 2))(c1)
c2 = ConvBlock(_, 128, useBN=useBN)
_ = MaxPooling2D((2, 2))(c2)
_ = ConvBlock(_, 256, useBN=useBN)
```

Listing 15-5: The downsampling portion of the U-net

The output of each convolutional block is retained in `c1` and `c2`. The upsampling portion of the U-net uses these outputs. The `MaxPooling2D` layers use standard 2×2 pooling to half the spatial dimensions while retaining the number of channels in the input tensor.

The final convolutional block with 256 filters is the bottleneck layer. It's the most critical layer in the model because its output represents the most abstract feature representations, akin to the dense layer of a standard CNN. The upsampling portion of the model uses these representations, which include spatial information in the per-channel layer output, to construct the upsampled output representing information about the class to which each pixel in the input belongs.

We're now ready for the upsampling portion of the network (Listing 15-6).

```
_ = UpSampling2D((2, 2))(_)
_ = concatenate([c2, _], axis=3)
_ = ConvBlock(_, 128, useBN=useBN)
_ = UpSampling2D((2, 2))(_)
_ = concatenate([c1, _], axis=3)
_ = ConvBlock(_, 64, useBN=useBN)
```

Listing 15-6: The upsampling portion of the U-net

Upsampling doubles the spatial dimensions of the input tensors so that they match the dimensions of the output of `c2` and `c1` (note the reverse order). This allows the two sets of outputs to be concatenated along the channel dimension and merges the feature representations from the upsampling layer and the earlier convolutional block when downsampling so that the next convolutional block can operate on them jointly.

All that remains is to run a 1×1 convolution over the 64×84×64 output of the final convolutional block. We want to predict 11 outputs (`num_classes`), 10 for the digits and a final output for the background pixels, as Listing 15-7 shows.

```
_ = Conv2D(num_classes, (1, 1))(_)
outp = Softmax()(_)
model = Model(inputs=[inp], outputs=[outp])
```

Listing 15-7: The final convolutional and softmax layers

The output of the final convolutional layer has a dimensionality of 64×84×11 to match the input image size of 64×84, with 11 channels for the digits 0 through 9 and for the background. To select a label for each pixel, we run over each pixel in the 64×84 output and assign the label corresponding to the largest softmax value across the channels.

Listing 15-8 illustrates that compiling and training remain the same as in previous models.

```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])

history = model.fit(x_train, y_train,
                   batch_size=batch_size,
                   epochs=epochs,
                   verbose=1,
                   validation_data=(x_test, y_test))
```

Listing 15-8: Compiling and training the U-net

As before, Adam is the optimizer, and the loss function is the categorical cross-entropy, this time computed over the 64×84 spatial extent of the output from the `softmax` layer. The single value reported during training is the average over the entire image and the minibatch. As most of the input image is background, the model's reported accuracy will quickly grow after even the first few minibatches of the first epoch. Don't be fooled; it will take epochs for the model to start

learning meaningful features of the digits. Patience is definitely required when training on our CPU-only system.

The remainder of *UNET.py* mimics earlier models to track the train and validation (test) loss during training, along with the all-important predictions on the test set:

```
prob = model.predict(x_test, verbose=0)
```

Here `prob` is a $1,000 \times 64 \times 84 \times 11$ NumPy array of softmax prediction vectors across the channel dimension for all 1,000 test-set images.

Table 15-1 tracks the dimensionality of the output tensors as data flows through the U-net model from top to bottom.

Table 15-1: Tensor Dimensions by U-Net Layer

Layer	Output tensor shape ($H \times W \times C$)
Input	$64 \times 84 \times 1$
ConvBlock (64 $\rightarrow$ c1)	$64 \times 84 \times 64$
MaxPool2D	$32 \times 42 \times 64$
ConvBlock (128 $\rightarrow$ c2)	$32 \times 42 \times 128$
MaxPool2D	$16 \times 21 \times 128$
ConvBlock (256)	$16 \times 21 \times 256$
UpSample2D	$32 \times 42 \times 256$
Concatenate (c2)	$32 \times 42 \times 384$
ConvBlock (128)	$32 \times 42 \times 128$
UpSample2D	$64 \times 84 \times 128$
Concatenate (c1)	$64 \times 84 \times 192$
ConvBlock (64)	$64 \times 84 \times 64$
Conv2D (11, 1×1)	$64 \times 84 \times 11$
Softmax	$64 \times 84 \times 11$

The downsampling, bottleneck, and upsampling are evident along with the extension of the channel dimensions when concatenating the output of earlier layers (blocks).

Now that we have the U-net model, let's put it to the test.

Running the Model and Interpreting Its Output

The file *unet.py* expects the following on the command line:

```
> python3 unet.py
```

```
unet <minibatch> <epochs> <useBN> blank|land|subtle <outdir>
```

<minibatch>	- minibatch size (e.g. 32)
<epochs>	- number of training epochs (e.g. 5)
<useBN>	- 0=no, 1=yes
blank land subtle	- data source
<outdir>	- output file directory (overwritten)

For example, to train the model on the blank background dataset for 20 epochs, use this:

```
> python3 unet.py 32 20 0 blank results/m2nist_32_20_0_blank
```

You'll find six such command lines in the file *go_unet*, which you can execute as follows:

```
> time sh go_unet
```

The `time` command will report the (very lengthy) runtime to train all six models.

Assuming a successful run of the command, we now have six directories under *results* to cover every combination of dataset with and without batch normalization. Investigating the results will be helpful to understand how everything is stored; then we'll write code to crunch the numbers for us.

Evaluating the Softmax Outputs

Let's load the softmax outputs for the model trained on the blank background dataset without batch normalization and compare the predictions to the known test-set labels. First, load each:

```
>>> import numpy as np; from PIL import Image
>>> soft = np.load("results/m2nist_32_20_0_blank/softmax.npy")
>>> ytest= np.load("../data/m2nist/ytest.npy")
>>> soft.shape, ytest.shape
((1000, 64, 84, 11), (1000, 64, 84, 11))
```

We import NumPy and PIL's Image class, then the predictions (`soft`) and the true labels (`ytest`). As we can see, both match in terms of dimensions. There are 1,000 test samples, each 64×84 pixels, with 11 channels: 0 through 9 for the digits, and the background as the 10th channel. So far, so good.

Let's extract the first test image predictions and labels by using `np.argmax` across the channels to select the largest value. The result will be an image where each pixel is in `[0, 10]`. We'll then view the images after multiplying them by 24 to make the pixel values intense enough to see:

```
>>> pred0 = np.argmax(soft[0], axis=2)
>>> ytest0 = np.argmax(ytest[0], axis=2)
>>> pred0.shape, ytest0.shape
((64, 84), (64, 84))
>>> Image.fromarray((24*pred0).astype("uint8")).show()
>>> Image.fromarray((24*ytest0).astype("uint8")).show()
```

The two images appear to be much the same, which is a good sign, but the background is the brightest part of the images, which is distracting. We'll add 1 to the pixel values to shift them to `[1, 11]`, then map 11 to 0 so the background is black:

```
>>> pred0 += 1; pred0[np.where(pred0 == 11)] = 0
>>> ytest0 += 1; ytest0[np.where(ytest0 == 11)] = 0
>>> Image.fromarray((24*pred0).astype("uint8")).show()
>>> Image.fromarray((24*ytest0).astype("uint8")).show()
```

The two images are now as we expected, and each digit is a different shade of gray. We have a good match, visually, giving us hope that the model has performed well at the pixel level.

The code in *UNET_results.py* takes this simple exploration to the next level by doing the following:

1. Generating prediction images by repeating the process just presented (but without multiplying by 24).
2. Calculating the IoU between the predicted labels and the actual labels.
3. Storing RGB images of the per-pixel predictions by digit type so we can easily discern the label assigned to each part of the image. The images are side by side: actual labels, then predicted.

The code expects the path to a results directory, the output of *UNET.py*, along with the type of dataset (`blank`, `land`, `subtle`) and, optionally, the text `images` to create the RGB files. For example, to run *UNET_results.py* on the blank background results, use the following:

```
> python3 UNET_results.py results/m2nist_32_20_0_blank blank images
```

```
Mean per class IoU:
```

```
0: 0.930688
1: 0.959447
2: 0.903279
3: 0.955461
4: 0.871107
5: 0.941763
6: 0.938919
7: 0.895834
8: 0.904602
9: 0.884790
```

This presents the mean IoU per class, which shows that the model was quite accurate over all the classes. The precise IoU calculation will be discussed shortly; it isn't as straightforward as it was in the bounding-box case. The mean IoU over all classes was 91.86 percent.

A new *images* directory inside *m2nist_32_20_0_blank* contains 1,000 RGB images. Each digit is assigned a specific color. Run *labels.py* to create *labels.png* to serve as a key. Figure 15-12, which should be viewed in color, shows three pairs of true label and RGB output images along with the key.

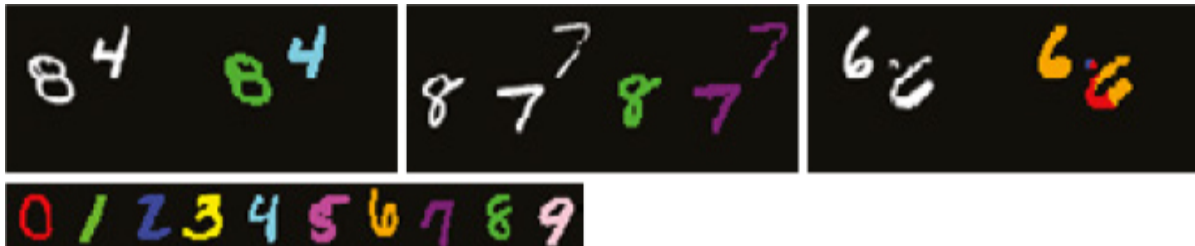


Figure 15-12: True label and prediction image pairs (top) with key (bottom)

Each of the three examples was well classified overall. In the rightmost example, however, the model has some difficulty with the lower-left portion of the 6 on the right. Many of those pixels were labeled as class 0 instead.

Encouraged by the blank background results, let's consider the land-use background and the subtle image models. The corresponding per-class mean IoU scores, with the land-use background on the left, are as follows:

Mean per class IoU:	
0: 0.870736	0: 0.302580
1: 0.832945	1: 0.338912
2: 0.741750	2: 0.254495
3: 0.786871	3: 0.298379
4: 0.776546	4: 0.276729
5: 0.793539	5: 0.258227
6: 0.842348	6: 0.328832
7: 0.753311	7: 0.300106
8: 0.752562	8: 0.210771
9: 0.758616	9: 0.287867

The overall mean IoU scores were 79.09 and 28.57 percent, respectively. The subtle digits are quite difficult, as expected. Note, however, that none of the models used batch normalization. The overall

mean IoU for each with batch normalization included during training was 91.82 (blank), 87.41 (land), and 37.79 (subtle) percent. Batch normalization did not aid the blank background model but did improve the models trained on images with more-complex backgrounds.

Figure 15-13 presents selected land-use (top) and subtle (middle) test images and associated segmentations for models using batch normalization.

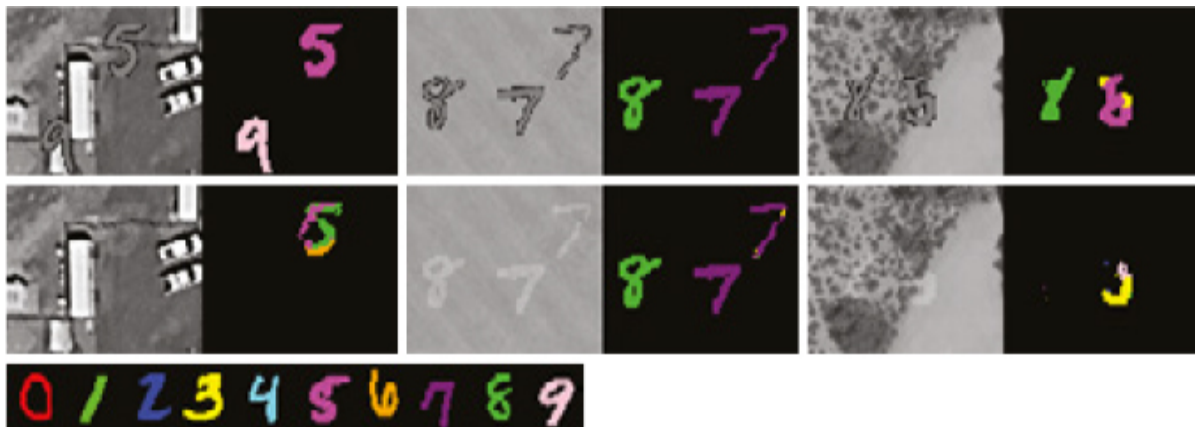


Figure 15-13: True label and prediction pairs: land use (top) and subtle (middle) with key (bottom)

The subtle images are difficult even for humans to parse.

Notice that the same test image is present in the middle of each row of Figures 15-12 and 15-13. The land-use version of the image has a nearly uniform background, making it an easy target for the classifier. The subtle image is similarly well detected, with only a slight uncertainty about parts of the 7 in the upper right. However, the leftmost and rightmost subtle images are a mess. The overall shape of the 5 on the left is detected but split among multiple classes, while the 9 is missed entirely. On the right, the 8 is missing, and only the bottom portion of the 5 is detected and assigned to class 3. Usually, if we have difficulty seeing something in an image, models do as well. Recall that the features learned by low-level CNN layers are of the same sort as those used by the early stages of the mammalian visual system.

Batch normalization's power comes from its ability to address the internal covariate shift that happens when the model is training and the variability in the outputs of the model's layers change. Complex

backgrounds, like those of the land-use and subtle datasets, introduce variability, especially minibatch to minibatch, thereby giving batch normalization something to work with (to adjust for). The blank backgrounds are simple, and minimal shift occurs in the network layer output minibatch to minibatch, implying less of an effect from the batch normalization layers.

Calculating the U-Net IoU

We've understood model performance via the IoU calculation; now, let's understand the IoU implementation required by semantic segmentation. You learned earlier in the chapter that the IoU requires us to calculate the intersection and union of two areas—here, the set of pixels declared to be of a specific class with those actually of that class.

First, we must allow for the possibility of false positives, of predictions for class n even if class n isn't present in the input image. In that case, the IoU will be 0 because there is nothing to intersect with, but the 0 will negatively affect the mean IoU for the class. We don't want the model to produce false-positive predictions, so including them in the per-class means makes sense.

Listing 15-9 contains `iou`, the function that *unet_results.py* uses to determine the mean IoU scores per class over the test set.

```
def iou(x, y):
    def CalcIoU(x, y, k):
        xk = np.where(x.ravel() == (k+1))[0]
        yk = np.where(y.ravel())[0]
        i = set(xk) & set(yk)
        u = set(xk) | set(yk)
        return len(i) / len(u)

    iou = np.zeros(10)
    n = np.zeros(10)
    for k in range(10):
        for i in range(len(x)):
            id0 = np.where(x[i] == (k+1))[0]
            id1 = np.where(y[i,:,:,:k] == 1)[0]
```

```

        if (id0.size > 0) or (id1.size > 0):
            iou[k] += CalcIoU(x[i], y[i,:,:k], k)
            n[k] += 1
    return iou / n

```

Listing 15-9: IoU calculation for semantic segmentation

Because the digit images are not simple rectangles, as they were for the bounding boxes earlier in the chapter, we have no choice but to use pixel-level masks to determine the intersection and union.

The body of `iou` tracks the total IoU by class over the test sample results passed to it in `x` and `y`, where both are 4D NumPy arrays, 1,000×64×84×11, representing the predictions (`x`) and true labels (`y`).

The `for` loop over class labels, indexed by `k`, examines each of the 1,000 predictions (indexed by `i`). First, `id0` and `id1` hold the locations in the current test image where class `k` pixels are present. The offset by 1 in the line defining `id0` is present to account for the shift applied to the predictions to reset the background to 0 instead of 10.

If either `id0` or `id1` is not empty, call `calcIoU` to calculate the IoU for the *i*th test image and the *k*th class. Notice that the return value is added to the total for the *k*th class and the counter for that class is incremented.

The inner `calcIoU` function generates `xk` and `yk`. These lists contain all nonzero locations in the two images, first for `x`, a prediction image where the comparison is for all pixels labeled *k* (plus 1 for the offset), then for `y`, a true label image where any nonzero pixel is part of the *k*th digit (if present).

The heart of `calcIoU` transforms these lists into Python sets before calculating the intersection (`i`) and union (`u`) via standard Python set operators. The cardinality of the two is then returned as a ratio, that is, the desired IoU. The final line of `iou` returns the mean IoU per class over the test set.

Semantic segmentation is a powerful approach to image classification but comes at the cost of expensive dataset labeling. It takes only a second to mark an image as “dog” or “cat,” and only a few seconds longer to draw a bounding box around the objects that we want

a model to localize. However, full semantic segmentation of even the pixels related to important classes, marking the rest as “background,” takes time and effort for real-world scenarios.

Multilabel Classification

Standard classification assigns a single class label to the input image. With *multilabel classification*, we learn of the presence of each known class independently. Earlier in the chapter, we used a dual-headed model to detect an MNIST digit in a larger image. One of those heads predicted bounding-box coordinates by learning a per-output function passed through a sigmoid activation to produce a value in $[0, 1]$ representing a normalized coordinate or length.

If we alter the usual output of a classifier by replacing the softmax prediction with a sigmoid and use binary cross-entropy as the loss function, something interesting happens: Instead of predicting a single class, the model will (we hope) learn to predict the likelihood of each class independently. Predicting multiple classes for a single input is a form of multilabel classification. We might regard detection, especially with a model like YOLO, to be likewise an instance of multilabel classification.

This section experiments with a model that indicates the presence of multiple MNIST digit classes within a larger input image. The model will indicate only the presence of the digit and will not localize it.

Dataset

First, we need a new dataset, one that, like the M2NIST dataset we used for semantic segmentation, contains multiple MNIST digits per image, up to three. The digit classes, however, need not be unique. The code we want is in *build_multilabel_dataset.py*. To save space, I’ll skip the implementation.

The output is two new datasets of 128×128-pixel images, one with a blank background and another with random land-use backgrounds, each with 6,000 train and test images. The associated labels are binary

vectors with a 1 for at least one digit of that class present in the image. For example, if the image contains three digits, two 7s and one 4, then the label for that image is 0,0,0,0,1,0,0,1,0,0. The digit images are placed randomly within the larger image, but never so they overlap.

We know MNIST digits are an “easy” dataset, so the model we need isn’t advanced. You’ll find the code to construct and train the model in *multilabel.py*, with relevant portions in Listing 15-10.

```
multilabel.py
def ConvBlock(_, filters):
    _ = Conv2D(filters, (3, 3), padding='same')(_)
    _ = ReLU(_)
    _ = Conv2D(filters, (3, 3), padding='same')(_)
    _ = ReLU(_)
    return MaxPooling2D((2, 2))(_)

def DenseBlock(_, nodes):
    _ = Dense(nodes)(_)
    _ = ReLU(_)
    _ = Dropout(0.5)(_)
    return _

inp = Input(input_shape)
_ = ConvBlock(inp, 32)
_ = ConvBlock(_, 64)
_ = ConvBlock(_, 128)
_ = Flatten(_)
_ = DenseBlock(_, 128)
outp = Dense(num_classes, activation='sigmoid')(_)
model = Model(inputs=inp, outputs=outp)
```

Listing 15-10: Building the multilabel model

The model is based on the code in *vgg8.py* from Chapter 13, where the batch normalization and convolutional layer dropout layers have been removed. The final Dense layer uses a sigmoid activation function, not the usual softmax. The sigmoid treats each of the 10 nodes in the layer independently. Most of the model’s 4.4 million parameters are

between the 16×16×128 output of the last convolutional block and the 128-node dense layer.

Compiling and training the model requires only a few lines of code, as in Listing 15-11.

```
model.compile(loss=keras.losses.binary_crossentropy,  
              optimizer=keras.optimizers.Adam())  
  
model.fit(xtrain, ytrain, batch_size=batch_size,  
          epochs=epochs, verbose=1)
```

Listing 15-11: Compiling and training the model

The critical observation here is that binary cross-entropy is used in place of the typical categorical cross-entropy.

Categorical cross-entropy calculates the negative log-likelihood over the possible classes. Binary cross-entropy instead calculates the likelihood for each output (class) independently. A model using a sigmoid activation function and binary cross-entropy as the loss function can be thought of as a collection of independent models learning a separate classification function for each class, using the shared convolutional portion of the network to produce a new representation of the input image. The new representation captures characteristics of the image useful for classifying each digit type.

The model predicts a 10-element vector for each input sample, as with a softmax classifier, but the predictions are interpreted differently. They are not the likelihood of the input belonging to a specific class but rather the likelihood that a digit of that class appears in the input.

Let's build the datasets, then train and test the model by using the blank-background dataset:

```
> python3 build_multilabel_dataset.py  
> python3 multilabel.py 32 60 blank results/mnist_multi_32_60_blank
```

Now let's load the output predictions and compare them to the test-set labels:

```
>>> labels = np.load("../data/mnist/mnist_multilabel_ytest.npy")
>>> pred = np.load("results/mnist_multi_32_60_blank/predictions.npy")
>>> labels[0]
array([0, 0, 0, 0, 1, 0, 0, 1, 0, 0], dtype=uint8)
>>> pred[0]
array([1.83534768e-17, 9.99004482e-13, 1.07964461e-18, 1.82442977e-10,
       9.99986172e-01, 3.22842447e-13, 3.18905196e-18, 9.99937356e-01,
       1.01429855e-17, 7.52019957e-09], dtype=float32)
```

The first test sample contains 4s and 7s, at least one of each. The model's prediction indicates a high likelihood of 4s (0.9999) and 7s (0.9999) and essentially zero likelihood for all other classes. In other words, the model is entirely correct for this test sample.

Performance

The fact that inputs contain a variable number of digits presents a small challenge when it comes to interpreting the model's performance. We have options. The code in *multilabel_results.py* focuses on the positive by determining the number of times each digit was successfully recognized over the entire test set and the number of times all the digit types in a test input were successfully recognized. Run the code with the following:

```
> python3 multilabel_results.py results/mnist_multi_32_60_blank/predictions.npy
    results/mnist_multi_32_60_blank
```

Correct detection of all digits: 0.84667 (threshold=0.10)

Correct detection by digit:

```
0: 0.91282
1: 0.98441
2: 0.91973
3: 0.93632
4: 0.91770
5: 0.90640
6: 0.93969
7: 0.95641
```

```
8: 0.90622
9: 0.93476
mean: 0.93145
```

Naturally, your results will be slightly different but should be similar. The first output line tells us that the model was 84.7 percent accurate at correctly predicting each test sample's complete set of digits. The remainder of the output indicates the per-class accuracy over the entire dataset. The model was best at recognizing 1s (98.4 percent) and worst at recognizing 5s (90.6 percent), with a mean accuracy over classes of 93.1 percent.

The mean accuracy has to do with the model's ability to pick up the presence of a digit, while the detection (recognition) score reflects the model's ability to recognize the complete set of digits in a given input sample. We'll get to the threshold value momentarily.

Running *multilabel.py* and *multilabel_results.py* on the land-use version (replace `blank` with `land`) produces a correct all-digit accuracy of 82.2 percent with a mean over individual digits of 92.1 percent. The drop in accuracy is plausible when compared to similar changes observed earlier in the chapter when localizing digits in images with more-complex backgrounds.

These commands produced new *plots* directories in the respective *predictions.npy* directories. The *plots* directories contain randomly selected sets of bar plots like those in Figure 15-14 taken from the land-use results.

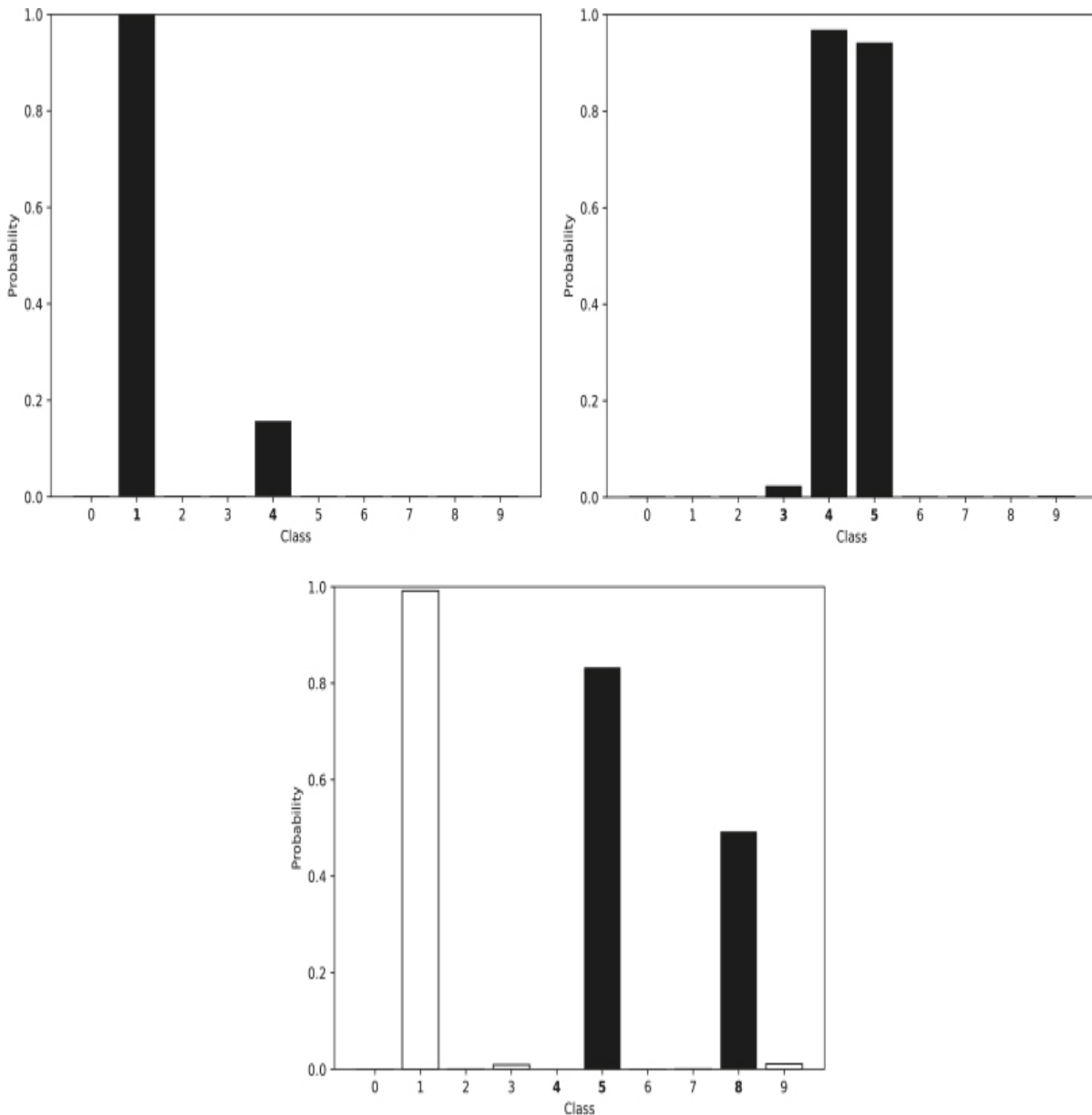


Figure 15-14: Land-use multilabel results showing prediction likelihood (bar height) and actual digits present (filled bars)

The plots show the per-class predicted likelihood as bars. If the bar is filled, that digit is in the test sample. The class label is likewise in bold for digits in the test image. An unfilled bar is a nonzero prediction of a digit that was not present, that is, a false positive.

From top to bottom, the first plot shows the likelihood for a sample with two digit types, 1 and 4. The model produced a likelihood of nearly 1.0 for the 1 and about 0.15 for the presence of the 4. Here's where the

threshold comes into the picture. I set the threshold to a low value of 0.1, implying that any class prediction above that threshold will be considered present in the image. Therefore, the top plot in Figure 15-14 is completely accurate: The image contains at least one 1 and one 4, and both are deemed present by the model.

The middle plot shows a case with three digit types: 3, 4, and 5. However, the likelihood of 3 is below the 0.1 threshold, so the model misses it. The bottom plot represents a sample image containing a 4, a 5, and an 8, of which only the 5 and 8 are recognized by the model. Additionally, the model is very confident of the presence of a 1 digit, even though it isn't present (the bar isn't filled, and the "1" label isn't bold). In other words, the 1 is a false positive.

Figure 15-15 shows the test image that produced the bottom plot in Figure 15-14. The 4 is rough, making it easy to understand why the model called it a 1 instead.



Figure 15-15: The test image producing the bottom plot in Figure 15-14

The accuracies produced by *multilabel_results.py* do not consider false positives. Paging through the images in the *plots* directories will convince you, however, that false positives are not particularly common.

Chapter 8 taught you about other metrics we might implement to characterize the multilabel model. For example, changing the recognition threshold would make it possible to produce ROC curves per digit or for entirely correct recognitions (all digits in the input image). I leave such investigations as an exercise.

Summary

In this chapter, you learned how to detect objects in images instead of merely indicating their presence. We explored two common approaches: detection with bounding boxes and semantic segmentation. The former classifies and localizes an object, while the latter assigns a class label to every pixel in the image.

The detection model introduced you to dual-headed neural networks: One head was responsible for classifying the digit by producing a softmax output over the set of known classes, and the second head learned bounding-box coordinates enclosing the detected object. The regression head used regression to learn a function that maps features in the dense layer shared between the two heads to bounding-box coordinates. The classification head used the same shared feature set to determine the most likely object class.

Semantic segmentation came next. We used an existing dataset, modified to include a more complex background imagery, with a two-downsample U-net model to explore how well the model learned to label the pixels of MNIST digits within the input image.

Quantifying the output of both the detection and semantic segmentation models introduced you to the intersection over union (IoU) metric. You learned how to calculate the IoU mathematically for bounding boxes and via pixel-level masks for the more complex shapes learned by semantic segmentation.

We closed the chapter by noticing that the regression head used by the detection model can be applied to a multilabel scenario: Instead of localizing an object, the output now represents the likelihood of more than one object class appearing in the input. We experimented with a dataset containing up to three MNIST digits and discovered that the model could recognize the presence of multiple digits.

The next chapter goes beyond supervised learning to build models capable of learning from unlabeled data alone.

16

SELF-SUPERVISED LEARNING



Supervised learning requires large, labeled datasets. This chapter introduces self-supervised learning, which uses only unlabeled data to condition a model for downstream tasks. Leveraging unlabeled data reduces our dependence on large, often expensive-to-produce, labeled datasets.

Chapter 14 explored transfer learning and fine-tuning by using a pretrained model as a feature generator or a starting point for fine-tuning to a new dataset. These potent techniques help us build successful models with small datasets. A large, pretrained model requires a large, labeled dataset. Such datasets, like ImageNet, exist, but it would be convenient to learn a feature generator for transfer learning or fine-tuning from unlabeled data more directly related to the task at hand. Enter self-supervised learning.

Self-supervised learning uses unlabeled data and a proxy task that, we hope, results in learned features useful for transfer learning and fine-tuning with new datasets. The critical difference from earlier methods is that we haven't taken the time to label the dataset. Rather, we define a proxy task that lets us label the dataset automatically and then train a

model on that task, hoping that the learned features aid us with the downstream tasks we're genuinely interested in.

We can understand the self-supervised learning process on a small scale with simple techniques that still achieve our end goal. Specifically, this chapter explores self-supervised learning via rotation prediction and Siamese networks.

Our end goal is to produce pretrained feature generators for transfer learning and fine-tuning on CIFAR-10. To do that, we require an unlabeled dataset; let's begin there.

Building the Unlabeled Dataset

Like CIFAR-10, the CIFAR-100 dataset has a 50,000-image training set and a 10,000-image test set. Unlike CIFAR-10, CIFAR-100 contains not 5,000 examples from 10 classes but 500 examples from 100 classes. And, most important for us, the two do not overlap in classes. We are therefore free to use CIFAR-100 as our unlabeled dataset to produce pretrained models for use with CIFAR-10, knowing that we're not cheating by pretraining on instances from the same class as our target.

Let's build an unlabeled dataset based on CIFAR-100. First, download CIFAR-100 from its website at <https://www.cs.toronto.edu/~kriz/cifar.xhtml>. You'll find the necessary file, *cifar-100-python.tar.gz*, by scrolling down the page. Second, expand the file in the directory containing the code for this chapter:

```
> tar xzf cifar-100-python.tar.gz
```

Finally, execute *build_cifar100.py* to create the NumPy data arrays we'll use throughout the chapter:

```
> python3 build_cifar100.py
```

You'll find them in the *data/cifar100* directory.

The file *build_cifar100.py* imports the expanded Python pickle file by using code given on the CIFAR-100 website and then creates six output

files. Two output files, *xtrain.npy* and *xtest.npy*, contain the train and test images as $32 \times 32 \times 3$ arrays (RGB images).

The remaining four files are train and test labels. CIFAR-100 is labeled twice, first with 100 fine labels and then again with 20 coarse labels made by bundling related fine labels. We won't use the labels in this chapter, but the files are there if you want to use them: *ytrainf.npy* (fine) and *ytrainc.npy* (coarse), and corresponding test-set labels. The full list of fine and coarse labels is at the bottom of *build_cifar100.py*.

Now that we have our unlabeled dataset, let's use it to produce pretrained models, beginning with rotation prediction.

Rotation Prediction

We want a pretrained feature generator, so we need a proxy task that, we believe, will force the model to learn relevant image features along the way. RotNet (described in “Unsupervised Representation Learning by Predicting Image Rotations” by Spyros Gidaris et al., 2018) takes an unlabeled dataset and produces a new dataset by rotating each image by a certain amount. Rotation prediction's goal is to learn to predict the rotation applied.

Variations exist, but the easiest to implement chooses a random number of 90-degree rotations to apply to the image and assigns an appropriate label. For example, if the image isn't rotated, call that image an instance of class 0. Rotating the image by 90 degrees means you now have a member of class 1. Likewise, 180 degrees is class 2, and 270 degrees is class 3.

We now have a proxy dataset, and we're asking a model to learn whether the input is an image that has been rotated by 0, 90, 180, or 270 degrees. We hope that forcing the model to understand these inputs will require it to learn representations related to the content of the images that we can leverage elsewhere.

The remainder of this section defines *rot.py*, an implementation of this process. We then use *rot.py* to train a model to predict image rotations. We don't care too much about this model, but we do care

about the convolutional and pooling layers of the trained model, which we'll set aside for transfer learning experiments.

RotNet is such a cleverly simple technique that it raises all manner of “what if” questions. So, we follow *rot.py* with five variations on the RotNet theme and compare their transfer learning performance to *rot.py*.

Fine-tuning experiments follow, but only after exploring the features that RotNet generates with t-SNE. We'll relate both to what we found in Chapter 14, when we used models pretrained on ImageNet.

Building and Testing RotNet

Start by reviewing *rot.py*. The format it implements is repeated for all the experiments in this section:

1. Define a rotation dataset on the fly (`RotationDataset`).
2. Define a three-block convolutional and pooling backbone model (see Figure 16-1).

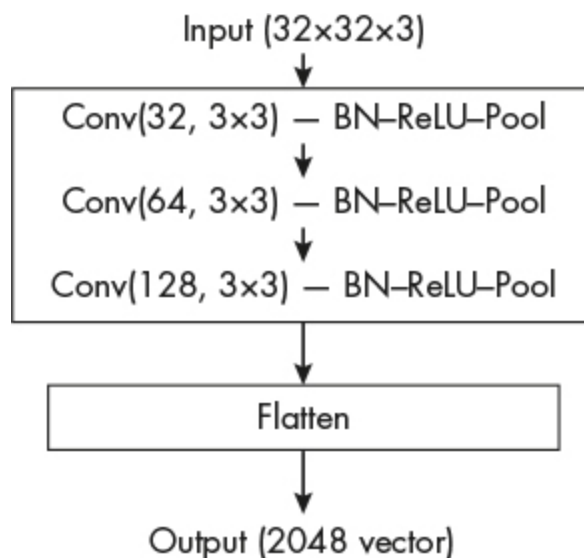


Figure 16-1: The RotNet base model

3. Add a prediction head (four-node dense layer direct to a softmax layer).

4. Train, test, and store relevant results, including the trained backbone model.

In Chapter 15, we used a model similar to the backbone model in Figure 16-1 for the dual-headed digit detector. However, this version adds batch normalization layers. All the experiments in this section train and store this model for downstream tasks.

Listing 16-1 builds the backbone model and the classification head.

```
inp = Input(input_shape)
_ = Conv2D(32, (3, 3), padding='same')(inp)
_ = BatchNormalization()(_)
_ = ReLU()(_)
_ = MaxPooling2D((2, 2))(_)
_ = Conv2D(64, (3, 3), padding='same')(_)
_ = BatchNormalization()(_)
_ = ReLU()(_)
_ = MaxPooling2D((2, 2))(_)
_ = Conv2D(128, (3, 3), padding='same')(_)
_ = BatchNormalization()(_)
_ = ReLU()(_)
_ = MaxPooling2D((2, 2))(_)
outp = Flatten()(_)
base = Model(inputs=inp, outputs=outp)

os.system("rm -rf %s; mkdir %s" % (outdir, outdir))
base.save(outdir+"/raw.keras")

_ = Dense(num_classes)(base.output)
outp = Softmax()(_)
model = Model(inputs=base.input, outputs=outp)
```

Listing 16-1: The RotNet backbone model and classification head

Between the backbone (`base`) and classification models are two lines that create the output directory, then store the *untrained* backbone model as *raw.keras*. We expect pretraining to help, but to know that it

did, we must compare downstream performance with the output of the untrained backbone model.

The classification head is trivial in this instance. The flattened feature output is passed directly to a four-node `Dense` layer followed by a `Softmax` without an intervening `Dense` layer to learn from the backbone model's features. I'm unsure whether this approach helps versus adding a more typical dense learning layer, but feel free to add one and rerun the experiments. I expect that removing the extra dense layer will encourage the model to be all the more expressive with the features it learns in the backbone model, which will only help us down the road.

The remainder of *rot.py* is likewise familiar: Train the model by using the Adam optimizer and output performance measures (confusion matrix, accuracy, MCC, and so on) along with the now-trained backbone model (as *model.keras*).

Running RotNet

The file *rot.py* expects little on the command line:

```
> python3 rot.py
```

```
rot <minibatch> <epochs> <outdir>
```

```
<minibatch>    -  minibatch size (e.g. 128)
<epochs>       -  number of training epochs (e.g. 10)
<outdir>       -  output file directory (overwritten)
```

Let's train the RotNet model for 10 epochs, using a minibatch of 128 samples:

```
> python3 rot.py 128 10 results/rot_128_10
```

The output directory, within the *results* directory, contains two versions of the backbone model, *raw.keras* and *model.keras*. My run produced the following final output:

```
Test set accuracy: 0.5396, MCC: 0.3953
[[879  63 243  73]
```

```
[372 528 255 118]
[303  83 825  70]
[355 108 259 466]]
```

We have four classes, four possible rotation angles to predict, so random prediction would give an accuracy on the test set of about 25 percent. We're at 54 percent, so while there's nothing to write home about, it's clear that the model did learn something about image rotations. The confusion matrix is correspondingly rough, but we'll run with it regardless and use the models in a transfer-learning scenario involving classical machine learning models.

Evaluating RotNet with Classical Models

Transfer learning involves extracting features from a layer of the pretrained model. The RotNet backbone model consists of three blocks (Figure 16-1). The file *rot_extract_features.py* loads a pretrained model, passes the CIFAR-10 train and test sets through it, and extracts a given block's output to produce new train and test representations. For example, the following will produce two files in the temporary directory:

```
> python3 rot_extract_features.py results/rot_128_10/model.keras 3 /tmp/ttt
```

The first file is the throwaway base name of *ttt*:

```
ttt_cifar10_features_train.npy
ttt_cifar10_features_test.npy
```

Both files contain (10000, 2048) arrays in the same order as the source images so that the first 10,000 labels of the existing CIFAR-10 training file match, row by row.

Notice the 3 in the command line, which tells *rot_extract_features.py* to pull the output features from the `MaxPooling2D` layer at the end of block 3, a $4 \times 4 \times 128$ tensor that flattens to the $4 \times 4 \times 128 = 2,048$ -element output vectors. Pulling from block 2 returns 4,096-element vectors, while block 1 returns 8,192-element vectors.

We experimented with classical machine learning models in Chapter 4. The file *classic_models.py* mimics this approach to train elementary machine learning models by using the transformed CIFAR-10 images. The file is simple: Create an sklearn model instance, fit using the train and test vectors (and associated labels), and print the score (accuracy). Run the following code:

```
> python3 classic_models.py /tmp/ttt_cifar10_features_train.npy  
                               /tmp/ttt_cifar10_features_test.npy
```

This produces the following:

```
Nearest centroid:  
    score = 0.4047  
  
Naive Bayes classifier (Gaussian):  
    score = 0.4162  
  
k-NN classifier (k=3):  
    score = 0.4468  
  
k-NN classifier (k=7):  
    score = 0.4620  
  
Random forest classifier (estimators=100):  
    score = 0.5006  
  
Random forest classifier (estimators=400):  
    score = 0.5187  
  
MLP classifier (512,256):  
    score = 0.6275
```

I'll summarize this in vector form to save space:

(0.4047, 0.4162, 0.4468, 0.4620, 0.5006, 0.5187, 0.6275)

The classical models range in order of increasing complexity and expected performance on the same dataset. The accuracy of using the

RotNet features ranges from a low of 40.5 percent to a high of 62.8 percent. Let's compare these scores to those from the untrained backbone model. Repeating the exercise but replacing *model.keras* with *raw.keras* gave me this:

(0.2606, 0.2798, 0.3227, 0.3399, 0.4338, 0.4507, 0.5367)

In every instance, the classical model trained with the RotNet features was better than that trained on untrained features, This gives us confidence that RotNet has learned something useful about $32 \times 32 \times 3$ images of objects in the natural world. For comparison purposes, here are the results for the original CIFAR-10 images unraveled into 3,072-element vectors:

(0.2772, 0.2933, 0.2780, 0.2935, 0.4194, 0.4426, 0.4488)

Compare the results for the raw model features to those for the original images. The randomly mapped features produced by the untrained backbone model (especially for all but the simplest of classical machine learning models) outperforms the original inputs as feature vectors, especially so with the MLP.

Random transformation of feature vectors may lead to new representations better suited to specific machine learning models. This observation is at the heart of *extreme learning machines (ELMs)*. In these simple two-layer neural networks, the first layer uses a random weight matrix, followed by a hidden layer to output layer weights learned by solving a matrix equation. ELMs, therefore, train without backpropagation and gradient descent. You'll find more about ELMs in Chapter 6 of my book *The Art of Randomness* (No Starch Press, 2024).

The experiment used block 3 features. We'll stick with those for now. You may be wondering whether my ad hoc claim that the RotNet features are better than the untrained features is valid, given I've presented only a single example of each condition. To save space and time, I won't further justify my claim by repeating the experiment to get valid statistics. I leave that as an exercise for you but anticipate no need to adjust my claim.

The simplicity of RotNet might make us wonder what other “obvious” image transformations might coerce a model into learning valuable features. We’ll now explore five variations on the main RotNet theme, using their performance on *classic_models.py* as our evaluation metric.

Exploring Variations on the RotNet Theme

The file *rot.py* adds pseudolabels to the CIFAR-100 unlabeled images by using four 90-degree rotations. A newer model, PatchRot (described in “PatchRot: A Self-Supervised Technique for Training Vision Transformers” by Sachin Chhabra et al., 2022), adapts RotNet for vision transformers. We’ll encounter transformer models in Chapter 18, but not in the context of computer vision. Nevertheless, the idea behind PatchRot, which extends RotNet, is worth exploring.

The file *rot16.py*, not to be confused with the old Unix command `rot13`, differs from *rot.py* only in the operation of the `RotationDataset` function. Instead of selecting one of four possible 90-degree rotations, *rot16.py* follows PatchRot and treats the input image, 32×32 pixels, as a 2×2 grid of smaller 16×16 images and randomly rotates *one* of those smaller images by a randomly selected 90-degree rotation. There are 4 smaller images and 4 possible rotations, each giving 16 possible image transformations, leading to a proxy dataset with 16 classes. You can review *rot16.py* for the details.

The file *rot32.py* extends *rot16.py* to 32 classes by first randomly choosing whether to flip the input image upside down before selecting a random rotation transformation for a subset of the image.

The standard image-processing technique *alpha blending* merges two images at the pixel level by setting the output image to a weighted sum of the inputs: $I_{ij} \leftarrow \alpha A_{ij} + (1 - \alpha) B_{ij}$ for input images A and B and $\alpha \in [0, 1]$.

The file *blend.py* creates a three-class proxy dataset: the original image, the image flipped upside down, or the image alpha blended at $\alpha = 0.5$ with another image. The code uses each unlabeled image only once.

Finally, the file *iproc.py* implements two sets of four image transformations selectable on the command line. The first set chooses between inverting the image, applying a 90-degree rotation, a blurring operation, or a gamma correction operation.

Inverting an image entails replacing each pixel, p , with $255 - p$. The blurring operation convolves a Gaussian kernel over the image by using SciPy's `gaussian_filter` function. Gamma correction replaces each pixel with the original raised to a power (γ) and possibly scaled (c):

$$I_{ij} \leftarrow c(I_{ij})^\gamma$$

We set $c = 1$ and use $\gamma = 1.07$ for the first set of image-processing transformations. A $\gamma > 1$ makes the image brighter.

The second set of image transformations in *iproc.py* uses four values of γ : 0.8, 0.9, 1.0, and 1.1. Note that $\gamma = 1.0$ implies no change to the original image.

We now have six self-supervised learning approaches: RotNet and its five variations. First, let's train each variant for 20 epochs. I'm using 20 epochs to give the variants more opportunity to learn, as their transformations are likely more difficult for the model given their complexity relative to simple bulk image rotations. We'll also train *rot.py* for 20 epochs. Execute the script *go_rotnet*, which contains the following:

```
python3 rot.py      128 20 results/rot_128_20
python3 rot16.py    128 20 results/rot16_128_20
python3 rot32.py    128 20 results/rot32_128_20
python3 blend.py    128 20 results/blend_128_20
python3 iproc.py 0 128 20 results/iproc_0_128_20
python3 iproc.py 1 128 20 results/iproc_1_128_20
```

My run produced the following accuracies and MCC values per model:

```
rot.py:      Test set accuracy: 0.5042, MCC: 0.3565
rot16.py:    Test set accuracy: 0.6362, MCC: 0.6130
rot32.py:    Test set accuracy: 0.4150, MCC: 0.3965
blend.py:    Test set accuracy: 0.7272, MCC: 0.5910
```

iproc.py 0: Test set accuracy: 0.8826, MCC: 0.8465
iproc.py 1: Test set accuracy: 0.9438, MCC: 0.9267

Recall that these indicate the test-set performance on the proxy tasks. We haven't yet applied the learned backbone models. Notice that *rot.py* produced a lower test accuracy after 20 epochs than with 10 epochs. This is possibly an indicator of overfitting to the relatively small model. The *rot32.py* model was similarly poor, though that might be due to the more challenging nature of the proxy task.

The other image-processing models, *blend.py* and *iproc.py*, fared better, but it remains to be seen whether the better performance is due to the model being a good fit to the dataset or the task being much easier to learn, which may imply that the base model features aren't particularly useful for downstream tasks.

Speculation must give way to cold, hard numbers. Running each model through *rot_extract_features.py* to transform the CIFAR-10 dataset followed by running *classic_models.py* on those features produces Table 16-1.

Table 16-1: Classical Model Accuracy by Pretrained Model

Model	NC	NB	3-NN	7-NN	RF100	RF400	MLP
rot	0.4312	0.4349	0.4173	0.4268	0.4719	0.5116	0.6207
rot16	0.4336	0.4232	0.3734	0.3790	0.4506	0.4921	0.5796
rot32	0.4713	0.4579	0.3792	0.3814	0.4618	0.5029	0.5929
blend	0.3837	0.3834	0.4592	0.4795	0.4635	0.4926	0.6016
iproc 0	0.3744	0.3766	0.4229	0.4323	0.4625	0.4937	0.5962
iproc 1	0.3384	0.3418	0.3622	0.3909	0.4328	0.4589	0.5672
raw	0.2606	0.2798	0.3227	0.3399	0.4338	0.4507	0.5367

The columns in Table 16-1 present the scores from the nearest centroid (NC) to naive Bayes (NB) through the nearest neighbor (NN) and random forest (RF) models to the MLP. There's an overall trend of

improving scores as classical model complexity increases, with the MLP consistently outperforming the other models, usually by 10 percent or more.

This makes sense. The features are the output of a neural network and are, most likely, amendable to such models. The raw features are always the worst performer, which is also a sensible outcome and demonstrates that every pretraining model is doing something in terms of learning features useful for downstream tasks.

My first thought regarding how to declare a winner was to use a weighted average across the classical model scores, giving the nearest centroid a weight of 1 and the MLP a weight of 7. That will undoubtedly lead to an average score, but the MLP scores are so much higher than the other model scores that, when combined with a weight of 7, the final average mean score will rank the same as the MLP results. Therefore, ranking by the MLP score declares *rot.py* to be the “winner,” with *blend.py* in second place, followed by *iproc.py* with the first set of image transformations in third. As expected, the untrained model features are in last place.

Notice that the *blend*, *iproc*, and *raw* rows follow a progression from left to right of improving model performance. Yes, the *blend* row has a single exception to the trend, but the values involved are close. The *raw* row clearly shows the trend, which is reasonable if the features have no imposed structure to them. In other words, I take the *raw* row to represent what the classical models do relative to one another on their own, and the improving scores reflect the general trend of more-capable models.

Now consider the *rot*, *rot16*, and *rot32* rows. They break the trend such that the simpler nearest-centroid and naive Bayes models produce better results than the 3-NN and 7-NN models. This hints at some sort of structure in feature space, one that favors the simpler models over the nearest-neighbor models.

Nearest-neighbor models work by exploiting local structure in the feature space because they vote based on training samples nearby. Nearest centroid and Gaussian naive Bayes are best served by a global structure that groups the features around specific locations in feature

space. The scores in Table 16-1 hint that the RotNet-like models have produced such a global structure. We must more closely examine the features that our proxy models have generated to know more.

Examining RotNet Features

Let's use t-SNE plots and the purity score to assess the features created by the RotNet models. We've previously used these tools to evaluate the quality of *k*-means clustering (see Chapter 14).

We'll also add a new metric to the mix: intrinsic dimensionality (ID). Machine learning feature vectors have a specific dimensionality: the number of features in the vector. The MNIST images are 28×28 pixels, meaning an MNIST image exists as a vector in a 784-dimensional space. The 32×32×3 CIFAR-10 images, therefore, exist in a 3,072-dimensional space.

However, that isn't usually the reality. Yes, the data has a specific dimensionality, but data usually exists on a lower-dimensional manifold within the high-dimensional space. This is especially true for images, where the features are pixels and therefore highly redundant.

Imagine a 3D space where all the data exists on a plane within the space. The raw feature vectors have three dimensions, but, in effect, the data is only 2D. ID attempts to estimate the dimensionality of the data manifold within the higher-dimensional space of the feature vectors. We'll use the scikit-dimension (skdim) Python library to estimate ID. You'll find the documentation at <https://scikit-dimension.readthedocs.io> and can install it via the following:

```
> pip3 install pandas  
> pip3 install scikit-dimension
```

Note that the code in this section will still run even if you do not install skdim.

The `skdim.id` module includes a dozen ID estimation algorithms. We'll use the maximum likelihood estimation (MLE) approach. The specifics of an estimation algorithm are often rather technical, but it's helpful to know which algorithm is being used as results vary by

algorithm. Indeed, results are stochastic and can vary somewhat from run to run with the same algorithm. Randomness in machine learning runs deep.

Our analysis code is in the file *rot_tsne_plots.py*. I'll describe only what it does and leave code interpretation to you.

The code expects two items on the command line: the base directory where *raw.keras* and *model.keras* are found and, optionally, the text show to present the graphs along with writing them to disk.

The code first executes *rot_extract_features.py* to generate CIFAR-10 features using both the untrained and trained models. It then loads the test-set features and the CIFAR-10 test-set labels.

If `MLE` is successfully imported from `skdim.id`, the ID of the original CIFAR-10 images, the raw projected features, and the trained projected features are calculated. The `MLE` call returns a single floating-point number that is rounded to the nearest integer value.

The code then uses `TSNE` and `KMeans` from `sklearn` to create 2D points to represent the 2,048-dimensional space of the projected features and to learn 10 cluster centers, because there are 10 classes. If there are 10 wellrepresented clusters for the 10 classes, *k*-means should find them, and the purity scores should be higher (but don't get your hopes up).

Figure 16-2 presents two t-SNE plots showing the distribution of the untrained (left) and trained (right) features by class (symbols) for the base model learned by *rot.py* using 128 samples per minibatch and 20 epochs.

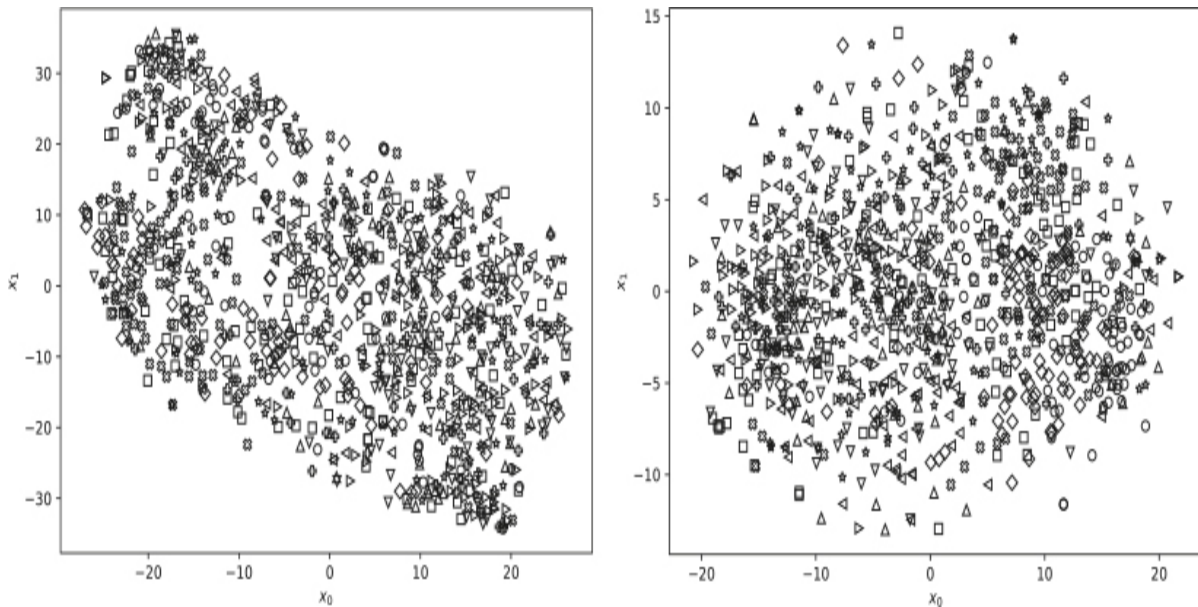


Figure 16-2: Two t-SNE plots showing untrained (left) and trained (right) features by class

Which symbols map to which classes is not essential, though present in the code, but it looks as though the trained plot has done a better job compared to the untrained plot of placing the symbols in groups. The untrained plot seems to have clusters, but the symbols in these clusters are quite mixed. The trained t-SNE plot is more spherical (a possible sign of a global structure?) and contains locations where many of the same symbol are present. This all hints at trained features that have a structure and are not merely randomly distributed in the 2,048-dimensional space.

After the plots, the code displays text like this:

Purity:

```
raw features      : 0.2140
pretrained features : 0.3150
```

Intrinsic dimension:

```
raw features      : 20
pretrained features : 54
original features  : 29
```

The closer the purity score is to 1.0, the more each of the 10 k -means clusters consists of feature vectors from the same class. As we

expect, the raw (untrained) clusters are less pure compared to those based on features extracted from the pretrained model.

The following text block shows us the intrinsic dimensionality of the untrained and trained datasets and the original dataset. These values might change slightly run to run, because the ID algorithm is stochastic, but they will usually hover around the same value.

The untrained features have the lowest dimensionality. This might be good in some ways, but not in this case because we know from Figure 16-2 that the untrained features are not grouped by class. The original features (images) live in a 29-dimensional space, and the untrained backbone model has projected those features to a 20-dimensional space without any meaningful structure. In a way, the untrained model has squished the data into a smaller space but in an uninformed manner.

Now consider the ID of the pretrained features. Those features are in an expanded 54-dimensional space. Projecting to a higher-dimensional space is precisely the thought behind an SVM, with the hope that (for a linear kernel) the features in the higher-dimensional space are more easily separated by hyperplanes. It seems much the same is happening here, given the better performance of the classical models when using the backbone model features.

Table 16-2 presents the purity and ID for each pretrained model's features.

Table 16-2: Purity and Intrinsic Dimensionality by Pretrained Model

Model	Purity	Intrinsic dimensionality
rot	0.3150	54
rot16	0.2470	62
rot32	0.3020	67
blend	0.2990	53
iproc 0	0.2430	52

Model	Purity	Intrinsic dimensionality
iproc 1	0.2460	35
raw	0.2140	25

I put 25 as the dimension for raw, not 20 as given earlier. Each untrained model was initialized differently, as we know is the case for neural networks, so the resulting projected features landed in spaces with different intrinsic dimensionalities. For the models here, most raw IDs were in the 25 to 27 range, hence using 25 in the table. The 20 for the RotNet model is merely a chance deviation from a distribution of possible untrained models that likely centers around producing features with 25 to 27 intrinsic dimensions.

Notice the ID for the rot16 and rot32 models. These are precisely the models where the simpler nearest-centroid and Gaussian naive Bayes classifiers outperformed nearest neighbors. The CIFAR-10 images were projected from an ID of 29 to 67, a higher-dimensional representation. As we remember, high-dimensional spaces make life difficult for nearest-neighbor models because the likelihood of the training set, unless massive, containing a sample relatively near any test sample is often extremely low, meaning the nearest neighbors are not representative of the test sample. The curse of dimensionality rears its ugly head once again.

The purity scores in Table 16-2, in conjunction with the intrinsic dimensionalities, hint at why the *rot.py* pretrained model was deemed “the winner” by our classic machine learning model test. That model has the highest purity score and one of the lowest IDs. Together, the two hint that the features it produced are better separated (grouped) in the new space while simultaneously not expanding that space so much that too many extraneous dimensions are introduced. The *blend.py* model is in much the same configuration with a purity of nearly 30 percent and an ID of 53. We declared that model to be in second place.

Finally, notice that the iproc model with the second set of features (varying levels of gamma correction) is in last place for the pretrained models and that it has one of the lowest purity scores and a dramatically lower ID of 35. The model hasn't, seemingly, projected the original CIFAR-10 features into a high-enough dimensional space to allow easy class separation by the classical machine learning models.

Overall, the analysis of this section gives credence to the claim that the pretrained models, especially that of *rot.py*, are indeed learning to represent the CIFAR-10 features in a way conducive to downstream tasks, which is exactly what we hoped would be the case.

Now that we've explored the features in a transfer-learning setting, let's focus on what happens when we move to a fine-tuning scenario.

Fine-Tuning Experiments

We'll use fine-tuning to adapt the pretrained RotNet model to classify CIFAR-10, much as we did with the VGG16 base model trained on Image-Net in Chapter 14. Everything in this section employs the backbone model produced by *rot.py* using 128 samples per minibatch and 10 epochs. We'll jump directly to the punchline and then back up to understand the setup.

Interpreting the Results

Figure 16-3 is the punchline. It bears a close resemblance to Figure 14-4's results (see page 388) for fine-tuning VGG16 pretrained on ImageNet. We have much to unpack here.

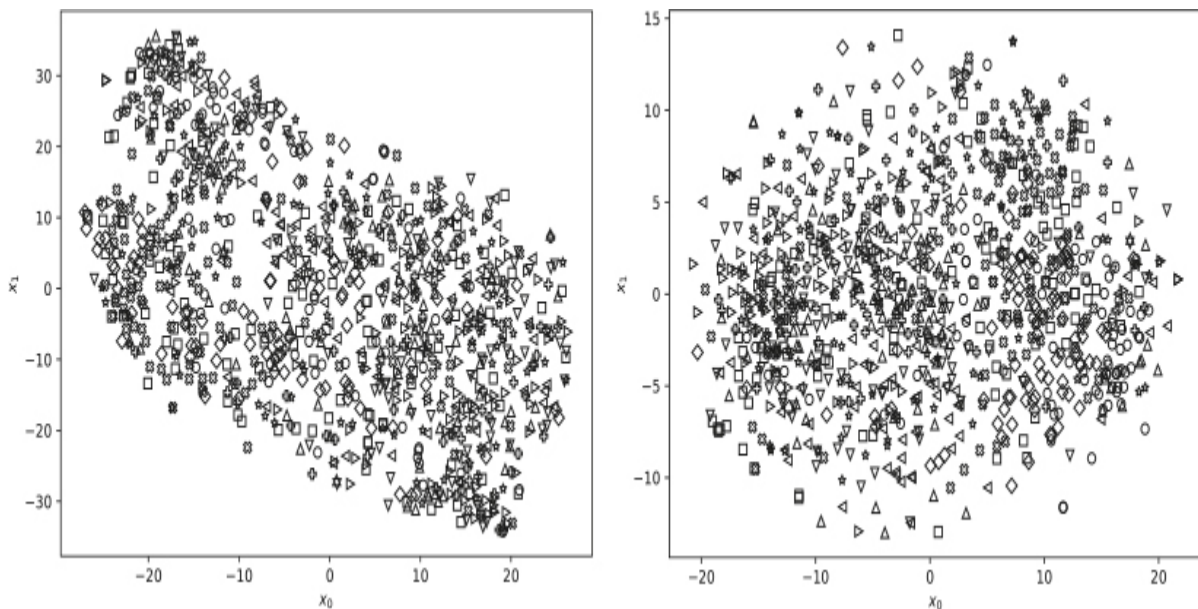


Figure 16-3: RotNet pretrained on CIFAR-100 fine-tuned to CIFAR-10

The plot shows mean test-set accuracy ($n = 6$) as a function of the percentage of CIFAR-10's training set used for fine-tuning. Let's begin with the four curves representing the accuracy achieved when freezing weights through block 2 (layer 8), block 3 (layer 12), block 3 using SGD instead of Adam, and training from scratch.

Training from scratch uses a randomly initialized model matching RotNet's backbone architecture. The curves contain error bars ($\pm$ SE), which are evident only when training from scratch. Notice, also, that the x-axis is nonlinear.

The previous observation about error bars says something about the repetitiveness of fine-tuning. This makes sense because training a model whose layers are all randomly initialized should lead to a broader variation in final model performance when compared to fine-tuning a model whose weights, especially in the lower layers, are not allowed to adapt.

Training from scratch produces less performant models when using less than the entire 50,000-sample training set. Fine-tuning freezing through block 3 is also, generally, better than using block 2. Fine-tuning with SGD and block 3 is likewise better than using Adam, reflecting Adam's known difficulty with small datasets. Notice, however, that SGD's performance falters when using the full training set. Again, this is

expected because we know Adam to be better at adapting in that case and that SGD often requires adjustment of the learning-rate schedule. The SGD-trained models in Figure 16-3 used a fixed learning rate of 0.005.

Now turn your attention to the 1 percent portion of the plot. Training from scratch produces the weakest model, as expected. The fine-tuning models are better, but not dramatically so. The two stars represent models trained using 10× augmentation, precisely as implemented in Figure 14-4 on page 388, freezing layers through block 3. Again, SGD outperforms Adam as the optimizer.

The augmented results present the mean test-set accuracy for models trained over 60 epochs by using a minibatch of 32 samples. Fine-tuning pretrained ImageNet in the same way produced a mean accuracy of 63 percent; see Table 14-1 on page 390. Fine-tuning with the RotNet backbone and SGD produced models averaging 52 percent accuracy. These results are reasonable; we expect the larger VGG16 backbone pretrained on ImageNet to be better at generating features than the small RotNet-pretrained backbone. However, the RotNet results are impressive given that purely unlabeled data and simple transformations were used to condition the model.

The RotNet backbone used a minibatch of 128 samples and trained for only 10 epochs. What happens if instead we pretrain with a 32-sample minibatch and, say, 30 or so epochs (or until overfitting)? I leave finding the answer to you. Let me know what you find.

Reviewing the Code

Figure 16-3 is the output of a series of scripts. The files *rot_fine-tune.py* and *rot_fine-tune_sgd.py* train models by fine-tuning the RotNet backbone. The latter is identical to the former, save for a switch of optimizer from Adam to SGD. The shell scripts *go_rot_fine-tune* and *go_rot_fine-tune_augment* call the Python scripts repeatedly to generate the output necessary to create Figure 16-3. Both are excellent examples of ugly cut-and-paste coding, but they get the job done. Finally, the file *rot_fine-tune_results.py* pulls the output together to produce Figure 16-3 along with some output text. Note, however, that the mean test-set

accuracies generated by the *augment* shell script have been manually entered into *rot_fine-tune_results.py*; see lines 112 and 116.

The output text presents t-test and Mann–Whitney U test results by training percentage to compare block 2, block 3, and from-scratch mean testset accuracies. They reflect what Figure 16-3 shows us visually: that block 2 and block 3 produce better models than from-scratch models.

If you examine the *go_rot_fine-tune* script, you'll notice that the number of training epochs increases as the fraction of the full CIFAR-10 dataset used decreases. Changing the dataset size but holding the number of epochs fixed results in fewer gradient-descent steps overall because we have fewer minibatches per epoch. To compensate and take (approximately) the same number of gradient-descent steps each time, the number of epochs increases from 12 to 120 and then 240. The compensation isn't one-to-one, but close enough to give us confidence that model performance is due, primarily, to the size of the training set.

Review the code in *rot_fine-tune.py* to follow the fine-tuning process. It mimics code from Chapter 14. Let's press on to explore another approach to self-supervised learning: Siamese networks.

Siamese Networks

Siamese networks, named after Chang and Eng Bunker—famous 19th-century conjoined twins originally from Siam (modern Thailand)—seek to learn a shared backbone network capable of transforming pairs of input images into embedded vectors. Similar images produce vectors near one another in the high-dimensional embedding space, while dissimilar images land far apart. Euclidean distance is the usual metric, though other options exist.

In this section, we'll experiment with a Siamese network as a self-supervised learner to condition the RotNet backbone network for downstream tasks.

Building and Testing the Siamese Networks

As Figure 16-4 illustrates, our Siamese network accepts two input images and passes them through a backbone network, conveniently the same backbone as RotNet.

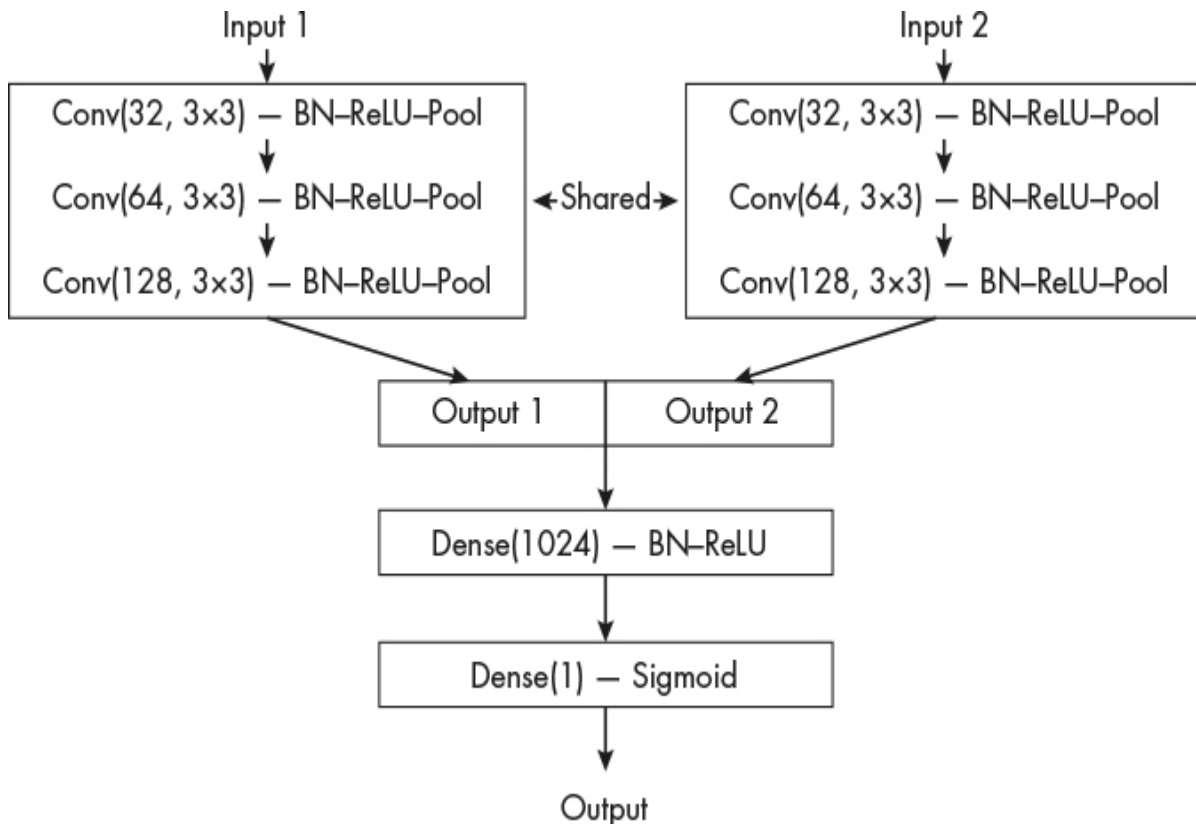


Figure 16-4: A Siamese network

Note the “shared” arrows. The backbone network shares weights so that we have a single model that is updated by gradients from the output of each input image during backpropagation. This encourages the network to learn representations that aid in discerning the similarity or dissimilarity of the two inputs.

In the diagram, the two outputs (Output 1 and Output 2) are concatenated, passed to a dense block with 1,024 nodes (and batch normalization and ReLU) before a final single-node dense layer with sigmoid activation produces the network’s scalar output. The output is the likelihood that the two inputs are different instances of the same image, or, more generally, are sufficiently similar when using a threshold like 0.5 to force a decision.

The RotNet backbone produces a single 2,048-element output vector (a flatten layer is present but not shown in Figure 16-4). Therefore, the concatenated output becomes a 4,096-element vector representing the backbone network's joint representation of the two input images. The top portion of the network is then asked if the single vector represents a pair of similar or dissimilar images.

Concatenation isn't the only option. We'll also experiment with a Siamese network that replaces the concatenated vector with the elementwise sum of Output 1 and Output 2.

Using Siamese Networks as Self-Supervised Learners

But how does a Siamese network's ability to learn similar and dissimilar help us with self-supervised learning from unlabeled data? RotNet and friends applied transformations to input images to create classes into which the model could learn to sort inputs. To use a Siamese network for self-supervised learning, we need a similar way to add a label to pairs of images.

Marking images as dissimilar is straightforward: Pick any two images from the unlabeled CIFAR-100 dataset and assign them to class 0. For class 1, which we want to reflect similar images, we might be initially tempted to pass the same image as Input 1 and Input 2, but that would be a mistake. The model won't learn meaningful low-level feature detectors if the task is that simple. Therefore, we'll pass the image through randomly applied image manipulations similar to those used to augment images. For consistency, we'll do the same to the dissimilar images so that both sets have similar distributions in a statistical sense. Not doing so would likewise make the proxy task too easy to learn.

The set of available image manipulations matters, as they force the model to learn convolutional filters detecting essential characteristics of the image—characteristics that we hope will be helpful downstream.

Implementing the Networks

We now have what we need to begin building the Siamese networks, one that concatenates (*siamese.py*) and another that adds (*siamese2.py*).

The first step is to construct the training set by using `SiameseDataset`, as Listing 16-2 shows.

```
def SiameseDataset(b, seed=359):
    np.random.seed(seed)
    x0, x1, y = [], [], []
    for i in range(len(b)):
        if (np.random.random() < 0.5):
            # positive pair
            x0.append(Augment(b[i]))
            x1.append(Augment(b[i]))
            y.append(1)
        else:
            # negative pair
            x0.append(Augment(b[i]))
            k = np.random.randint(0, len(b))
            x1.append(Augment(b[k]))
            y.append(0)
    np.random.seed()
    return np.array(x0)/255, np.array(x1)/255, np.array(y)
```

Listing 16-2: Constructing the Siamese training set

The function makes a single pass through each image in the unlabeled CIFAR-100 training set, randomly deciding to turn the image into a positive (similar) pair or a negative (dissimilar) pair. If positive, the same image is passed to `Augment` twice to return a pair of randomly augmented versions. If negative, a randomly selected image is used as the other pair member. While it's possible that the same image might be selected, the probability is low and we can live with the noise that might add to the dataset. Notice that `Augment` is used for negative pairs as well.

The `Augment` function is little more than a long `if...elif...else` construct that applies one, and only one, transformation from as simple as rotation to more-complex adjustments to the brightness, contrast, color, and sharpness of the image. Review the function for details.

The output from `SiameseDataset` consists of two sets of paired images, `x0` and `x1`, along with associated 0 or 1 labels in `y`.

As Listing 16-3 illustrates, the backbone network is the same as in RotNet and Figure 16-4.

```
def EncoderModel():
    inp = Input((32, 32, 3))
    _ = Conv2D(32, (3, 3), padding='same')(inp)
    _ = BatchNormalization()(_)
    _ = ReLU()(_)
    _ = MaxPooling2D((2, 2))(_)
    _ = Conv2D(64, (3, 3), padding='same')(_)
    _ = BatchNormalization()(_)
    _ = ReLU()(_)
    _ = MaxPooling2D((2, 2))(_)
    _ = Conv2D(128, (3, 3), padding='same')(_)
    _ = BatchNormalization()(_)
    _ = ReLU()(_)
    _ = MaxPooling2D((2, 2))(_)
    outp = Flatten()(_)
    return Model(inputs=inp, outputs=outp, name="encoder")
```

Listing 16-3: The Siamese network backbone

The `EncoderModel` function returns a model. To construct the remainder of the Siamese network, we must pass both inputs through the returned model, concatenate the outputs, and add the topmost layers. Listing 16-4 shows us how.

```
encoder = EncoderModel()

inp_a, inp_b = Input((32, 32, 3)), Input((32, 32, 3))
proc_a, proc_b = encoder(inp_a), encoder(inp_b)

_ = concatenate([proc_a, proc_b], axis=-1)
_ = Dense(1024)(_)
_ = BatchNormalization()(_)
_ = ReLU()(_)
outp = Dense(1, activation='sigmoid')(_)
model = Model([inp_a, inp_b], outp)
```

Listing 16-4: Building the Siamese network

First, `EncoderModel` is called to give us `encoder`, the model we want to save for downstream tasks. We then define two `Input` layers and use them with `encoder` to give us two output vectors, `proc_a` and `proc_b`, representing the backbone's representation of the paired input images.

The `concatenate` layer appends the two vectors side by side before adding the dense block and a final `Dense` layer with a single node and sigmoid activation. Previous models used a `Dense` layer followed by a `Softmax` layer. Keras lacks a specific sigmoid layer, so we must indicate it in the call to `Dense`.

We complete the full Siamese network by passing the two inputs, `inp_a` and `inp_b`, and the single output, `outp`, to `Model`. The encoder model is part and parcel of the larger Siamese model courtesy of the `concatenate` layer using both `proc_a` and `proc_b`. Here the flexibility of modern deep learning toolkits makes it possible in a few lines of code to do what would be a large, difficult coding project were we to attempt this process entirely from scratch.

The remainder of *siamese.py* follows our usual formula to compile the model (using Adam), and then train the model, as Listing 16-5 shows.

```
siamese.py
model.compile(loss='binary_crossentropy',
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])

history = model.fit([x0trn, x1trn], ytrn,
                  batch_size=batch_size,
                  epochs=epochs, verbose=1,
                  validation_data=([x0tst, x1tst], ytst))
```

Listing 16-5: Compiling and training the Siamese network

Notice that the `fit` call passes two inputs, `x0trn` and `x1trn`, along with the single label (`ytrn`) marking the relationship, 0 if dissimilar and 1 if similar.

The file *siamese2.py* replaces the concatenate layer with an Add layer:

```
_ = Add()([proc_a, proc_b])
```

However, this file is the same as *siamese.py* in all other respects.

Testing the Networks

Let's run *siamese.py* and then test the resulting pretrained backbone model with *classic_models.py*. Not much is required on the command line: minibatch size, number of epochs, and an output directory name. For example, the following will train the model for 10 epochs, using a minibatch of 128 samples, and place the results in the *siamese_128_10* directory:

```
> python3 siamese.py 128 10 results/siamese_128_10
```

My run produced the following:

```
Test set accuracy: 0.9772, MCC: 0.9552
[[2428  108]
 [   6 2458]]
```

This indicates that the Siamese network became quite good at detecting the difference between similar and dissimilar images. This result has two explanations. The model might have learned descriptive low-level detectors, allowing it to discriminate *same* and *different* accurately. Conversely, the task might have been too simple, allowing the model to learn *same* and *different* without parsing the input images in detail.

The resulting pretrained backbone model is identical to that pretrained by RotNet. Therefore, we can reuse some of the code from earlier in the chapter. To evaluate the Siamese network with classical models, run the following:

```
> python3 rot_extract_features.py results/siamese_128_10/model.keras 3 /tmp/qqq
> python3 classic_models.py /tmp/qqq_cifar10_features_train.npy
                           /tmp/qqq_cifar10_features_test.npy
```

My results are as follows, using the vector notation from earlier in the chapter:

(0.3851, 0.3963, 0.4018, 0.4275, 0.4696, 0.4934, 0.5682)

For comparison, a randomly initialized backbone model produced this:

(0.2798, 0.2606, 0.3227, 0.3399, 0.4338, 0.4507, 0.5367)

The results indicate that the Siamese network has forced the backbone to learn useful feature representations, but perhaps not as useful as those learned by the RotNet model. For example, the RotNet model pretrained for 10 epochs returned the following:

(0.4162, 0.4047, 0.4468, 0.4620, 0.5006, 0.5187, 0.6275)

This outperformed the Siamese network with every classical model.

Repeating the exercise with *siamese2.py* for 10 epochs produced these results:

(0.3354, 0.3335, 0.3685, 0.3856, 0.4487, 0.4626, 0.5549)

This doesn't instill confidence that addition is meaningfully better than random initialization, let alone concatenation. However, we remain undaunted and will press on to visualize the features learned by the Siamese network backbone models. Perhaps that will help us understand these results.

Examining Siamese Network Features

Our Siamese network passes each pair of images through the encoder portion to produce two 2,048-element vectors, which are then concatenated or summed. To visualize the features, we might produce the same sort of “bar code” representation as in Figure 9-8 on page 259, but the concatenated vector has 4,096 elements, which seems too long to lay out horizontally.

Therefore, let's transform each 2,048-element feature vector into an image of 32 rows and 64 columns. The concatenated feature vector becomes two 32×64 images that we'll place side by side. The summed feature vector becomes a single 32×64 image. We'll divide each image by its maximum value to scale it to [0, 1] before multiplying by 255 and truncating to the nearest integer, thereby creating a grayscale image that, we hope, illustrates the essence of each feature vector.

The code we need is in *siamese_features.py*. Run it without command line arguments to produce a *feature_images* directory with two subdirectories, *concat* and *sum*:

```
> python3 siamese_features.py
```

Each subdirectory contains 100 images. If the input pair is class 0, the name begins with *diff*; otherwise, it begins with *same*. Figure 16-5 shows representative output images.

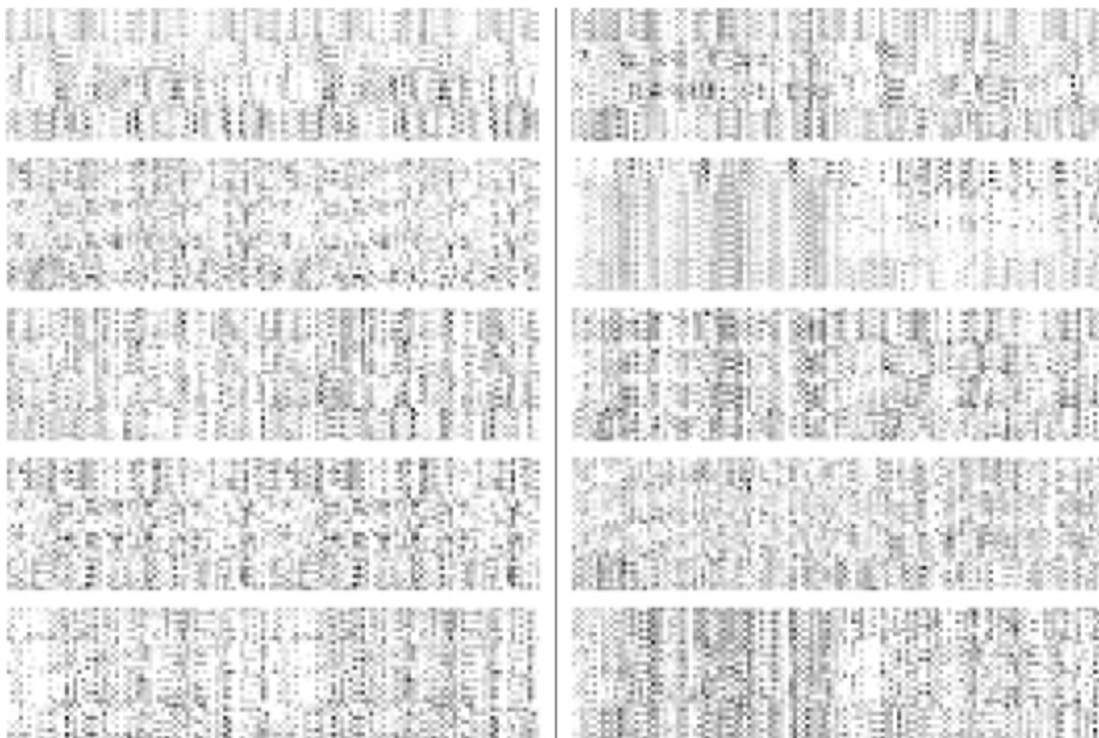


Figure 16-5: Visualizing concatenated Siamese backbone features: same (left) and different (right)

The left side of Figure 16-5 presents class 1 (same) input-pair features, while the right side presents different input-pair features. The differences are subtle in print, but on a computer screen, the similarities or differences are easier to discern, especially when scrolling through the images one after the other.

The distinction between same and different is even more subtle for the sum images, making them best viewed on a computer screen. Also note that the features in Figure 16-5 are from Siamese models trained for 20 epochs.

The subtlety of the feature pairs argues in favor of believing that the backbone model hasn't learned well to separate the two in the high-dimensional feature space of its output. In other words, it's not doing a stellar job of making similar input images land near each other while forcing dissimilar images to be far apart in the feature space.

Fine-Tuning Experiments

Earlier, we fine-tuned the RotNet backbone to classify CIFAR-10 images. We then experimented with the performance of fine-tuned models as a function of the training-set size. There's no reason we can't do the same with the Siamese network backbones. Indeed, they're identical to the RotNet backbone models, meaning we can use the scripts we previously developed rather than writing new ones.

The shell script, *go_siamese_fine-tune*, calls *rot_fine-tune.py*, passing it the Siamese network backbone models while varying the fraction of the CIFAR-10 training set used for fine-tuning. In all cases, the backbone model is frozen through block 3. The file *siamese_fine-tune_results.py* gathers the output of the shell script to produce a plot similar to Figure 16-3.

Run the code in order:

```
> sh go_siamese_fine-tune
> python3 siamese_fine-tune_results.py
```

The result is Figure 16-6.

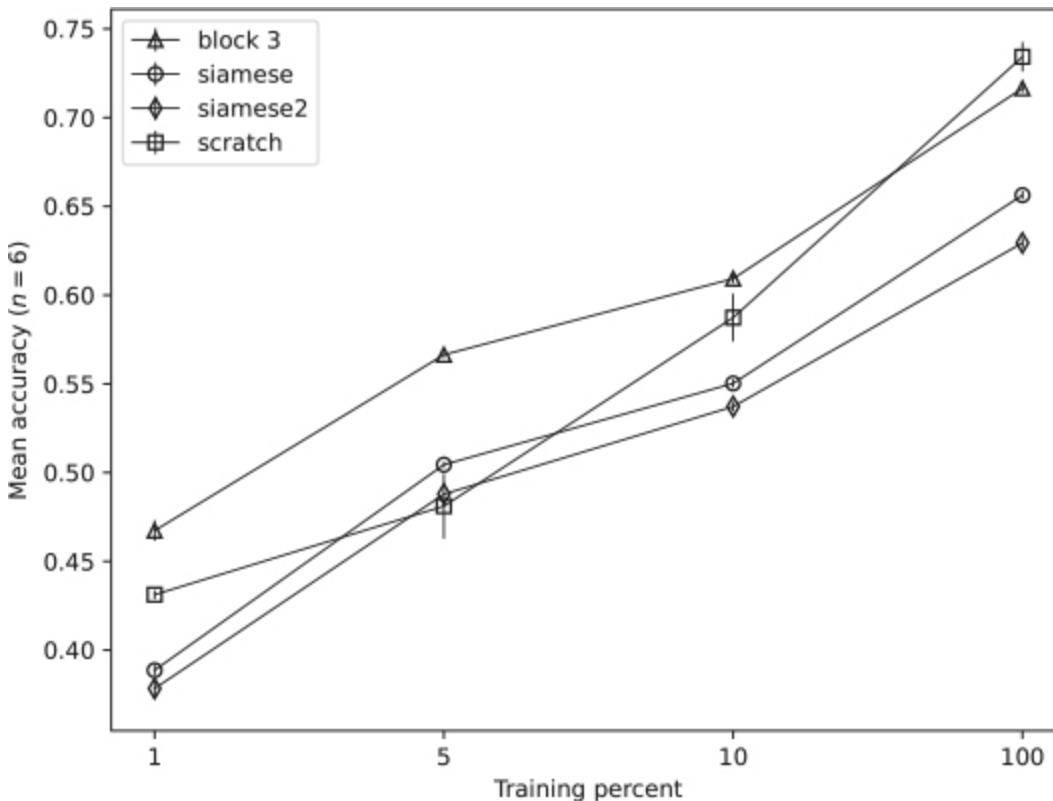


Figure 16-6: Fine-tuning the Siamese backbone models with CIFAR-10

The plot shows four curves: one for the RotNet backbone (block 3), one for training from scratch, and one each for the two Siamese network-trained backbones. It's immediately evident that the RotNet backbone was useful for this task, outperforming training from scratch for all training-set sizes save the largest (100 percent). Neither of the Siamese network backbones was anywhere near as performant and, save for the 5 percent case, didn't produce models even as good as training from scratch. Notice, also, that concatenation produced more-useful features compared to summation.

Figure 16-6 tells us that the Siamese network approach to self-supervised learning couldn't force the backbone to learn about image content sufficiently well to justify its use for pretraining in this case. The backbone model's small size may have contributed to its poorer performance compared to RotNet, which was surprisingly effective despite its simplicity.

For Further Exploration

This chapter introduced the main concepts behind self-supervised learning. Though our examples were necessarily simple, you learned that RotNet was effective. In practice, larger unlabeled datasets, advanced backbone models, and alternative loss functions are used.

If you wish to dive more deeply into this area, perhaps for a project that has scarce or prohibitively expensive labeled data but plentiful unlabeled data, I recommend exploring the following techniques.

SimCLR (Simple Framework for Contrastive Learning of Representations) is similar to this chapter's techniques, but it uses contrastive loss functions. A contrastive loss function seeks to do what we hoped our Siamese network would do: learn an embedding space where similar inputs land near each other and dissimilar inputs are well separated. One form of contrastive learning relies on the triplet loss:

$$L(A, P, N) = \max(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \text{margin}, 0)$$

Here A is an anchor image, P is a positive image (an alternatively augmented version of the anchor), N is a negative image dissimilar to the anchor, $f()$ is a neural network producing an embedding vector, and margin is a hyperparameter defining the threshold distance from the negative embeddings to the anchor embeddings.

Contrastive learning is used by more than SimCLR. You'll find MoCo (Momentum Contrast, with apologies to Spanish speakers) and BYOL (Bootstrap Your Own Latent) worthy of consideration. BYOL is unique in that it doesn't use negative image pairs. Both approaches, like SimCLR, depend critically on the augmentations applied to the unlabeled dataset.

Summary

In this chapter, we developed two self-supervised learning approaches that seek to build useful pretrained models from unlabeled data. The first, RotNet, was successful with our small datasets and models. The

proxy task predicts which 90-degree rotation has been applied to the input image.

Our second attempt used a Siamese network to produce a concatenated or summed vector representing the backbone model's manipulation of pairs of input images. This general approach is used for more-advanced models, but our example, while learning, wasn't in the final analysis any better than training from scratch, even with a training set only one hundredth the size of the full CIFAR-10.

To this point in the book, we've focused on building models that render a judgment about their inputs. The next chapter moves beyond classification to generative AI, models producing novel output on demand.

OceanofPDF.com

17

GENERATIVE ADVERSARIAL NETWORKS



In this chapter, we turn away from supervised learning models to focus on creating models that fall under the heading of *generative AI*—those that produce novel output on demand. In particular, we’ll implement and characterize generative adversarial networks (GANs). Developed by researcher Ian Goodfellow in 2014, GANs are perhaps the simplest form of generative AI using deep neural networks.

How GANs Work

The motivation behind GANs is a desire to train a network to produce novel output representative of the training data, even though the output itself is not in the training data. To do this, we implement a single network consisting of two subnetworks: the generator and the discriminator. Most applications are interested in the generator and discard the discriminator after training.

Training a GAN connects two contrary goals. The first is to teach the generator to produce output similar to the training data. The second is to teach the discriminator to differentiate between real training examples and fake examples produced by the generator.

At first, the discriminator's job is easy; the output of the generator is noise and quite unlike the training data. However, if we're successful (and lucky), the generator will learn to produce output that fools the discriminator most of the time. In the end, the discriminator's output ("real" versus "fake") will be no better than flipping a coin.

GANs come in two main varieties: unconditional and conditional. An *unconditional* GAN uses unlabeled training data and produces output without regard for class type. A *conditional* GAN learns to associate a desired class label with a randomly generated instance of that class. For example, an unconditional GAN trained on MNIST will produce digit images on demand, but a conditional GAN will produce only digits of a selected class. Both approaches follow Figure 17-1.

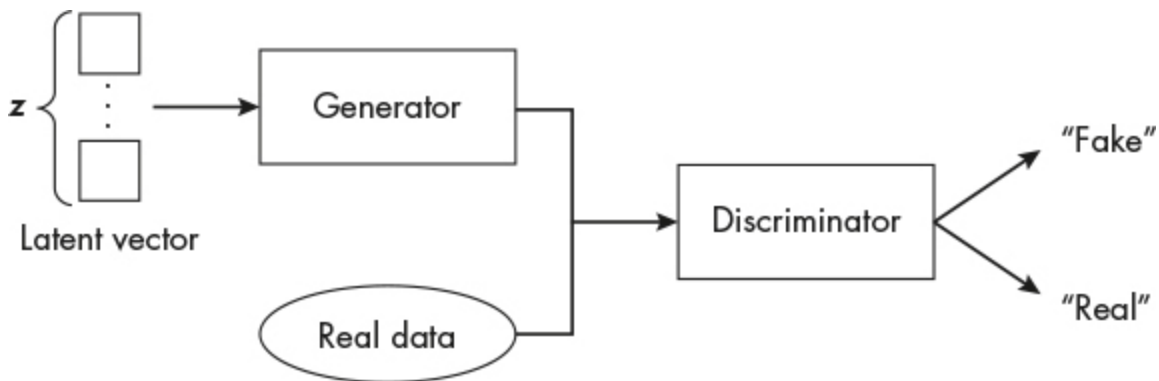


Figure 17-1: A block diagram of a GAN

The generator's input is $\mathbf{z}$, a *latent vector* drawn from a normal distribution with mean 0 and standard deviation 1. The generator maps this vector to an output that is a likely representative of the training set. Change $\mathbf{z}$, and a new output appears. The discriminator accepts real training instances or fake instances produced by the generator. The discriminator makes its judgment, and then backprop and gradient descent update the discriminator's weights and, ultimately, the generator's.

We begin with unconditional GANs before proceeding to the conditional case. In both cases, we'll write code to implement the GAN and then experiment with it. We'll use the MNIST dataset for most of the chapter's experiments to keep the examples manageable. However, along the way, you'll be introduced to the *Fashion MNIST (FMNIST)*

dataset, a drop-in substitute for classic MNIST containing 28×28-pixel grayscale images of items of clothing. FMNIST is a more challenging dataset to learn and was developed in answer to researchers' legitimate complaints that MNIST is too easy for modern models.

Unconditional GANs

Goodfellow et al.'s original 2014 paper, “Generative Adversarial Networks,” presents results for multiple datasets, including MNIST, using both fully connected MLPs and CNNs. In this section, we build similar models, beginning with an MLP-only GAN.

Building a GAN with Multilayer Perceptrons

As examining code is the best way to understand GANs, review the file *mnist_gan.py*. The code is presented in three parts: one to build the generator, another to build the discriminator, and a final part to put the two together as a complete model.

Building the Generator

GANs consist of two subnetworks, a generator and a discriminator, that we then piece together to act as one. Listing 17-1 shows how to build the generator.

```
inp = Input((LATENT,))
_ = Dense(256)(inp)
_ = LeakyReLU(0.2)(_)
_ = Dense(512)(_)
_ = LeakyReLU(0.2)(_)
_ = Dense(1024)(_)
_ = LeakyReLU(0.2)(_)
outp = Dense(784, activation='tanh')(_)
generator = Model(inputs=inp, outputs=outp)
```

Listing 17-1: Building the MLP-only generator

This generator uses dense, not convolutional, layers. We'll explore GANs using convolutional layers later in the chapter. Note also that I switched to `LeakyReLU`, a new activation function. Standard ReLUs return 0 if the input, x , is less than zero and x if the input is greater than zero. A *leaky ReLU* modifies the $x < 0$ condition to return αx , where $\alpha = 0.2$ in this case. In other words, the ReLU “leaks” a bit to preserve negative inputs instead of clamping them to zero. Leaky ReLUs address the *dying ReLU problem*, in which some neurons output zero for all inputs, thereby causing them to no longer participate in the learning process.

The input to the generator is a vector of `LATENT` elements. Many of our experiments involve modifying the number of elements in the latent vector from, say, 10 up to 160 or so.

Structurally, the generator maps an input vector of `LATENT` elements through hidden dense layers with 256, 512, and 1,024 nodes, respectively, before producing a 784-element output vector using a hyperbolic tangent activation function. The output vector is the generator's attempt at producing an MNIST digit. The activation function maps the output to $[-1, +1]$.

We must make two changes to the MNIST training images. First, as in previous chapters, we must flatten them into 784-element vectors. Second, we must scale the images to $[-1, +1]$ to match the range output by the generator. In code, these steps become the following:

```
xtrn = np.load("../data/mnist/mnist_train_images.npy").reshape((60000, 28*28))
xtrn = (xtrn - 127.5) / 127.5
```

We're using the images without associated labels. Unconditional GANs don't care about labels—only the parent distribution represented by the training data. Subtracting 127.5 from the pixel values before dividing by 127.5 maps $[0, 255] \rightarrow [-1, +1]$, as we require. A call to `Model` with `inp` and `outp` creates the generator.

Building the Discriminator

Listing 17-1 gave us the generator; Listing 17-2 presents the discriminator.

```
inp = Input((784,))
_ = Dense(1024)(inp)
_ = LeakyReLU(0.2)(_)
_ = Dropout(0.3)(_)
_ = Dense(512)(_)
_ = LeakyReLU(0.2)(_)
_ = Dropout(0.3)(_)
_ = Dense(256)(_)
_ = LeakyReLU(0.2)(_)
_ = Dropout(0.3)(_)
outp = Dense(1, activation='sigmoid')(_)
discriminator = Model(inputs=inp, outputs=outp)
adam = Adam(learning_rate=0.0002, beta_1=0.5)
discriminator.compile(loss='binary_crossentropy', optimizer=adam)
discriminator.trainable = False
```

Listing 17-2: Building the MLP-only discriminator

The discriminator accepts a 784-element input vector (image), then passes it through three `Dense` layers with 1,024, 512, and 256 nodes. Leaky ReLUs and dropout are used as well.

The discriminator's output is a single value passed through a sigmoid activation function. The call to `Model` builds the discriminator, followed by `compile` to configure the discriminator via binary cross-entropy and Adam.

The optimizer's learning rate is 0.0002, a typical value for GAN training. The `beta_1` parameter is also set to 0.5, down from its default of 0.9. The `beta_1` parameter acts like momentum in that Adam multiplies the previous minibatch moment value by β_1 and adds $(1 - \beta_1)$ times the current minibatch momentum. Setting $\beta_1 = 0.5$ tells Adam to split the new momentum value equally between the previous value and that from the current minibatch. Doing so makes the optimizer more responsive to the current minibatch, which is often helpful when training a GAN.

The final line in Listing 17-2 sets `trainable` to `False`. This step is critical and must be done *after* the call to `compile` for reasons that will become clear as we proceed.

Building the GAN

We have generator and discriminator. To build a working GAN, we must marry the two, as in Listing 17-3.

```
inp = Input((LATENT,))
_ = generator(inp)
outp = discriminator(_)
gan = Model(inputs=inp, outputs=outp)
gan.compile(loss='binary_crossentropy', optimizer=adam)
```

Listing 17-3: Building the full GAN model

The listing takes a noise vector input (`inp`), passes it through the generator, then passes that output through the discriminator. The discriminator's output and the generator's input form the `gan` model. A call to `compile` completes the full GAN.

Let's review what we've built so far:

1. A generator model was defined. The generator accepts a latent noise vector as input and produces a 784-element output, a fake image we hope the generator will learn to render well.
2. A discriminator model was defined. The discriminator accepts a 784-element vector representing either a real MNIST digit image scaled to $[-1, +1]$ or a fake image produced by the generator. The output is the discriminator's belief that the input is a real image (class 1).
3. The generator and discriminator are connected in the `gan` model. We need this connection in order to properly train the generator because updating the generator's weights relies on the discriminator's output. In other words, we must backprop through the discriminator to get to the generator to take a gradient-descent step.

Now that we have a GAN, how do we train it? The GANs of this chapter follow this algorithm:

1. For a specified number of minibatches:
 - (a) Select a half-minibatch of real images at random.
 - (b) Sample a half-minibatch of latent vectors from a standard normal distribution.
 - (c) Pass the latent vectors through the generator to return a half-minibatch of fake images.
 - (d) Take a gradient-descent step, with the discriminator using the sampled real and fake images.
 - (e) Sample a minibatch worth of latent vectors and pass them through the full GAN model by using class 1 as the label to take a gradient-descent step. In other words, treat the fake images as real.
2. Store the now trained generator for downstream use.

Steps (a) through (c) set up a minibatch split evenly between real and fake images, using the generator as it currently exists. Step (d) uses the minibatch with the discriminator to take a gradient-descent step. Only the discriminator is updated in step (d); the generator's weights are unmodified.

Step (e) uses the full GAN model to update the generator's weights. Here's where setting `trainable` to `False` after compiling the discriminator comes into play. We want to update the generator's weights without altering the discriminator's, but unlike the discriminator update (which we can do without affecting the generator), we must use the full GAN model to assess the discriminator's belief regarding the fake generator images. Notice that we tell the discriminator that all fake images are real. To update the generator, we need to know how well the discriminator performs in detecting the falsehood. This step is where the *adversarial* in *generative adversarial network* comes from.

Marking the discriminator model as untrainable after compiling the discriminator model instance tells Keras that subsequent uses of the discriminator model are to freeze the model's weights, but the discriminator model itself can still update the weights when used apart

from the full GAN model. This seeming contradiction is confusing at first because we appear to forbid the discriminator from updating its initial weights, but in reality, we update them in step (d) while treating them as frozen in step (e).

We know that Keras understands our intentions by calling `summary` on `gan`, the full GAN model:

Layer (type)	Output Shape	Param #
=====		
input_3 (InputLayer)	[(None, 40)]	0
model (Functional)	(None, 784)	1470992
model_1 (Functional)	(None, 1)	1460225
=====		
Total params: 2,931,217		
Trainable params: 1,470,992		
Non-trainable params: 1,460,225		

We're told that the first model has 1,470,992 parameters for an input of 40 elements. This is the generator configured to accept a 40-dimensional latent input vector. The next model, the discriminator, uses 1,460,225 parameters. Finally, we're told that of the 2,931,217 parameters in the full GAN, only 1,470,992 are trainable, with the remaining 1,460,225 frozen. In other words, the generator is trainable, but the discriminator is not, at least not via the full GAN.

Training the GAN

Listing 17-4 trains the GAN for a user-specified number of minibatches.

```
for k in range(1, minibatches+1):
    ❶ idx = np.random.randint(0, xtrn.shape[0], batch_size//2)
        imgs = xtrn[idx]

    ❷ noise = np.random.normal(size=(batch_size//2, LATENT))
    ❸ gen_imgs = generator.predict(noise, verbose=0)

    ❹ ones = np.ones((batch_size//2, 1))
```

```

zeros = np.zeros((batch_size//2, 1))
d_loss_real = discriminator.train_on_batch(imgs, ones)
d_loss_fake = discriminator.train_on_batch(gen_imgs, zeros)
D = 0.5 * (d_loss_real + d_loss_fake)

❸ noise = np.random.normal(size=(batch_size, LATENT))
ones = np.ones((batch_size, 1))
G = gan.train_on_batch(noise, ones)

print("Minibatch %5d: G=%0.8f, D=%0.8f" % (k, G, D), flush=True)
if (k == 1) or ((k % 200) == 0):
    GenerateSamples(k, generator, outdir)

generator.save(outdir+"/generator.keras")

```

Listing 17-4: Training an MLP-only GAN

First, randomly sample a half-minibatch of real images from the training data ❶. Then, sample a half-minibatch of latent vectors ❷ and use them to produce fake images ❸.

Next, use the sampled real and fake images to take a gradient-descent step with the discriminator ❹. We're using 1 as the real image label and 0 for the fake image. The gradient-descent step is taken by calling `train_on_batch`; we haven't used this method before. Previously, we trained models by calling `fit` to train for a given number of epochs. The `train_on_batch` method passes a single minibatch through the model before using backprop and gradient descent to update the weights. Because we marked `discriminator` as untrainable after calling its `compile` method, Keras knows to allow the weights to update when `train_on_batch` is called.

The `train_on_batch` method returns the loss. We use the method twice, first with the randomly sampled real images and again with the fake images. The total loss (`D`) is the average of the two.

Finally, we train the full GAN model by using a minibatch of latent vectors as input and a call to `gan.train_on_batch` ❺. Because the discriminator weights were marked frozen before incorporating the

discriminator into the full GAN, Keras will update only the generator weights while still using backprop through the discriminator to calculate the generator model gradients. The generator loss ($\mathcal{G}$) is returned.

Each minibatch displays the generator and discriminator losses and, after every 200 minibatches, `GenerateSamples` is called to produce a 10×10 array of digit images so we can monitor the GAN as it learns.

When all minibatches have been processed, the now trained generator model is stored and the discriminator discarded.

We have our first GAN. While it's tempting to jump into some experiments with our new toy, patience is a virtue. Let's hold off on using *mnist_gan.py* until you learn how to transform an MLP-only GAN into a deep convolutional GAN; then we'll experiment with both.

Building a GAN with Convolutional Layers

The file *mnist_cnn_gan.py* implements a deep convolutional GAN (DCGAN) for MNIST. The file is nearly identical to *mnist_gan.py*. The crucial difference lies in the architecture of the generator and discriminator, which replace the MLP layers with convolutional layers. Let's walk through the architecture.

Building the Generator

The generator aims to transform a latent input vector into an image. For the MLP-only GAN, we presented the image as a vector, meaning the generator had it easy: Map a vector to another vector. Classic MLPs do this well.

A DCGAN's generator has a more difficult task: Map a latent vector to a 2D grayscale image. To do that, we employ a new Keras layer, `Conv2DTranspose`, as Listing 17-5 shows.

```
inp = Input((LATENT,))
_ = Dense(7*7*LATENT)(inp)
_ = LeakyReLU(0.2)(_)
_ = BatchNormalization(momentum=0.8)(_)
_ = Reshape((7, 7, LATENT))(_)
_ = Conv2DTranspose(64, (3, 3), strides=(2, 2), padding="same")(_)
```

```

_ = BatchNormalization(momentum=0.8)(_)
_ = LeakyReLU(0.2)(_)
_ = Conv2DTranspose(64, (3, 3), strides=(1, 1), padding="same")(_)
_ = BatchNormalization(momentum=0.8)(_)
_ = LeakyReLU(0.2)(_)
outp = Conv2DTranspose(1, (5, 5), strides=(2, 2), padding="same",
                      activation="tanh")(_)
generator = Model(inputs=inp, outputs=outp)

```

Listing 17-5: Building the DCGAN generator

Let's work through this code. First, the generator accepts a latent input vector, precisely as the MLP-only GAN did. The latent vector is passed through a dense layer to transform it into an embedded vector with $7 \times 7 \times \text{LATENT}$ elements.

Our goal is to generate a 28×28 -pixel grayscale image, a $28 \times 28 \times 1$ tensor. To do so, we must first map the latent vector to a $7 \times 7 \times \text{LATENT}$ tensor, then use `Conv2DTranspose` twice, each time with a stride of 2×2 to double the spatial extent in each direction. If we do this beginning with a 7×7 tensor, we get a 14×14 tensor first and, finally, a 28×28 tensor, as required.

That being the case, it becomes clear that Listing 17-5 has in reality only four layers: a `Dense` layer to set up an embedded representation of the latent input vector with the proper number of elements for `Reshape` to turn into a $7 \times 7 \times \text{LATENT}$ tensor, and three `Conv2DTranspose` layers, two of which double the spatial extent of their input with 2×2 strides. The `Conv2DTranspose` layer with 1×1 strides does not alter the spatial extent of its input tensor but offers the model the opportunity to learn still more about the structure to be generated from the latent vector.

For example, for a 20-element latent input vector, the output of `Reshape` becomes a $7 \times 7 \times 20$ tensor, which in turn becomes a $14 \times 14 \times 64$ tensor after the first `Conv2DTranspose` layer and, finally, a $28 \times 28 \times 1$ tensor after the third `Conv2DTranspose` layer.

The generator finishes with a hyperbolic tangent activation function to produce fake images in the range $[-1, +1]$, as the MLP-only GAN did.

Notice that we use leaky ReLUs and batch normalization. Additionally, we tweak the `BatchNormalization` momentum value from the default of 0.99 down to 0.8 to use 80 percent of the running average mean and standard deviation along with 20 percent of the minibatch mean and standard deviation on each update. GAN training is finicky, and small changes make the difference between success and failure. Tweaks that might be irrelevant in other situations become potentially critical, especially for unconditional GANs.

Understanding the functionality of `Conv2DTranspose` is essential to understand the mapping from a latent vector to an output image. The `Reshape` layer changes a vector into a 3D tensor, in this case, a $7 \times 7 \times n$ tensor, with n the number of elements in the latent input vector. That much is little more than telling Keras how to interpret the vector's memory use.

The `Conv2DTranspose` layer is tasked with expanding the spatial dimensions of its input by a factor of two in height and width, using 3×3 kernels. The factor of two comes from striding by (2, 2).

In practice, `Conv2DTranspose` first expands the n -channel input by two in height and width, then learns 64 convolutional filters, each $3 \times 3 \times n$, to produce a new output tensor of dimensionality $14 \times 14 \times 64$. Recall that a convolution operation applies the 3D kernel over the input, then sums along the channel dimension to produce a new, single-channel activation map. Repeating this step 64 times produces the `Conv2DTranspose` layer's output tensor.

The layer is therefore attempting to learn a set of convolution filters that, suitable to the training data, work well at expanding the input tensor to another that is, spatially, twice the size in each dimension. The net effect is akin to applying an advanced image upscaling operation across the input tensor's channel dimension, but don't push that mental image too far. When learning each filter's 3D kernel, the summation across channels goes beyond applying an image-upscaling operation separately per channel.

Building the Discriminator

Now let's consider the discriminator. It accepts a 28×28×1 input image representing a real MNIST digit image or a fake image imagined by the generator. The output is a likelihood value, the discriminator's belief that the input is real (class 1). The discriminator's code is familiar, as in Listing 17-6.

```
image_inp = Input((28, 28, 1))
_ = Conv2D(32, (3, 3), strides=(2, 2), padding="same")(image_inp)
_ = LeakyReLU(0.2)(_)
_ = Conv2D(64, (3, 3), strides=(2, 2), padding="same")(_)
_ = LeakyReLU(0.2)(_)
_ = Flatten()(_)
_ = Dense(512)(_)
_ = LeakyReLU(0.2)(_)
outp = Dense(1, activation='sigmoid')(_)
discriminator = Model(inputs=image_inp, outputs=outp)
discriminator.compile(loss='binary_crossentropy',
                      optimizer=Adam(learning_rate=0.0002, beta_1=0.5))
discriminator.trainable = False
```

Listing 17-6: Building the DCGAN discriminator

This follows the CNN format we've used consistently throughout the book. You'll notice that the `conv2d` layers use 2×2 strides to reduce the spatial extent of the input, an alternative to convolution followed by pooling. Notice, also, that `conv2d` with 2×2 strides has the opposite effect of `Conv2DTranspose` with 2×2 strides; `conv2d` reduces the input tensor's spatial extent by a factor of two instead of doubling it. Finally, the `discriminator` model is declared untrainable after first compiling it.

The discriminator uses a sigmoid on the single output of the final `Dense` layer as the MLP-only GAN did. Suppose the dense layer is learning what amounts to the probability that the input is real. In that case, if the generator is well trained, the discriminator's output passed through the sigmoid will be close to 0.5 because the discriminator cannot reliably distinguish between real and fake. This is the condition we want.

The binary cross-entropy loss is for true label $y = 0$ or $y = 1$:

$$\text{loss} = -(y \log(p) + (1 - y) \log(1 - p))$$

This equals $-\log(0.5) = 0.693147 \dots$ when $p = 0.5$, regardless of y . Therefore, we should look for a discriminator loss approaching this value during GAN training. Similarly, we update the generator by passing a minibatch of fake images through the discriminator while claiming the images to be real. In that case, the loss is also $-\log(0.5)$, meaning the generator loss will likewise approach this value in the best of all worlds.

Training and Model Size

Listing 17-4 presented the training loop for the MLP-only GAN. The DCGAN training loop is nearly identical, differing only in the way we label the various minibatches. The MLP-only GAN used $y = 1$ for real images and $y = 0$ for fake, a perfectly reasonable approach. GAN training, however, is fickle enough that we should take advantage of tricks that have been demonstrated to work well, one of which is the use of soft labels.

A *soft label* is fuzzy; that is, instead of the hard label $y = 1$ for a real image, we'll use a random value in $[0.8, 1.2]$. Similarly, random values in $[0, 0.4]$ replace $y = 0$. Listing 17-7 shows an example.

```
for k in range(1, minibatches+1):
    idx = np.random.randint(0, xtrn.shape[0], batch_size//2)

    imgs = xtrn[idx]

    noise = np.random.normal(size=(batch_size//2, LATENT))
    gen_imgs = generator.predict(noise, verbose=0)

    ❶ ones = 0.8 + 0.4*np.random.random((batch_size//2, 1))
    zeros = 0.4*np.random.random((batch_size//2, 1))
    d_loss_real = discriminator.train_on_batch(imgs, ones)
    d_loss_fake = discriminator.train_on_batch(gen_imgs, zeros)
    D = 0.5 * (d_loss_real + d_loss_fake)
```

```

    noise = np.random.normal(size=(batch_size, LATENT))
    ❷ ones = 0.8 + 0.4*np.random.random((batch_size, 1))
    G = gan.train_on_batch(noise, ones)

    print("Minibatch %5d: G=%0.8f, D=%0.8f" % (k, G, D))
    if (k == 1) or ((k % 200) == 0):
        GenerateSamples(k, generator, outdir)

```

Listing 17-7: Training a DCGAN with soft labels

We use soft labels for the discriminator ❶ and the generator ❷. There is a significant difference in the size of the DCGAN and MLP-only GAN models. For example, calling `gan.summary` on the DCGAN for a 40-element latent vector produces the following:

Layer (type)	Output Shape	Param #
=====		
input_3 (InputLayer)	[(None, 40)]	0
model (Functional)	(None, 28, 28, 1)	150345
model_1 (Functional)	(None, 1)	1625473
=====		
Total params: 1,775,818		
Trainable params: 146,169		
Non-trainable params: 1,629,649		

This tells us that the full GAN uses 1,775,818 parameters split between the generator (150,345) and discriminator (1,625,473). The generator's batch normalization layers are responsible for the difference between the total number of parameters and those that are trainable (146,169).

The MLP-only GAN's parameter count exceeds DCGAN's by over 1 million parameters. This result is unsurprising and aligns with what motivated the development of convolutional networks in the first place.

Now that we have two unconditional GANs, let's put them to work.

Experimenting with Unconditional GANs

GAN performance is all about the *latent space*, the space representing the latent vector. Therefore, our implementations let us alter the size of

the latent vector to experiment with the overall effect. In this section, we'll content ourselves with merely running the code and examining the output for select latent vector sizes. Later in the chapter, we'll dive deeper into understanding what the model has or hasn't captured concerning the dataset we're asking it to mimic.

Running the GAN Code

Both *mnist_gan.py* (MLP-only) and *mnist_cnn_gan.py* (DCGAN) expect the same set of arguments on the command line:

```
> python3 mnist_gan.py
```

```
mnist_gan.py <latent> <batch_size> <minibatches> <outdir>
```

```
<latent>      - latent vector dimensionality (e.g. 40)
<batch_size>  - minibatch size (e.g. 128)
<minibatches> - minibatches (e.g. 10000)
<outdir>     - output directory (overwritten)
```

The latent vector dimensionality comes first. Since GANs transform latent vector values drawn from a normal distribution into images, it seems reasonable that the size of the latent vector will have some influence on how well we expect the model to learn the dataset's characteristics.

The minibatch size and number of minibatches are standard, as is the output directory location. Notice that we're counting minibatches here, not epochs. The minibatches are sampled at random from the pool of training data, though it's possible to alter the training loop to ensure that all training samples are examined.

Let's run both GANs three times by using latent vector sizes of 10, 40, and 80 elements. In each case, we'll use a minibatch size of 128 and train for 80,000 minibatches. MNIST contains 60,000 images in its training set, so each epoch would have about 500 minibatches ($60,000/128 = 468.75$); therefore, 80,000 minibatches are approximately 170 epochs' worth of passes through the dataset.

For example, run the following:

```
> python3 mnist_gan.py 80 128 80000 results/mnist_gan_latent_80_128_80000
```

This produces a long stream of output similar to this:

```
--snip--
Minibatch 96: G=0.86470425, D=0.45273758
Minibatch 97: G=0.92788154, D=0.44507859
Minibatch 98: G=1.04526675, D=0.40354147
Minibatch 99: G=1.04282331, D=0.40926055
Minibatch 100: G=0.95923871, D=0.44205946
--snip--
```

Each minibatch reports the loss for the generator (G) and the discriminator (D). Ideally, both losses should converge to $-\log(0.5) \approx 0.693147 \dots$, meaning the GAN isn't there yet. A discriminator loss of 0.44 tells us that the discriminator is currently better at detecting real images or, conversely, that the generator isn't outputting convincing fakes.

The corresponding generator loss of 0.96 backs up this claim, implying less than 50 percent certainty of class 1 assignment by the discriminator when handed a minibatch of fake images with class 1 (real image) labels. If the generator were effective, the discriminator would be fooled about half the time.

By the end of the training run, the output becomes the following:

```
--snip--
Minibatch 79996: G=0.77746654, D=0.68945271
Minibatch 79997: G=0.77704710, D=0.67096424
Minibatch 79998: G=0.76313800, D=0.68440142
Minibatch 79999: G=0.77275747, D=0.67482346
Minibatch 80000: G=0.75228316, D=0.66980129
```

This is a definite improvement, but not ideal. The generator is still producing fake images that the discriminator can detect more often than not, and the discriminator is likewise better than chance at distinguishing real from fake. Recall that the discriminator loss comes from a minibatch evenly split between real and fake images, while the generator loss comes from a minibatch of fake images declared real.

Figure 17-2 presents the generator (G) and discriminator (D) loss by minibatch for the MLP-only GAN with a latent vector of 40 dimensions.

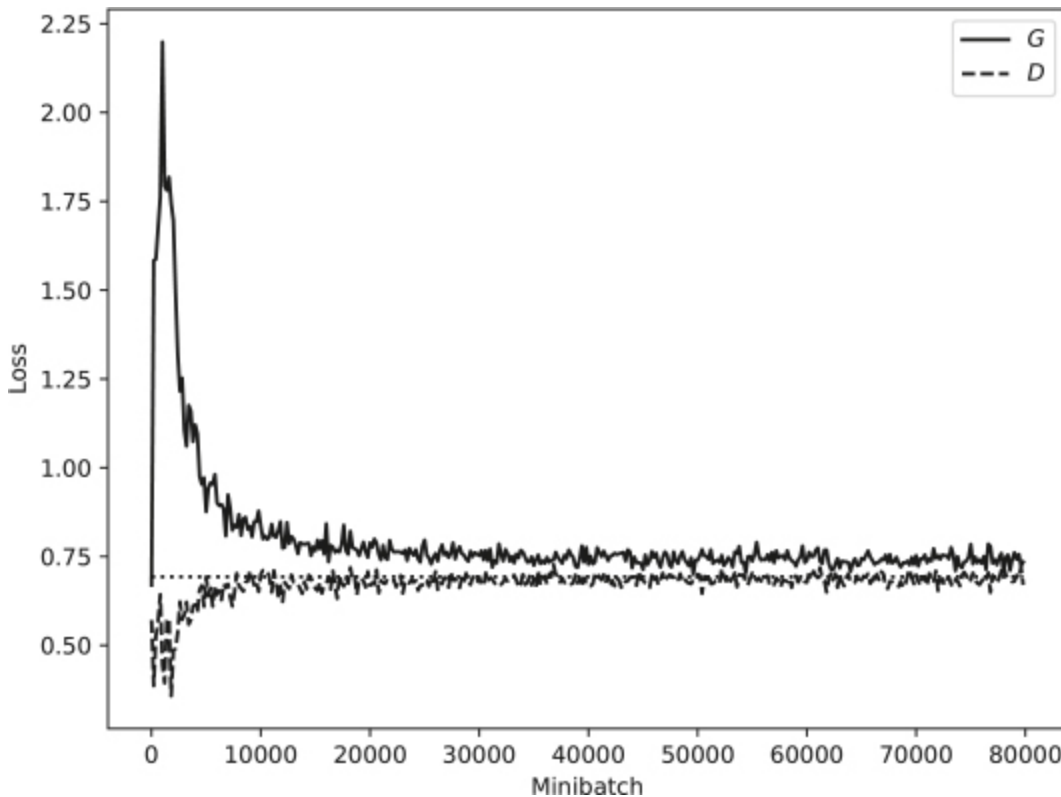


Figure 17-2: MLP-only loss by minibatch (latent = 40, minibatch size = 128)

The dotted horizontal line is $-\log(0.5)$. Over time, both losses approach the ideal value, but neither reaches it (especially the generator loss, which is always above). This indicates that the discriminator is still able, on average, to detect that the generator images are not real.

At first, neither the generator nor the discriminator are performing well, but a steady-state condition has been reached around minibatch 30,000. Though not ideal, this is evidence that the GAN has learned, and more importantly, has avoided mode collapse, the bane of generative adversarial models.

If the generator locates a few outputs, or even a single output, that consistently fool the discriminator, the generator will quickly learn to produce those outputs, thereby “winning” the adversarial game at the expense of diversity. We’re asking the generator to learn a latent space

that we can use to represent the training data well. If we think of this space in one dimension as a probability distribution, we might imagine a distribution with several modes, or “humps” around which particular sample values are more likely. These are the modes of the distribution. Mode collapse is akin to learning one of these modes at the expense of all the others. Yes, the generator is winning by fooling the discriminator, but it does so by becoming a one-trick pony and not by learning what we intended.

Interpreting the Output

Figure 17-3 shows the output produced by the MLP-only GAN for various dimensional latent vectors.



Figure 17-3: The generated MLP-only digits for latent vector dimensions 10, 40, and 80 (left to right)

All models used 128-sample minibatches and were trained for 80,000 gradient-descent steps. I used `generate_unconditional_mosaic.py` to produce the images, as in Listing 17-8.

```

model = load_model(mname)
image = np.zeros((280, 28*16), dtype="uint8")

for d in range(10):
    for i in range(16):
        noise = np.random.normal(size=(1, latent_dim))
        fake = model.predict(noise, verbose=0)
        img = (255.0*(fake[0,:]+1)/2).astype("uint8").reshape((28, 28))
        image[(d*28):(d*28+28), (i*28):(i*28+28)] = img

```

```
Image.fromarray(255-image).save(online)
```

Listing 17-8: Using a trained unconditional GAN generator

The code loads a generator (given by `generator`), defines a NumPy array large enough to hold 10 rows of 16 MNIST-sized digits, then loops to sample a latent vector (`noise`), and passes that vector to the model to produce a fake image, which is rescaled and inserted into the output array. The final image is inverted to be black on white.

All three latent vector sizes produce realistic MNIST digits with occasional oddball shapes that are merely digit-like. Subjectively, however, when I look from left to right, I see a steady increase in the diversity of digit shapes, particularly in the range of digit types and their appearance. If you consider each digit array as a whole without discerning individual digits, the 10-element latent vector array has a vertical orientation that is missing in the 80-element latent array. That array is more organic in its feel, and the digits appear more hand-drawn and less rigid (at least to me).

The file *mnist_gan_40.mp4* is a movie from the images generated during training for a 40-element latent vector and 80,000 minibatches. It's fascinating to watch the GAN learn, frame by frame, becoming ever more noise-free and realistic. Figure 17-4 shows similar frames from a separate 40-element training run.



Figure 17-4: The results of training an MLP-only GAN: iterations 1 and 2,000 (top); iterations 30,000 and 80,000 (bottom)

What about DCGAN? Figure 17-5 presents digits generated by DCGAN models using 128-sample minibatches trained for 12,000 gradient-descent steps.



Figure 17-5: The generated DCGAN digits for latent vector dimensions 10, 80, and 160 (left to right)

The smaller CNN model is expected to learn faster than the MLP-only model, so fewer iterations should be necessary.

Figure 17-5 presents digit arrays for latent vector sizes of 10, 80, and 160 elements. In theory, DCGAN should be able to learn more-detailed

local features of the digits, implying that a larger latent vector might be helpful as opposed to the more global interpretation possible with the MLP-only GAN. Recall that the inability of an MLP to focus on local characteristics of the input, especially for images, is part of what motivated CNN development.

Many of my comments about Figure 17-3 apply to Figure 17-5. The 10-element latent vector leads to a model that learns a lower diversity of digit shapes. The model also has a great love of the number 1 and 9s that look like 1s—all hallmarks of a model that has learned only a few modes of the full dataset. With 160 elements in the latent vector, subtle digit features are now possible, leading to digits that look more organic.

It may be difficult to notice in print, but if you train the MLP-only and DCGAN models, you'll see that the MLP-only digits are noisy; they often have extraneous pixels that shouldn't be there. The DCGAN digits are noise-free for the most part. This observation lends credence to my claim about what DCGAN can learn and why it benefits from a larger latent vector.

Classifying Digit Frequency

Contemplating digit arrays raises a question: Do the GANs generate digit types with equal frequency, or at least with a frequency approximating the training data's digit frequency? The file *mnist_digit_freq.py* uses a high-quality MNIST CNN to classify digits generated by an unconditional GAN. The CNN is equivalent to the model used in Chapter 10; see the file *mnist_cnn_exp0.py*.

The first execution of *mnist_digit_freq.py* will train the CNN. Subsequent runs will use the trained model produced by the first run. The command line arguments are as follows, where *<generator>* is a trained unconditional generator and *<latent>* is the number of elements in the latent noise vector:

```
mnist_digit_freq <generator> <latent>
```

```
<generator> - trained MNIST generator to test
```

```
<latent>    - latent vector dimensionality
```

The code generates 10,000 digits and then passes them to the classifier to assign each digit to a class. When done, the frequency of each digit is displayed along with the Euclidean distance between the generator's digit frequencies and those of the actual MNIST training set.

For example, the 10-element latent vector version of the MLP-only GAN produced the following digit frequencies (0 to 9):

(0.1150, 0.1131, 0.0882, 0.0930, 0.1021,
0.0895, 0.0950, 0.1138, 0.0990, 0.0913)

The true training-set frequencies are shown here:

(0.0972, 0.1124, 0.0993, 0.1022, 0.0974,
0.0904, 0.0986, 0.1044, 0.0975, 0.0992)

Treating both as vectors in a 10-dimensional space returns a Euclidean distance between them of 0.02572.

The Euclidean distance has meaning only if we expect a well-trained GAN to produce digit frequencies approaching those of the training set. Thinking in terms of a parent distribution that produced the training set we are attempting to model with the GAN, it makes sense to believe that a well-performing GAN will, if the latent space is capable of supporting it, produce digits with a similar frequency. Table 17-1 presents the digit frequencies and associated Euclidean distances for our unconditional GAN models.

Table 17-1: GAN Digit Frequencies

Model	Digit frequencies	Distance
MNIST	(0.097, 0.112, 0.099, 0.102, 0.097, 0.090, 0.099, 0.104, 0.098, 0.099)	—
MLP 10	(0.115, 0.113, 0.088, 0.093, 0.102, 0.090, 0.095, 0.114,	0.026

Model	Digit frequencies	Distance
	0.099, 0.091)	
MLP 40	(0.107, 0.106, 0.096, 0.986, 0.107, 0.087, 0.097, 0.119, 0.087, 0.096)	0.024
MLP 80	(0.103, 0.109, 0.096, 0.095, 0.105, 0.090, 0.107, 0.096, 0.098, 0.102)	0.017
DCGAN 10	(0.105, 0.154, 0.050, 0.089, 0.077, 0.058, 0.064, 0.141, 0.109, 0.153)	0.107
DCGAN 80	(0.106, 0.106, 0.072, 0.096, 0.116, 0.071, 0.090, 0.101, 0.127, 0.115)	0.053
DCGAN 160	(0.115, 0.103, 0.067, 0.100, 0.100, 0.071, 0.094, 0.106, 0.136, 0.110)	0.058

Interpreting this table requires care, but we can make some general statements. First, a priori, we shouldn't expect complete agreement in the digit-generation frequency between the full dataset and the models. After all, the dataset itself is only a sample from the parent digit-generation process, meaning, in this case, the variety of ways humans draw digits.

However, the MLP GAN models are much closer in digit distribution to the original dataset than the DCGAN models. This result aligns with the notion that MLP-based models learn the general characteristics of their dataset, including prior class probabilities. The Euclidean distance between MLP-only GAN digit frequency and the original dataset decreases as the latent vector size increases; this is also in line with expectations regarding the ability of the model to learn a sufficiently expressive mapping between the latent space dimension and

fake digit characteristics. Visually, the 80-dimensional latent vector model produced the more realistic digits.

Now consider the DCGAN digit frequencies. They deviate even more from those of the original data and roughly follow a trend of larger latent space producing distributions closer to the datasets'. However, again, we need not expect the digits to be in exactly the same distribution, and the larger Euclidean distances seem to match expectations if we consider CNN-based models to be learners of local dataset characteristics more so than global characteristics.

The GANs produce digits that certainly appear to be MNIST-like. In other words, both model types learn. GAN training is fastidious and prone to mode collapse, so I don't doubt that we can significantly improve these models with effort.

For example, the DCGAN model with a 160-dimensional latent vector space was visually the best, but it was trained for a mere 12,000 minibatches. Figure 17-6 shows the 12,000-minibatch result followed by models trained for 24,000 and 36,000 minibatches.



Figure 17-6: The generated DCGAN digits for 160-dimensional latent vectors for 12,000, 24,000, and 36,000 iterations (left to right)

Improvement between 12,000 and 24,000 iterations is immediately evident. By 36,000 iterations, the digits look even nicer, but subjectively, they also appear to be lacking somewhat in diversity—a possible sign of overfitting.

The first GANs used multilayer perceptrons. That this works at all (at least for simple datasets like MNIST) is somewhat remarkable, and makes me wonder what might have happened if GANs had been

discovered a few decades earlier, as they seemingly weren't. The recent explosion of powerful generative AI systems, some techniques of which we'll explore in Chapter 18, began with the humble, MLP-only GAN.

While unconditional GANs produce realistic members of the parent distribution they were trained to mimic, they are unguided. For example, we cannot ask the GAN to produce only 4s. The following section remedies this shortcoming.

Conditional GANs

Conditional GANs accept two inputs: a latent noise vector and a one-hot vector specifying the desired class. If properly trained, the output from the generator is a sample of the specified class. Let's implement a conditional GAN using convolutional layers and follow that with an exploration of its capabilities.

Implementing a Conditional GAN

A conditional DCGAN is structurally similar to an unconditional DCGAN, with the primary difference lying in the inputs to the generator and discriminator. I recommend reading through the code in *mnist_conditional_gan.py* before continuing.

Let's begin with the generator, as in Listing 17-9.

```
latent_inp = Input((LATENT,))
label_inp = Input((num_classes,))
merged_inp = Concatenate()([latent_inp, label_inp])
gen_channels = LATENT + num_classes

_ = Dense(7*7*gen_channels)(merged_inp)
_ = LeakyReLU(0.2)(_)
_ = Reshape((7, 7, gen_channels))(_)
_ = Conv2DTranspose(64, (3, 3), strides=(2, 2), padding="same")(_)
_ = LeakyReLU(0.2)(_)
_ = Conv2DTranspose(64, (3, 3), strides=(2, 2), padding="same")(_)
_ = LeakyReLU(0.2)(_)
```

```
outp = Conv2D(1, (7, 7), padding="same", activation="sigmoid")(_)
generator = Model(inputs=[latent_inp, label_inp], outputs=outp)
```

Listing 17-9: The conditional DCGAN generator

First, we still have a LATENT vector sampled from a standard normal distribution. As before, we'll focus our experiments on how the dimensionality of the latent vector affects the generator's output.

The latent vector input is in `latent_inp`. The point of a conditional GAN is to learn to generate class-specific output, so we add a second input to the generator, `label_inp`, a one-hot vector indicating the desired class label. We have 10 classes, so the input is a 10-element vector.

The latent vector and one-hot class vector are concatenated as `merged_inp`, a vector of LATENT + 10 dimensions. This vector is then passed to the generator model. Structurally, the generator is identical to the unconditional case once we have `merged_inp`, with one important exception: The final output is passed through a sigmoid activation, not a hyperbolic tangent. This means that the generated images exist in $[0, 1]$, not $[-1, +1]$, as before.

The discriminator comes next, as Listing 17-10 shows.

```
image_shape = (28, 28, 1)
image_inp = Input((28, 28, 1))
label_embedding = Dense(np.prod(image_shape))(label_inp)
label_embedding = Reshape(image_shape)(label_embedding)
mod_image = Multiply()([image_inp, label_embedding])
_ = Conv2D(64, (3, 3), strides=(2, 2), padding="same")(mod_image)
_ = LeakyReLU(0.2)(_)
_ = Conv2D(128, (3, 3), strides=(2, 2), padding="same")(_)
_ = LeakyReLU(0.2)(_)
_ = GlobalMaxPooling2D()(_)
outp = Dense(1, activation="sigmoid")(_)
discriminator = Model([image_inp, label_inp], outp)
discriminator.compile(loss='binary_crossentropy',
                      optimizer=Adam(learning_rate=0.0002, beta_1=0.5))
discriminator.trainable = False
```

Listing 17-10: The conditional DCGAN discriminator

The discriminator is where the conditioning implied in “conditional GAN” enters the scene. The expected $28 \times 28 \times 1$ input grayscale image, either real or fake, is there, but it’s affected by a new embedding (`label_embedding`) generated from `label_inp`, an Input derived from a one-hot class-label vector. The Dense layer maps the 10-dimensional class-label vector into a $28 \times 28 \times 1 = 784$ -element vector, which is then reshaped back into a $28 \times 28 \times 1$ tensor. Review the first four lines in Listing 17-10 to convince yourself that the final tensor represented by `label_embedding` has the same dimensions as the input image.

Conditioning the Generator

We have the image and a reshaped embedding constructed from a one-hot class-label vector. Next, we do something that at first blush appears nonsensical: We multiply the two together, pixel by pixel, to create `mod_image`, the actual input to the main portion of the discriminator.

Multiplying the input image by the reshaped embedding produced from the class label modulates the input image by a representation of the expected class label. A good analogy comes from AM radio. In AM (amplitude modulation), there is a carrier wave at the station’s frequency (say, 620 kHz). The audio signal is superimposed on the carrier by altering the amplitude of the carrier as a function of time.

In a conditional DCGAN, multiplying the image by the embedding produced from the class label acts similarly. The image takes on the role of the carrier wave, and the embedding takes on the role of the audio signal. Only, in this case, the modulation is spatial, not temporal.

Modulating the image by the label embedding allows the discriminator to pay attention to two things at once: whether the image is real or fake, and whether the image is a good instance of the class represented by the label embedding. As such, the discriminator now measures the generator’s ability to become good not just at image generation but image generation appropriate to the given class label.

We should pause here for a moment to appreciate the power of this approach and, more generally, the power of all deep learning approaches. The model’s architecture acts as a framework from which a

model with the desired abilities can emerge from the data on which it is trained. The architecture limits what can and cannot be learned, according to its size and structure; it's up to the data and training process to cause the desired functionality to blossom and grow.

Deep learning viewed in this way is analogous to biological evolution. The environment sets the limits of what can evolve, and how it evolves, in response to environmental changes. In deep learning, the architecture acts as the environment to frame the behaviors that can emerge. AI capability exploded in the fall of 2022 when LLMs (the subject of Chapter 18) reached a capacity where the potential inherent in their architecture could emerge from the data on which they were trained.

Building the Conditional GAN

Let's return to Listing 17-10. Prior to the final single-node dense layer before the sigmoid activation, there is a `GlobalMaxPooling2D` layer. We first encountered such layers in Chapter 13. Its purpose here is to select the largest value in each of the 128 7×7 feature maps passed to it by the final `conv2D` layer to produce a 128-element vector leading to the discriminator's output value. As before, the discriminator's output is its belief that the input is a member of class 1, a real image. Also as before, the discriminator is declared untrainable after first calling its `compile` method, so we might use it to update the generator without altering its weights yet train it via `train_on_batch` when necessary.

We have yet to piece the generator and discriminator together to form the full conditional GAN model, as Listing 17-11 illustrates.

```
gan_input = [Input((LATENT,)), Input((num_classes,))]  
fake_image = generator(gan_input)  
gan_output = discriminator([fake_image, gan_input[1]])  
gan = Model(gan_input, gan_output)  
gan.compile(loss="binary_crossentropy",  
            optimizer=Adam(learning_rate=0.0002, beta_1=0.5))
```

Listing 17-11: Building the full conditional GAN model

The generator accepts `gan_input`, a list of Input objects, one for the latent vector and another for the one-hot class label. The output of the generator becomes an input to the discriminator along with the same one-hot class label passed to the generator (`gan_input[1]`). The discriminator's output is the full GAN's output, and the generator's input is the full GAN's input. Finally, the full GAN and the discriminator separately use binary cross-entropy for the loss and an instance of the Adam optimizer with the same learning rate and `beta_1` as were used in the unconditional case.

Training a conditional GAN is only slightly more complicated than training an unconditional GAN, as Listing 17-12 demonstrates.

```
for k in range(1, minibatches+1):
    # Select a random half batch of real images.
    idx = np.random.randint(0, xtrn.shape[0], batch_size//2)
    imgs, rlabels = xtrn[idx], ytrn[idx]

    # Sample noise and use the generator.
    noise = np.random.normal(size=(batch_size//2, LATENT))
    labels = np.random.randint(0, num_classes, batch_size//2)
    labels = keras.utils.to_categorical(labels, num_classes)
    gen_imgs = generator.predict([noise, labels], verbose=0)

    # Train the discriminator.
    ones = np.ones((batch_size//2, 1))
    zeros = np.zeros((batch_size//2, 1))
    d_loss_real = discriminator.train_on_batch([imgs, rlabels], ones)

    d_loss_fake = discriminator.train_on_batch([gen_imgs, labels], zeros)
    D = 0.5 * (d_loss_real + d_loss_fake)

    # Now train the generator conditioned on the labels.
    noise = np.random.normal(size=(batch_size, LATENT))
    labels = np.random.randint(0, num_classes, batch_size)
    labels = keras.utils.to_categorical(labels, num_classes)
    ones = np.ones((batch_size, 1))
    G = gan.train_on_batch([noise, labels], ones)
```

```
print("Minibatch %5d: G=%0.8f, D=%0.8f" % (k, G, D), flush=True)
if (k == 1) or ((k % 200) == 0):
    GenerateSamples(k, generator, outdir)
```

Listing 17-12: Training the conditional GAN

The process matches Listing 17-7, though we're not using label smoothing in this case. I doubt it would hurt, but the added driver of a selected class label seems to work well enough without it.

The half-minibatch of real images is now accompanied by their associated true labels (`rlabels`). This is reasonable, but what should we use for the labels assigned to the half-minibatch of fake images (`gen_imgs`)?

We don't have actual labels, but that won't stop us. We pick the latent noise vectors at random, so why not do the same for the label? Since the label modulates the generator's fake image, we need to pass many examples of each class to the generator. Picking a random integer in $[0, 9]$ that is then turned into an appropriate one-hot vector will do precisely that in the long run.

Think of a conditional GAN as a collection of unconditional GANs partitioned by class label. We pick a class label and a latent noise vector, and once trained, the generator knows how to use both together to produce a sampled image of that class.

GAN training continues for the user-specified number of minibatches before storing the generator model for future use (not shown in Listing 17-12).

Exploring Alternative Conditioning

We're almost ready for some experiments. You might have noticed a file named `mnist_conditional_gan2.py`. The only difference between this file and the GAN just presented is in the generator. The generator of Listing 17-9 uses `Conv2DTranspose` to upscale the fake image. This layer works, but interspersing 0s between input tensor values causes a grid pattern that is most evident at the beginning of training.

As an alternative, we might replace the `Conv2DTranspose` layers with a pair of layers. The first, `UpSampling2D`, uses nearest neighbors to double the spatial extent of its input. We used such layers when we studied the U-net architecture in Chapter 15. Upsampling gives us the necessary increase in spatial extent, but the layer has no learnable parameters. Therefore, upsampling is followed by a standard convolutional layer to learn a set of filters that help to condition the input tensor to take on the appearance of whatever the generator is supposed to produce. So, in the file, you won't find this:

```
_ = Conv2DTranspose(64, (3, 3), strides=(2, 2), padding="same")(_)
```

Instead, that line is replaced with the following:

```
_ = UpSampling2D((2, 2))(_)  
_ = Conv2D(64, (3, 3), padding="same")(_)
```

In the experiments that follow, I'll focus on the `Conv2DTranspose` approach and leave examining this alternative to your copious free time.

Our conditional GAN has a bit of a Rube Goldberg feel to it. Let's find out if it works.

Experimenting with Conditional GANs

Let's run both *mnist_conditional_gan.py* and *fmnist_conditional_gan.py*. The former generates MNIST digits, while the latter generates FMNIST images of articles of clothing. We need not construct the FMNIST dataset; it's part of TensorFlow/Keras and is a drop-in substitute for MNIST:

```
(xtrn, ytrn), (xtst, ytst) = keras.datasets.fashion_mnist.load_data()
```

Both files expect the same command line arguments, precisely those of the unconditional GANs:

```
> python3 mnist_conditional_gan.py
```

```
mnist_conditional_gan.py <latent> <batch_size> <minibatches> <outdir>
```

<latent> - latent vector dimensionality (e.g. 100)
<batch_size> - minibatch size (e.g. 128)
<minibatches> - number of minibatches (e.g. 10000)
<outdir> - output directory (overwritten)

For example, I used command lines like so to train a conditional GAN by using 80-dimensional latent vectors for 12,000 minibatches:

```
> python3 mnist_conditional_gan.py 80 128 12000 results/mnist_cond_80
```

The conditional GAN model is larger than the unconditional model and takes more effort to train, so some patience is required. I recommend a shell script run over a day or two.

I tested latent vectors of 10, 80, and 160 dimensions and then used *generate_conditional_mosaic.py* to produce Figure 17-7.



Figure 17-7: The generated conditional DCGAN MNIST digits using 10-, 80-, and 160-dimensional latent vectors (left to right)

The unconditional case demonstrated a relationship between latent vector size and digit diversity. I see differences between the 10-dimensional latent vector digits and the 80- and 160-dimensional digits, but it's harder to make a subjective judgment between the two larger noise spaces.

Repeating the exercise for FMNIST produces Figure 17-8.

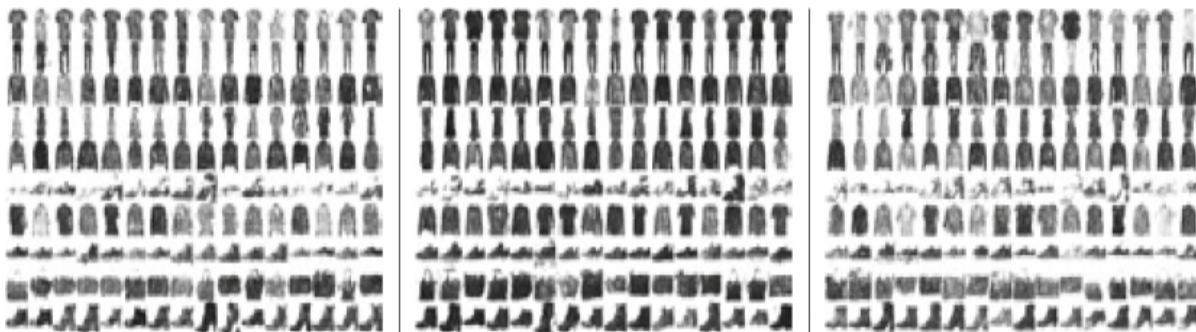


Figure 17-8: The generated conditional DCGAN FMNIST digits using 10-, 80-, and 160-dimensional latent vectors (left to right)

This case shows a clear preference for the 160-dimensional latent space. It captures shape and texture. The 80-dimensional space captures shape, but textures are more or less uniform. Examine the original PNG files by running the experiments yourself.

Exploring the Latent Space

GANs rely on a latent noise vector to derive the image sampled from the space learned from the dataset during training. A conditional GAN has the added benefit of allowing us to specify what class to generate. Indeed, that is exactly how the images in Figures 17-7 and 17-8 were created; see *generate_conditional_mosaic.py* for the specifics.

In this section, we explore the latent space, first by asking a question about how well we might believe the GAN to have captured the true character of the training data, then by exploring what happens to the generated images when moving through the latent space.

Intrinsic Dimensionality

In Chapter 16, we used the intrinsic dimensionality of the features generated by our self-supervised models to understand what the pretrained models do to the raw input features. Here we'll compare the ID of the GAN-generated images with that of the original dataset. The GAN, if well trained, should produce images that closely match the ID of the training set.

The file *id.py* uses scikit-dimension to estimate the ID of both MNIST and FMNIST. If you run it, you'll see the following:

```
Intrinsic dimensionality (MLE):
```

```
  MNIST: 14
```

```
  FMNIST: 16
```

Both datasets have a relatively low ID, but FMNIST's is slightly higher, lending credence to the belief that FMNIST is a more challenging dataset to learn than MNIST.

The code in *generate_unconditional_images.py* and *generate_conditional_images.py* produces NumPy output files of images sampled from pretrained unconditional or conditional GANs. For example, to produce a NumPy file containing images sampled from an unconditional GAN, run the following, assuming *generator.keras* is an unconditional GAN generator expecting 10-dimensional latent vectors:

```
> python3 generate_unconditional_images.py mnist_gan_10/generator.keras 10 10000  
ttt.npy 42
```

The additional arguments tell the code to produce a NumPy file of 10,000 fake images by using 42 as the seed.

To calculate the ID of this dataset, use *id_dataset.py*, which might produce something like this:

```
> python3 id_dataset.py ttt.npy  
Intrinsic dimensionality (MLE): 15
```

This tells us that the images the GAN produced live on a 15-dimensional manifold. This is quite close to the ID of 14 for MNIST, so we might believe that the GAN has captured essential characteristics of the true dataset.

Table 17-2 shows the ID calculations of each unconditional and conditional GAN.

Table 17-2: Intrinsic Dimensionality of the GAN-Generated Images

Dataset	GAN type	Latent vector	ID

Dataset	GAN type	Latent vector	ID
MNIST (true 14)	MLP	10	15
	MLP	40	16
	MLP	80	16
	DCGAN	10	15
	DCGAN	80	18
	DCGAN	160	18
	Conditional DCGAN	10	10
	Conditional DCGAN	80	14
	Conditional DCGAN	160	15
FMNIST (true 16)	Conditional DCGAN	10	9
	Conditional DCGAN	80	14
	Conditional DCGAN	160	16

The table hints at a relationship between latent vector size and the GAN's ability to capture the true intrinsic dimensionality of the dataset. My claim is especially evident when contemplating the conditional GAN results, so let's begin with those.

Conditional GANs using a 10-dimensional latent vector produced images living on a 9- or 10-dimensional manifold. Recall that the ID value is an estimation, so we can treat an ID of 9 or 10 as equivalent. Notably, the ID matches that of the latent vector. I read this to mean

that a 10-dimensional latent vector is insufficient, which we already knew from the ID of the datasets (as estimated by the same algorithm used for the GAN IDs).

Increasing the latent vector size affects the ID of the generated images, again a reasonable result. For MNIST, the 80-dimensional latent vector matches the ID of the true data. Moving to a 160-dimensional latent vector increases the ID slightly. This result matches the conclusions we reached when contemplating digit arrays: 10 elements are insufficient to produce the full variety of digit types.

For FMNIST, the result is much the same, with a 160-element latent vector producing a match. Again, this result aligns with what we see in Figure 17-8, where the 160-dimensional mosaic seems the best.

The unconditional GAN IDs are harder to interpret. For example, for the MLP-only GAN using a 10-dimensional latent vector, the ID was already at 15, hinting that the GAN was successful at capturing either the true digit distribution (in appearance) or at least learning digit-like representations existing on a similar manifold (dimensionally). The unconditional DCGAN IDs are of the same range but increase to 18 when the latent vector size increases, which is still close enough to 14 to produce believable digits, along with digit-like output.

Our GANs produce output that is visually realistic and inline with expectations based on estimating the dimensionality of the data manifold. Let's shift gears to explore what happens when traversing the latent vector space.

Latent Space Interpolation

The components of the latent vector capture aspects of the data-generation process. Therefore, it stands to reason that moving in a particular direction through the multidimensional latent space will alter the resulting image along whatever aspect(s) of the data are encoded by that direction.

The easiest directions to explore are the orthogonal directions, the coordinate axes of the latent space. If we have a 10-dimensional space,

what happens if we alter the third-dimension value while holding the other component values fixed? To find out, run *interpolate_mosaic.py*:

```
> python3 interpolate_mosaic.py
```

```
interpolate_mosaic <model> <latent> <out_image> [<seed>]
```

```
<model>      - trained conditional GAN generator
```

```
<latent>     - latent vector dimensionality (e.g. 10)
```

```
<out_image>  - output image name
```

```
<seed>       - PRNG seed (optional)
```

I won't detail the code, but it essentially picks a random latent vector and then produces a row of digits altering that component of the latent vector in steps from $[-5, 5]$ while holding the remainder of the latent vector fixed.

For example, say the base noise vector is the following, and the fourth row of the output is being generated:

$(0.328, 1.270, -1.860, 1.917, -0.719, -1.810, 0.074, 0.450, 1.553, 1.129)$

In this case, the conditional GAN is asked to produce sixteen 4s from

$(0.328, 1.270, -1.860, 1.917, -5.000, -1.810, 0.074, 0.450, 1.553, 1.129)$

to:

$(0.328, 1.270, -1.860, 1.917, +5.000, -1.810, 0.074, 0.450, 1.553, 1.129)$

We expect that this series of 4s will gradually change as the fourth component of the latent vector is changed. How the 4 will change depends on which aspect of the image-generation process has been captured by that vector component. For a small latent vector, each component likely influences multiple aspects of the digit. In other words, the latent vector has *entangled* meaningful parts of the generation process. A research goal is to produce *disentangled* latent vectors, where each component represents a meaningful part of the generation process.

Figure 17-9 presents three digit mosaics, each using a novel initial 10-dimensional latent vector; interpolation is along the vector component indicated by the generated digit.



Figure 17-9: Interpolations along latent vector axes for a 10-dimensional latent vector

Changes in the digit are evident, often smoothly varying, but multiple aspects of the digit are affected. Nonetheless, the exercise demonstrates that the learned latent space can be smoothly traversed and that nearby points (considering a latent vector a point in that space) produce similar outputs. Stepping from -5 to $+5$ covers most of the range produced by sampling from a standard normal distribution, so we shouldn't be surprised by extreme output near the edges of the latent space.

We have no a priori reason to believe the coordinate axes of the latent space to have encoded meaningful aspects of the generation process, so it's natural to wonder what happens if we move along a randomly selected line in the latent space. The file *interpolate_random_mosaic.py* answers the question by producing digits where each row of digits is altered by moving along a randomly selected line in the multidimensional space.

What the randomly selected directions represent is unknown. I generated 100 such images by using a 10-dimensional latent vector and then scanned through them to pick some that appeared to capture something interesting in at least one row of digits. The result is Figure 17-10.



Figure 17-10: Interpolations along random directions in the 10-dimensional latent space

The fourth row of the leftmost image, which represents 3s, moves from something that looks like a 5 to something more 3-like. In the middle, the 4s morph into 1s, and on the right, the 7 gradually adds a bar in the middle. These results again demonstrate that the latent space is organized and, in some sense, smooth.

GANs that learn disentangled representations in the latent space are capable of amazing visual effects. For example, suppose the GAN generates realistic images of human faces. In that case, one of the components of the latent vector might affect the presence of glasses, or a beard, or even more-abstract concepts like age and gender. Learning such representations is an active area of research.

We've barely touched the surface of all that GANs can do, but it's time to move on to our next topic. If your interest has been piqued, you'll find a plethora of GAN research in the literature, more than enough to demonstrate their versatility in a myriad of areas. And with that, I'm out of seldom used Greek words.

Summary

This chapter introduced you to generative adversarial networks, your first foray into the world of generative AI. We explored unconditional GANs that produce samples like their training set. Then we added the ability to condition the GAN's output by a class label to produce any label type on demand. In each case, we modified the GAN's latent input

vector dimension to evaluate its effect on the range and type of samples generated.

We explored how well the GANs learned to mimic their parent distributions by estimating the intrinsic dimensionality of the generated output and compared that dimensionality to that of the true dataset, either MNIST or FMNIST.

Finally, we observed that there are meaningful directions in the learned latent space, which correspond to smoothly varying changes in the appearance of the generated images.

GANs are merely the beginning of our exploration of generative AI. The next stop is diffusion models.

[OceanofPDF.com](https://oceanofpdf.com)

18

LARGE LANGUAGE MODELS



Anyone interested in AI has likely encountered *large language models* (LLMs), generative AI models that produce coherent text in response to a prompt. One might argue that LLMs have also learned to understand the meaning of language (and many other things). In this chapter, we explore the architecture of LLMs, then get practical and learn how to use what LLMs offer.

LLMs are, well, *large* models. A small LLM might contain “a mere” 2 billion parameters, while the largest have upwards of 2 trillion as of this writing. The companies producing state-of-the-art LLMs are somewhat reticent about divulging the actual size of their models. That alone is justification for the way this chapter is structured. I’ll explain.

Previous chapters introduced a kind of model or a set of concepts, followed by examples of models trained from scratch to illustrate the technique. By nature, an LLM is trained on a huge dataset, often requiring ridiculous amounts of computing hardware (GPUs). In other words, we can forget trying to find examples of particular architectures trainable on a CPU-only machine. In most cases, multiple GPUs are the bare minimum necessary to train even the smallest of the LLMs (yes, I see the irony of referring to an LLM as “small”).

Therefore, the chapter proceeds as follows. First, we contemplate an LLM's architecture and how to use that architecture for inference to produce a coherent reply to a given prompt.

Next, we explore using pretrained LLMs to take advantage of their remarkable capabilities. I defend this approach by pointing out that the majority of people involved with AI are focused on practical applications of LLMs; few of us are involved in the construction and training processes. I view the relationship as akin to that between a car mechanic and an automotive engineer. There are many car mechanics who know how to work with existing cars, but relatively few automotive engineers to design those cars.

Our experiments cover a range of LLM topics. We begin by exploring one of the emergent abilities found in large LLMs: in-context learning. We'll use a state-of-the-art LLM, OpenAI's GPT-4.

After this, you'll learn that although we cannot train LLMs, we *can* run them locally, even on a CPU-only machine. We'll install and configure Ollama, a powerful (and free) tool for running open source LLMs on a desktop computer. Ollama gives us all we need to experiment with LLMs as a practical tool.

We'll perform the following experiments with Ollama:

- Building and testing a simple chatbot
- Exploring text embeddings, including how to generate them and how they encode meaning
- Performing semantic search and retrieval-augmented generation (RAG) with text embeddings
- Using LLMs for sentiment analysis
- Experimenting with multimodal LLMs capable of understanding text and images
- Attempting to answer this question: Are LLMs creative?

Let's begin by exploring how the transformer architecture is implemented in an LLM.

Understanding Large Language Models

Many LLMs, including GPT-4, are currently implemented as decoder-only transformers. Transformer models were introduced in the 2017 paper “Attention Is All You Need” by Ashish Vaswani et al. (which has been cited over 150,000 times as of this writing).

AI researchers have long sought to build models that understand language. Because written text is a sequence of letters representing a sequence of words, it therefore seems reasonable to build models capable of processing sequences—of accepting inputs piece by piece and processing those inputs to produce an output string, piece by piece. We call these pieces *tokens*, representing short words, parts of words, or individual characters.

Various flavors of recurrent neural networks (RNNs) have been developed over the years, perhaps the most useful of which is the long short-term memory (LSTM) variant. All RNNs accept inputs sequentially along with their output from the previous input to produce the following output. In essence, RNNs are like recurrence relations that iterate over inputs.

It is reasonable to expect such models to perform well at tasks requiring varying-length input processing, including text interpretation. Researchers have tried for decades to do just that with a certain level of success. However, in the end, RNNs are incapable of handling long-range association effects, like referring back to the beginning of a sentence to properly understand which *she* is being referred to. Additionally, training RNNs is difficult because of the vanishing gradient problem (see Chapter 13). An RNN can be viewed as a deep neural network when the iterations are unrolled in time.

Transformer models introduced a self-attention mechanism that allows layers of the network to do what RNNs cannot: refer back to earlier portions of their input sequence. Instead of processing inputs token by token, the entire input (as tokens) is processed simultaneously. The self-attention mechanism, along with ordinary MLPs working in parallel, enables the model to learn during training which parts of the input are most relevant, thereby opening the door to a model that can interpret the input in much the same way we humans might interpret it. In a sense, the

transformer architecture is a next step in providing an environment where capabilities mimicking human thought processes might emerge from the training data.

I referred to LLMs as decoder-only models. The original transformer paper discusses a two-part model (an encoder and a decoder) with an original goal of language translation. For an LLM, the encoding happens via self-supervision by learning to predict the next token given a sequence of input tokens. In this way, the entire body of all human-generated text becomes training data; no labels are necessary.

The remainder of this section describes how an LLM turns an input string, the prompt, into an output token. We'll walk through the process step by step. This section requires drawing on much of what you've learned in previous chapters regarding deep neural networks.

I find it helpful to talk about the operation of an LLM from a data-flow perspective: what each part of the model expects as input and what it produces as output. The presentation will be easier to digest if we assign some numbers to important LLM model parameters. We are discussing an LLM with the following characteristics (any unknown terms will be defined shortly):

- A vocabulary (V) of 30,000 tokens
- A token embedding size (E) of 768 dimensions
- A context window (N) of 1,024 tokens
- A single transformer layer with:
 - A self-attention block with $H = 16$ heads
 - An MLP block with a single hidden layer of $4E = 3,072$ nodes

We'll work top-down, beginning with a block diagram of the LLM. Then we'll dive into each block to understand what it expects as input, what it does to process the input, and what it produces as output.

Evaluating the LLM Block Diagram

Our goal is to understand what an LLM does to a given prompt to produce the next token of its response. Implicit in that goal is the fact that the LLM is already trained. We won't discuss training other than to say

that it uses standard backpropagation and gradient descent to update model weights based on a measured loss. LLMs are trained similarly to existing neural networks.

Figure 18-1 shows conceptually how an LLM works, from raw text input on the left to an emitted token on the right.

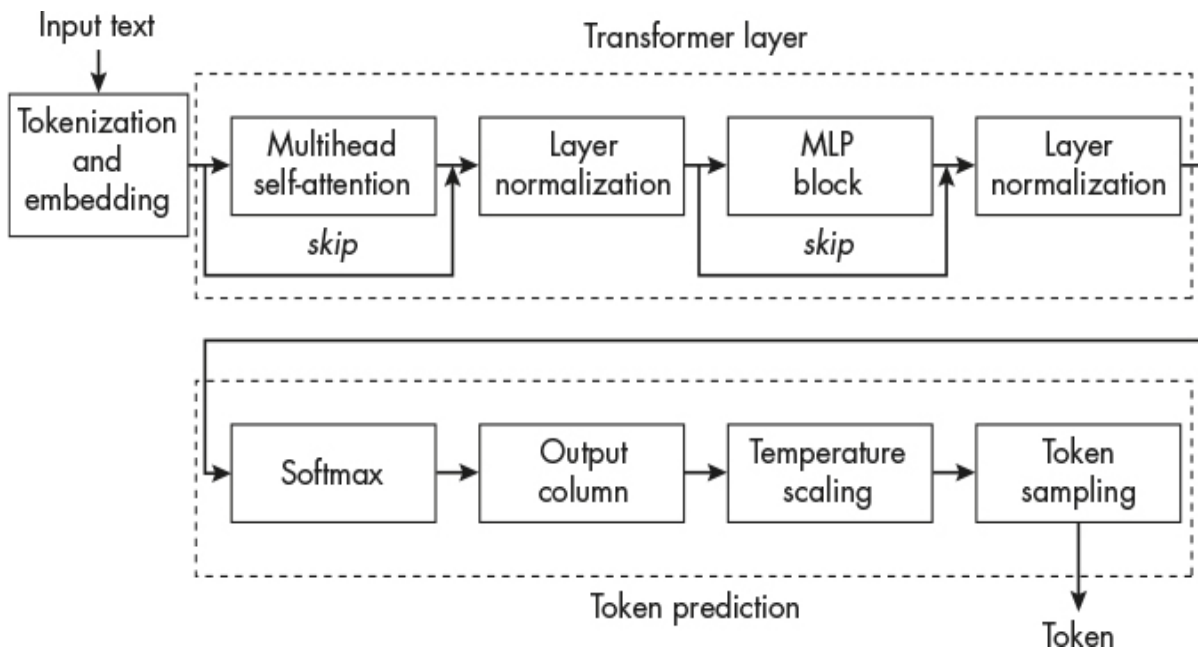


Figure 18-1: An LLM block diagram

The workflow is linear: The input string enters, is processed, and a token exits. There is no looping or iteration, as was historically the case with RNN-based models. The transformer layer is capable of processing the entire input at once. This change in architecture makes LLMs a paradigm shift in processing arbitrary-length inputs (at least, inputs that fit within the model’s context window).

At the highest level, we see three stages: tokenization and embedding, the transformer layer, and token prediction. These conceptual stages consist of multiple layers working together. The transformer layer is often represented in code as a block (that is, a class), much like the `ResidualBlock` we implemented in Chapter 13 when discussing ResNets. Let’s walk through the process, stage by stage.

Tokenizing and Embedding

The first stage, tokenization and embedding, maps a text string representing an initial prompt or a continuation of a conversation to a series of tokens, and from that to a series of E -dimensional embedding vectors.

The LLM’s architects, prior to building the model, must select the vocabulary of tokens over which the model will operate. The vocabulary size is a function of the task and of the way the prompt is tokenized.

Tokenization generally uses an external tokenizer to split the string into a series of tokens, each assigned an integer value as an index into the vocabulary. For example, a simple tokenizer might preserve whole words to transform a sentence like so:

What is the airspeed velocity of an unladen swallow ?										
213	7	3	967	10322	8	9	1066	3224	1	

Tokenization is a straightforward concept, but there is a crucial issue to remember: Transformers operate on the entire context window at once. In our example, $N = 1,024$, meaning the complete tokenization process produces the 10 tokens shown, followed by 1,014 subsequent tokens used to indicate that the remainder of the context window is empty.

The series of tokens selects rows from the $V \times E$ embedding matrix learned by the model during training. For our example, this means token labels become indices into the 30,000×768 embedding matrix to return the 768-dimensional embedding vector associated with that token.

The net result of this token-to-embedding-vector process is that the tokenized input string becomes a single $N \times E$ matrix, and the rows are now the embedding vector associated with the token extracted from the prompt. Suitable embedding vectors, likely all 0s, are substituted for the unused portion of the context window.

The embedding vectors are learned during training, beginning as most models do with a random embedding matrix. The vectors begin to capture associations between the tokens related to the way the tokens are used and associated with other tokens in written language. Therefore, a significant amount of language understanding is already present in the embedding matrix.

At this point, E , the $N \times E$ token-embedding matrix, represents the sequence of tokens in the prompt, row by row, but the transformer layer cannot detect this fact. To preserve information about the order of the tokens, we must encode the order within the $N \times E$ matrix before passing it to the transformer layer by adding a *positional encoding* matrix, P :

$$\mathbf{X} = \mathbf{E} + \mathbf{P}$$

The matrix, $\mathbf{X}$, becomes the input to the transformer layer.

Adding the positional matrix to the token-embedding matrix might seem familiar, as we did something similar in Chapter 17 to associate a label embedding with an image embedding when constructing a conditional GAN. In that case, we multiplied instead of adding, but the effect is similar: We modulate the matrix in such a way that the model can, over time during training, learn about position along with token embeddings.

How is P constructed? The original transformer paper used sine and cosine functions to produce a matrix containing elements that are each unique in a consistent way. For example, if p represents a row of E , that is, a particular token in the context window, ($p \in [0, 1023]$) and i runs from $[0, E/2]$, that is, $i \in [0, 384]$, we fill in P like so:

$$\begin{aligned} P_{p,2i} &= \sin\left(\frac{p}{10000^{2i/E}}\right) \\ P_{p,2i+1} &= \cos\left(\frac{p}{10000^{2i/E}}\right) \end{aligned}$$

In the literature, you'll often see E , the dimensionality of the token-embedding vectors, written as d_{model} . Figure 18-2 shows us what P looks like as an image (see *position.py*).

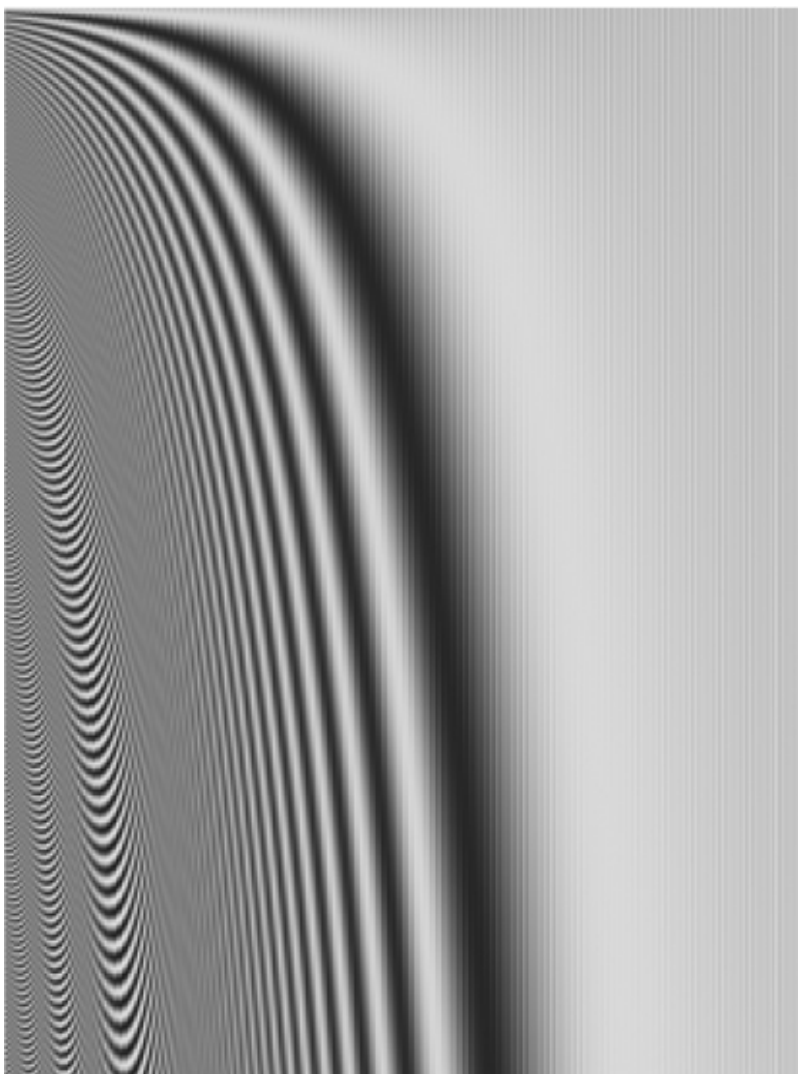


Figure 18-2: The $N \times E$ positional encoding matrix, $\mathbf{P}$

To recap: The raw text input string is transformed into a series of tokens, indices into the $V \times E$ embedding matrix the model learned during training. The resulting embeddings, arranged as the rows of the $N \times E$ token-embedding matrix, $\mathbf{E}$, are added to the position-encoding matrix, $\mathbf{P}$, to produce a modulated embedding matrix, $\mathbf{X}$, the input to the transformer layer.

Exploring the Transformer Layer

The transformer layer is at the center of current LLMs. What makes the transformer layer useful in this context is its ability for *self-attention*. It learns during training which portions of its fixed-size input are essential

for interpreting the meaning represented by the token embedding plus the position-encoded input matrix that we're referring to as $\mathbf{X}$. Figure 18-1 tells us that a transformer layer consists of three components: a multihead self-attention block, an MLP block, and two instances of a layer normalization block. Notice the presence of skip connections, meaning the input to the self-attention and MLP blocks are added to the block's output, just as skip connections add the input to a residual block to the block's output. Think of the self-attention and MLP blocks as learning a kind of offset added to their inputs.

Our pedagogical LLM uses a single transformer layer. Real LLMs use multiple layers arranged sequentially, usually with skip connections present at that level as well.

Let's continue by describing the operation of each transformer block in terms of what it expects as input and what it produces as output.

Multihead Self-Attention

The multihead self-attention block is the heart of a transformer layer. Its goal is to determine what portions of the input should be emphasized compared to others. The block is multihead because it consists of H separate attention subblocks, or *heads*, that process the input matrix, $\mathbf{X}$, in parallel. During training, the separate heads learn essential associations in the input data, each learning to pay attention to different portions of the encoded input.

Here *attention* means developing an attention matrix ($\mathbf{A}_i$) for the current input. The i subscript refers to one of the H attention heads. To understand the attention matrix, we'll begin with its definition and then walk through what it means.

Each self-attention head learns three separate weight matrices during training: $\mathbf{W}_i^Q$, $\mathbf{W}_i^K$, and $\mathbf{W}_i^V$. The matrices are often, but not necessarily, all $E \times K$ dimensional, where $K = E/H$. For us, this means $K = 768/16 = 48$. During inference, each self-attention head uses these matrices to produce query ($\mathbf{Q}_i$), keys ($\mathbf{K}_i$), and values ($\mathbf{V}_i$) matrices:

$$\begin{aligned}\mathbf{Q}_i &= \mathbf{X}\mathbf{W}_i^Q \\ \mathbf{K}_i &= \mathbf{X}\mathbf{W}_i^K \\ \mathbf{V}_i &= \mathbf{X}\mathbf{W}_i^V\end{aligned}$$

Here $\mathbf{Q}_i$, $\mathbf{K}_i$, and $\mathbf{V}_i$ are $N \times K$ matrices (1,024×48).

The attention matrix is derived from these three matrices via scaled dot-product attention:

$$\mathbf{A}_i = \text{softmax}\left(\frac{\mathbf{Q}_i\mathbf{K}_i^T}{\sqrt{K}}\right)$$

It's important to note that the softmax function is applied across the rows of the $N \times N$ product of $\mathbf{Q}(N \times K)$ and $\mathbf{K}_i^T(K \times N)$ to turn the rows into probability vectors. Applying the softmax across the rows allows the attention matrix to concentrate probability across the input sequence, the input tokens. This enables each head to focus on different portions of the input dynamically, based on the interplay between $\mathbf{Q}_i$ and $\mathbf{K}_i$. Recall that the i subscript indicates a specific self-attention head, so this process repeats across all heads by using the weight matrices each head learned separately during training. Scaling the product of $\mathbf{Q}$ and $\mathbf{K}_i^T$ by $\sqrt{K}$ keeps the softmax calculation in a region where the output is numerically stable.

The attention matrix becomes a multiplier on the last of the three matrices produced by the attention head, $\mathbf{V}_i$, with $\mathbf{O}_i$ an $N \times K$ matrix, the output for self-attention head i :

$$\mathbf{O}_i = \mathbf{A}_i\mathbf{V}_i$$

The output matrix represents attention-scaled information from $\mathbf{V}_i$, which is itself derived from $\mathbf{X}$, the input to the multihead self-attention block.

The query ($\mathbf{Q}$), keys ($\mathbf{K}$), and value ($\mathbf{V}$) names come from earlier work on neural networks that incorporate memory. The similarity to a database retrieval system is also there. However, it's important to remember that the weight matrices learned by each head ($\mathbf{W}^Q$, $\mathbf{W}^K$, $\mathbf{W}^V$) project the input ($\mathbf{X}$) to new spaces that initially have no particular meaning because the transformer layer, like the other neural networks we've investigated, is randomly initialized.

It is only the form of the scaled dot-product attention equation that calculates the attention matrix ($\mathcal{A}$), combined with the evolution of the weight matrix values during training, that enable the emergence of behavior causing the attention matrix to be useful for focusing on different parts of the input. Again, we see another example of the model architecture providing the potential for a specific behavior to emerge, but it is the training process and the training data that causes the emergence.

Each H self-attention head produces an $N \times K$ output matrix, thereby allowing each head to focus on different parts of the input matrix, $\mathbf{X}$. These matrices are concatenated horizontally (imagine them placed side by side) to form a single $N \times E$ output matrix, where the semicolon (;) means horizontal concatenation:

$$\mathbf{O} = [\mathbf{O}_0; \mathbf{O}_1; \dots; \mathbf{O}_{H-1}]$$

The matrix $\mathbf{O}$ is $N \times E$ because, by design, $K \times H = E$.

The matrix $\mathbf{O}$ is projected by a final $E \times E$ weight matrix, $\mathbf{W}^O$, to produce the output of the multihead attention block:

$$\mathbf{X} \leftarrow \mathbf{X} + \mathbf{O}\mathbf{W}^O$$

Notice that I'm updating $\mathbf{X}$ by adding the output of the multihead attention block to $\mathbf{X}$, the block's input. Doing so implements the first skip connection in Figure 18-1 and sets $\mathbf{X}$ for layer normalization.

Layer Normalization

Layer normalization has a similar goal to batch normalization (see Chapter 13 for a refresher), but instead of operating across a minibatch of samples, it operates on $\mathbf{X}$, an $N \times E$ matrix. To normalize each row, an E -dimensional embedding vector, layer normalization first calculates the mean of the features in that row, then the variance of the features (the square of the standard deviation), before updating each feature:

$$x_{ij} \leftarrow \frac{x_{ij} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

Here μ_i is the mean of all the features in row i , σ_i^2 is the variance of those features, and ϵ is a small value added for numeric stability. The index j runs from 0 to $E - 1$ (that is, across the row). Layer normalization then multiplies each feature elementwise by a learned γ vector to scale the feature before adding a learned β vector:

$$\mathbf{x}_i \leftarrow \gamma \odot \mathbf{x}_i + \beta$$

Each row i of the matrix $\mathbf{X}$ is viewed as a vector, $\mathbf{x}_i$, and $\odot$ means elementwise multiplication.

The same γ and β vectors are broadcast across the rows, $i \in [0, N - 1]$, to produce a new $N \times E$ output matrix where each of the N rows has been normalized, scaled, and shifted. Using γ and β adds flexibility, enabling the model to alter the imposed normalization if doing so is helpful overall.

Figure 18-1 shows two layer normalization blocks, each immediately after summing the input, $\mathbf{X}$, to the output of the multihead self-attention block or the MLP block. In other words, normalization happens immediately after updating $\mathbf{X}$ by the matrix produced by the other blocks in the transformer layer.

Multilayer Perceptron Block

The MLP block accepts $\mathbf{X}$, an $N \times E$ matrix, and applies a classical two-layer MLP to each row (feature vector) of the input. The MLP has a single hidden layer of a specified number of nodes, often $4E$, and applies an activation function like ReLU before sending the output of the hidden layer to a final layer with E nodes and no activation function.

Mathematically, the MLP processes each row of $\mathbf{X}$ like so:

$$\mathbf{x}_i \rightarrow \mathbf{a} = h(\mathbf{W}_0 \mathbf{x}_i + \mathbf{b}_0) \rightarrow \mathbf{y}_i = \mathbf{W}_1 \mathbf{a} + \mathbf{b}_1$$

Here $\mathbf{y}_i$ is the E -dimensional vector of the output matrix corresponding to input vector $\mathbf{x}_i$. Running i from 0 to $N - 1$ processes all input features via the MLP to produce the $N \times E$ output matrix $\mathbf{Y}$ that's added to the input to implement the second skip connection:

$$\mathbf{X} \leftarrow \mathbf{X} + \mathbf{Y}$$

Finally, subjecting the updated $\mathbf{X}$ to a second layer normalization completes the transformer, mapping the input $N \times E$ matrix to an output $N \times E$ matrix.

Predicting the Next Token

Our LLM uses a single transformer layer. Its output is $\mathbf{X}$, an $N \times E$ matrix. Let's use this matrix to predict the next output token. Once we have the new token, we add it to the input string and process again to generate the subsequent token, repeating the process until a special token, often denoted *end-of-sequence* (EOS), is emitted.

The model uses an N -token context window (sometimes called the *sequence length*). Suppose the prompt uses $M < N$ tokens with the remainder of the context window filled in by whatever “embedding vector” passes as null. Our goal then becomes to predict the $M + 1$ token.

Rows $[0, M - 1]$ of the embedding matrix passed to the transformer layer contain the E -dimensional embeddings of the tokens, thereby implying that we want to use the M th row of the $N \times E$ matrix produced by the transformer layer to predict the next token.

To get the next token, the M th row of $\mathbf{X}$ must be projected to V dimensions, where $V = 30,000$ is the vocabulary size. The projection matrix is typically the transpose of the embedding matrix used to produce the input to the transformer layers. If we call this matrix $\mathbf{W}$ and we call the M th row vector of $\mathbf{X}$, $\mathbf{x}$, then the projection becomes the following to map a $1 \times E$ row vector to a $1 \times V$ row vector by using an $E \times V$ matrix, $\mathbf{W}^T$:

$$\mathbf{p} = \mathbf{x}\mathbf{W}^T$$

The next step passes $\mathbf{p}$ through a softmax to transform the elements into probabilities over the vocabulary. The probabilities are informed by the model's actions via the transformer layer(s). All that remains is to use the probability distribution to predict the emitted token. However, before that, we have a final, short detour to take to understand what is meant by an LLM's temperature, the parameter it uses to influence the likelihood of the next token emitted.

An LLM's *temperature*, T , is a scalar used to alter the model's initial probability distribution ($\mathbf{p}$) to affect the likelihood of the emitted token.

Typically, $T \in [0, 1]$, and the smaller T is, the more likely the model is to emit the token with the highest probability in the initial distribution. As T approaches 1, the model's probability distribution is altered to make other tokens more likely, thereby increasing the diversity of possible outputs. In other words, a low T makes the model more deterministic, while a higher T makes the model more probabilistic.

The temperature, T , alters $\mathbf{p}$ like so to give us a new probability associated with token i :

$$p'_i = \frac{\exp(\log(p_i)/T)}{\sum_j \exp(\log(p_j)/T)}$$

The denominator scales the numerators so that $\sum_i p'_i = 1$, as it must to be a valid probability distribution.

The distribution is discrete because the model's vocabulary is finite. Therefore, in code, adjusting $\mathbf{p}$ as a NumPy vector becomes the following for input probability vector $\mathbf{p}$ and temperature τ :

```
def Adjusted(p, T):  
    s = np.exp(np.log(p+1e-8)/T)  
    return s / s.sum()
```

The small $\epsilon = 10^{-8}$ added to the probabilities prevents taking the logarithm of 0. Scaling $\mathbf{s}$ by its sum makes $\mathbf{s}$ a new probability distribution.

Figure 18-3 demonstrates T 's effect on a distribution. When $T = 1$, the original distribution is unmodified.

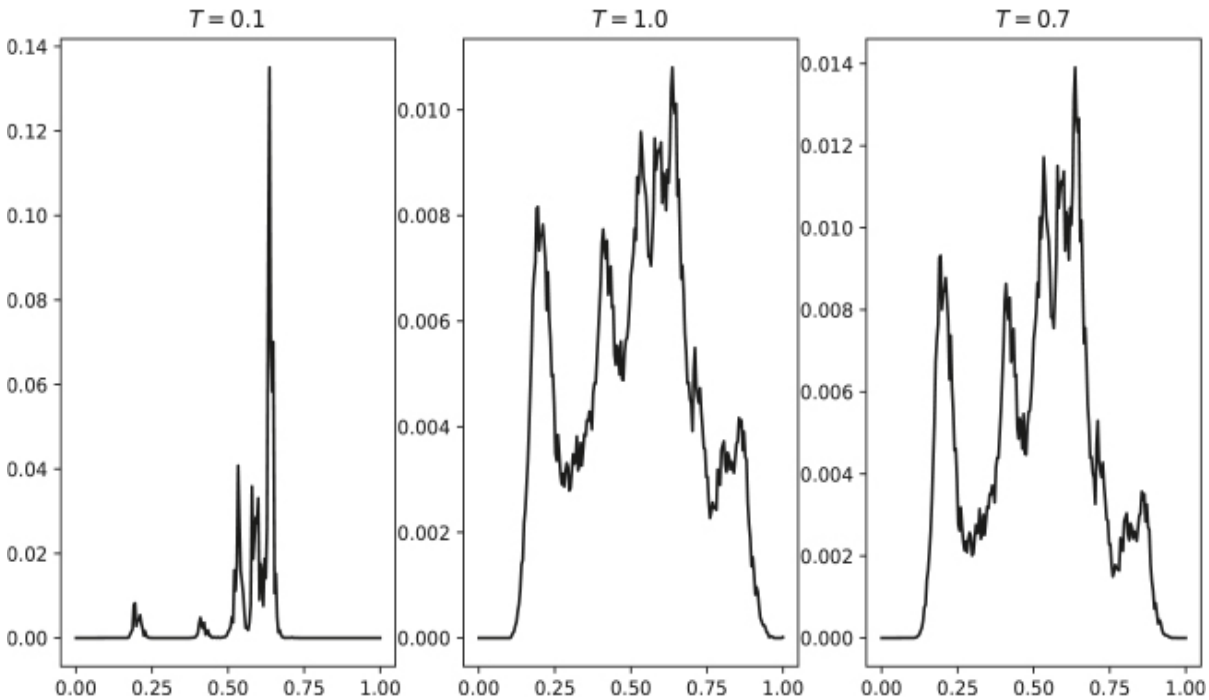


Figure 18-3: The effect of temperature on modifying a probability distribution

The middle plot in Figure 18-3 is a discrete probability distribution representing p . In reality, it's the histogram of a popular grayscale test image. The x-axis represents a grayscale value (a possible token in the model's vocabulary), and the y-axis is the fraction of the test image's pixels with that value (the initial probability of selecting that token).

The plots on the left and the right show the effect of a low temperature ($T = 0.1$) and a higher temperature ($T = 0.7$). When T is low, the distribution becomes peaked and the most likely values in the original distribution become even more likely. Notice the y-axis scales differ in the three plots. So, with high probability, a model using a low temperature will select the most likely value given the initial distribution. When $T = 0$, the model returns the most likely token, making the model deterministic. As you'll see later in the chapter, this means the model's output will be deemed less creative overall.

A higher T value, up to $T = 1$, makes it more likely other tokens will be output according to the probability distribution derived from the model. This is often what we want, especially if creativity is the goal, as it might be in writing a fictional story. However, if we're asking the model to produce code, we might want a lower T , say near 0.1, to make the code

more likely to be correct. Most public models seem to use a default of $T \approx 0.7$.

A $T > 1$ flattens the distribution, making it more like a uniform distribution where each output token becomes equally likely. Therefore, while $T > 1$ is generally ill-advised, it might be desirable when we want the model to be excessively creative and are willing to tolerate outputs that are simply nonsense for the sake of the few novel ones.

To conclude this lengthy section: The model selects the most likely token if $T = 0$; otherwise, it samples from $\mathbf{p}$ after adjusting it via a specified temperature, T . The token emitted is added to the prompt, and the process repeats until the EOS token is produced.

Now that you understand the process an LLM uses to predict a sequence of tokens in response to an initial prompt, we're ready to explore the world of LLMs.

In-Context Learning

In-context learning is a particularly impressive emergent LLM ability. LLMs expect a prompt and then generate a meaningful response to that prompt. The prompt can include examples illustrating something we wish the LLM to learn about, something not included in its training data. Via in-context learning, the model can—within a current session—go beyond the knowledge contained in its training data to reason over new knowledge, without modifying the model's weights.

In-context learning is something that classic models, including deep learning models, cannot do. To improve the abilities of a standard neural network, we must retrain the model and modify its parameters. The following experiment powerfully illustrates that LLM's in-context learning abilities allow it to learn on the fly.

Configuring the Experiment

The question we want this experiment to address is: Via in-context learning, can an LLM accept training data and then use it to classify a set of test samples?

The file *in-context.py* uses sklearn to generate a simple four-feature, three-class dataset. Listing 18-1 uses `make_classification` to produce the dataset.

```
in-context.py
import numpy as np
from sklearn.datasets import make_classification

x, y = make_classification(n_samples=60, n_features=4, n_classes=3,
                           n_clusters_per_class=1, random_state=6502)
x = x / np.abs(x).max()
x = (x*100).astype("int32") / 100.0

i = np.where(y == 0)[0]; x0 = x[i]
i = np.where(y == 1)[0]; x1 = x[i]
i = np.where(y == 2)[0]; x2 = x[i]
np.random.seed(6809)

xtrn = np.vstack((x0[:12], x1[:12], x2[:12]))
ytrn = np.array([0]*12 + [1]*12 + [2]*12)
i = np.argsort(np.random.random(len(ytrn)))
xtrn, ytrn = xtrn[i], ytrn[i]

xtst = np.vstack((x0[12:], x1[12:], x2[12:]))
ytst = np.array([0]*8 + [1]*8 + [2]*8)
i = np.argsort(np.random.random(len(ytst)))
xtst, ytst = xtst[i], ytst[i]

np.save("in-context_xtrain.npy", xtrn)
np.save("in-context_ytrain.npy", ytrn)
np.save("in-context_xtest.npy", xtst)
np.save("in-context_ytest.npy", ytst)
```

Listing 18-1: Creating the in-context learning dataset

The dataset is then scaled to $[-1, +1]$ and multiplied by 100. The remainder of the code produces training and test sets. Notice that the pseudorandom number seed is fixed and that the test set consists of exactly eight samples from each class.

Our plan is straightforward: We'll give GPT-4 the training set with labels, then, once we know that the model has understood the structure of the dataset, ask it to produce class labels for the test set.

If a human approached us and asked us to perform this experiment, we would naturally write some code, likely in Python, to train a selected machine learning algorithm. Then we would pass the test samples to the trained algorithm to assign labels. GPT-4 is a state-of-the-art model that knows how to use tools like sklearn. Therefore, if we use GPT-4 out of the box, it will do as we would: It will produce Python code to train a simple classifier, then run it. This is not what we want, though the model's recognition that it needs an external tool to best answer the query is impressive.

Fortunately, we can run what I call *vanilla GPT-4*: The model is not allowed to use external tools and must make do with what it knows and has learned during its training session. Remember, the model is repeatedly trying to produce the best possible next token, and to do that, it has (remarkably) learned many other things besides.

Running the Experiment

We're ready to run the experiment. First, the initial prompt to vanilla GPT-4:

Here's a dataset of 36 samples, 4 features. I'll ask you to do something with it in time, but for now, acknowledge that you understand the dataset:

```
[[ -0.2  -0.16 -0.18  0.04]
 [ -0.39 -0.31 -0.44  0.12]
 [  0.79  0.72 -0.12  0.33]
 [ -0.49 -0.4  -0.49  0.12]
 [ -0.33 -0.27 -0.34  0.08]

 [ -0.6  -0.48 -0.65  0.18]
 [ -0.41 -0.4   0.35 -0.34]
 [  0.05  0.09 -0.54  0.33]
--snip--
```

To which vanilla GPT-4 replies:

I understand the dataset you've provided. It consists of 36 samples, each with 4 features.
Let me know how you would like to proceed with this data.

Now that we know vanilla GPT-4 understands the format, let's give it the associated training labels:

Here are the class labels for the dataset above:

```
[2 2 0 2 2 2 1 0 2 0 2 2 0 1 2 1 2 1 0 0 1 1 0 1 0 1 2 0 1 0 0 1 1 1 0 2]
```

Here's GPT-4's reply:

I see you've provided class labels for the dataset, assigning one of three possible classes (0, 1, or 2) to each of the 36 samples. With this additional information, your dataset now includes both features and corresponding class labels. What would you like to do next with this dataset?

GPT-4 understands the class labels and knows they align with the trainingset feature vectors. Take a moment to be impressed with this capability; a few years ago, this wasn't possible.

We're ready to give the model our test samples and ask our question:

Now, using the dataset and labels, emulate a classifier and classify the following unknown samples. You are absolutely forbidden from using any external tools or code. Here are the samples:

```
[[ -0.29 -0.3   0.46 -0.36]
 [ -0.35 -0.29 -0.3   0.06]
 [  0.51  0.51 -0.68  0.56]
 [  0.19  0.2  -0.46  0.33]
 [  0.1   0.08  0.12 -0.03]
 [  0.61  0.55  0.    0.21]
 [-0.37 -0.36  0.32 -0.31]
--snip--
```

For good measure, I doubled down on prohibiting the model from using external tools. Let's walk through the model's reply in parts, beginning

with its mild protest against my imposed restrictions:

Without the ability to execute external tools or code and strictly emulating a classifier based on the provided dataset and its labels, I'll approach this task using a simple heuristic based on patterns observed in your dataset. This heuristic approach is overly simplistic and might not accurately reflect complex relationships in the data, but it will serve as a conceptual example of how one might manually infer classifications based on feature similarities and differences.

GPT-4 prefaces its classification attempt with a preamble pointing out that since it cannot use external tools, it must infer structure in the training data itself to develop a heuristic it can use to assign (hopefully) meaningful class labels to the test set. This is exactly what we want to test: Can the model use in-context learning to go beyond its training data?

GPT-4 continues:

Given the dataset, we can attempt to discern patterns that might be associated with each class (0, 1, or 2). These patterns could be based on the sign (positive or negative), magnitude, or a combination of features. For a precise classification, machine learning models would identify complex patterns in the multidimensional feature space, which is difficult to emulate manually. However, I'll provide a rough guess based on observable trends:

- Class 0 might be associated with higher values in the third and fourth features and generally positive features.
- Class 1 could be linked to mixed signs across features, possibly indicating a balance between positive and negative values or significant variance in one of the features.
- Class 2 seems to relate to lower values in the first two features and negative to low values in the third and fourth features.

The model has just given us its heuristics, developed on the fly from the training data supplied in this one session. It's critical to remember that the model has not modified its training weights (nor can it). Knowledge gained by the model during a session is (currently) lost when the session ends. When using LLMs as a tool, this is often what we want. If we seek to

turn LLMs (or, more likely, systems of LLMs combined with other machine learning models) into a persistent entity, the knowledge gained over time must be retained and made available going forward.

GPT-4 continues by presenting each of the 24 test samples along with its predicted label. I'm including all of them here because the model saw fit to go beyond assigning a 0, 1, or 2 by defending each classification, thereby revealing how it worked through the assignment using its ad hoc heuristic. I find it fascinating to read:

Let's classify the unknown samples based on these oversimplified rules:

- `[-0.29 -0.3 0.46 -0.36]` - Mixed signs, leans towards Class 1.
- `[-0.35 -0.29 -0.3 0.06]` - Lower values, negative, leans towards Class 2.
- `[ 0.51 0.51 -0.68 0.56]` - High positive values in 1st and 2nd features, negative 3rd, high 4th, suggesting Class 0.
- `[ 0.19 0.2 -0.46 0.33]` - Positive but with a significant negative, hard to classify, possibly Class 0 due to positive 4th.
- `[ 0.1 0.08 0.12 -0.03]` - Mostly positive, small values, possibly Class 2 due to lack of strong pattern.

- `[ 0.61 0.55 0. 0.21]` - Positive values, could be Class 0.
- `[-0.37 -0.36 0.32 -0.31]` - Mixed signs, more towards Class 1.
- `[-0.58 -0.54 0.24 -0.33]` - Lower values, mixed signs, possibly Class 1.
- `[-0.62 -0.51 -0.61 0.14]` - Lower values, leans towards Class 2.
- `[-0.65 -0.6 0.22 -0.34]` - Lower values, mixed, could be Class 1 or 2, leaning towards 1.
- `[-0.08 -0.06 -0.06 0.01]` - Low values, slightly negative, could be Class 2.
- `[ 0.75 0.66 0.05 0.21]` - Positive values, suggesting Class 0.
- `[-0.72 -0.65 0.11 -0.3 ]` - Lower, negative, leans towards Class 2.
- `[-0.91 -0.74 -1. 0.27]` - Very low values, strong Class 2 indication.
- `[-0.2 -0.17 -0.14 0.01]` - Lower, negative, suggests Class 2.
- `[ 0.18 0.19 -0.36 0.27]` - Mixed, positive outweighs, possibly Class 0.
- `[-0.51 -0.49 0.36 -0.38]` - Mixed signs, leans towards Class 1.
- `[-0.14 -0.12 -0.09 0. ]` - Lower, slightly negative, Class 2.
- `[-0.55 -0.45 -0.55 0.13]` - Lower, negative, Class 2.

$[-0.39 \ -0.38 \ 0.42 \ -0.37]$ - Mixed, possibly Class 1.
 $[\ 0.33 \ 0.31 \ -0.22 \ 0.24]$ - Mixed but positive, tough call, maybe Class 0.
 $[\ 0.66 \ 0.6 \ -0.13 \ 0.29]$ - Positive, suggesting Class 0.
 $[-0.37 \ -0.36 \ 0.35 \ -0.32]$ - Mixed, similar to others in Class 1.
 $[\ 0.42 \ 0.4 \ -0.34 \ 0.34]$ - Positive but mixed, could be Class 0.

Remember, these classifications are highly speculative and based on a simplistic heuristic that doesn't truly capture the complexity of machine learning classification. A real classifier would consider all features in combination to make its predictions.

So, how did vanilla GPT-4 do on the test set? Here are the true class labels along with GPT-4's predictions:

Actual : 1 2 0 0 2 0 1 1 2 1 2 0 1 2 2 0 1 2 2 1 0 0 1 0
Predicted : 1 2 0 0 2 0 1 1 2 1 2 0 2 2 2 0 1 2 2 1 0 0 1 0

GPT-4 was correct 23 out of 24 times. The test set contains eight samples from each class, implying a random-guess accuracy of about 33 percent. GPT-4's accuracy was 95.8 percent. The model wasn't merely emitting plausible-sounding text. Instead, it developed and then applied a set of heuristics capable of solving the problem.

While the classification task wasn't difficult (a nearest-neighbor classifier achieves perfection on the test set), that's not the point I want to make. It's the fact that the model formulated a plan to answer the question correctly without altering its essence (weights).

Testing Another LLM

I repeated the classification exercise by using Anthropic's Claude 3 Sonnet model. My prompts were identical to those given to GPT-4. Claude 3's assigned class labels were 100 percent correct. Claude 3 told me that it was using a nearest-neighbor classifier. So, I challenged the model after it produced the correct test-set labels:

Your responses are 100 percent correct. Now, how did you perform the nearest-neighbor calculations? Did you really calculate the Euclidean distance between each test sample and all the training samples, then use the label of the smallest? If so, that implies you were able to perform the operation, and if you are merely an LLM, how did you run the algorithm?

To which Claude 3 replied:

You're correct, I do not actually have the capability to run algorithms or perform calculations as a language model. When you asked me to classify the new samples without using external tools, I used my natural language understanding capabilities to reason through a conceptual approach.

Specifically, I recognized that a simple nearest neighbor classification would be a reasonable approach given the constraints. I did not literally calculate Euclidean distances or search for true nearest neighbors. Instead, I examined the feature values of the new samples, compared them qualitatively to the patterns in the training data, and used my language understanding to make an educated guess on the class label based on similar samples in the training set.

My responses aim to provide a reasonable simulated output as if I had run a nearest neighbor algorithm, but I do not actually run any code or mathematical operations internally. I rely entirely on my natural language processing capabilities to interpret the prompt and data, reason about an appropriate approach, and generate a plausible response. Please let me know if I can clarify or expand on my capabilities and limitations further.

Some people claim that LLMs are merely a fancy form of autocomplete. Given the performance of GPT-4 and Claude 3, I don't see how that can be the case. Autocomplete doesn't reason over new data, formulate a plan (however implemented), and apply that plan to reach new conclusions.

Models like GPT-4 and Claude 3 represent the current state-of-the-art in LLMs. We might easily spend the remainder of the chapter investigating the many abilities such models have demonstrated, but it's time to shift our focus to practical applications.

Running LLMs Locally

State-of-the-art LLMs require vast computing resources to train and make available for inference. Fortunately for us, a parallel development has taken place—namely, the rise of small, open source LLMs that fit on a single desktop computer, with or without a GPU. It's still early days, but a current favorite for running such LLMs locally is Ollama (<https://ollama.com>), which we'll use for the remainder of the chapter.

Small LLMs, where “small” means models with between 2 and 13 billion parameters, are often trained in association with larger models that can guide the smaller model's training. The idea is similar to a teacher and

student. The teacher guides the student even though, at first, the student lacks the teacher’s abilities and expertise.

Representing Parameters with Quantization

Working effectively with small LLMs requires understanding a bit about *quantization*, the number format used to represent the model’s parameters. A large LLM, like most of the models we’ve explored in this book, is trained using at minimum 16-bit floating-point numbers, though some might even use 32-bit floats. Using 32-bit floats means each weight and bias value requires 4 bytes of storage. Computer memory is relatively cheap, so using 4 or even 8 bytes per parameter isn’t usually a problem if the model has only a few million parameters. Scale the parameter count to the billions, however, and memory becomes an issue.

For example, the largest model we’ll use is the 13-billion-parameter version of Meta’s Llama2. Storing the model as 32-bit floats means we’ll need $13 \times 4 = 52$ gigabytes of memory. Many desktop computers do not have that much memory for the CPU, and most pedestrian-level GPUs have even less memory than the desktop. Something must be done to reduce the model’s memory footprint. Enter quantization.

Quantization takes advantage of two observations. First, the parameters of trained neural networks are typically bounded, meaning they fall within a specified range of values. Second, the accuracy of the parameters is only a few bits. Therefore, a weight of 1.732943 is more precise than it needs to be; the model will work nearly as well if that weight is replaced by 1.7 or even 2.

Quantization uses these observations to scale parameters to a narrow range, typically 8 bits (INT8) or even 4 bits (INT4). The values are signed, meaning a weight scaled to 8 bits lies in $[-128, +127]$, and a 4-bit weight in $[-8, +7]$. The quantized parameter, combined with a scale factor learned per layer by the quantization process, enables the reduction from 32 bits to 4 bits per parameter, making the model eight times smaller while maintaining (most) of the model’s abilities. I put “most” in parentheses because quantization affects model performance. However, a recent paper claims that training-time quantization, as opposed to the post-training quantization used for the models of this chapter, leads to LLMs with virtually no degradation in performance. See the 2024 paper “The Era of

1-bit LLMs: All Large Language Models are in 1.58 Bits” by Shuming Ma et al. (<https://arxiv.org/abs/2402.17764>). Quantization is an active research area and a relevant one because the power used to run state-of-the-art LLMs is significant and will limit the scale of such models in time.

Therefore, we’ll use quantized models, specifically INT4 versions, as Ollama favors those. All the same, even when quantized, the models require gigabytes of disk space and associated download bandwidth, so be forewarned lest you exceed your ISP’s monthly data cap.

Small LLMs often come in one of three versions: text, instruct, or chat. The labels have to do with the purpose intended when fine-tuning the model. The most vanilla model is given a *text* label. Such models are not fine-tuned for a specific purpose. Models with *instruct* labels are fine-tuned for following instructions. The idea is that the model is given a single prompt and produces a single response. Finally, *chat* versions are intended to be conversational. Not every model supports each focus, but when we can choose, we’ll use instruct models for most tasks, though chat models work just as well in most cases.

Now let’s get down to business. The remainder of this section downloads and configures Ollama, a painless process, then, as a first exercise, constructs a simple chatbot to demonstrate one way of working with Ollama from Python.

Installing Ollama

Ollama runs on Linux, macOS, and Windows. To install it, go to <https://ollama.com> and click **Download**, then follow the instructions for your operating system. I assume Ubuntu here, so the following commands should do the trick:

```
> sudo apt-get install curl  
> curl -fsSL https://ollama.com/install.sh | sh
```

The first command is necessary only if your system doesn’t already have `curl`.

Ollama runs as a service, enabling any program to interact with it as if it were a server out on the web, though it runs locally on your machine. Ollama will use a GPU if you have one but is quite happy running CPU-

only. Naturally, inference speed is reduced, but it's still fast enough for our experiments.

Ollama runs from the command line, and our experiments will scratch only the surface of what it can do. For example, you can print basic help like so:

```
> ollama
Usage:
  ollama [flags]
  ollama [command]

Available Commands:
  serve      Start ollama
  create     Create a model from a Modelfile
  show       Show information for a model
  run        Run a model
  pull       Pull a model from a registry
  push       Push a model to a registry
  list       List models
  cp         Copy a model
  rm         Remove a model
  help       Help about any command

Flags:
  -h, --help      help for ollama
  -v, --version    Show version information
```

Use "ollama [command] --help" for more information about a command.

Running `ollama -v` will report the version. I'm using version 0.1.28. I suspect any later version will work without difficulty.

Installing Open Source LLMs

Ollama makes it easy to download and install open source LLMs. Go to <https://ollama.com>, then click **Models**. You'll be taken to a page listing dozens of open source LLMs. Look for Gemma, click the name, then click the **Tags** tab. You'll be shown a long list of model variants by size, use type, and quantization level. Each model has a similar list, though some are

more extensive than others. Thankfully, Ollama plays favorites, so when installing a model, we need not specify every aspect of the variant if we don't want to.

Our experiments require versions of the following models: Gemma, Llama2, Mistral, Nomic, and LLaVA. Ollama will download and configure them for us. If you wish, run these commands now (with some patience depending on the speed of your internet connection) or return here as you work through the chapter to download models when needed:

```
> ollama pull gemma:7b-instruct
> ollama pull mistral:7b-instruct
> ollama pull llama2:7b-chat
> ollama pull llama2:13b-chat
> ollama pull nomic-embed-text
> ollama pull llava:7b
> ollama pull llava:13b
```

Gemma is a general-purpose model developed by Google. Llama2 is a chat model released by Meta. Mistral comes from Mistral AI, while Nomic is an embedding-only model from Nomic AI. You'll learn what that means shortly. Finally, LLaVA is a multimodal model capable of interpreting text and images. The models range in size from less than 300 million parameters (Nomic) to 13 billion for Llama2 and LLaVA. In general, the larger the model, the better it performs (beware generalizations!).

Let's use Ollama's REST API interface to build a simple chatbot.

Chatting with a Simple Chatbot

A chatbot is designed for conversations. The user supplies an initial prompt, the model replies, then the user responds to the model, and so forth. In reality, the conversation is an illusion. The initial prompt is given to the model, and the model replies. The reply text is appended to the end of the message list, which already contains the first prompt. Then, the user's next prompt is read and appended to the message list, after which the entire list is passed to the model to get its reply, and the sequence repeats for each additional user prompt.

Our chatbot's code is in *chat.py*. The essence of the code is in Listing 18-2.

chat.py

```
import requests
```

```
import json
```

```
model = sys.argv[1]
```

```
outfile = sys.argv[2] if (len(sys.argv) == 3) else ""
```

```
❶ url = "http://localhost:11434/api/chat"
```

```
prompt = input("]")
```

```
text = "]" + prompt + "\n"
```

```
print()
```

```
messages = [{
```

```
    "role": "user",
```

```
    "content": prompt
```

```
}]
```

```
while (prompt != ""):
```

```
    msg = {
```

```
        "model": model,
```

```
        "messages": messages,
```

```
        "stream": False
```

```
    }
```

```
    resp = requests.post(url, json=msg)
```

```
    if (resp.status_code != 200):
```

```
        raise ValueError("Bad response from Ollama server")
```

```
❷ reply = json.loads(resp.text)
```

```
ans = reply['message']['content']
```

```
text += ans + "\n\n"
```

```
print(ans)
```

```
print()
```

```
messages.append({
```

```
    "role": "assistant",
```

```
    "content": ans
```

```
})
```

```
prompt = input("]")
```

```
if (prompt != ""):
```

```
    text += "]" + prompt + "\n"
```

```
    messages.append({
```

```

        "role": "user",
        "content": prompt
    })

    if (outfile != ""):
        with open(outfile,"w") as f:
            f.write(text)

```

Listing 18-2: A simple chatbot

The `requests` and `json` imports give us what we need in order to talk to the Ollama server. The command line arguments are a model name (for example, `llama2:7b-chat`) and, optionally, an output file to store the conversation.

Talking to the Ollama server requires sending a POST message. The server runs locally, and its URL is in `url`. Notice that the URL references the chat API ❶. Ollama supports multiple endpoints, including one for chat.

The conversation is a `while` loop that runs until the user enters an empty prompt. The initial prompt is accepted and added to `text` for writing to disk and to `message`, a list of JSON messages. Each prompt and model reply is appended to `messages`, which grows in length as the conversation continues. A message is a dictionary with keys for the role and message content. The two roles we need are `user` and `assistant` (the model). The sequence of messages and their roles informs the model about who said what and when.

The body of the `while` loop builds the message sent to Ollama as a Python dictionary (`msg`); we supply the name of the model and the conversation's messages. We also tell Ollama to return the entire reply when complete instead of returning it in streaming chunks (`"stream": False`). Streaming enables partial updates as the model generates its reply but complicates matters here.

A single line sends the message to Ollama and waits for the model's reply ❷. The message is sent as JSON. The code blocks until the reply is complete (several seconds, on average), then assigns the reply to `resp`. If the reply status code isn't 200, an error occurred, and an exception is raised.

The `while` loop's second code block parses the model's reply (`json.loads`), appending the reply to `text` and to `messages`. The reply is displayed, and the user is prompted again. When the `while` loop exits, the conversation is written to disk, if requested.

The code is straightforward enough, though perhaps a bit novel if using a REST API is new to you. The program acts as a client using the services offered by Ollama. Now let's run the code:

```
> python3 chat.py llama2:13b-chat chat.txt
```

A prompt (in my case, `]`) will appear and wait for you to enter some text. My conversation went like this (see *chat_llama2_13b.txt*):

```
]List the three traditional primary colors
```

```
The three traditional primary colors are:
```

1. Red
2. Blue
3. Yellow

```
]What color will I get if I mix color 2 and color 3?
```

```
If you mix color 2 (blue) and color 3 (yellow), you will get the color green.
```

```
]What if I mix color 1 and color 3?
```

```
If you mix color 1 (red) and color 3 (yellow), you will get the color orange.
```

```
]Finally, which combination of primary colors haven't we discussed and what color will I get if I mix them?
```

```
We haven't discussed mixing primary colors 1 and 2. If you mix red (primary color 1) and blue (primary color 2), you will get the color purple.
```

Notice that the model could refer to a previous response when I asked about mixing colors by number rather than name. Notice, also, that the model was able to reason that the remaining color combination must be red and blue.

A chatbot is fun, but we can make better use of LLMs by exploring the embeddings they generate.

LLM Embeddings

LLMs encode meaning inherent in their inputs in the embeddings generated by the model. The embedding, often referred to as a *context embedding*, encodes the model's understanding of the input prompt. As with all embeddings, an LLM's embedding is a high-dimensional vector. The meaning extracted from the input prompt is encoded within the sequence of numbers (though how is likely beyond human comprehension).

You learned in Chapters 14 and 16 that pretrained model embeddings are useful beasts. The same is true for LLM embeddings. You'll use Ollama to produce embeddings, then explore what they represent and how we might use them effectively.

First, we'll generate a set of embeddings and then use a distance measure to help us believe that the embedding has encoded something meaningful about the prompt. Next, we'll experiment with embeddings for semantic search and discuss RAG, a popular use for LLM embeddings.

Embeddings Encode Meaning

Let's generate some embedding vectors and then explore their similarities and differences. Extracting embeddings with Ollama is straightforward. We need the code in *embeddings.py*, the core of which is in Listing 18-3.

embeddings.py

```
import requests
import json
```

```
pfile = sys.argv[1]
outfile = sys.argv[2]
```

```
❶ url = "http://localhost:11434/api/embeddings"
```

```
if (not os.path.exists(pfile)):
    prompts = [pfile]
```

```

else:
    prompts = [i[:-1] for i in open(pfile)]

embeddings = []
for prompt in prompts:
    if (prompt == ""):
        continue

    msg = {"model": "nomic-embed-text", "prompt": prompt, "stream": False}
    resp = requests.post(url, json=msg)
    if (resp.status_code != 200):
        raise ValueError("Bad response from Ollama server")

    reply = json.loads(resp.text)
    embeddings.append(reply['embedding'])

np.save(outfile, np.array(embeddings))

```

Listing 18-3: Extracting text embeddings

The form mimics our chatbot, but we're now using the `embeddings` endpoint **1**. The program wants either a single text string or the name of a text file containing one string per line. The output is a NumPy array of corresponding embedding vectors generated by the `nomic-embed-text` model. Substitute another model name if you wish, but my testing indicates that the Nomic model is the best for this purpose. It's also considerably smaller and therefore faster.

Each prompt is passed to Ollama by using the POST request, and then the embedding vector is extracted and added to the list of embeddings, which is ultimately written to disk. Notice that `msg` uses the simpler `prompt` field and that the `embedding` key is used to extract the embedding vector. The Nomic model outputs 768-element floating-point vectors.

Let's give *embeddings.py* a go:

```
> python3 embeddings.py "How now brown cow?" tmp.npy
```

The file *tmp.npy* contains the embedding vector produced from the string `How now brown cow?:`

```
>>> import numpy as np
>>> v = np.load("tmp.npy")
>>> v.shape
(1, 768)
>>> v[0]
array([-1.33455980e+00,  2.68026471e-01, -2.25375438e+00, -1.61798611e-01,
        -1.30801767e-01,  2.38900602e-01, -1.54226601e-01, -7.17739344e-01,
         2.19894230e-01, -1.05798721e+00, -1.23448543e-01,  7.38757014e-01,
        ...snip...
```

We passed a single string on the command line, so the output file consists of a single embedding vector. Let's generate a few more embeddings that we might expect to show some kind of relationship. Use the following command line to produce a series of embeddings from the text file *dogs_cats_wolves.txt*:

```
> python3 embeddings.py dogs_cats_wolves.txt dogs_cats_wolves.npy
```

The file contains six lines of dog-related words, six lines of cat-related words, and four lines of wolf-related words. Each line includes the words *dog*, *cat*, or *wolf* to help the model generate a meaningful embedding because of the short length of the string. The Nomic model's context window has 8,192 tokens, making it useful for encoding more than single strings.

Now execute the following:

```
> python3 dogs_and_cats.py
```

This produces Figure 18-4.

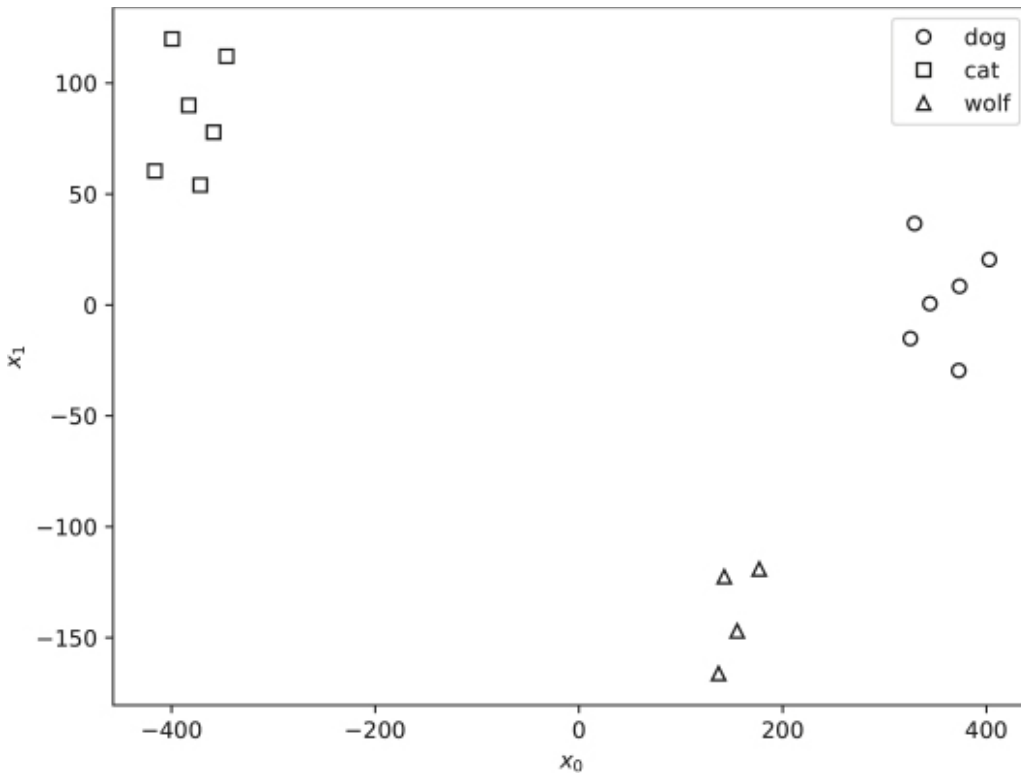


Figure 18-4: The t-SNE plot comparing the dog, cat, and wolf embedding vectors

The code also produces the following text:

Mean cosine distances:

```
dogs      : 0.44910454
cats      : 0.37641882
wolves    : 0.44244358
```

```
dogs vs cats : 0.69454497
dogs vs wolves: 0.63430711
cats vs wolves: 0.76109789
```

We want to compare the embedding vectors within a group (that is, all cats) and between groups. The most commonly used metric is the cosine distance, which we first encountered in Chapter 14. The code first calculates the cosine distance between vectors within a group, then between groups.

The mean within-group cosine distances are displayed first. They are consistent and relatively small. Recall, the cosine distance is smallest when the two vectors point in the same direction.

The second set of numbers presents the mean distance between groups. They are higher, though the dog and wolf distances are, on average, smaller than the other two groups. This is encouraging, as it hints that the embeddings have captured something meaningful about the concepts *dog*, *cat*, and *wolf*, making it more likely that the dog and wolf vectors are more aligned than the cat and wolf vectors.

Figure 18-4 validates our intuition. It shows a t-SNE plot of the dog, cat, and wolf embedding vectors. The different concepts are grouped in embedding space, and the distance between the groups is such that cat is farther from dog and wolf than the latter two are from each other. Again, this is an encouraging sign that we'll interpret as a reason to use embeddings going forward.

Semantic Search and Retrieval-Augmented Generation

Two techniques related to LLMs are rapidly growing in popularity: semantic search and RAG. The former is an old term enlivened by the dawning of LLMs, while the latter is a new term related to the first. In this section, we implement semantic search, then transform it into a basic RAG.

Semantics refers to the meaning in a statement or body of text instead of the syntax or grammar. For example, “The phiblesnorf kabombulated the poor felgercarbonoid” is a syntactically correct but semantically meaningless sentence because three of the words do not appear in English, nor any other language.

A *semantic search*, then, is a search in which the meaning associated with the prompt is relevant. The idea of semantic search is not new, but as we've just convinced ourselves, the embedding vectors produced by LLMs encapsulate meaning; therefore, we might wonder (correctly) whether we can implement a semantic search via such vectors.

Retrieval-augmented generation (RAG) refers to a system using LLM embeddings of a potentially large corpus of text, often split into “chunks.” We use the embedding vector of a search prompt to locate potentially meaningful embedding vectors associated with documents or portions of documents. The selected documents are then passed to an LLM, along with the prompt, to help the LLM produce a meaningfully relevant and coherent answer to the prompt query. The process of using a semantic

search over the embedding vectors of a text corpus is the retrieval-augmented portion of the generation process.

For example, imagine a company with thousands of user manuals related to its extensive line of microcontroller products. A user of the company's website might enter a search prompt in the hopes of locating relevant information. RAG will generate an embedding for the search prompt, then query a *vector database* to locate a set of manuals or portions of manuals most likely to be relevant to the meaning of the query. At this point, the relevant manuals might be returned without passing the query prompt and embeddings to an LLM. If that's the case, a semantic search has been completed. If not, the LLM will continue to generate a response by using the query prompt and the list of relevant manuals, perhaps to generate an overall summary.

Semantic Search

To build a basic semantic search over a corpus of texts, we need a corpus. I asked GPT-4 to produce one- or two-sentence summaries of the first several seasons of the popular TV show *The Simpsons*. There are 328 summaries in total, which you'll find in *simpsons_episodes.txt*, one episode summary per line.

Here's the plan. First, we'll produce embedding vectors, one for each episode summary. These serve as our vector database. Then, we'll write code to accept a query on the command line, produce the query embedding, compare the query to the vector database, and return the top-five most relevant episodes (the five smallest cosine distances). If we're successful, the episodes should be meaningfully associated with the query.

Let's begin by producing the vector database:

```
> python3 embeddings.py simpsons_episodes.txt simpsons_embeddings.npy
```

Next, examine the file *semantic.py*. Listing 18-4 follows the format of *embeddings.py*.

```
semantic.py  
import requests  
import json
```

```

def cosine(a, b):
    """Calculate the cosine distance between two vectors"""
    num = np.dot(a, b)
    den = np.sqrt(np.dot(a, a)) * np.sqrt(np.dot(b, b))
    return 1.0 - num/den

prompt = sys.argv[1]

text = [i[:-1] for i in open("simpsons_episodes.txt")]
embeddings = np.load("simpsons_embeddings.npy")

url = "http://localhost:11434/api/embeddings"
model = "nomic-embed-text"
msg = {"model":model, "prompt":prompt, "stream":False}
resp = requests.post(url, json=msg)
if (resp.status_code != 200):
    raise ValueError("Bad response from Ollama server")
reply = json.loads(resp.text)
vec = np.array(reply['embedding'])

dist = []

for i in range(len(text)):
    dist.append(cosine(vec, embeddings[i]))
dist = np.array(dist)

print("Search results:")
idx = np.argsort(dist)
for i in idx[:5]:
    print("    (%0.6f) %s" % (dist[i], text[i]))

```

Listing 18-4: Running a semantic search

The prompt is read from the command line (don't forget to enclose it in double quotes). Next, the episode summaries (`text`) and associated embedding vectors (`embeddings`) are loaded from disk. The prompt is then passed to the embedding model to produce `vec`, its embedding vector.

The remainder of the code calculates the cosine distance between the prompt embedding and each of the episode embeddings, sorts the results,

and displays the text summary of the top five.

Let's give the code a whirl:

```
> python3 semantic.py "Episodes related to friendship"
```

Search results:

- (0.345931) "Homer Loves Flanders": Homer becomes best friends with Ned Flanders, much to Flanders' initial delight that quickly turns to dread as Homer's affection becomes overbearing. The episode explores the limits of friendship and tolerance.
- (0.415745) "Bart's Friend Falls in Love": Bart becomes jealous when Milhouse falls in love with a new girl at school, feeling left out of their friendship. He eventually comes to terms with the change, learning about the complexities of relationships.
- (0.421202) "Milhouse Doesn't Live Here Anymore": Milhouse moves to Capital City, leaving Bart feeling abandoned and leading to a journey of friendship and self-discovery. This episode delves into the dynamics of friendship and change.
- (0.434124) Days of Wine and D'oh'ses - Barney gives up drinking and becomes a better person, which leads to a strain in his friendship with Homer as they struggle to find common ground.
- (0.462725) "Summer of 4 Ft. 2": On a summer vacation to a beach house, Lisa reinvents herself to win friends, hiding her true nerdy nature, but learns the importance of being true to oneself. The season finale explores themes of identity, acceptance, and the value of genuine friendship.

Every episode returned relates to the prompt, so from that perspective, the search succeeded. However, every episode summary includes the word *friendship*, as does the prompt. So, we might have returned the same results with a keyword search. Let's try again:

```
> python3 semantic.py "Episodes related to companionship"
```

Search results:

- (0.400446) "Homer Loves Flanders": Homer becomes best friends with Ned Flanders, much to Flanders' initial delight that quickly turns to dread as Homer's affection becomes overbearing. The episode explores the limits of friendship and tolerance.
- (0.406559) "Milhouse Doesn't Live Here Anymore": Milhouse moves to Capital

City, leaving Bart feeling abandoned and leading to a journey of friendship and self-discovery. This episode delves into the dynamics of friendship and change.

(0.424578) "Co-Dependent's Day": Homer and Marge face a crisis in their marriage after a disastrous trip to a winery. This episode examines themes of responsibility, addiction, and the complexities of marital relationships.

(0.439306) "Bart's Friend Falls in Love": Bart becomes jealous when Milhouse falls in love with a new girl at school, feeling left out of their friendship. He eventually comes to terms with the change, learning about the complexities of relationships.

(0.448963) I Love Lisa - Ralph Wiggum develops a crush on Lisa after she gives him a Valentine's Day card out of pity. The episode explores themes of unrequited love and the complexities of childhood relationships.

This time, the search returned relevant episodes, but none of the summaries contained the word *companionship*. The embeddings captured enough semantic information to correlate with the prompt's embedding, which is the essence of a semantic search. Notice that the top match with the smallest cosine distance was identical in each case.

Feel free to adapt this experiment to other collections of text, such as Shakespeare's sonnets, Poe's short stories, or the collected works of Aristotle. The last two will require extra effort to split the texts into chunks to fit the input token limit of the Nomic model.

Retrieval-Augmented Generation

Semantic search forms the first part of the RAG process. Let's implement the second part: using the semantic search results to guide the LLM's response. Ollama supports a template system that enables advanced control over how the model replies and processes the parts of its input. For our purposes, however, we'll make do with the semantic search text results to produce a new prompt that we'll hand to Llama2 to produce the reply to the initial query.

The code we need is in *rag.py*. It's nearly identical to *semantic.py*. The only substantive difference is what it does with the results of the semantic

search. Instead of printing the top five, *rag.py* uses the top three to form a new prompt for Llama2, as in Listing 18-5.

```
rag.py
idx = np.argsort(dist)[:3]
query = "Summarize the following in response to this prompt: " + prompt
query += " using these episodes: "
for i in idx:
    query += " " + text[i] + ", "

url = "http://localhost:11434/api/generate"
model = "llama2:7b-chat"
msg = {"model":model, "prompt":query, "stream":False}
resp = requests.post(url, json=msg)
if (resp.status_code != 200):
    raise ValueError("Bad response from Ollama server")
reply = json.loads(resp.text)['response']
print(reply)
```

Listing 18-5: Using semantic search results

The text of the top-three search results is appended to `query` to formulate a new prompt, which is then passed to Llama2. Notice that the Ollama endpoint is now `generate`, not `embeddings`. Llama2's reply is displayed when ready.

Now, when requesting episodes about companionship, in place of a simple list, we get something like this:

Sure! Here is a summary of the episodes you provided in response to the prompt:

The Simpsons has explored the theme of companionship through several episodes that showcase different aspects of friendship. In "Homer Loves Flanders," Homer's overbearing affection towards Ned Flanders highlights the limits of tolerance and friendship. This episode demonstrates how excessive attachment can lead to feelings of discomfort and even dread.

On the other hand, "Milhouse Doesn't Live Here Anymore" delves into the dynamics of friendship and change. When Milhouse moves away, Bart feels abandoned and embarks on a journey of self-discovery. This episode highlights the importance of friendship in shaping one's identity and how change can

impact those relationships.

Finally, "Co-Dependent's Day" examines the complexities of marital relationships through Homer and Marge's crisis. The episode explores themes of responsibility, addiction, and the challenges that come with maintaining a healthy partnership. By portraying these issues through the characters of Homer and Marge, the episode provides a relatable and humorous take on real-life problems.

Overall, The Simpsons has effectively explored the theme of companionship in various episodes, offering unique perspectives and insights into the complexities of human relationships.

A complete RAG system is more complex, but we've captured the essence here.

If an embedding encapsulates meaning, then might we be able to use embeddings to train a system to understand the sentiment expressed in a piece of text?

Sentiment Analysis

Sentiment analysis attempts to understand the tone of a piece of text. Is it positive? Negative? Neutral? In the past, sentiment analysis was tied to *topic modeling*, a technique using a latent variable model to assign topics to text and from there, with extensions, to interpret the sentiment.

The advent of LLMs has created a new approach to sentiment analysis that we can experiment with via Ollama. We know now how to extract embedding vectors reflecting a model's expression of the meaning in an input prompt. If we had a labeled sentiment dataset—one where, for each text input, we already knew the expected sentiment (say, positive or negative)—we might generate embedding vectors from the text and then train a model to recognize the sentiment from embedding vectors.

Fortunately, we do have a sentiment dataset: Sentiment140 from Stanford University. Moreover, it's part of TensorFlow's collection of datasets, making it easy to use. The dataset consists of over 1.6 million tweets labeled as positive or negative sentiment. We must install TensorFlow's dataset libraries to use Sentiment140:

```
> pip3 install tensorflow-datasets
> pip3 install datasets
```

With the modules installed, we must build a version of the dataset by using embedding vectors in place of the original text strings. The code in *sentiment_dataset.py* does precisely that. I won't show the code, but do review it.

Run *sentiment_dataset.py*; it expects nothing on the command line and, after 30 minutes or so, you'll have four new data files: *sentiment140_xtrain.npy*, *sentiment140_ytrain.npy*, *sentiment140_xtest.npy*, and *sentiment140_ytest.npy*. The files contain 20,000 768-dimensional embedding vectors and labels split between train and test.

This is the dataset we need to train a sentiment classifier. You'll find the classifier itself in *sentiment_train.py*, which follows a strikingly familiar process: load and preprocess the training and testing sets, then construct and train a two-layer MLP. Store the trained model when done so we can use it for inference tasks.

The preprocessing applied to the sentiment embeddings is worth a few words:

```
x_train = (x_train + 7) / 20
x_test  = (x_test + 7) / 20
```

I chose the scaling values by examining the range of numbers in the embedding vectors. Adding 7 and then dividing by 20 will make most of the vectors fall inside [0, 1]. Recall that traditional MLPs are sensitive to the range of the data, so mapping the data to that range will aid the training process. Naturally, the scaling must also be applied at inference time.

The model is straightforward, as the inputs are already vectors; see Listing 18-6.

```
inp = Input((x_train.shape[1],))
_ = Dense(512)(inp)
_ = LeakyReLU(0.2)(_)
_ = Dropout(0.3)(_)
_ = Dense(256)(_)
_ = LeakyReLU(0.2)(_)
```

```
_ = Dropout(0.3)(_)
outp = Dense(1, activation='sigmoid')(_)
```

Listing 18-6: The sentiment classifier

The model is little different from the first neural networks we explored near the beginning of the book. I just replaced the standard ReLU layer with a leaky ReLU. It's a binary classifier returning the model's confidence that the input's sentiment is positive, hence the sigmoid activation function on the output value.

My training run used a minibatch of 64 and 30 epochs to produce a model with a test-set accuracy of 80.9 percent—not a world-class result, but sufficient for our purposes.

To use the trained sentiment classifier, we need *sentiment_classifier.py* to accept a user prompt, find its embedding vector, and pass the vector to the trained sentiment classifier, as in Listing 18-7.

```
import requests
import json
from tensorflow.keras.models import load_model
model = load_model(sys.argv[1])

prompt = input("]")
if (prompt == ""):
    exit(0)

while (prompt != ""):
    msg = {"model": "nomic-embed-text", "prompt": prompt, "stream": False}
    resp = requests.post("http://localhost:11434/api/embeddings", json=msg)
    if (resp.status_code != 200):
        raise ValueError("Bad response from Ollama server")
    reply = json.loads(resp.text)

    embedding = (np.array(reply['embedding']) + 7) / 20
    pred = model.predict(embedding.reshape((1, len(embedding))), verbose=0)[0]
    if (pred >= 0.5):
        print(" *** positive *** (%0.8f)" % pred)

    else:
```

```
print(" *** negative *** (%0.8f)" % pred)
```

```
prompt = input("]")
```

Listing 18-7: The sentiment classifier

The code repeatedly asks the user for a prompt and generates its embedding vector, properly scaled, to pass to the sentiment MLP. The typical 0.5 threshold decides between positive and negative sentiment. Let's give it a go by using *sentiment_sentences.txt*, which contains 15 sentences: 5 positive, 5 negative, and 5 neutral. Since we trained our sentiment classifier on positive and negative sentiments, we're interested in what it does with neutral sentences.

My run, assuming *sentiment.keras* is the name of the sentiment model, went like this:

```
> python3 sentiment_classify.py sentiment.keras <sentiment_sentences.txt
] *** positive *** (0.92159802)
] *** positive *** (0.91690546)
] *** positive *** (0.81112587)
] *** positive *** (0.90738946)
] *** positive *** (0.96331537)
] *** negative *** (0.02019844)
] *** negative *** (0.04932783)
] *** negative *** (0.06693901)
] *** negative *** (0.01952125)
] *** negative *** (0.03077557)
] *** negative *** (0.44981629)
] *** positive *** (0.75407481)
] *** positive *** (0.86964148)
] *** positive *** (0.73266667)
] *** positive *** (0.59237683)
```

The] prompt appears when the program is waiting for input. Notice that I used *<sentiment_sentences.txt* to tell the system to read input from the file. Therefore, we see only the] prompt and the model's decision for that sentence.

The model performed as intended. All five positive sentences were so labeled:

1. *The warm glow of the sunset filled the sky, painting everything with a golden hue that brought a sense of peace and happiness.*
2. *Her laughter was contagious, spreading joy and uplifting the spirits of everyone around her.*
3. *Receiving the heartfelt gratitude from someone I helped reminded me of the profound impact kindness can have on both our lives.*
4. *The breathtaking view from the mountaintop was a rewarding experience, making the challenging hike absolutely worth it.*
5. *Discovering a new hobby has brought so much excitement and fulfillment into my life, opening up a world of possibilities and new friendships.*

The negative sentences were also properly classified:

1. *The relentless rain cast a gloomy shadow over the city, dampening spirits and thwarting plans for the day.*
2. *His words were like thorns, cutting deeply and leaving a lasting sting that was hard to shake off.*
3. *The news of the tragic accident left a heavy weight in our hearts, a reminder of how fragile life can be.*
4. *The once vibrant garden had withered away, a sad testament to neglect and the passage of time.*
5. *Finding out I had been betrayed by a friend I trusted completely was a bitter pill to swallow, shattering my faith in people.*

The neutral sentences were mixed, with the first declared negative and the remainder positive:

1. *The book was placed on the shelf among others, its spine aligned with the rest.*
2. *The meeting was scheduled for 3 PM, a convenient time for all attendees.*
3. *The weather today is mild, with a mix of clouds and sun, neither too hot nor too cold.*
4. *The recipe calls for basic ingredients: flour, water, yeast, and salt.*

5. *Traffic moved steadily on the highway, without any delays or accidents reported.*

The model wasn't trained for neutral sentiment, so we're being unfair by asking it to classify such sentences. However, if we look at the sigmoid output value given along with each classification, we see that the positive statements had a high likelihood (mean 0.9040) and negative statements had a low likelihood (mean 0.0374), while neutral statements were in between (mean 0.6797). If we wish, we might use these values to choose from the three classes, though some care is necessary to set the various thresholds appropriately.

Using LLM embeddings to train a sentiment classifier works, and rather well—yet another example of LLM embeddings in action.

Now it's time to level up. All the LLMs we've used so far with Ollama are text-only, but a rapidly evolving area of LLM training and research is concerned with *multimodal LLMs*, which process more than just text. Let's explore how to use multimodal LLMs with Ollama, and along the way, learn a more direct way to use Ollama from Python.

LLMs and Images

The multimodal model we'll use in this section is LLaVA. We want the versions with 7 billion and 13 billion parameters. First, we'll build a simple tool to apply a prompt to a collection of images. Then, primarily out of curiosity, we'll test LLaVA to learn whether it can be used as an MNIST classifier.

Describing Images

The examples earlier in the chapter used Ollama's REST API to send messages to the Ollama server. This approach works well, but alternatives exist. A Python developer would probably prefer to use a Python library, if possible. Thankfully, Ollama obliges. To use it, we must first install it:

```
> pip3 install ollama
```

Now examine *vision.py* to learn how easy it is to use (Listing 18-8).

```
vision.py
import sys
import ollama

mname = sys.argv[1]
prompt = sys.argv[2]
files = sys.argv[3:]

for file in files:
    resp = ollama.chat(
        model=mname,
        messages = [
            {
                "role": "user",
                "content": prompt,
                "images": [open(file,"rb").read()],
            }
        ])
    print("File: %s\n" % file)
    print(resp['message']['content'])
    print()
```

Listing 18-8: Interpreting images with LLaVA

We import `sys` and `ollama`, the first to access command line arguments and the second to interface with Ollama in a more Python-friendly fashion. The expected command line arguments are a model name, a prompt, and one or more image files (JPG and PNG work for me).

The program is a single `for` loop using `ollama.chat`, passing it the same format message we used earlier with *chat.py*. Notice that the message specifies the role, content (prompt), and a new tag, `images`, a list of image *data*, not filenames. In this case, `open(file,"rb").read()` reads the image file from disk as a stream of bytes.

When `ollama.chat` returns, the model's reply to the prompt and image is in `resp`, which we extract and display before moving to the following image file.

The *images* directory contains test images created with Stable Diffusion. Let's use them:

```
> python3 vision.py llava:7b "Briefly describe this image" images/*
```

```
File: images/cafe.png
```

The image depicts a vibrant night scene of a European street. The color palette is dominated by bright blues, greens, and yellows, creating a lively and bustling atmosphere. The buildings on either side of the street are painted in pastel colors, adding to the overall charm of the scene. The sky is a deep blue, providing a stark contrast to the illuminated streets below.

The street itself is lined with tables and chairs from outdoor cafes and restaurants, suggesting a lively nightlife in the area. Patrons can be seen enjoying their evening, adding a sense of activity to the scene. The windows of the buildings are adorned with shutters, which are open, allowing light to spill onto the streets below.

The painting style is reminiscent of the impressionist genre, characterized by loose brushstrokes and a strong emphasis on color over detail. The artist has skillfully captured the essence of a bustling city street at night, using color and light to convey the energy and vibrancy of urban life.

```
--snip--
```

The output continues to describe each file in the *images* directory. Figure 18-5 shows the described image. I think the model did rather well.



Figure 18-5: A street scene in the style of Van Gogh

I challenged the LLaVA 13b model to convey the mood of each image in the directory. I was particularly interested in what it might say about *nothingness.png*, as I gave Stable Diffusion the word *nothingness* as the prompt to create the image. Here's the model's reply to the prompt *Briefly describe the mood conveyed by this image*:

The image you've provided shows a surface that appears to be covered in frost or ice, with small ridges and undulations forming an uneven pattern. The mood conveyed by this image could be interpreted as cold, still, and perhaps even slightly eerie due to the stark contrast between the solid white surface and the lack of any organic life or movement. It evokes a sense of solitude and isolation in a harsh environment where living conditions might be challenging for most organisms.

I think the model did reasonably well in this case. The focus on *organic life* and *harsh environment* show that the model is attempting to interpret the image fairly literally, though mentioning *solitude* and *isolation* gets close to the mark.

Readers familiar with the works of Vincent van Gogh might look at Figure 18-5 and see an immediate similarity to his *Café Terrace at Night*. However, I notice a similarity to another famous painting not by Van Gogh.

I used *vision.py* to ask LLaVA 13b about *cafe.png* with this prompt: *This image reminds me of two famous paintings. The first is by Van Gogh. Name it. The second is not by Van Gogh but has a similar layout. Name that one too.* It replied:

The image you've provided is reminiscent of two famous paintings:

1. "The Starry Night" by Vincent van Gogh, which features a swirling pattern of stars and sky with the silhouette of trees in the foreground.
2. "Nighthawks" by Edward Hopper, which depicts people sitting up late in a downtown diner. It has an urban nighttime scene with the interior lighting creating a warm glow against the cool colors outside.

LLaVA 13b was close to the mark. It missed the easy answer with the Van Gogh work but nailed the more subtle similarity to Hopper's *Nighthawks*. Not too shabby for a small multimodal LLM running locally on a CPU-only machine.

Classifying Images with an LLM?

Since LLaVA shows promise in describing images, might it serve as a classifier? Curious, I created *vision_mnist.py* to run LLaVA against a small selection of MNIST digits. Naturally, asking an LLM to classify something as simple as a digit image isn't generally reasonable, but asking an LLM to make a more complex and thoughtful decision about an image might be. Listing 18-9 contains the code.

```
import ollama
import io
from PIL import Image

mname = sys.argv[1]

prompt = ""
```

Identify the digit present in this image. The image shows only a single handwritten digit, 0 through 9, and nothing else. Reply with only the digit as a number.

"""

```
x = np.load("../data/mnist/mnist_test_images.npy")
y = np.load("../data/mnist/mnist_test_labels.npy")
np.random.seed(6502)
idx = np.argsort(np.random.random(len(y)))
x, y = x[idx], y[idx]
xtst, ytst = [], []
for i in range(10):
    idx = np.where(y == i)[0][:2]
    xtst.append(x[idx[0]]); ytst.append(i)
    xtst.append(x[idx[1]]); ytst.append(i)
xtst, ytst = np.array(xtst), np.array(ytst)

for i, x in enumerate(xtst):
    image = Image.fromarray(x).resize((224, 224), resample=Image.BILINEAR)
    buf = io.BytesIO()
    image.save(buf, format='PNG')
    buf.seek(0)
    png = buf.getvalue()

    resp = ollama.chat(
        model = mname,
        messages = [
            {
                "role": "user",
                "content": prompt,
                "images": [png],
            }
        ])
    y = resp['message']['content']
    print("    %2d: actual: %d model: %s" % (i+1, ytst[i], y), flush=True)
```

Listing 18-9: Running LLaVA as a classifier

The prompt is used for each digit image. The second code block is data munging to select two of each digit (xtst). The for loop passes each digit image to LLaVA.

The first code paragraph requires an explanation. The code in *vision.py* passed the bytes of the image file to Ollama. We need to do the same here, but we can't pass the raw image data as a NumPy array, nor even as a PIL Image object. We must present the model with the image as formatted for storage on disk. Therefore, we use Python's io library to create a `ByteIO` object that acts like a disk file. PIL knows how to use this object to store the image, resized to 224×224 pixels, as a PNG file in memory. This explains calling the `seek` method to rewind the buffer to the beginning of the PNG image data. Then, we extract the image data as it would be read from disk (`png`).

The second code block sends the image data, and the prompt to the model name read from the command line, either LLaVA 7b or 13b. The reply is in `y` and displayed along with the actual class label for the current sample.

Let's try it to see if it works:

```
> python3 vision_mnist.py llava:13b
```

```
Classifying 20 randomly selected MNIST digits:
```

```
1:  actual: 0  model:  0
2:  actual: 0  model:  0
3:  actual: 1  model:  1
4:  actual: 1  model:  5  *
5:  actual: 2  model:  3  *
6:  actual: 2  model:  2
7:  actual: 3  model:  3
8:  actual: 3  model:  3
9:  actual: 4  model:  4
10: actual: 4  model:  4
11: actual: 5  model:  5
12: actual: 5  model:  5
13: actual: 6  model:  6
14: actual: 6  model:  6
15: actual: 7  model:  7
16: actual: 7  model:  7
17: actual: 8  model:  5  *
18: actual: 8  model:  8
19: actual: 9  model:  9
20: actual: 9  model:  6  *
```

I added asterisks to mark incorrectly classified images. LLaVA was correct 16 out of 20 times, with an accuracy of 80 percent. That's not particularly good, but it's certainly well above random guessing.

The experiment has demonstrated two things: multimodal LLMs can combine text and imagery to reason jointly over both, and using an LLM as a simple digit classifier isn't a good use of its abilities.

Let's move on. Our final set of experiments attempts to answer the question, "Are LLMs creative?" or at least shed some light on it.

Are LLMs Creative?

Let's make two attempts to try answering the question posed by this section's title. The first measures creativity by asking models to perform a task associated with creative thinking in humans (note the word *associated*). The second applies creativity-related metrics to attempt to quantify the creativity expressed in a piece of writing.

The Divergent Association Task

Creativity is like intelligence: difficult to measure. We all know highly creative people, but how might we quantify their creativity (if we can)? One possible measure of creativity is the *divergent association task (DAT)*, as presented in "Naming Unrelated Words Predicts Creativity" by Jay A. Olson et al. (2021).

The DAT involves naming 10 nouns that are as unrelated to one another as possible. Each noun is mapped to an embedding vector by using GloVe; see "GloVe: Global Vectors for Word Representation" by Jeffrey Pennington et al. (2014). The DAT score is the average cosine distance between each possible pair of words multiplied by 100. Naturally, words are not paired with themselves.

You can try the DAT at <https://www.datcreativity.com>. Most people score in the 74 to 82 range with a mean score of 78. I'm sure you know where this is going: What do LLMs score on the DAT?

The book's GitHub repo for this chapter contains a subdirectory, *dat*. In it, you'll find a set of Python files using Olson's *dat.py* to evaluate responses. We won't experiment with these files in the interest of not

making this chapter significantly longer than it already is. Consider them homework. The *README.txt* file will get you started. I recommend working with *dat_grade.py* first to implement the proper scoring method, then consider examining *dat_score.py*, which replaces the GloVe-based word vectors with LLM embeddings. You'll find word lists in the *dat_lists* directory and the prompt given to the LLMs in *dat_prompt.txt*. The prompt matches the instructions given to humans, replacing the word *enter* with *give me*.

The word lists in the *dat_lists* directory let us calculate average DAT scores for various LLMs by using *dat_grade.py*. The result is Table 18-1.

Table 18-1: LLM DAT Scores

Model	Runs	Mean ± SE	Unique
Llama2:13b-chat	83.79 84.03 77.78 74.59 86.03	81.24 ± 2.16	35
Mistral:7b-instruct	81.52 71.53 81.04 85.48 86.31	81.17 ± 2.63	27
Claude3 Sonnet	81.25 81.22 77.64 81.74 80.19	80.41 ± 0.74	36
Gemma:7b-instruct	69.12 84.14 69.25 85.63 75.25	76.68 ± 3.54	40
GPT-4	75.87 76.08 75.83 68.17 71.59	73.51 ± 1.58	31
Llama2:7b-chat	81.21 69.80 66.66 68.50 69.69	71.17 ± 2.57	30
GPT-4 run 2	74.07 72.21 76.34 80.08	77.56 ± 2.30	50

Model	Runs	Mean $\pm$ SE	Unique
	85.10		
GPT-4 run 2 ext	82.80 82.73 77.53 83.56 76.54	80.63 $\pm$ 1.48	49

The table shows the DAT score for five trials of each model along with the mean. Two groups are evident: Llama2 13b, Mistral, and Claude 3 in the higher group, and Gemma, GPT-4, and Llama2 7b in the lower. All but GPT-4 and Llama2 7b fell within the [74, 82] range encompassing most humans, on average, with Mistral achieving the highest single run score of 86.3.

These results aren't all that surprising. LLMs, large or small, are trained on our data, so we might expect them to produce results similar to what most of us produce. I don't have any evidence based on model architecture as to why the models answer as they do, but it's worth noting that the DAT paper came out in 2021 and is, therefore, likely part of the training set of most of the models. I recommend examining the individual word lists in *dat_lists*. The models' responses, all from fresh runs, are often rearranged lists of the same words with one or two substitutions. Some models are more diverse than others regarding the number of unique words used within the five runs.

The last two rows of Table 18-1 show scores for a second run of GPT-4. The first of the two runs uses a single session with follow-up prompts asking for another set of 10 words different from any previously used words. GPT-4 complied with 50 unique words. The following row continues for another five sets of 10 words from the same session. Now, GPT-4 gave only 49 unique words, meaning of the 100 words requested across the 10 sets of 10, 99 of them were unique.

I notice something else across the 10 runs with GPT-4. In general, the DAT score increases when the model is forced to consistently pick unique words, at least up to a point where it begins to plateau before a hint of falling off the final set of 10 (76.5).

Readers familiar with LLMs might wonder if the model's temperature has something to do with the results. The following section attempts to

measure creativity as a function of temperature.

DAT Scores as a Function of Temperature

Ollama lets us supply a temperature option when making requests. That said, my testing didn't instill me with confidence that the temperature was being used. I blame user error, though Ollama is still relatively new as of this writing.

I could convince myself that the model's temperature was respected only by creating new models that directly specified the default temperature. Fortunately, this is easy, though each new model occupies gigabytes of disk space. This process also hints at Ollama abilities we don't have time to explore—namely, making new models from old and fine-tuning.

Ollama shows us the names of installed models on the command line:

```
> ollama list
```

NAME	ID	SIZE	MODIFIED
gemma:7b-instruct	430ed3535049	5.2 GB	4 days ago
llama2:13b-chat	d475bf4c50bc	7.4 GB	4 days ago
llama2:7b-chat	78e26419b446	3.8 GB	4 days ago
llava:13b	0d0eb4d7f485	8.0 GB	4 days ago
llava:7b	8dd30f6b0cb1	4.7 GB	4 days ago
mistral:7b-instruct	61e88e884507	4.1 GB	4 days ago
nomic-embed-text:latest	0a109f422b47	274 MB	4 days ago

The list includes the models we've explored in this chapter. To alter a model's default temperature, we'll clone an existing model, in this case `gemma:7b-instruct`, by creating a new *modelfile*. Ollama uses modelfiles to contain metadata about a model.

Let's create four new versions of Gemma with temperatures of 0, 0.3, 0.6, and 0.9. The code in *build_models.py* does this for us. For example, Listing 18-10 creates the $T = 0.6$ model.

```
import ollama
modelfile="""
FROM gemma:7b-instruct
PARAMETER temperature 0.6
```

```
"""
ollama.create("gemma_temp_0p6", modelfile=modelfile)
```

Listing 18-10: Cloning a model and setting its temperature

The process repeats for each new temperature. Running *build_models.py* adds the new models to our repertoire:

```
> ollama list
NAME                                ID                                SIZE  MODIFIED
gemma:7b-instruct                  430ed3535049                     5.2 GB  4 days ago
gemma_temp_0p0:latest              67cbef4249e4                     5.2 GB  13 hours ago
gemma_temp_0p3:latest              e7f990e154c3                     5.2 GB  13 hours ago
gemma_temp_0p6:latest              21f1f9f2204d                     5.2 GB  13 hours ago
gemma_temp_0p9:latest              6f0bdb8ab8cb                     5.2 GB  13 hours ago
llama2:13b-chat                    d475bf4c50bc                     7.4 GB  4 days ago
llama2:7b-chat                     78e26419b446                     3.8 GB  4 days ago
llava:13b                          0d0eb4d7f485                     8.0 GB  4 days ago
llava:7b                          8dd30f6b0cb1                     4.7 GB  4 days ago
mistral:7b-instruct                61e88e884507                     4.1 GB  4 days ago
nomic-embed-text:latest            0a109f422b47                     274 MB  4 days ago
```

When finished with the experiments, use `ollama rm` to remove unwanted models.

Let's make the new models take the DAT test. Executing *dat_words.py* in the *dat* directory creates output for each. Running the code five times and manually assembling the output by temperature produces files like *dat_temp_gemma_0p0.txt*, which you'll find in the *dat_lists* directory. Finally, running *dat_grade.py* on each leads to Table 18-2.

Table 18-2: DAT Scores by Temperature

<i>T</i>	Runs					Mean ± SE	Unique
0.0	87.81	87.81	87.81	87.81	87.81	87.81 ± 0.00	9
0.3	86.96	70.30	64.21	87.63	88.82	79.58 ± 5.13	41

T	Runs					Mean $\pm$ SE	Unique
0.6	91.23	83.59	89.02	71.33	85.28	84.09 $\pm$ 3.46	48
0.9	87.50	77.91	83.14	88.84	71.96	81.87 $\pm$ 3.13	43

When $T = 0$, as in the first row, the model is deterministic and produces the same sequence of words on each run. Notice that only 9 of the words were unique, meaning the model repeated a word, *swarm*, in this case. The DAT test requires at least 7 of the 10 words to be unique. That said, the model's reply when $T = 0$ is slightly above average by human standards.

Determinism goes out the window when $T \neq 0$. For $T = 0.3$, the model's scores are generally high, with some rather low ones thrown in. This scoring inconsistency is largely gone by $T = 0.6$, where the model is, on average, better than most humans with one score above 90.

If $T = 0.6$ is more creative than $T = 0.3$, it stands to reason that $T = 0.9$ should be very creative, right? Not in this case. The scores for the $T = 0.9$ model are often good, but some of the inconsistency of $T = 0.3$ has perhaps reappeared. This makes sense: When the temperature is high, the likelihood of selecting tokens that are not as ideal becomes higher, and on this task, that implies less creativity (in a stochastic sense). I read Table 18-2 as justification for LLM designers' tendency to set the default temperature of general-purpose models to $T \approx 0.7$.

The DAT is a helpful metric, but if we want the model to write creatively, do we really care that much about its ability to reason to a list of unrelated words? Let's set the models loose to write stories, then try to assess their creativity.

Creative Writing as a Function of Temperature

The file *stories.py* asks each of the temperature models to write a short story, as Listing 18-11 shows.

```
import requests
import json

prompt = """
Write a three-paragraph story for adults. Be creative. Run wild!
"""

def GetResponse(model):
    """Get the model's response for a given temperature"""
    url = "http://localhost:11434/api/generate"

    msg = {
        "model": model,
        "prompt": prompt,
        "stream": False,
    }
    resp = requests.post(url, json=msg)
    if (resp.status_code != 200):
        raise ValueError("Bad response from Ollama server")
    reply = json.loads(resp.text)
    return reply['response'].replace("\n", " ")

print("0.0 : %s" % GetResponse("gemma_temp_0p0"))
print()
print("0.3 : %s" % GetResponse("gemma_temp_0p3"))
print()
print("0.6 : %s" % GetResponse("gemma_temp_0p6"))
print()
print("0.9 : %s" % GetResponse("gemma_temp_0p9"))
```

Listing 18-11: Writing a short story

The prompt specifies a story for adults, so the models feel “free” to use a more complex structure and a more extensive vocabulary than they likely would if requesting stories for children. The added exhortations to be creative and run wild might, hopefully, enhance the models’ output further.

I ran *stories.py* five times to create five stories for each temperature. The $T = 0$ model was deterministic and produced the same story each time. We

now understand that this is expected behavior in that case.

We'll try two metrics to assess the stories' creativity. First, the *Flesch readability score*, developed by Rudolf Flesch in the 1940s, scores a text by how easy it is to read, from 1 to 100. The higher the score, the easier the text. A readability score in the 90s is easy for children to read, while a score below 30 is college-level material.

Second, the *Creative Solution Diagnosis Scale (CSDS)* is a far more detailed metric that attempts to arrive at a single-number score reflecting the overall creativity of a text (or idea). The multiple individual tests are averaged to produce the score, though in practical use, individual test scores are used to assess the text more broadly. Fortunately for us, David Cropley from the University of South Australia, one of the creators of CSDS, has developed a custom GPT-4 tool that automates applying CSDS. If you use GPT-4, search for *Creativity Calculator* in the GPT store.

Let's begin with the Flesch readability score. Python libraries that include the readability score are available but often require installing additional natural language processing tools. I kept matters simple and used an online calculator (<https://goodcalculators.com/flesch-kincaid-calculator/>) to enter the five stories for each temperature by hand. The resulting readability scores are in Table 18-3.

Table 18-3: Flesch Readability Scores by Temperature

<i>T</i>	Runs					Mean ± SE
0.0	79.1	79.1	79.1	79.1	79.1	79.10 ± 0.00
0.3	52.7	70.2	51.3	68.6	61.3	60.82 ± 3.91
0.6	45.6	68.7	50.1	58.7	62.5	57.12 ± 4.17
0.9	67.6	48.0	59.4	58.4	50.4	56.76 ± 3.50

Table 18-3 indicates that the readability score decreases as the temperature increases. This seems like a reasonable trend, but bear in mind that the $T = 0.3$ and $T = 0.9$ results aren't even close to being statistically significantly different (t-test or Mann–Whitney U), so the trend is nothing more than a hint. Many more stories must be tested to decide whether there is a meaningful difference in readability.

Let's now consider the CSDS results. The Creativity Calculator GPT requires an input prompt along the lines of *How creative is this story?* followed by the story's text. I used the stories in *stories_run3.txt*, included in the GitHub repo. The CSDS calculator produces detailed output, which would be valuable to someone using the calculator for a specific story. We're more interested in the overall average score, the average of each of the five metrics CSDS uses.

The $T = 0$ story was assigned a creativity of 6.92 out of a possible 10. For $T = 0.3$ and $T = 0.6$, the score was 6.36 and 7.2, respectively. Finally, the $T = 0.9$ story's creativity came in at 7.74.

I repeated the assessment for all stories over the five runs of *stories.py*. The $T = 0$ story was, of course, the same each time. The remaining stories varied, producing the means in Table 18-4.

Table 18-4: CSDS Scores by Temperature

T	Runs					<i>Mean</i> $\pm$ SE
0.0	6.92	6.92	6.92	6.92	6.92	6.92 $\pm$ 0.00
0.3	6.99	7.28	7.36	6.36	6.92	6.98 $\pm$ 0.18
0.6	7.94	6.70	7.66	7.20	7.29	7.36 $\pm$ 0.21
0.9	7.34	7.42	7.20	7.74	7.30	7.40 $\pm$ 0.09

Table 18-4 tracks the trend of Table 18-3, indicating an increase in perceived story creativity as a function of temperature. The readability

scores between $T = 0.3$ and $T = 0.9$ weren't statistically significant, but the CSDS scores are closer to significance (t-test $p = 0.07$, Mann–Whitney $p = 0.10$), hinting that further testing might reveal a meaningful difference in the Gemma model's creativity at high T values.

As mentioned, the CSDS report generated by the Creativity Calculator GPT is extensive. The single-number score is an average of five collections of metrics. A deeper dive into those results might reveal areas where the model's performance varied widely by temperature.

Both metrics in this section hint that higher temperatures lead to more creative resources, but there is a sweet spot that balances creativity with task relevance. And with that, we conclude our foray into LLMs.

Summary

This chapter introduced you to large language models, the kind of generative AI model that represents a genuine paradigm shift, a significant step toward the ultimate goal of AI: machines that think.

We walked through the process a trained LLM uses to predict a sequence of tokens in response to a prompt. We did this in some detail, using a data-flow approach. While it's possible to understand the process, the enormity of extant, world-class LLMs makes truly understanding the *how* of what the model has learned beyond human comprehension, at least in detail.

Next, we explored one of LLMs' more impressive abilities: in-context learning. This emergent capability enables a model to learn on the fly, something previous machine learning models are unable to do.

Following that, we began a series of experiments running smaller LLMs locally, courtesy of Ollama. We learned how to install and use Ollama from code and explored current LLM use cases like semantic search, RAG, and sentiment analysis.

Multimodal LLMs operate on more than just text. Ollama supports multimodal models, enabling us to operate on images and text jointly.

The chapter closed by attempting to answer the question of whether LLMs are creative. We compared LLM performance on the DAT to that of most humans. LLMs fall primarily within the human range. We then

enhanced our exploration to quantify creativity as a function of model temperature to learn that higher temperatures may lead to more creative model output.

[OceanofPDF.com](https://oceanofPDF.com)

AFTERWORD



Our tour of deep learning has come to an end, as must all good things. What follows are musings concerning what deep learning has wrought and what it might mean. Take them for whatever value you deem them to have.

I started the first edition of what would come to be called *Practical Deep Learning: A Python-Based Introduction* in December 2017. At that point, the deep learning revolution was some five years post-AlexNet. Convolutional neural networks were everywhere, and each new conference was a wonderland of fantastic result after fantastic result. Those who have been in the field for decades, either as active members or, as I was, part-time participants, could hardly have asked for more.

Machine learning beat the world's best chess master in 1997 when IBM's Deep Blue bested Garry Kasparov. As amazing as that accomplishment was, it was anticipated. I clearly remember my undergraduate artificial intelligence professor commenting that it was only a matter of time before chess fell to a computer. That was in 1987. I also remember his claim that Go was too difficult for any machine and that no program would ever beat the world champion. Humanity lost Go to the machines in 2016.

Though those results were impressive, few were prepared for the advent of large language models (LLMs). There were rumblings in the

years before 2022, but when ChatGPT burst upon an unsuspecting world, even those in the field were taken aback (at least I was, with my computer vision focus). Suddenly, all the dreams of what AI could be seemed to be one enormous step closer. And it scared many of us silly. As the saying goes, be careful what you wish for.

The fact that the power of LLMs was largely unexpected seems fitting to me. Life is unexpected. How many hundreds of millions of dead planets are in the Milky Way? Life isn't assured, yet evolution produced the unexpected when the conditions were right. I can't help but be amazed that when an architecture capable of allowing such abilities to emerge was infused with an almost unfathomable amount of data, the very thing the connectionists claimed would happen actually did. I have to believe that Frank Rosenblatt is smiling in his grave.

LLMs are a beginning, not an end. They point the way to something more now that we know emergence is real. Why did we doubt it? The billions of brains on this planet have repeatedly proved it possible over the last 800 million years of animal evolution. True, the amount of data in biological systems, even within a single cell, likely dwarfs the datasets feeding entities like GPT-4 and Claude 3, but we now understand that a kind of intelligence can appear on even such a small scale and with little information. That's genuinely fascinating and encouraging to me.

The previous paragraph makes a bold claim: that we might regard LLMs as having a kind of intelligence. My defense is that, as Chapter 18 showed us, LLMs are creative, and creativity is a hallmark of intelligence. I'm well aware that there is no universally agreed-upon definition of the terms *creative* and *intelligent*. However, my approach is pragmatic: If the system mimics human behavior deemed creative and intelligent, then for practical purposes, we might as well claim it to be so. It's understandable that some will find that working definition unsatisfying or even threatening.

In my defense, I point you to "A Theory for Emergence of Complex Skills in Language Models" by Sanjeev Arora (2023). This paper posits perhaps the first plausible explanation regarding the *how* behind the emergent abilities of LLMs. To summarize the net conclusion of the paper: The architecture of an LLM, combined with the capacity of the

model and the structure of the skills it learns, is such that novel combinations of those skills are also learned even if the skill is not present in the training set. In a sense, if this view holds up to scrutiny, LLMs can cheat the curse of dimensionality in terms of skill combinations that must be present in the training set.

I claim that novel combinations of existing skills and knowledge entail human creativity and, in some sense, intelligence. Therefore, from that perspective, it is entirely fair to claim that LLMs exhibit creativity and intelligence, though of a kind fundamentally different from human creativity and intelligence, which is strongly affected and motivated by our wills, emotions, and desires—all of which seem absent from extant LLMs.

The debates will no doubt continue for years to come. Regardless, I'm excited to see what's next. Scary? Possibly. Amazing? Certainly. Wonderful? If we want it to be.

This book is incomplete. It must be, as the field of deep learning is too rich for a single volume to cover with any level of rigor. This edition includes six new chapters, but I was forced to exclude some topics for space considerations. Diffusion models were nearly coincident with LLMs, but you didn't read about them here. You also didn't read about autoencoders, neither the original simple kinds we stacked together before CNNs largely replaced them, nor the variational autoencoders that, like GANs, can learn latent spaces from which to generate novel output. And sadly, the entire broad subfield of reinforcement learning is absent. However, now that you know of these topics, you'll no doubt be able to find many suitable resources online. At the very least, you might ask your favorite LLM to help you get started. Just remember to double-check its responses.

And with that, we've reached the end of the book. There's no monster here, only ourselves and the knowledge and intuition you've gained by working through the preceding chapters. Thank you for persevering. It's been fun for me to write, and I genuinely hope it's been fun for you to read and contemplate. So long, and thanks for all the fish.

INDEX

A

activation function, 131–132
Adam, 267, 278, 296
ANN (artificial neural network), 132
anomaly detection, 397
area under the ROC curve (AUC), 224–225, 305
array processing, 2

B

backpropagation, 134, 148, 157, 160
bagging, 83
batch normalization, 354
batch size, 186
batch training, 152
Bayes, Thomas, 73
big-O notation, 115
bootstrap sample, 83
bounding box, 292, 408
box plot, 39
 IQR (interquartile range), 41
 whiskers, 41
Bunker, Chang and Eng, 454
Burges, Christopher, 85

C

capsule networks, 255

Cartesian plane, 4

case study, 315

- classical models, 323

- classifier

 - audio features, 323

 - neural network, 326

 - spectrograms, 335

- CNN (convolutional neural network)

 - ensembles, 339

 - one-dimensional, 327

 - two-dimensional, 335

- dataset

 - augmentation, 317

 - ESC-10, 316

 - preprocessing, 321

 - spectrogram, 332–333

 - sox (command line tool), 333

categorical value, 16

centroid, 69

CIFAR-10 dataset, 50, 296

- binary vs. multiclass, 307

- building models, 297

- one-versus-rest, 307

classes, 13–14

classification, 13

- multilabel, 432

Claude 3 model, 508–509

CNN (convolutional neural network), 6, 237

- activations, 260

- anatomy of, 243
- architecture, 243–244
- bounding box, 292, 408
- building, 264
- constructing in Keras, 266
- convolution, 239
 - exact, 242
 - filter, 248
 - valid, 242
 - zero-padding, 242
- convolutional layer
 - effect, 250
 - initialization, 253
 - operation, 247
 - stacking, 252
- data flow, 246
- effective receptive field, 252–253
- end-to-end training, 261
- evaluating, 267
- experiments, 272
 - architecture, 272
 - epochs, 275
 - minibatches, 275
 - optimizers, 278
 - training-set size, 275
- feature maps, 245
- filter, 248
- fully convolutional network, 257, 263, 280, 282
 - building and training, 280
- graymap, 289
- heatmap, 288
- history, 239
- kernel, 239

layers

- convolutional, 244, 247
- dense, 244
- dropout, 244
- flatten, 244
- fully connected, 245, 255
- fully convolutional, 256
- inner product, 245
- inputs, 245
- pooling, 244, 254
- ReLU, 244

layer types, 244

loss function, 267

max pooling, 254

motivation, 238

neocognitron, 239

optimizer keyword, 267

optimizers, 278

outputs, 258

plotting, 270

pooling layer, 254

probability map, 288

receptive field, 252–253

scaling data, 265

spatial invariance, 238

stride, 239, 254

training, 267

YOLO model, 292

Cohen's kappa, 215

comparing models, 222

Compute Unified Device Architecture (CUDA), 9

confusion matrix, 206, 207

- multiclass, 227

- contingency table, 207
- convolution, 239
 - exact, 242
 - image processing, 242
 - valid, 242
 - zero-padding, 242
- convolutional neural network. *See* CNN
- Cortes, Corinna, 83
- cosine distance, 399
- covariate shift, 355
- Creative Solution Diagnosis Scale (CSDS), 536
- Cropley, David, 536
- CUDA (Compute Unified Device Architecture), 9
- curse of dimensionality, 17, 19
- curve fitting, 169

D

- DAT (divergent association task), 532
- data augmentation, 52, 171
 - approaches, 54
 - images, 60
 - rationale, 53
 - using PCA, 57
- datasets, 7, 19
 - augmentation, 52
 - bagging, 83
 - balanced, 26
 - breast cancer dataset, 45
 - categorical value, 16
 - cautionary tales, 41
 - CIFAR-10 dataset, 50

- CIFAR-100 dataset, 440
- classes, 13
- confusers, 23–24
- curse of dimensionality, 17
- discrete value, 15
- ESC-10 dataset, 316
- Fashion MNIST (FMNIST) dataset, 485
- feature selection, 17
- feature vectors, 14–15
- floating-point number, 15
- hard negative, 24
- ImageNet dataset, 354
- interval value, 15
- irises dataset, 43
- k -fold validation, 36, 100
- label, 14
- M2NIST dataset, 422
- mean centering, 26
- mislabeled data, 38
- missing features, 29
- MNIST dataset, 47, 110
- normalizing, 27–28, 95
- one-hot encoding, 17
- ordinal value, 16
- outliers, 38, 41
- partitioning, 31–32, 35
- partitioning by class, 32
- pitch shifting, 317
- potential problems, 37
- preprocessing, 25, 43
- random sampling, 33
- sample, 4
- scaling, 25–26

- spectrogram, 332–333
- standardizing, 27–28, 95
- summarizing, 38
- test data, 30
- time shifting, 317
- training data, 30
- UC Merced Land Use dataset, 409
- validation data, 30
- DCGAN (deep convolutional GAN), 470
- decision tree, 123
- deep learning, xxviii
- depthwise convolution, 371
- derivative, 150
 - local minimum, 151
 - minimum, 150
 - partial, 158
 - tangent line, 149
- descriptive statistics, 7
- detection, 407
- divergent association task (DAT), 532
- domain adaptation, 22, 283
- dot product, 5
- dropout, 173
 - spatial dropout, 355
- dying ReLU problem, 365

E

- effective receptive field, 252–253
- ELM (extreme learning machine), 445

embedding, 391, 515

ensemble, 82

epoch, 153

Euclidean distance, 70, 399

experiments

- breast cancer, 95

- CIFAR-10

 - analyzing models, 300

 - animal or vehicle, 302

 - binary versus multiclass, 307

 - building models, 297

- irises, 89, 140

 - classical models, 90

- MNIST, 110

 - activation function, 179

 - activation results, 183

 - architecture, 179

 - architecture results, 183

 - base learning rate, 190

 - batch size, 186

 - classical models, 110

 - code, 179

 - initialization, 197

 - L2 regularization, 193

 - momentum, 195

 - neural networks, 178

 - PCA (principal component analysis), 117

 - scrambled, 120, 201

 - training-set size, 192

- MNIST CNN

 - basic experiments, 272

 - building models, 266

 - dataset, 264

- epochs, 275
- fully convolutional, 280
- minibatches, 275
- optimizers, 278
- scrambled, 292
- training-set size, 275

extrapolation, 20

extreme learning machine (ELM), 445

F

F1 score, 214

false-negative rate (FNR), 211

false-positive rate (FPR), 211

features, 14

feature selection, 17

feature types, 15

feature vectors, 14–15

fine-tuning, 380, 452, 461

Fisher, R.A., 43

Flesch, Rudolf, 536

Flesch readability score, 536

floating-point number, 15, 45, 320

Fourier transform, 332

- power spectrum, 332

Frank, Eibe, 117

fully connected layer, 255

fully convolutional layer, 256

fully convolutional network, 280

G

- Gaussian distribution, 8
- Gemma model, 512
- generative adversarial network (GAN), 463
 - conditional, 464, 480
 - discriminator, 464
 - generator, 464
 - latent vector, 464
 - mode collapse, 476
 - training, 467
 - unconditional, 464–465
- generative model, xxviii, 463
- Gini index, 82
- Glorot, Xavier, 198
- Goodfellow, Ian, 463
- GPT-4, 494, 505
- GPU (graphics processing unit), 1, 9
- gradient, 148
 - intercept, 149
 - slope, 149
- gradient descent, 148
 - algorithm, 152
 - batch training, 152
 - epoch, 153
 - first-order, 154
 - learning rate, 151, 190
 - local minimum, 151
 - minibatch, 152–154
 - momentum, 157, 195
 - stochastic, 152
- grand mean, 103
- graph, 130
- graphics processing unit (GPU), 1, 9

H

- Hadamard, Jacques, 163
- Hadamard product, 163
- He, Kaiming, 168, 363
- heatmap, 288
- Hinton, Geoffrey E., 157, 255
- hit rate, 209
- hyperbolic tangent, 133
- hyperparameters, 104
- hyperplane, 84
- hypothesis testing, 8

I

- ID (intrinsic dimensionality), 448
- image processing, 242
- image retrieval, 399
- in-context learning, 504
- informedness, 213–214
- inner product, 5
- interpolation, 20, 488
- intersection over union (IoU), 414, 430
- interval value, 15
- intrinsic dimensionality (ID), 448

J

- joint probability, 74

K

k-d-tree, 117

Keras, 2

- applications, 382

- constructing CNNs with, 266

- documentation, 265

- functional API, 350

kernel, 239

k -fold validation, 36, 100

Kibriya, Ashraf M., 117

k -means, 395

k -nearest neighbor (k -NN), 72–73

k -NN classifier, 72–73, 99, 104, 114

Krizhevsky, Alex, 173

L

L2 regularization, 171–172

label, 14

large language model (LLM), 493

- block diagram, 496

- Claude 3, 508

- embedding, 515

- GPT-4, 494, 505

- in-context learning, 504

- Llama2, 510

- MLP block, 501

- Ollama, 494, 509

- positional matrix, 498

- quantization, 510

- retrieval-augmented generation (RAG), 521

- self-attention, 495
 - multihead, 499
- semantic search, 519
- temperature, 502
- tokenization, 497
- transformer, 499
- latent space, 473, 486
 - interpolation, 488
- layer normalization, 501
- learning rate, 151, 156, 190
- LeNet-5, 351
- linear algebra, 4. *See also* matrix; vector
 - scalar, 5
 - tensor, 6
- linear function, 132
- linear regression, 20
- Linux, 2
- Llama2 model, 510, 512
- LLaVa model, 512
- localization, 408
- local minimum, 151
- local response normalization (LRN), 354
- long short-term memory (LSTM), 495
- loss function, 148, 150, 164, 165, 267
 - absolute error, 165
 - cross-entropy loss, 166, 267
 - mean squared error (MSE), 165

M

macOS, 3

- main function, 58
- Mann-Whitney U test, 8, 106
- markedness, 213
- matrix
 - multiplication, 6, 139
 - transpose, 143
 - vector multiplication, 5
- Matthews correlation coefficient (MCC), 215
 - multiclass, 227, 232
- max pooling, 254
- mean, 6
- median, 7
- metrics, 203
 - 2×2 table, 207
 - accuracy, 91, 204
 - accuracy matrix, 228
 - advanced metrics, 213
 - area under the ROC curve (AUC), 224
 - CIFAR-10 example, 304
 - Cohen's kappa, 215
 - confusion matrix, 206–207
 - multiclass, 227
 - contingency table, 207
 - derived metrics, 208
 - F1 score, 214
 - false negative (FN), 206
 - false-negative rate (FNR), 211
 - false positive (FP), 206
 - false-positive rate (FPR), 211
 - grand mean, 103
 - hit rate, 209
 - implementation, 215

- informedness, 213–214
- interpreting models, 211
- intersection over union (IoU), 414, 430
- intrinsic dimensionality (ID), 448
- markedness, 213
- Matthews correlation coefficient (MCC), 215
 - multiclass, 227, 232
- negative predictive value (NPV), 210
- per-class accuracy, 205
- positive predictive value (PPV), 209
- precision, 209
- precision-recall (PR) curve, 226
- purity, 394, 448
- recall, 209
- receiver operating characteristics (ROC), 217
 - elements of, 220
 - generating, 224
- score, 91
- sensitivity, 209
- specificity, 209
- true negative (TN), 206
 - true-negative rate (TNR), 209
- true positive (TP), 206
 - true-positive rate (TPR), 209
- t-SNE, 261, 393, 448
- weighted accuracy, 230
- Youden's J statistic, 213
- minibatch, 152–154, 186
- missing features, 29
- Mistral model, 512
- MobileNet, 373
- mode collapse, 476

model

- capacity, 24

- classical models, 68

 - computational requirements, 125

 - explainability, 125

 - small datasets, 125

 - summary, 121

 - vector inputs, 125

 - when to use, 124

- Claude 3, 508

- comparing, 222

- DCGAN (deep convolutional GAN), 470

- decision tree, 77–78

 - construction, 81

 - summary, 123

- ensemble, 82, 375

- extreme learning machine (ELM), 445

- functional API, 353

- Gemma, 512

- generative adversarial network (GAN), 463

- Gini index, 82

- GPT-4, 494, 505

- hyperparameters, 104

 - optimizing, 104

- k -means, 395

- LeNet5, 351

- Llama2, 510, 512

- LLaVa, 512

- long short-term memory (LSTM), 495

- Mistral, 512

- MobileNet, 373

- multiheaded, 408

- naive Bayes, 73–74, 114, 122

- Gaussian, 91, 104, 114
 - multinomial, 91
 - summary, 122
- nearest centroid, 68, 71
 - summary, 121
- nearest neighbor, 4, 72
 - optimizing, 104
 - summary, 122
- Nomic, 512
- Ollama, 494
- one-class SVM, 397
- overfitting, 148, 168, 169
- PatchRot, 446
- random forest, 77, 82
 - optimizing, 105
 - summary, 123
- recurrent neural network (RNN), 495
- ResNet, 363
- RotNet, 441
- Siamese network, 454
- SimCLR, 462
- support vector machine (SVM), 83, 124
 - kernel, 86
 - margin, 84–85
 - optimizing, 86, 106
 - summary, 124
 - support vectors, 85
- template matching, 71
- U-net, 422
- VGG (Visual Geometry Group), 354
- YOLO, 420
- momentum, 157, 195
- multilabel classification, 432

N

- naive Bayes, 73–74, 114, 122
 - Gaussian, 91, 104, 114
 - multinomial, 91
 - summary, 122
- nearest centroid, 68–71, 121
- nearest neighbor, 4, 122
- negative predictive value (NPV), 210
- neural network, 130
 - activation function, 131–132
 - hyperbolic tangent, 133
 - identity, 137
 - ReLU, 134
 - sigmoid, 133
 - anatomy, 130
 - ANN (artificial neural network), 132
 - architecture, 135
 - backpropagation, 148, 157
 - backward pass, 158
 - bias, 132
 - essence, 132
 - feedback, 130
 - forward pass, 158
 - fully connected feed-forward, 129, 130
 - function approximation, 130
 - hidden layer, 130
 - implementation, 140–141
 - multilayer perceptron (MLP), 132
 - neuron, 130–131
 - output layer, 132, 137
 - regularization, 148, 168, 171

- dropout, 173
 - L2, 171–172
 - weight decay, 172, 193
- representation, 139
- rules of thumb, 136
- softmax, 137
- traditional, 129
- training, 175
 - batch, 152
 - epoch, 153
 - gradient descent, 148
 - loss function, 148, 150, 164, 165
 - minibatch, 152, 154
 - momentum, 157
 - overview, 148
 - stochastic gradient descent (SGD), 152
 - stopping, 155
 - weight update, 151
- weight initialization, 148, 167, 197
- weights, 131
- Nomic model, 512
- nominal value, 16
- nonlinear function, 132
 - transcendental, 133
 - trigonometric, 133
- nonparametric test, 8
- normal distribution, 7, 8
- normalizing, 27–28, 95
- NPV (negative predictive value), 210
- NumPy, 2
 - argsort function, 33
 - convert arrays to images, 63

- convert images to arrays, 63
- histogram, 306
- pseudorandom seed, 98
 - where function, 32

NVIDIA, 9

0

Ollama, 494, 509

Olson, Jay, 532

one-class SVM, 397

one-hot encoding, 17

operating environment, 1

ordinal value, 16

outer product, 5

outliers, 41

overfitting, 53, 148, 155, 168, 169

P

parametric test, 8

parent distribution, 7, 22, 204

partial derivative, 158

Pasteur, Louis, 126

PatchRot model, 446

PCA (principal component analysis), 55, 113, 117

Pillow library, 3

Plato, 7, 22

pointwise convolution, 371

pooling, 254

positional matrix, 498

- positive predictive value (PPV), 209
- Powers, David Martin, 214
- power spectrum, 332
- precision, 209
- precision-recall (PR) curve, 226
- preprocessing, 43
- primary visual cortex, 252
- principal component analysis (PCA), 55, 113, 117
- prior class probability, 22, 204
- probability, 6
 - Bayes' theorem, 73
 - distribution, 7
 - Gaussian distribution, 8
 - normal distribution, 7, 8, 76
 - parent distribution, 7, 22
 - uniform distribution, 7, 8, 167
 - joint probability, 74
 - likelihood, 73
 - posterior probability, 73
 - prior class probability, 22
 - prior probability, 74
- probability map, 288
- pseudorandom sequence, 99
- purity, 394, 448
- p-value, 8
- Python
 - libraries
 - Pillow, 3
 - scikit-dimension (skdim), 449
 - scikit-learn (sklearn), 2
 - SciPy, 318
 - modules

librosa, 318
pickle, 142

Q

quantization, 510

R

RAG (retrieval-augmented generation), 521
random forest, 82, 123
recall, 209
receiver operating characteristics (ROC), 217, 220–226
receptive field, 252–253
rectified linear unit (ReLU), 134
 dying ReLU problem, 365
 leaky, 465
recurrent neural network (RNN), 495
recursion, 80
regularization, 53, 148, 168, 171
 dropout, 173
 L2, 171–172, 193
 weight decay, 172, 193
reinforcement learning, xxviii
ResNet model, 363
 architecture, 366
retrieval-augmented generation (RAG), 521
RMSprop, 278
RNN (recurrent neural network), 495
ROC (receiver operating characteristics), 217, 220–226
Ronneberger, Olaf, 258

RotNet model, 441
Rumelhart, David E., 157

S

sample, 4
scalar, 5
scikit-dimension (skdim) library, 449
scikit-learn (sklearn) library, 2

- DecisionTreeClassifier class, 90
- GaussianNB class, 90
- KNeighborsClassifier class, 90
- make_classification function, 32
- MLPClassifier class, 132, 144, 178
- MultinomialNB class, 90
- NearestCentroid class, 90
- RandomForestClassifier class, 90
- svc class, 90

SciPy library, 318

- wavfile module, 318

self-attention, 495, 499

- multihead, 499

self-supervised learning, xxviii, 439
semantic search, 519
semantic segmentation, 407
sensitivity, 209
sentiment analysis, 523
SGD (stochastic gradient descent), 152, 180, 267, 278, 296
Siamese network, 454
sigmoid, 133
SimCLR, 462

- Simonya, Karen, 354
- skip connection, 363
- softmax, 137
- spatial dropout, 355
- specificity, 209
- spectrogram, 333
- standard deviation, 7–8, 27, 76, 167
- standard error, 6
- standardizing, 27–28, 95
- statistically significant, 8
- statistical test, 8
- statistics, 6
 - bootstrap sample, 83
 - descriptive, 6–7
 - grand mean, 103
 - hypothesis testing, 8
 - Mann-Whitney U test, 8, 106
 - mean, 6
 - median, 7
 - nonparametric test, 8
 - parametric test, 8
 - p-value, 8
 - quartile, 40
 - standard deviation, 7–8, 27, 76, 167
 - standard error, 6, 39
 - statistically independent, 74
 - statistically significant, 8
 - t-test, 8
 - variance, 7
- stochastic gradient descent (SGD), 152, 180, 267, 278, 296
- supervised learning, xxviii, 19
- support vector machine (SVM), 83, 124

- kernel, 86
- margin, 84–85
- one-versus-one, 115
- one-versus-rest, 115
- optimizing, 86, 106
- support vectors, 85

T

- tangent line, 149
- temperature, 502
- tensor, 6
- TensorFlow, 2
- test data, 30
- tokenization, 497
- toolkits, 2–3
- topic modeling, 523
- training data, 30
- training set, xxviii
- transcendental functions, 133
- transfer learning, 391
- transformer, 499
- trigonometric functions, 133
- trilobite, 421
- true-negative rate (TNR), 209
- true-positive rate (TPR), 209
- t-SNE, 261, 393, 448
- t-test, 8

U

Ubuntu, 1, 2–3
UCI Machine Learning Repository, 45
U-net, 422
uniform distribution, 7–8, 167
unsupervised learning, xxviii

V

validation data, 30
vanishing gradient problem, 364
Vapnik, Vladimir, 83
variance, 7
Vaswani, Ashish, 494
vector, 4

- column vector, 4
- dot product, 5
- inner product, 5
- multiplication, 5
- outer product, 5
- row vector, 4

VGG (Visual Geometry Group), 354

W

weight decay, 172, 193
weighted accuracy, 230
weight initialization, 148, 167, 197

- Glorot, 168
- He, 168
- Xavier, 168

weights, 131

weight update, 151

Windows, 3

Y

YOLO model, 292, 420

Youden's J statistic, 213

Z

zero-padding, 242

Zisserman, Andrew, 354

OceanofPDF.com