



MACHINE LEARNING
MASTERY

Deep Learning FOR COMPUTER VISION

Image Classification,
Object Detection, and
Face Recognition in
Python



Jason Brownlee

Disclaimer

The information contained within this eBook is strictly for educational purposes. If you wish to apply ideas contained in this eBook, you are taking full responsibility for your actions.

The author has made every effort to ensure the accuracy of the information within this book was correct at time of publication. The author does not assume and hereby disclaims any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from accident, negligence, or any other cause.

No part of this eBook may be reproduced or transmitted in any form or by any means, electronic or mechanical, recording or by any information storage and retrieval system, without written permission from the author.

Acknowledgements

Special thanks to my proofreader Sarah Martin and my technical editors Arun Koshy, Andrei Cheremskoy, and Michael Sanderson.

Copyright

Deep Learning for Computer Vision

© Copyright 2019 Jason Brownlee. All Rights Reserved.

Edition: v1.81

Contents

Copyright	i
Contents	ii
Preface	iii
Introductions	v
Welcome	v
I Foundations	1
1 Introduction to Computer Vision	3
1.1 Overview	3
1.2 Desire for Computers to See	3
1.3 What Is Computer Vision	4
1.4 Challenge of Computer Vision	5
1.5 Tasks in Computer Vision	6
1.6 Further Reading	7
1.7 Summary	8
2 Promise of Deep Learning for Computer Vision	9
2.1 Overview	9
2.2 Promises of Deep Learning	9
2.3 Types of Deep Learning Network Models	13
2.4 Types of Computer Vision Problems	14
2.5 Further Reading	14
2.6 Summary	15
3 How to Develop Deep Learning Models With Keras	16
3.1 Keras Model Life-Cycle	16
3.2 Keras Functional Models	21
3.3 Standard Network Models	23
3.4 Further Reading	27
3.5 Summary	28

II Image Data Preparation	29
4 How to Load and Manipulate Images With PIL/Pillow	31
4.1 Tutorial Overview	31
4.2 How to Install Pillow	32
4.3 How to Load and Display Images	32
4.4 How to Convert Images to NumPy Arrays and Back	34
4.5 How to Save Images to File	37
4.6 How to Resize Images	38
4.7 How to Flip, Rotate, and Crop Images	40
4.8 Extensions	43
4.9 Further Reading	44
4.10 Summary	44
5 How to Manually Scale Image Pixel Data	45
5.1 Tutorial Overview	45
5.2 Sample Image	46
5.3 Normalize Pixel Values	47
5.4 Center Pixel Values	48
5.5 Standardize Pixel Values	50
5.6 Extensions	53
5.7 Further Reading	53
5.8 Summary	54
6 How to Load and Manipulate Images with Keras	55
6.1 Tutorial Overview	55
6.2 Test Image	56
6.3 Keras Image Processing API	56
6.4 How to Load an Image with Keras	57
6.5 How to Convert an Image With Keras	58
6.6 How to Save an Image With Keras	59
6.7 Extensions	60
6.8 Further Reading	61
6.9 Summary	61
7 How to Scale Image Pixel Data with Keras	62
7.1 Tutorial Overview	62
7.2 MNIST Handwritten Image Classification Dataset	63
7.3 <code>ImageDataGenerator</code> Class for Pixel Scaling	64
7.4 How to Normalize Images With <code>ImageDataGenerator</code>	65
7.5 How to Center Images With <code>ImageDataGenerator</code>	67
7.6 How to Standardize Images With <code>ImageDataGenerator</code>	69
7.7 Extensions	71
7.8 Further Reading	71
7.9 Summary	72

8 How to Load Large Datasets From Directories with Keras	73
8.1 Tutorial Overview	73
8.2 Dataset Directory Structure	74
8.3 Example Dataset Structure	75
8.4 How to Progressively Load Images	77
8.5 Extensions	80
8.6 Further Reading	80
8.7 Summary	80
9 How to Use Image Data Augmentation in Keras	82
9.1 Tutorial Overview	82
9.2 Image Data Augmentation	83
9.3 Sample Image	83
9.4 Image Augmentation With <code>ImageDataGenerator</code>	84
9.5 Horizontal and Vertical Shift Augmentation	86
9.6 Horizontal and Vertical Flip Augmentation	89
9.7 Random Rotation Augmentation	91
9.8 Random Brightness Augmentation	92
9.9 Random Zoom Augmentation	94
9.10 Extensions	95
9.11 Further Reading	95
9.12 Summary	96
III Convolutions and Pooling	97
10 How to Use Different Color Channel Ordering Formats	99
10.1 Tutorial Overview	99
10.2 Images as 3D Arrays	99
10.3 Manipulating Image Channels	100
10.4 Keras Channel Ordering	103
10.5 Extensions	106
10.6 Further Reading	106
10.7 Summary	106
11 How Convolutional Layers Work	108
11.1 Tutorial Overview	108
11.2 Convolution in Convolutional Neural Networks	109
11.3 Convolution in Computer Vision	110
11.4 Power of Learned Filters	111
11.5 Worked Example of Convolutional Layers	112
11.6 Extensions	119
11.7 Further Reading	119
11.8 Summary	120

12 How to Use Filter Size, Padding, and Stride	121
12.1 Tutorial Overview	121
12.2 Convolutional Layer	122
12.3 Problem of Border Effects	124
12.4 Effect of Filter Size (Kernel Size)	126
12.5 Fix the Border Effect Problem With Padding	128
12.6 Downsample Input With Stride	129
12.7 Extensions	131
12.8 Further Reading	132
12.9 Summary	132
13 How Pooling Layers Work	133
13.1 Tutorial Overview	133
13.2 Pooling Layers	134
13.3 Detecting Vertical Lines	135
13.4 Average Pooling Layer	137
13.5 Max Pooling Layer	140
13.6 Global Pooling Layers	142
13.7 Extensions	144
13.8 Further Reading	144
13.9 Summary	145
IV Convolutional Neural Networks	146
14 ImageNet, ILSVRC, and Milestone Architectures	148
14.1 Overview	148
14.2 ImageNet Dataset	149
14.3 ImageNet Large Scale Visual Recognition Challenge (ILSVRC)	149
14.4 Deep Learning Milestones From ILSVRC	150
14.5 Further Reading	152
14.6 Summary	153
15 How Milestone Model Architectural Innovations Work	155
15.1 Tutorial Overview	155
15.2 Architectural Design for CNNs	156
15.3 LeNet-5	156
15.4 AlexNet	157
15.5 VGG	159
15.6 Inception and GoogLeNet	161
15.7 Residual Network or ResNet	162
15.8 Further Reading	163
15.9 Summary	164

16 How to Use 1x1 Convolutions to Manage Model Complexity	165
16.1 Tutorial Overview	165
16.2 Convolutions Over Channels	166
16.3 Problem of Too Many Feature Maps	166
16.4 Downsample Feature Maps With 1x1 Filters	167
16.5 Examples of How to Use 1x1 Convolutions	167
16.6 Examples of 1x1 Filters in CNN Model Architectures	170
16.7 Extensions	173
16.8 Further Reading	173
16.9 Summary	174
17 How To Implement Model Architecture Innovations	175
17.1 Tutorial Overview	175
17.2 How to implement VGG Blocks	176
17.3 How to Implement the Inception Module	181
17.4 How to Implement the Residual Module	185
17.5 Extensions	188
17.6 Further Reading	188
17.7 Summary	189
18 How to Use Pre-Trained Models and Transfer Learning	190
18.1 Tutorial Overview	190
18.2 What Is Transfer Learning?	191
18.3 Transfer Learning for Image Recognition	191
18.4 How to Use Pre-Trained Models	192
18.5 Models for Transfer Learning	193
18.6 Examples of Using Pre-Trained Models	196
18.7 Extensions	202
18.8 Further Reading	203
18.9 Summary	203
V Image Classification	205
19 How to Classify Black and White Photos of Clothing	207
19.1 Tutorial Overview	207
19.2 Fashion-MNIST Clothing Classification	208
19.3 Model Evaluation Methodology	209
19.4 How to Develop a Baseline Model	210
19.5 How to Develop an Improved Model	219
19.6 How to Finalize the Model and Make Predictions	228
19.7 Extensions	233
19.8 Further Reading	233
19.9 Summary	234

20 How to Classify Small Photos of Objects	235
20.1 Tutorial Overview	235
20.2 CIFAR-10 Photo Classification Dataset	236
20.3 Model Evaluation Test Harness	237
20.4 How to Develop a Baseline Model	243
20.5 How to Develop an Improved Model	249
20.6 How to Finalize the Model and Make Predictions	265
20.7 Extensions	269
20.8 Further Reading	269
20.9 Summary	270
21 How to Classify Photographs of Dogs and Cats	271
21.1 Tutorial Overview	271
21.2 Dogs vs. Cats Prediction Problem	272
21.3 Dogs vs. Cats Dataset Preparation	272
21.4 Develop a Baseline CNN Model	278
21.5 Develop Model Improvements	288
21.6 Explore Transfer Learning	295
21.7 How to Finalize the Model and Make Predictions	298
21.8 Extensions	303
21.9 Further Reading	303
21.10Summary	304
22 How to Label Satellite Photographs of the Amazon Rainforest	305
22.1 Tutorial Overview	305
22.2 Introduction to the Planet Dataset	306
22.3 How to Prepare Data for Modeling	306
22.4 Model Evaluation Measure	316
22.5 How to Evaluate a Baseline Model	320
22.6 How to Improve Model Performance	327
22.7 How to Use Transfer Learning	336
22.8 How to Finalize the Model and Make Predictions	346
22.9 Extensions	352
22.10Further Reading	352
22.11Summary	353
VI Object Detection	354
23 Deep Learning for Object Recognition	356
23.1 Overview	356
23.2 What is Object Recognition?	357
23.3 R-CNN Model Family	359
23.4 YOLO Model Family	363
23.5 Further Reading	365
23.6 Summary	367

24 How to Perform Object Detection With YOLOv3	368
24.1 Tutorial Overview	368
24.2 YOLO for Object Detection	369
24.3 Experiencor YOLO3 for Keras Project	369
24.4 Object Detection With YOLOv3	370
24.5 Extensions	387
24.6 Further Reading	387
24.7 Summary	388
25 How to Perform Object Detection With Mask R-CNN	390
25.1 Tutorial Overview	390
25.2 Mask R-CNN for Object Detection	391
25.3 Matterport Mask R-CNN Project	392
25.4 Object Detection With Mask R-CNN	392
25.5 Extensions	402
25.6 Further Reading	402
25.7 Summary	403
26 How to Develop a New Object Detection Model	405
26.1 Tutorial Overview	405
26.2 How to Install Mask R-CNN for Keras	406
26.3 How to Prepare a Dataset for Object Detection	408
26.4 How to Train Mask R-CNN Model for Kangaroo Detection	425
26.5 How to Evaluate a Mask R-CNN Model	430
26.6 How to Detect Kangaroos in New Photos	436
26.7 Extensions	443
26.8 Further Reading	443
26.9 Summary	444
VII Face Recognition	446
27 Deep Learning for Face Recognition	448
27.1 Overview	448
27.2 Faces in Photographs	449
27.3 Process of Automatic Face Recognition	449
27.4 Face Detection Task	450
27.5 Face Recognition Tasks	452
27.6 Deep Learning for Face Recognition	453
27.7 Further Reading	455
27.8 Summary	456
28 How to Detect Faces in Photographs	457
28.1 Tutorial Overview	457
28.2 Face Detection	458
28.3 Test Photographs	458
28.4 Face Detection With OpenCV	460

28.5 Face Detection With Deep Learning	465
28.6 Extensions	476
28.7 Further Reading	476
28.8 Summary	477
29 How to Perform Face Identification and Verification with VGGFace2	479
29.1 Tutorial Overview	479
29.2 Face Recognition	480
29.3 VGGFace and VGGFace2 Models	480
29.4 How to Install the keras-vggface Library	482
29.5 How to Detect Faces for Face Recognition	483
29.6 How to Perform Face Identification With VGGFace2	487
29.7 How to Perform Face Verification With VGGFace2	491
29.8 Extensions	494
29.9 Further Reading	494
29.10 Summary	495
30 How to Perform Face Classification with FaceNet	496
30.1 Tutorial Overview	496
30.2 Face Recognition	497
30.3 FaceNet Model	497
30.4 How to Load a FaceNet Model in Keras	498
30.5 How to Detect Faces for Face Recognition	499
30.6 How to Develop a Face Classification System	501
30.7 Extensions	514
30.8 Further Reading	515
30.9 Summary	516
VIII Appendix	517
A Getting Help	518
A.1 Computer Vision Textbooks	518
A.2 Programming Computer Vision Books	518
A.3 Official Keras Destinations	519
A.4 Where to Get Help with Keras	519
A.5 How to Ask Questions	520
A.6 Contact the Author	520
B How to Setup Python on Your Workstation	521
B.1 Overview	521
B.2 Download Anaconda	521
B.3 Install Anaconda	523
B.4 Start and Update Anaconda	525
B.5 Install Deep Learning Libraries	528
B.6 Further Reading	529
B.7 Summary	529

C How to Setup Amazon EC2 for Deep Learning on GPUs	530
C.1 Overview	530
C.2 Setup Your AWS Account	531
C.3 Launch Your Server Instance	532
C.4 Login, Configure and Run	536
C.5 Build and Run Models on AWS	537
C.6 Close Your EC2 Instance	538
C.7 Tips and Tricks for Using Keras on AWS	539
C.8 Further Reading	540
C.9 Summary	540
IX Conclusions	541
How Far You Have Come	542

Preface

We are awash in images such as photographs, videos, YouTube, Instagram, and increasingly from live video. Every day, I get questions asking how to develop machine learning models for image data. Working with images can be challenging as it requires drawing upon knowledge from diverse domains such as digital signal processing, machine learning, statistics, and these days, deep learning.

I designed this book to teach you step-by-step how to bring modern deep learning methods to your computer vision projects. I chose the programming language, programming libraries, and tutorial topics to give you the skills you need.

Python is the go-to language for applied machine learning and deep learning, both in terms of demand from employers and employees. This is partially because there is renaissance Python-based tools for machine learning. I have focused on showing you how to use the best of breed Python tools for computer vision such as PIL/Pillow, as well as the image handling tools provided with the Keras deep learning library. Key to getting results is speed of development, and for this reason, we use the Keras deep learning library as you can define, train, and use complex deep learning models with just a few lines of Python code. There are three key areas that you must know when working with image data:

- How to handle image data. This includes how to load images, load datasets of image data, and how to scale image data to make it ready for modeling.
- How models work. This mainly includes intuitions for how the layers of a convolutional neural network operate on images and how to configure these layers.
- How to use modern models. This includes both innovations in the model architectures as well as the specific models used on a variety of different computer vision tasks.

These key topics provide the backbone for the book and the tutorials you will work through. I believe that after completing this book, you will have the skills that you need to both work through your own computer vision projects and bring modern deep learning methods to bear.

Jason Brownlee
2019

Introductions

Welcome

Welcome to *Deep Learning for Computer Vision*. Computer vision is the area of study dedicated to helping computers see and understand the meaning in digital images such as photographs and videos. It is an old field of study, up until recently dominated by specialized hand-crafted methods designed by digital signal processing experts and statistical methods. Within the last decade, deep learning methods have demonstrated state-of-the-art results on challenging computer vision tasks such as image classification, object detection, and face recognition. This book is designed to teach you step-by-step how to bring modern deep learning models to your own computer vision projects.

Who Is This Book For?

Before we get started, let's make sure you are in the right place. This book is for developers that know some applied machine learning and some deep learning. Maybe you want or need to start using deep learning for computer vision on your research project or on a project at work. This guide was written to help you do that quickly and efficiently by compressing years of knowledge and experience into a laser-focused course of hands-on tutorials. The lessons in this book assume a few things about you, such as:

- You know your way around basic Python for programming.
- You know your way around basic NumPy for array manipulation.
- You know your way around basic scikit-learn for machine learning.
- You know your way around basic Keras for deep learning.

For some bonus points, perhaps some of the below points apply to you. Don't panic if they don't.

- You may know how to work through a predictive modeling problem end-to-end.
- You may know a little bit of computer vision background.
- You may know a little bit of computer vision such as PIL/Pillow or OpenCV.

This guide was written in the top-down and results-first machine learning style that you're used to from MachineLearningMastery.com.

About Your Outcomes

This book will teach you how to get results as a machine learning practitioner interested in using deep learning on your computer vision project. After reading and working through this book, you will know:

- About the promise of neural networks and deep learning methods in general for computer vision problems.
- How to load and prepare image data, such as photographs, for modeling using best-of-breed Python libraries.
- How specialized layers for image data work, including 1D and 2D convolutions, max and average pooling, and intuitions for the impact that each layer has on input data.
- How to configure convolutional layers, including aspects such as filter size, stride, and pooling.
- How key modeling innovations for convolutional neural networks work and how to implement them from scratch, such as VGG blocks, inception models, and resnet modules.
- How to develop, tune, evaluate and make predictions with convolutional neural networks on standard benchmark computer vision datasets for image classification, such as Fashion-MNIST and CIFAR-10.
- How to develop, tune, evaluate, and make predictions with convolutional neural networks on entirely new datasets for image classification, such as satellite photographs and photographs of pets.
- How to use techniques such as pre-trained models, transfer learning and image augmentation to accelerate and improve model development.
- How to use pre-trained models and develop new models for object recognition tasks, such as object localization and object detection in photographs, using techniques like R-CNN and YOLO.
- How to use deep learning models for face recognition tasks, such as face identification and face verification in photographs, using techniques like Google’s FaceNet and Oxford’s VGGFace.

This book will NOT teach you how to be a research scientist nor all the theory behind why specific methods work. For that, I would recommend good research papers and textbooks. See the *Further Reading* section at the end of each tutorial for a solid starting point.

How to Read This Book

This book was written to be read linearly, from start to finish. That being said, if you know the basics and need help with a specific method or type of problem, then you can flip straight to that section and get started. This book was designed for you to read on your workstation, on

the screen, not on a tablet or eReader. My hope is that you have the book open right next to your editor and run the examples as you read about them.

This book is not intended to be read passively or be placed in a folder as a reference text. It is a playbook, a workbook, and a guidebook intended for you to learn by doing and then apply your new understanding with working Python examples. To get the most out of the book, I would recommend playing with the examples in each tutorial. Extend them, break them, then fix them. Try some of the extensions presented at the end of each lesson and let me know how you do.

About the Book Structure

This book was designed around major deep learning techniques that are directly relevant to computer vision problems. There are a lot of things you could learn about deep learning and computer vision, from theory to abstract concepts to APIs. My goal is to take you straight to developing an intuition for the elements you must understand with laser-focused tutorials. The tutorials were designed to focus on how to get results with deep learning methods. As such, the tutorials give you the tools to both rapidly understand and apply each technique or operation. There is a mixture of both tutorial lessons and projects to both introduce the methods and give plenty of examples and opportunity to practice using them.

Each of the tutorials is designed to take you about one hour to read through and complete, excluding the extensions and further reading. You can choose to work through the lessons one per day, one per week, or at your own pace. I think momentum is critically important, and this book is intended to be read and used, not to sit idle. I would recommend picking a schedule and sticking to it. The tutorials are divided into seven parts; they are:

- **Part 1: Foundations.** Discover a gentle introduction to computer vision and the promise of deep learning in the field of computer vision, as well as tutorials on how to get started with Keras.
- **Part 2: Data Preparation.** Discover tutorials on how to load images, image datasets, and techniques for scaling pixel data in order to make images ready for modeling.
- **Part 3: Convolutions and Pooling.** Discover insights and intuitions for how the building blocks of convolutional neural networks actually work, including convolutions and pooling layers.
- **Part 4: Convolutional Neural Networks.** Discover the convolutional neural network model architectural innovations that have led to impressive results and how to implement them from scratch.
- **Part 5: Image Classification.** Discover how to develop, tune, and evaluate deep convolutional neural networks for image classification, including problems like Fashion-MNIST and CIFAR-10, as well as entirely new datasets.
- **Part 6: Object Detection.** Discover deep learning models for object detection such as R-CNN and YOLO and how to both use pre-trained models and train models for new object detection datasets.

- **Part 7: Face Recognition.** Discover deep learning models for face recognition, including FaceNet and VGGFace and how to use pre-trained models for face identification and face verification.

Each part targets a specific learning outcome, and so does each tutorial within each part. This acts as a filter to ensure you are only focused on the things you need to know to get to a specific result and do not get bogged down in the math or near-infinite number of digressions. The tutorials were not designed to teach you everything there is to know about each of the methods. They were designed to give you an understanding of how they work, how to use them, and how to interpret the results the fastest way I know how: to learn by doing.

About Python Code Examples

The code examples were carefully designed to demonstrate the purpose of a given lesson. For this reason, the examples are highly targeted.

- Models were demonstrated on real-world datasets to give you the context and confidence to bring the techniques to your own computer vision problems.
- Model configurations used were discovered through trial and error and are skillful, but not optimized. This leaves the door open for you to explore new and possibly better configurations.
- Code examples are complete and standalone. The code for each lesson will run as-is with no code from prior lessons or third parties needed beyond the installation of the required packages.

A complete working example is presented with each tutorial for you to inspect and copy-and-paste. All source code is also provided with the book and I would recommend running the provided files whenever possible to avoid any copy-paste issues. The provided code was developed in a text editor and intended to be run on the command line. No special IDE or notebooks are required. If you are using a more advanced development environment and are having trouble, try running the example from the command line instead.

Neural network algorithms are stochastic. This means that they will make different predictions when the same model configuration is trained on the same training data. On top of that, each experimental problem in this book is based around generating stochastic predictions. As a result, this means you will not get exactly the same sample output presented in this book. This is by design. I want you to get used to the stochastic nature of the neural network algorithms. If this bothers you, please note:

- You can re-run a given example a few times and your results should be close to the values reported.
- You can make the output consistent by fixing the NumPy random number seed.
- You can develop a robust estimate of the skill of a model by fitting and evaluating it multiple times and taking the average of the final skill score (highly recommended).

All code examples were tested on a POSIX-compatible machine with Python 3 and Keras 2. All code examples will run on modest and modern computer hardware and were executed on a CPU or GPU. A GPU is not required but is recommended for some of the presented examples. Advice on how to access cheap GPUs via cloud computing is provided in the appendix. I am only human and there may be a bug in the sample code. If you discover a bug, please let me know so I can fix it and correct the book and send out a free update.

About Further Reading

Each lesson includes a list of further reading resources. This may include:

- Research papers.
- Books and book chapters.
- Webpages.
- API documentation.
- Open Source Projects.

Wherever possible, I try to list and link to the relevant API documentation for key objects and functions used in each lesson so you can learn more about them. When it comes to research papers, I try to list papers that are first to use a specific technique or first in a specific problem domain. These are not required reading but can give you more technical details, theory, and configuration details if you're looking for it. Wherever possible, I have tried to link to the freely available version of the paper on arxiv.org. You can search for and download any of the papers listed on Google Scholar Search scholar.google.com. Wherever possible, I have tried to link to books on Amazon.

I don't know everything, and if you discover a good resource related to a given lesson, please let me know so I can update the book.

About Getting Help

You might need help along the way. Don't worry; you are not alone.

- **Help with a Technique?** If you need help with the technical aspects of a specific operation or technique, see the *Further Reading* section at the end of each tutorial.
- **Help with APIs?** If you need help with using the Keras library, see the list of resources in the *Further Reading* section at the end of each lesson, and also see *Appendix A*.
- **Help with your workstation?** If you need help setting up your environment, I would recommend using Anaconda and following my tutorial in *Appendix B*.
- **Help running large models?** I recommend renting time on Amazon Web Service (AWS) EC2 instances to run large models. If you need help getting started on AWS, see the tutorial in *Appendix C*.
- **Help in general?** You can shoot me an email. My details are in *Appendix A*.

Summary

Are you ready? Let's dive in! Next up you will discover a concrete understanding of the field of computer vision.

Part I

Foundations

Overview

In this part you will discover a gentle introduction to computer vision, the promise of deep learning for computer vision and how to get started developing deep learning models with the Keras deep learning library. After reading the chapters in this part, you will know:

- Computer vision is the challenging sub-field of artificial intelligence focused on developing systems to help computers *see* (Chapter 1).
- Deep learning is delivering on the promise of automated feature learning, better performance, end-to-end models and more (Chapter 2).
- How to develop simple deep learning models and the modeling life-cycle using the Keras deep learning framework (Chapter 3).

Chapter 1

Introduction to Computer Vision

Computer Vision, often abbreviated as CV, is defined as a field of study that seeks to develop techniques to help computers *see* and understand the content of digital images such as photographs and videos. The problem of computer vision appears simple because it is trivially solved by people, even very young children. Nevertheless, it largely remains an unsolved problem based both on the limited understanding of biological vision and because of the complexity of vision perception in a dynamic and nearly infinitely varying physical world. In this tutorial, you will discover a gentle introduction to the field of computer vision. After reading this tutorial, you will know:

- The goal of the field of computer vision and its distinctness from image processing.
- What makes the problem of computer vision challenging.
- Typical problems or tasks pursued in computer vision.

Let's get started.

1.1 Overview

This tutorial is divided into four parts; they are:

1. Desire for Computers to See
2. What Is Computer Vision
3. Challenge of Computer Vision
4. Tasks in Computer Vision

1.2 Desire for Computers to See

We are awash in images. Smartphones have cameras, and taking a photo or video and sharing it has never been easier, resulting in the incredible growth of modern social networks like Instagram. YouTube might be the second largest search engine and hundreds of hours of video are uploaded every minute and billions of videos are watched every day.

The internet is comprised of text and images. It is relatively straightforward to index and search text, but in order to index and search images, algorithms need to know what the images contain. For the longest time, the content of images and video has remained opaque, best described using the meta descriptions provided by the person that uploaded them. To get the most out of image data, we need computers to “see” an image and understand the content. This is a trivial problem for a human, even young children.

- A person can describe the content of a photograph they have seen once.
- A person can summarize a video that they have only seen once.
- A person can recognize a face that they have only seen once before.

We require at least the same capabilities from computers in order to unlock our images and videos.

1.3 What Is Computer Vision

Computer vision is a field of study focused on the problem of helping computers to see.

At an abstract level, the goal of computer vision problems is to use the observed image data to infer something about the world.

— Page 83, *Computer Vision: Models, Learning, and Inference*, 2012.

It is a multidisciplinary field that could broadly be called a subfield of artificial intelligence and machine learning, which may involve the use of specialized methods and make use of general learning algorithms.

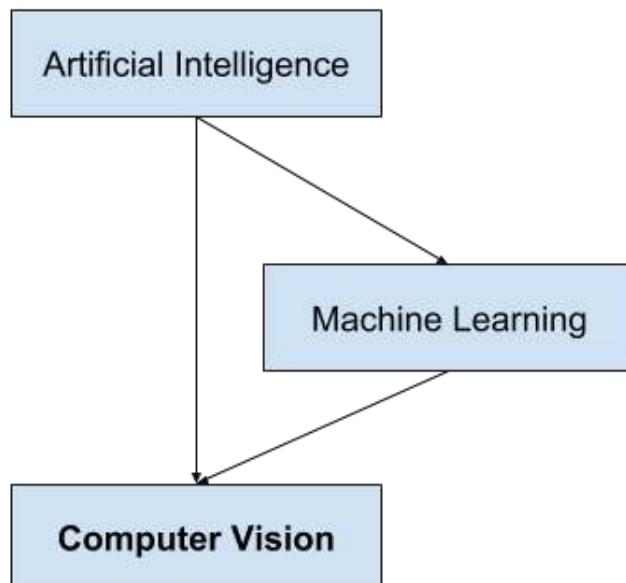


Figure 1.1: Overview of the Relationship of Artificial Intelligence and Computer Vision.

As a multidisciplinary area of study, it can look messy, with techniques borrowed and reused from a range of disparate engineering and computer science fields. One particular problem in vision may be easily addressed with a hand-crafted statistical method, whereas another may require a large and complex ensemble of generalized machine learning algorithms.

Computer vision as a field is an intellectual frontier. Like any frontier, it is exciting and disorganized, and there is often no reliable authority to appeal to. Many useful ideas have no theoretical grounding, and some theories are useless in practice; developed areas are widely scattered, and often one looks completely inaccessible from the other.

— Page xvii, *Computer Vision: A Modern Approach*, 2002.

The goal of computer vision is to understand the content of digital images. Typically, this involves developing methods that attempt to reproduce the capability of human vision. Understanding the content of digital images may involve extracting a description from the image, which may be an object, a text description, a three-dimensional model, and so on.

Computer vision is the automated extraction of information from images. Information can mean anything from 3D models, camera position, object detection and recognition to grouping and searching image content.

— Page ix, *Programming Computer Vision with Python*, 2012.

1.3.1 Computer Vision and Image Processing

Computer vision is distinct from image processing. Image processing is the process of creating a new image from an existing image, typically simplifying or enhancing the content in some way. It is a type of digital signal processing and is not concerned with understanding the content of an image. A given computer vision system may require image processing to be applied to raw input, e.g. pre-processing images. Examples of image processing include:

- Normalizing photometric properties of the image, such as brightness or color.
- Cropping the bounds of the image, such as centering an object in a photograph.
- Removing digital noise from an image, such as digital artifacts from low light levels.

1.4 Challenge of Computer Vision

Helping computers to see turns out to be very hard.

The goal of computer vision is to extract useful information from images. This has proved a surprisingly challenging task; it has occupied thousands of intelligent and creative minds over the last four decades, and despite this we are still far from being able to build a general-purpose “seeing machine.”

— Page 16, *Computer Vision: Models, Learning, and Inference*, 2012.

Computer vision may appear easy, perhaps because it is so effortless for humans. Initially, it was believed to be a trivially simple problem that could be solved by a student connecting a camera to a computer. After decades of research, *computer vision* remains unsolved, at least in terms of meeting the capabilities of human vision.

Making a computer see was something that leading experts in the field of Artificial Intelligence thought to be at the level of difficulty of a summer student's project back in the sixties. Forty years later the task is still unsolved and seems formidable.

— Page xi, *Multiple View Geometry in Computer Vision*, 2004.

One reason is that we don't have a strong grasp of how human vision works. Studying biological vision requires an understanding of the perception organs like the eyes, as well as the interpretation of the perception within the brain. Much progress has been made, both in charting the process and in terms of discovering the tricks and shortcuts used by the system, although like any study that involves the brain, there is a long way to go.

Perceptual psychologists have spent decades trying to understand how the visual system works and, even though they can devise optical illusions to tease apart some of its principles, a complete solution to this puzzle remains elusive

— Page 3, *Computer Vision: Algorithms and Applications*, 2010.

Another reason why it is such a challenging problem is because of the complexity inherent in the visual world. A given object may be seen from any orientation, in any lighting conditions, with any type of occlusion from other objects, and so on. A true vision system must be able to *see* in any of an infinite number of scenes and still extract something meaningful. Computers work well for tightly constrained problems, not open unbounded problems like visual perception.

1.5 Tasks in Computer Vision

Nevertheless, there has been progress in the field, especially in recent years with commodity systems for optical character recognition and face detection in cameras and smartphones.

Computer vision is at an extraordinary point in its development. The subject itself has been around since the 1960s, but only recently has it been possible to build useful computer systems using ideas from computer vision.

— Page xviii, *Computer Vision: A Modern Approach*, 2002.

The 2010 textbook on computer vision titled *Computer Vision: Algorithms and Applications* provides a list of some high-level problems where we have seen success with computer vision.

- Optical character recognition (OCR)
- Machine inspection
- Retail (e.g. automated checkouts)

- 3D model building (photogrammetry)
- Medical imaging
- Automotive safety
- Match move (e.g. merging CGI with live actors in movies)
- Motion capture (mocap)
- Surveillance
- Fingerprint recognition and biometrics

It is a broad area of study with many specialized tasks and techniques, as well as specializations to target application domains.

Computer vision has a wide variety of applications, both old (e.g., mobile robot navigation, industrial inspection, and military intelligence) and new (e.g., human computer interaction, image retrieval in digital libraries, medical image analysis, and the realistic rendering of synthetic scenes in computer graphics).

— Page xvii, *Computer Vision: A Modern Approach*, 2002.

It may be helpful to zoom in on some of the simpler computer vision tasks that you are likely to encounter or be interested in solving given the vast number of digital photographs and videos available. Many popular computer vision applications involve trying to recognize things in photographs; for example:

- **Object Classification:** What broad category of object is in this photograph?
- **Object Identification:** Which type of a given object is in this photograph?
- **Object Verification:** Is the object in the photograph?
- **Object Detection:** Where are the objects in the photograph?
- **Object Landmark Detection:** What are the key points for the object in the photograph?
- **Object Segmentation:** What pixels belong to the object in the image?
- **Object Recognition:** What objects are in this photograph and where are they?

Other common examples are related to information retrieval; for example: finding new images like an existing image or images.

1.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

1.6.1 Books

- *Computer Vision: Models, Learning, and Inference*, 2012.
<https://amzn.to/2rxrd0F>
- *Programming Computer Vision with Python*, 2012.
<https://amzn.to/2QKTAAL>
- *Multiple View Geometry in Computer Vision*, 2004.
<https://amzn.to/2LfHLE8>
- *Computer Vision: Algorithms and Applications*, 2010.
<https://amzn.to/2LcIt4J>
- *Computer Vision: A Modern Approach*, 2002.
<https://amzn.to/2rv7AqJ>

1.6.2 Articles

- Computer vision, Wikipedia.
https://en.wikipedia.org/wiki/Computer_vision
- Machine vision, Wikipedia.
https://en.wikipedia.org/wiki/Machine_vision
- Digital image processing, Wikipedia.
https://en.wikipedia.org/wiki/Digital_image_processing

1.7 Summary

In this tutorial, you discovered a gentle introduction to the field of computer vision. Specifically, you learned:

- The goal of the field of computer vision and its distinctness from image processing.
- What makes the problem of computer vision challenging.
- Typical problems or tasks pursued in computer vision.

1.7.1 Next

In the next section, you will discover the promise of deep learning for tasks in computer vision.

Chapter 2

Promise of Deep Learning for Computer Vision

The promise of deep learning in the field of computer vision is better performance by models that may require more data but less digital signal processing expertise to train and operate. There is a lot of hype and large claims around deep learning methods, but beyond the hype, deep learning methods are achieving state-of-the-art results on challenging problems. Notably, on computer vision tasks such as image classification, object recognition, and face detection. In this tutorial, you will discover the specific promises that deep learning methods have for tackling computer vision problems. After reading this tutorial, you will know:

- The promises of deep learning for computer vision.
- Examples of where deep learning has or is delivering on its promises.
- Key deep learning methods and applications for computer vision.

Let's get started.

2.1 Overview

This tutorial is divided into three parts; they are:

1. Promises of Deep Learning
2. Types of Deep Learning Network Models
3. Types of Computer Vision Problems

2.2 Promises of Deep Learning

Deep learning methods are popular, primarily because they are delivering on their promise. That is not to say that there is no hype around the technology, but that the hype is based on very real results that are being demonstrated across a suite of very challenging artificial intelligence problems from computer vision and natural language processing. Some of the first

large demonstrations of the power of deep learning were in computer vision, specifically image recognition. More recently in object detection and face recognition. In this tutorial, we will look at five specific promises of deep learning methods in the field of computer vision. In summary, they are:

- **The Promise of Automatic Feature Extraction.** Features can be automatically learned and extracted from raw image data.
- **The Promise of End-to-End Models.** Single end-to-end models can replace pipelines of specialized models.
- **The Promise of Model Reuse.** Learned features and even entire models can be reused across tasks.
- **The Promise of Superior Performance.** Techniques demonstrate better skill on challenging tasks.
- **The Promise of General Method.** A single general method can be used on a range of related tasks.

We will now take a closer look at each. There are other promises of deep learning for computer vision; these were just the five that I chose to highlight.

2.2.1 Promise 1: Automatic Feature Extraction

A major focus of study in the field of computer vision is on techniques to detect and extract features from digital images. Extracted features provide the context for inference about an image, and often the richer the features, the better the inference. Sophisticated hand-designed features such as scale-invariant feature transform (SIFT), Gabor filters, and histogram of oriented gradients (HOG) have been the focus of computer vision for feature extraction for some time, and have seen good success.

The promise of deep learning is that complex and useful features can be automatically learned directly from large image datasets. More specifically, that a deep hierarchy of rich features can be learned and automatically extracted from images, provided by the multiple deep layers of neural network models.

They have deeper architectures with the capacity to learn more complex features than the shallow ones. Also the expressivity and robust training algorithms allow to learn informative object representations without the need to design features manually.

— *Object Detection with Deep Learning: A Review*, 2018.

Deep neural network models are delivering on this promise, most notably demonstrated by the transition away from sophisticated hand-crafted feature detection methods such as SIFT toward deep convolutional neural networks on standard computer vision benchmark datasets and competitions, such as the ImageNet Large Scale Visual Recognition Competition (ILSVRC).

ILSVRC over the past five years has paved the way for several breakthroughs in computer vision. The field of categorical object recognition has dramatically evolved [...] starting from coded SIFT features and evolving to large-scale convolutional neural networks dominating at all three tasks of image classification, single-object localization, and object detection.

— *ImageNet Large Scale Visual Recognition Challenge*, 2015.

2.2.2 Promise 2: End-to-End Models

Addressing computer vision tasks traditionally involved using a system of modular models. Each model was designed for a specific task, such as feature extraction, image alignment, or classification. The models are used in a pipeline with a raw image at one end and an outcome, such as a prediction, at the other end. This pipeline approach can and is still used with deep learning models, where a feature detector model can be replaced with a deep neural network.

Alternately, deep neural networks allow a single model to subsume two or more traditional models, such as feature extraction and classification. It is common to use a single model trained directly on raw pixel values for image classification, and there has been a trend toward replacing pipelines that use a deep neural network model with a single model trained end-to-end directly.

With the availability of so much training data (along with an efficient algorithmic implementation and GPU computing resources) it became possible to learn neural networks directly from the image data, without needing to create multi-stage hand-tuned pipelines of extracted features and discriminative classifiers.

— *ImageNet Large Scale Visual Recognition Challenge*, 2015.

A good example of this is in object detection and face recognition where initially superior performance was achieved using a deep convolutional neural network for feature extraction only, where more recently, end-to-end models are trained directly using multiple-output models (e.g. class and bounding boxes) and/or new loss functions (e.g. contrastive or triplet loss functions).

2.2.3 Promise 3: Model Reuse

Typically, the feature detectors prepared for a dataset are highly specific to that dataset. This makes sense, as the more domain information that you can use in the model, the better the model is likely to perform in the domain. Deep neural networks are typically trained on datasets that are much larger than traditional datasets, e.g. millions or billions of images. This allows the models to learn features and hierarchies of features that are general across photographs, which is itself remarkable.

If this original dataset is large enough and general enough, then the spatial hierarchy of features learned by the pretrained network can effectively act as a generic model of the visual world, and hence its features can prove useful for many different computer vision problems, even though these new problems may involve completely different classes than those of the original task.

— Page 143, *Deep Learning with Python*, 2017.

For example, it is common to use deep models trained in the large ImageNet dataset, or a subset of this dataset, directly or as a starting point on a range of computer vision tasks.

... it is common to use the features from a convolutional network trained on ImageNet to solve other computer vision tasks

— Page 426, *Deep Learning*, 2016.

This is called transfer learning, and the use of pre-trained models that can take days and sometimes weeks to train has become standard practice.

The pre-trained models can be used to extract useful general features from digital images and can also be fine-tuned, tailored to the specifics of the new task. This can save a lot of time and resources and result in very good results almost immediately.

A common and highly effective approach to deep learning on small image datasets is to use a pretrained network.

— Page 143, *Deep Learning with Python*, 2017.

2.2.4 Promise 4: Superior Performance

An important promise of deep neural networks in computer vision is better performance. It is the dramatically better performance with deep neural networks that has been a catalyst for the growth and interest in the field of deep learning. Although the techniques have been around for decades, the spark was the outstanding performance by Alex Krizhevsky, et al. in 2012 for image classification.

The current intensity of commercial interest in deep learning began when Krizhevsky et al. (2012) won the ImageNet object recognition challenge ...

— Page 371, *Deep Learning*, 2016.

Their deep convolutional neural network model, at the time called SuperVision, and later referred to as AlexNet, resulted in a leap in classification accuracy.

We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

— *ImageNet Classification with Deep Convolutional Neural Networks*, 2012.

The technique was then adopted for a range of very challenging computer vision tasks, including object detection, which also saw a large leap in model performance over then state-of-the-art traditional methods.

The first breakthrough in object detection was the RCNN which resulted in an improvement of nearly 30% over the previous state-of-the-art.

— *A Survey of Modern Object Detection Literature using Deep Learning*, 2018.

This trend of improvement has continued year-over-year on a range of computer vision tasks. Performance has been so dramatic that tasks previously thought not easily addressable by computers and used as CAPTCHA to prevent spam (such as predicting whether a photo is of a dog or cat) are effectively *solved* and models on problems such as face recognition achieve better-than-human performance.

We can observe significant performance (mean average precision) improvement since deep learning entered the scene in 2012. The performance of the best detector has been steadily increasing by a significant amount on a yearly basis.

— *Deep Learning for Generic Object Detection: A Survey*, 2018.

2.2.5 Promise 5: General Method

Perhaps the most important promise of deep learning is that the top-performing models are all developed from the same basic components. The impressive results have come from one type of network, called the convolutional neural network, comprised of convolutional and pooling layers. It was specifically designed for image data and can be trained on pixel data directly (with some minor scaling).

Convolutional networks provide a way to specialize neural networks to work with data that has a clear grid-structured topology and to scale such models to very large size. This approach has been the most successful on a two-dimensional, image topology.

— Page 372, *Deep Learning*, 2016.

This is different from the broader field that may have required specialized feature detection methods developed for handwriting recognition, character recognition, face recognition, object detection, and so on. Instead, a single general class of model can be configured and used across each computer vision task directly. This is the promise of machine learning in general; it is impressive that such a versatile technique has been discovered and demonstrated for computer vision.

Further, the model is relatively straightforward to understand and to train, although it may require modern GPU hardware to train efficiently on a large dataset, and may require model hyperparameter tuning to achieve bleeding-edge performance.

2.3 Types of Deep Learning Network Models

Deep Learning is a large field of study, and not all of it is relevant to computer vision. It is easy to get bogged down in specific optimization methods or extensions to model types intended to lift performance. From a high-level, there is one method from deep learning that deserves the most attention for application in computer vision. It is:

- Convolutional Neural Networks (CNNs).

The reason that CNNs are the focus of attention for deep learning models is that they were specifically designed for image data.

Additionally, both of the following network types may be useful for interpreting or developing inference models from the features learned and extracted by CNNs; they are:

- Multilayer Perceptrons (MLP).
- Recurrent Neural Networks (RNNs).

The MLP or fully-connected type neural network layers are useful for developing models that make predictions given the learned features extracted by CNNs. RNNs, such as LSTMs, may be helpful when working with sequences of images over time, such as with video.

2.4 Types of Computer Vision Problems

Deep learning will not solve computer vision or artificial intelligence. To date, deep learning methods have been evaluated on a broader suite of problems from computer vision and achieved success on a small set, where success suggests performance or capability at or above what was previously possible with other methods. Importantly, those areas where deep learning methods are showing the greatest success are some of the more end-user facing, challenging, and perhaps more interesting problems. Five examples include:

- Optical Character Recognition.
- Image Classification.
- Object Detection.
- Face Detection.
- Face Recognition.

All five tasks are related under the umbrella of *object recognition*, which refers to tasks that involve identifying, localizing, and/or extracting specific content from digital photographs.

Most deep learning for computer vision is used for object recognition or detection of some form, whether this means reporting which object is present in an image, annotating an image with bounding boxes around each object, transcribing a sequence of symbols from an image, or labeling each pixel in an image with the identity of the object it belongs to.

— Page 453, *Deep Learning*, 2016.

2.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

2.5.1 Books

- *Deep Learning*, 2016.
<https://amzn.to/2EvccDe>
- *Deep Learning with Python*, 2017.
<https://amzn.to/2Exheit>

2.5.2 Papers

- *ImageNet Large Scale Visual Recognition Challenge*, 2015.
<https://arxiv.org/abs/1409.0575>
- *ImageNet Classification with Deep Convolutional Neural Networks*, 2012.
<https://dl.acm.org/citation.cfm?id=3065386>
- *Object Detection with Deep Learning: A Review*, 2018.
<https://arxiv.org/abs/1807.05511>
- *A Survey of Modern Object Detection Literature using Deep Learning*, 2018.
<https://arxiv.org/abs/1808.07256>
- *Deep Learning for Generic Object Detection: A Survey*, 2018.
<https://arxiv.org/abs/1809.02165>

2.6 Summary

In this tutorial, you discovered the specific promises that deep learning methods have for tackling computer vision problems. Specifically, you learned:

- The promises of deep learning for computer vision.
- Examples of where deep learning has or is delivering on its promises.
- Key deep learning methods and applications for computer vision.

2.6.1 Next

In the next section, you will discover how to develop simple models using the Keras deep learning Python library.

Chapter 3

How to Develop Deep Learning Models With Keras

Deep learning neural networks are very easy to create and evaluate in Python with Keras, but you must follow a strict model life-cycle. In this chapter you will discover the step-by-step life-cycle for creating, training and evaluating deep learning neural networks in Keras and how to make predictions with a trained model. You will also discover how to use the functional API that provides more flexibility when designing models. After reading this chapter you will know:

- How to define, compile, fit and evaluate a deep learning neural network in Keras.
- How to select standard defaults for regression and classification predictive modeling problems.
- How to use the functional API to develop standard Multilayer Perceptron, convolutional and recurrent neural networks.

Let's get started.

Note: It is assumed that you have a basic familiarity with deep learning and Keras. Nevertheless, this chapter should provide a refresher for the Keras API, and perhaps an introduction to the Keras functional API. See the Appendix [B](#) for installation instructions, if needed.

3.1 Keras Model Life-Cycle

Below is an overview of the 5 steps in the neural network model life-cycle in Keras:

1. Define Network.
2. Compile Network.
3. Fit Network.
4. Evaluate Network.
5. Make Predictions.

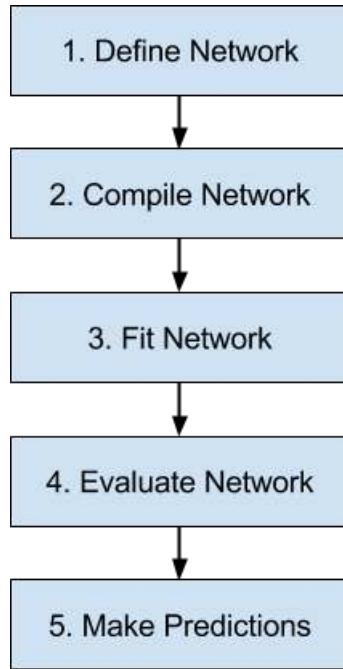


Figure 3.1: 5 Step Life-Cycle for Neural Network Models in Keras.

Let's take a look at each step in turn using the easy-to-use Keras Sequential API.

3.1.1 Step 1. Define Network

The first step is to define your neural network. Neural networks are defined in Keras as a sequence of layers. The container for these layers is the `Sequential` class. Create an instance of the `Sequential` class. Then you can create your layers and add them in the order that they should be connected. For example, we can do this in two steps:

```

...
model = Sequential()
model.add(Dense(2))
  
```

Listing 3.1: Sequential model with one `Dense` layer with 2 neurons.

But we can also do this in one step by creating an array of layers and passing it to the constructor of the `Sequential` class.

```

...
layers = [Dense(2)]
model = Sequential(layers)
  
```

Listing 3.2: Layers for a Sequential model defined as an array.

The first layer in the network must define the number of inputs to expect. The way that this is specified can differ depending on the network type, but for a Multilayer Perceptron model this is specified by the `input_dim` attribute. For example, a small Multilayer Perceptron model with 2 inputs in the visible layer, 5 neurons in the hidden layer and one neuron in the output layer can be defined as:

```
...
model = Sequential()
model.add(Dense(5, input_dim=2))
model.add(Dense(1))
```

Listing 3.3: Sequential model with 2 inputs.

Think of a Sequential model as a pipeline with your raw data fed in at the bottom and predictions that come out at the top. This is a helpful conception in Keras as components that were traditionally associated with a layer can also be split out and added as separate layers, clearly showing their role in the transform of data from input to prediction. For example, activation functions that transform a summed signal from each neuron in a layer can be extracted and added to the Sequential as a layer-like object called the `Activation` class.

```
...
model = Sequential()
model.add(Dense(5, input_dim=2))
model.add(Activation('relu'))
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

Listing 3.4: Sequential model with `Activation` functions defined separately from layers.

The choice of activation function is most important for the output layer as it will define the format that predictions will take. For example, below are some common predictive modeling problem types and the structure and standard activation function that you can use in the output layer:

- **Regression:** Linear activation function, or `linear` (or `None`), and the number of neurons matching the number of outputs.
- **Binary Classification (2 class):** Logistic activation function, or `sigmoid`, and one neuron the output layer.
- **Multiclass Classification (>2 class):** Softmax activation function, or `softmax`, and one output neuron per class value, assuming a one hot encoded output pattern.

3.1.2 Step 2. Compile Network

Once we have defined our network, we must compile it. Compilation is an efficiency step. It transforms the simple sequence of layers that we defined into a highly efficient series of matrix transforms in a format intended to be executed on your GPU or CPU, depending on how Keras is configured. Think of compilation as a precompute step for your network. It is always required after defining a model.

Compilation requires a number of parameters to be specified, tailored to training your network. Specifically, the optimization algorithm to use to train the network and the loss function used to evaluate the network that is minimized by the optimization algorithm. For example, below is a case of compiling a defined model and specifying the stochastic gradient descent (`sgd`) optimization algorithm and the mean squared error (`mean_squared_error`) loss function, intended for a regression type problem.

```
...
model.compile(optimizer='sgd', loss='mean_squared_error')
```

Listing 3.5: Example of compiling a defined model.

Alternately, the optimizer can be created and configured before being provided as an argument to the compilation step.

```
...
algorithm = SGD(lr=0.1, momentum=0.3)
model.compile(optimizer=algorithm, loss='mean_squared_error')
```

Listing 3.6: Example of defining the optimization algorithm separately.

The type of predictive modeling problem imposes constraints on the type of loss function that can be used. For example, below are some standard loss functions for different predictive model types:

- **Regression:** Mean Squared Error or `mean_squared_error`.
- **Binary Classification (2 class):** Logarithmic Loss, also called cross-entropy or `binary_crossentropy`.
- **Multiclass Classification (>2 class):** Multiclass Logarithmic Loss or `categorical_crossentropy`.

The most common optimization algorithm is stochastic gradient descent, but Keras also supports a suite of other state-of-the-art optimization algorithms that work well with little or no configuration. Perhaps the most commonly used optimization algorithms because of their generally better performance are:

- **Stochastic Gradient Descent**, or `sgd`, that requires the tuning of a learning rate and momentum.
- **Adam**, or `adam`, that requires the tuning of learning rate.
- **RMSprop**, or `rmsprop`, that requires the tuning of learning rate.

Finally, you can also specify metrics to collect while fitting your model in addition to the loss function. Generally, the most useful additional metric to collect is accuracy for classification problems. The metrics to collect are specified by name in an array. For example:

```
...
model.compile(optimizer='sgd', loss='mean_squared_error', metrics=['accuracy'])
```

Listing 3.7: Example of defining metrics when compiling the model.

3.1.3 Step 3. Fit Network

Once the network is compiled, it can be fit, which means adapting the model weights in response to a training dataset. Fitting the network requires the training data to be specified, both a matrix of input patterns, X , and an array of matching output patterns, y . The network is trained using the backpropagation algorithm and optimized according to the optimization algorithm and loss function specified when compiling the model.

The backpropagation algorithm requires that the network be trained for a specified number of epochs or exposures to the training dataset. Each epoch can be partitioned into groups of input-output pattern pairs called batches. This defines the number of patterns that the network is exposed to before the weights are updated within an epoch. It is also an efficiency optimization, ensuring that not too many input patterns are loaded into memory at a time. A minimal example of fitting a network is as follows:

```
...
history = model.fit(X, y, batch_size=10, epochs=100)
```

Listing 3.8: Example of fitting a compiled model.

Once fit, a history object is returned that provides a summary of the performance of the model during training. This includes both the loss and any additional metrics specified when compiling the model, recorded each epoch. Training can take a long time, from seconds to hours to days depending on the size of the network and the size of the training data.

By default, a progress bar is displayed on the command line for each epoch. This may create too much noise for you, or may cause problems for your environment, such as if you are in an interactive notebook or IDE. You can reduce the amount of information displayed to just the loss each epoch by setting the verbose argument to 2. You can turn off all output by setting verbose to 0. For example:

```
...
history = model.fit(X, y, batch_size=10, epochs=100, verbose=0)
```

Listing 3.9: Example of turning off verbose output when fitting the model.

3.1.4 Step 4. Evaluate Network

Once the network is trained, it can be evaluated. The network can be evaluated on the training data, but this will not provide a useful indication of the performance of the network as a predictive model, as it has seen all of this data before. We can evaluate the performance of the network on a separate dataset, unseen during training. This will provide an estimate of the performance of the network at making predictions for unseen data in the future.

The model evaluates the loss across all of the test patterns, as well as any other metrics specified when the model was compiled, like classification accuracy. A list of evaluation metrics is returned. For example, for a model compiled with the accuracy metric, we could evaluate it on a new dataset as follows:

```
...
loss, accuracy = model.evaluate(X, y)
```

Listing 3.10: Example of evaluating a fit model.

As with fitting the network, verbose output is provided to give an idea of the progress of evaluating the model. We can turn this off by setting the verbose argument to 0.

```
...  
loss, accuracy = model.evaluate(X, y, verbose=0)
```

Listing 3.11: Example of turning off verbose output when evaluating a fit model.

3.1.5 Step 5. Make Predictions

Once we are satisfied with the performance of our fit model, we can use it to make predictions on new data. This is as easy as calling the `predict()` function on the model with an array of new input patterns. For example:

```
...  
predictions = model.predict(X)
```

Listing 3.12: Example of making a prediction with a fit model.

The predictions will be returned in the format provided by the output layer of the network. In the case of a regression problem, these predictions may be in the format of the problem directly, provided by a linear activation function. For a binary classification problem, the predictions may be an array of probabilities for the first class that can be converted to a 1 or 0 by rounding.

For a multiclass classification problem, the results may be in the form of an array of probabilities (assuming a one hot encoded output variable) that may need to be converted to a single class output prediction using the `argmax()` NumPy function. Alternately, for classification problems, we can use the `predict_classes()` function that will automatically convert the predicted probabilities into class integer values.

```
...  
predictions = model.predict_classes(X)
```

Listing 3.13: Example of predicting classes with a fit model.

As with fitting and evaluating the network, verbose output is provided to give an idea of the progress of the model making predictions. We can turn this off by setting the verbose argument to 0.

```
...  
predictions = model.predict(X, verbose=0)
```

Listing 3.14: Example of disabling verbose output when making predictions.

3.2 Keras Functional Models

The sequential API allows you to create models layer-by-layer for most problems. It is limited in that it does not allow you to create models that share layers or have multiple input or output layers. The functional API in Keras is an alternate way of creating models that offers a lot more flexibility, including creating more complex models.

It specifically allows you to define multiple input or output models as well as models that share layers. More than that, it allows you to define ad hoc acyclic network graphs. Models are

defined by creating instances of layers and connecting them directly to each other in pairs, then defining a Model that specifies the layers to act as the input and output to the model. Let's look at the three unique aspects of Keras functional API in turn:

3.2.1 Defining Input

Unlike the Sequential model, you must create and define a standalone `Input` layer that specifies the shape of input data. The input layer takes a `shape` argument that is a tuple that indicates the dimensionality of the input data. When input data is one-dimensional, such as for a Multilayer Perceptron, the shape must explicitly leave room for the shape of the minibatch size used when splitting the data when training the network. Therefore, the shape tuple is always defined with a hanging last dimension `(2,)`, this is the way you must define a one-dimensional tuple in Python, for example:

```
...
visible = Input(shape=(2,))
```

Listing 3.15: Example of defining input for a functional model.

3.2.2 Connecting Layers

The layers in the model are connected pairwise. This is done by specifying where the input comes from when defining each new layer. A bracket or functional notation is used, such that after the layer is created, the layer from which the input to the current layer comes from is specified. Let's make this clear with a short example. We can create the input layer as above, then create a hidden layer as a `Dense` that receives input only from the input layer.

```
...
visible = Input(shape=(2,))
hidden = Dense(2)(visible)
```

Listing 3.16: Example of connecting a hidden layer to the visible layer.

Note it is the `(visible)` layer after the creation of the `Dense` layer that connects the input layer's output as the input to the `Dense` hidden layer. It is this way of connecting layers pairwise that gives the functional API its flexibility. For example, you can see how easy it would be to start defining ad hoc graphs of layers.

3.2.3 Creating the Model

After creating all of your model layers and connecting them together, you must define the model. As with the Sequential API, the model is the thing you can summarize, fit, evaluate, and use to make predictions. Keras provides a `Model` class that you can use to create a model from your created layers. It requires that you only specify the input and output layers. For example:

```
...
visible = Input(shape=(2,))
hidden = Dense(2)(visible)
model = Model(inputs=visible, outputs=hidden)
```

Listing 3.17: Example of creating a full model with the functional API.

Now that we know all of the key pieces of the Keras functional API, let's work through defining a suite of different models and build up some practice with it. Each example is executable and prints the structure and creates a diagram of the graph. I recommend doing this for your own models to make it clear what exactly you have defined. My hope is that these examples provide templates for you when you want to define your own models using the functional API in the future.

3.3 Standard Network Models

When getting started with the functional API, it is a good idea to see how some standard neural network models are defined. In this section, we will look at defining a simple Multilayer Perceptron, convolutional neural network, and recurrent neural network. These examples will provide a foundation for understanding the more elaborate examples later.

3.3.1 Multilayer Perceptron

In this section, we define a Multilayer Perceptron model for binary classification. The model has 10 inputs, 3 hidden layers with 10, 20, and 10 neurons, and an output layer with 1 output. Rectified linear activation functions are used in each hidden layer and a sigmoid activation function is used in the output layer, for binary classification.

```
# example of a multilayer perceptron
from keras.utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
visible = Input(shape=(10,))
hidden1 = Dense(10, activation='relu')(visible)
hidden2 = Dense(20, activation='relu')(hidden1)
hidden3 = Dense(10, activation='relu')(hidden2)
output = Dense(1, activation='sigmoid')(hidden3)
model = Model(inputs=visible, outputs=output)
# summarize layers
model.summary()
# plot graph
plot_model(model, to_file='multilayer_perceptron_graph.png')
```

Listing 3.18: Example of defining an MLP with the functional API.

Running the example prints the structure of the network.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 10)	0
dense_1 (Dense)	(None, 10)	110
dense_2 (Dense)	(None, 20)	220
dense_3 (Dense)	(None, 10)	210
dense_4 (Dense)	(None, 1)	11

```
=====
Total params: 551
Trainable params: 551
Non-trainable params: 0
=====
```

Listing 3.19: Summary of MLP model defined with the functional API.

A plot of the model graph is also created and saved to file.

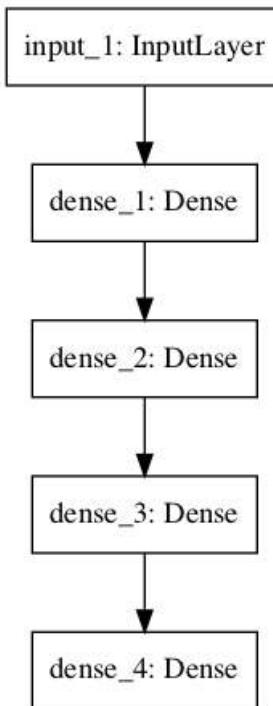


Figure 3.2: Plot of the MLP Model Graph.

Note, creating plots of Keras models requires that you install `pydot` and `pygraphviz` (the `graphviz` library and the Python wrapper). Instructions for installing these libraries vary for different systems. If this is a challenge for you (e.g. you're on Windows), consider commenting out the calls to `plot_model()` when you see them.

3.3.2 Convolutional Neural Network

In this section, we will define a convolutional neural network for image classification. The model receives black and white 64×64 images as input, then has a sequence of two convolutional and pooling layers as feature extractors, followed by a fully connected layer to interpret the features and an output layer with a sigmoid activation for two-class predictions. If this is your first CNN, don't worry. We will cover convolutional layers in detail in Chapter 11 and Chapter 12, and pooling layers in Chapter 13.

```
# example of a convolutional neural network
from keras.utils import plot_model
from keras.models import Model
from keras.layers import Input
```

```

from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv2D
from keras.layers.pooling import MaxPooling2D
visible = Input(shape=(64,64,1))
conv1 = Conv2D(32, (4,4), activation='relu')(visible)
pool1 = MaxPooling2D()(conv1)
conv2 = Conv2D(16, (4,4), activation='relu')(pool1)
pool2 = MaxPooling2D()(conv2)
flat1 = Flatten()(pool2)
hidden1 = Dense(10, activation='relu')(flat1)
output = Dense(1, activation='sigmoid')(hidden1)
model = Model(inputs=visible, outputs=output)
# summarize layers
model.summary()
# plot graph
plot_model(model, to_file='convolutional_neural_network.png')

```

Listing 3.20: Example of defining an CNN with the functional API.

Running the example summarizes the model layers.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 64, 64, 1)	0
conv2d_1 (Conv2D)	(None, 61, 61, 32)	544
max_pooling2d_1 (MaxPooling2D)	(None, 30, 30, 32)	0
conv2d_2 (Conv2D)	(None, 27, 27, 16)	8208
max_pooling2d_2 (MaxPooling2D)	(None, 13, 13, 16)	0
flatten_1 (Flatten)	(None, 2704)	0
dense_1 (Dense)	(None, 10)	27050
dense_2 (Dense)	(None, 1)	11
Total params:	35,813	
Trainable params:	35,813	
Non-trainable params:	0	

Listing 3.21: Summary of CNN model defined with the functional API.

A plot of the model graph is also created and saved to file.

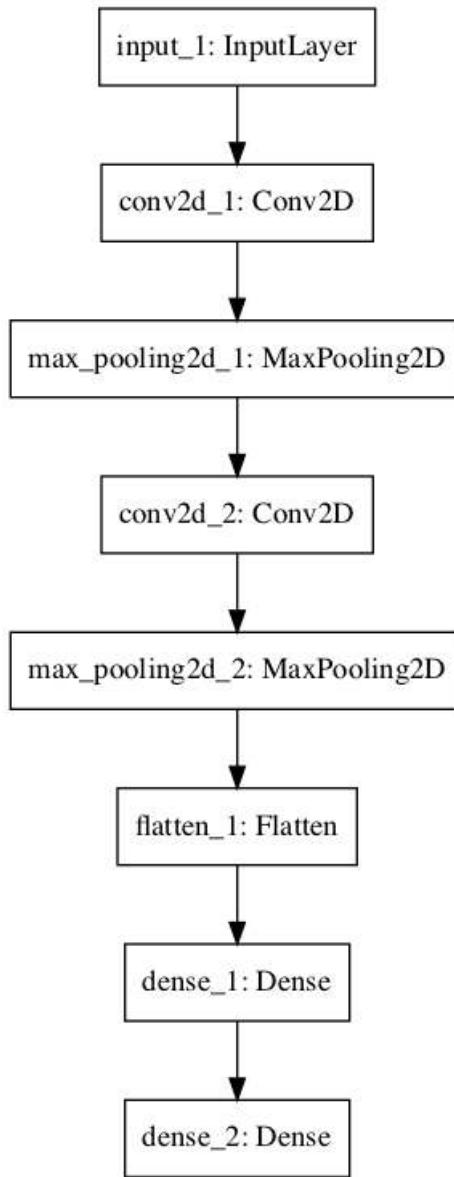


Figure 3.3: Plot of the CNN Model Graph.

3.3.3 Recurrent Neural Network

In this section, we will define a long short-term memory recurrent neural network for sequence classification. The model expects 100 time steps of one feature as input. The model has a single LSTM hidden layer to extract features from the sequence, followed by a fully connected layer to interpret the LSTM output, followed by an output layer for making binary predictions.

```

# example of a recurrent neural network
from keras.utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers.recurrent import LSTM
visible = Input(shape=(100,1))
hidden1 = LSTM(10)(visible)
  
```

```

hidden2 = Dense(10, activation='relu')(hidden1)
output = Dense(1, activation='sigmoid')(hidden2)
model = Model(inputs=visible, outputs=output)
# summarize layers
model.summary()
# plot graph
plot_model(model, to_file='recurrent_neural_network.png')

```

Listing 3.22: Example of defining an RNN with the functional API.

Running the example summarizes the model layers.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 100, 1)	0
lstm_1 (LSTM)	(None, 10)	480
dense_1 (Dense)	(None, 10)	110
dense_2 (Dense)	(None, 1)	11
<hr/>		
Total params: 601		
Trainable params: 601		
Non-trainable params: 0		
<hr/>		

Listing 3.23: Summary of RNN model defined with the functional API.

A plot of the model graph is also created and saved to file.

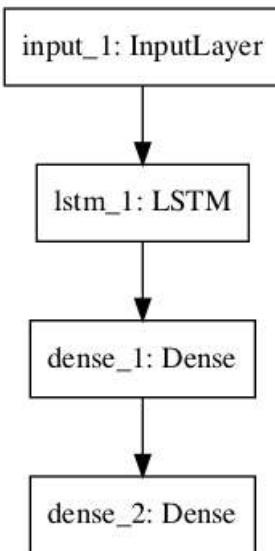


Figure 3.4: Plot of the RNN Model Graph.

3.4 Further Reading

This section provides more resources on the topic if you are looking go deeper.

- Keras documentation for Sequential Models.
<https://keras.io/models/sequential/>
- Keras documentation for Functional Models.
<https://keras.io/models/model/>
- Getting started with the Keras Sequential model.
<https://keras.io/models/model/>
- Getting started with the Keras functional API.
<https://keras.io/models/model/>
- Keras documentation for optimization algorithms.
<https://keras.io/optimizers/>
- Keras documentation for loss functions.
<https://keras.io/losses/>

3.5 Summary

In this tutorial, you discovered the step-by-step life-cycle for creating, training and evaluating deep learning neural networks in Keras and how to use the functional API that provides more flexibility when designing models. Specifically, you learned:

- How to define, compile, fit and evaluate a deep learning neural network in Keras.
- How to select standard defaults for regression and classification predictive modeling problems.
- How to use the functional API to develop standard Multilayer Perceptron, convolutional and recurrent neural networks.

3.5.1 Next

This was the final tutorial in this part on the foundations. In the next part you will discover the image data preparation required prior to modeling.

Part II

Image Data Preparation

Overview

In this part you will discover how to load and prepare image data ready for modeling. After reading the chapters in this part, you will know:

- How to load and handle image data using the PIL/Pillow standard Python library (Chapter [4](#)).
- How to manually standardize and normalize pixel values to prepare images for modeling (Chapter [5](#)).
- How to load and handle image data directly using the Keras deep learning library (Chapter [6](#)).
- How to scale pixel values to prepare images for modeling using Keras (Chapter [7](#)).
- How to load large image datasets from directories using Keras (Chapter [8](#)).
- How to use data augmentation to expand the size of an image training dataset (Chapter [9](#)).

Chapter 4

How to Load and Manipulate Images With PIL/Pillow

Before you can develop predictive models for image data, you must learn how to load and manipulate images and photographs. The most popular and de facto standard library in Python for loading and working with image data is Pillow. Pillow is an updated version of the Python Image Library, or PIL, and supports a range of simple and sophisticated image manipulation functionality. It is also the basis for simple image support in other Python libraries such as SciPy and Matplotlib. In this tutorial, you will discover how to load and manipulate image data using the Pillow Python library. After completing this tutorial, you will know:

- How to install the Pillow library and confirm it is working correctly.
- How to load images from file, convert loaded images to NumPy arrays, and save images in new formats.
- How to perform basic transforms to image data such as resize, flips, rotations, and cropping.

Let's get started.

4.1 Tutorial Overview

This tutorial is divided into six parts; they are:

1. How to Install Pillow
2. How to Load and Display Images
3. How to Convert Images to NumPy Arrays and Back
4. How to Save Images to File
5. How to Resize Images
6. How to Flip, Rotate, and Crop Images

4.2 How to Install Pillow

The Python Imaging Library, or PIL for short, is an open source library for loading and manipulating images. It was developed and made available more than 25 years ago and has become a de facto standard API for working with images in Python. The library is now defunct and no longer updated and does not support Python 3. Pillow is a PIL library that supports Python 3 and is the preferred modern library for image manipulation in Python. It is even required for simple image loading and saving in other Python scientific libraries such as SciPy and Matplotlib.

The Pillow library is installed as a part of most SciPy installations; for example, if you are using Anaconda. If you manage the installation of Python software packages yourself for your workstation, you can easily install Pillow using pip; for example:

```
sudo pip install Pillow
```

Listing 4.1: Example of installing Pillow with Pip.

For more help installing Pillow manually, see:

- [Pillow Installation Instructions.¹](#)

Pillow is built on top of the older PIL and you can confirm that the library was installed correctly by printing the version number; for example:

```
# check Pillow version number
import PIL
print('Pillow Version:', PIL.__version__)
```

Listing 4.2: Example of checking the PIL and Pillow versions.

Running the example will print the version number for Pillow; your version number should be the same or higher.

```
Pillow Version: 6.1.0
```

Listing 4.3: Example output from checking the Pillow version.

Now that your environment is set up, let's look at how to load an image.

4.3 How to Load and Display Images

We need a test image to demonstrate some important features of using the Pillow library. In this tutorial, we will use a photograph of the Sydney Opera House, taken by Ed Dunens² and made available on Flickr under a creative commons license, some rights reserved.

¹<https://pillow.readthedocs.io/en/stable/installation.html>

²<https://www.flickr.com/photos/blachswan/36102705716/>



Figure 4.1: Photograph of the Sydney Opera House.

Download the photograph and save it in your current working directory with the file name `opera_house.jpg`.

- Download Photo (`opera_house.jpg`).³

Images are typically in PNG or JPEG format and can be loaded directly using the `open()` function on `Image` class. This returns an `Image` object that contains the pixel data for the image as well as details about the image. The `Image` class is the main workhorse for the Pillow library and provides a ton of properties about the image as well as functions that allow you to manipulate the pixels and format of the image.

The `format` property on the image will report the image format (e.g. JPEG), the `mode` will report the pixel channel format (e.g. RGB or CMYK), and the `size` will report the dimensions of the image in pixels (e.g. 640×480). The `show()` function will display the image using your operating systems default application. The example below demonstrates how to load and show an image using the `Image` class in the Pillow library.

```
# load and show an image with Pillow
from PIL import Image
# load the image
image = Image.open('opera_house.jpg')
# summarize some details about the image
print(image.format)
print(image.mode)
print(image.size)
# show the image
image.show()
```

Listing 4.4: Example of loading and displaying an image with PIL.

³<https://machinelearningmastery.com/wp-content/uploads/2019/01/Sydney-Opera-House.jpg>

Running the example will first load the image, report the format, mode, and size, then show the image on your desktop.

```
JPEG  
RGB  
(640, 360)
```

Listing 4.5: Example output from loading and displaying an image with PIL.

The image is shown using the default image preview application for your operating system, such as Preview on MacOS.

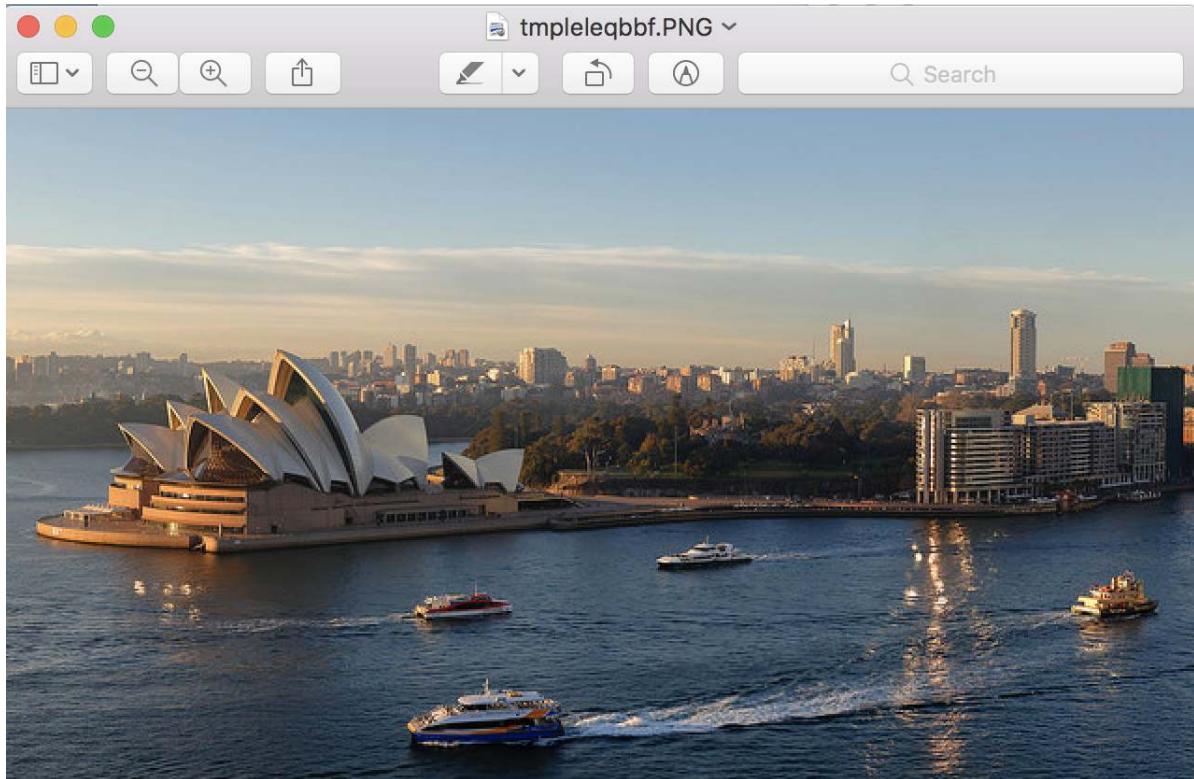


Figure 4.2: Sydney Opera House Displayed Using the Default Image Preview Application.

Now that you know how to load an image, let's look at how you can access the pixel data of images.

4.4 How to Convert Images to NumPy Arrays and Back

Often in machine learning, we want to work with images as NumPy arrays of pixel data. With Pillow installed, you can also use the Matplotlib library to load the image and display it within a Matplotlib frame. This can be achieved using the `imread()` function that loads the image as an array of pixels directly and the `imshow()` function that will display an array of pixels as an image. The example below loads and displays the same image using Matplotlib that, in turn, will use Pillow under the covers.

```
# load and display an image with Matplotlib  
from matplotlib import image
```

```
from matplotlib import pyplot
# load image as pixel array
data = image.imread('opera_house.jpg')
# summarize shape of the pixel array
print(data.dtype)
print(data.shape)
# display the array of pixels as an image
pyplot.imshow(data)
pyplot.show()
```

Listing 4.6: Example of converting an image to an array with PIL.

Running the example first loads the image and then reports the data type of the array, in this case, 8-bit unsigned integers, then reports the shape of the array, in this case, 360 pixels high by 640 pixels wide and three channels for red, green, and blue. **Note**, the wide and height order may be listed differently (e.g. reversed), depending on your platform.

```
uint8
(360, 640, 3)
```

Listing 4.7: Example output from converting an image to an array with PIL.

Finally, the image is displayed using Matplotlib.

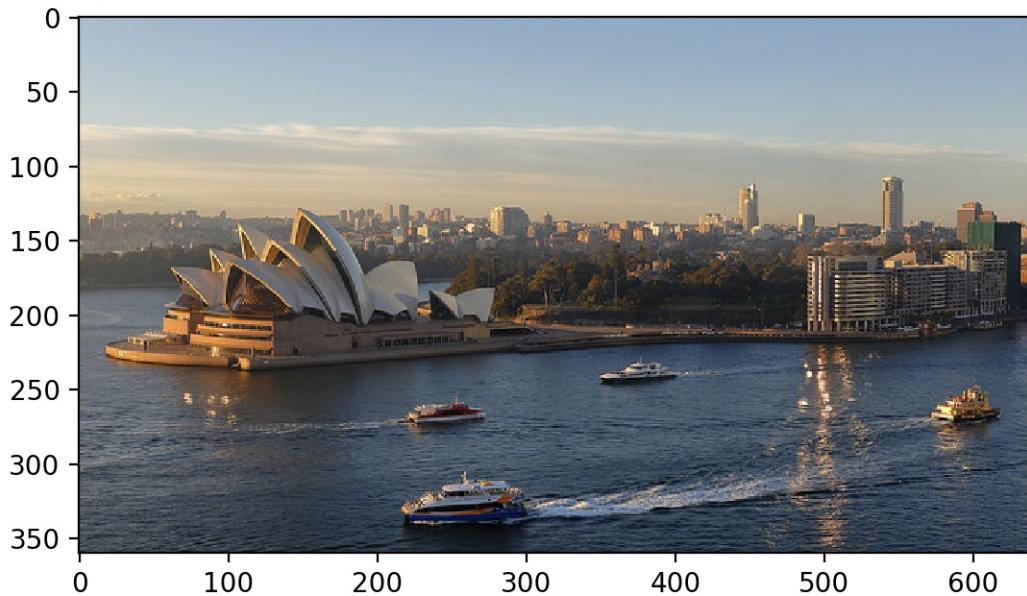


Figure 4.3: Sydney Opera House Displayed Using Matplotlib.

The Matplotlib wrapper functions can be more effective than using Pillow directly. Nevertheless, you can access the pixel data from a Pillow Image. Perhaps the simplest way is to construct a NumPy array and pass in the Image object. The process can be reversed, converting a given array of pixel data into a Pillow Image object using the `Image.fromarray()` function. This can be useful if image data is manipulated as a NumPy array and you then want to save it later as a PNG or JPEG file. The example below loads the photo as a Pillow Image object and converts it to a NumPy array, then converts it back to an Image object again.

```
# load image and convert to and from NumPy array
from PIL import Image
from numpy import asarray
# load the image
image = Image.open('opera_house.jpg')
# convert image to numpy array
data = asarray(image)
# summarize shape
print(data.shape)
# create Pillow image
image2 = Image.fromarray(data)
# summarize image details
print(image2.format)
print(image2.mode)
print(image2.size)
```

Listing 4.8: Example of alternate way of converting an image to an array with PIL.

Running the example first loads the photo as a Pillow image then converts it to a NumPy array and reports the shape of the array. Finally, the array is converted back into a Pillow image and the details are reported.

```
(360, 640, 3)
JPEG
RGB
(640, 360)
```

Listing 4.9: Example output from alternate way of converting an image to an array with PIL.

Both approaches are effective for loading image data into NumPy arrays, although the Matplotlib `imread()` function uses fewer lines of code than loading and converting a Pillow Image object and may be preferred. For example, you could easily load all images in a directory as a list as follows:

```
# load all images in a directory
from os import listdir
from matplotlib import image
# load all images in a directory
loaded_images = list()
for filename in listdir('images'):
    # load image
    img_data = image.imread('images/' + filename)
    # store loaded image
    loaded_images.append(img_data)
print('> loaded %s %s' % (filename, img_data.shape))
```

Listing 4.10: Example of loading all images from a directory with PIL.

Now that we know how to load images as NumPy arrays, let's look at how to save images to file.

4.5 How to Save Images to File

An image object can be saved by calling the `save()` function. This can be useful if you want to save an image in a different format, in which case the `format` argument can be specified, such as PNG, GIF, or PEG. For example, the code listing below loads the photograph in JPEG format and saves it in PNG format.

```
# example of saving an image in another format
from PIL import Image
# load the image
image = Image.open('opera_house.jpg')
# save as PNG format
image.save('opera_house.png', format='PNG')
# load the image again and inspect the format
image2 = Image.open('opera_house.png')
print(image2.format)
```

Listing 4.11: Example of saving an image in PNG format.

Running the example loads the JPEG image, saves it in PNG format, then loads the newly saved image again, and confirms that the format is indeed PNG.

PNG

Listing 4.12: Example output from saving an image in PNG format.

Saving images is useful if you perform some data preparation on the image before modeling. One example is converting color images (RGB channels) to grayscale (1 channel). There are a number of ways to convert an image to grayscale, but Pillow provides the `convert()` function and the mode 'L' will convert an image to grayscale.

```
# example of saving a grayscale version of a loaded image
from PIL import Image
# load the image
image = Image.open('opera_house.jpg')
# convert the image to grayscale
gs_image = image.convert(mode='L')
# save in jpeg format
gs_image.save('opera_house_grayscale.jpg')
# load the image again and show it
image2 = Image.open('opera_house_grayscale.jpg')
# show the image
image2.show()
```

Listing 4.13: Example of saving an image in grayscale format.

Running the example loads the photograph, converts it to grayscale, saves the image in a new file, then loads it again and shows it to confirm that the photo is now grayscale instead of color.



Figure 4.4: Example of Grayscale Version of Photograph.

4.6 How to Resize Images

It is important to be able to resize images before modeling. Sometimes it is desirable to thumbnail all images to have the same width or height. This can be achieved with Pillow using the `thumbnail()` function. The function takes a tuple with the height and width, and the image will be resized so that the height and width of the image are equal or smaller than the specified shape.

For example, the test photograph we have been working with has the width and height of (640, 360). We can resize it to (100, 100), in which case the largest dimension, in this case, the width, will be reduced to 100, and the height will be scaled in order to retain the aspect ratio of the image. The example below will load the photograph and create a smaller thumbnail with a width and height of 100 pixels.

```
# create a thumbnail of an image
from PIL import Image
# load the image
image = Image.open('opera_house.jpg')
# report the size of the image
print(image.size)
# create a thumbnail and preserve aspect ratio
image.thumbnail((100,100))
# report the size of the modified image
print(image.size)
# show the image
image.show()
```

Listing 4.14: Example of resizing an image as a thumbnail.

Running the example first loads the photograph and reports the width and height. The image is then resized. In this case, the width is reduced to 100 pixels and the height is reduced to 56 pixels, maintaining the aspect ratio of the original image.

```
(640, 360)  
(100, 56)
```

Listing 4.15: Example output from resizing an image as a thumbnail.

We may not want to preserve the aspect ratio, and instead, we may want to force the pixels into a new shape. This can be achieved using the `resize()` function that allows you to specify the width and height in pixels and the image will be reduced or stretched to fit the new shape. The example below demonstrates how to resize a new image and ignore the original aspect ratio.

```
# resize image and force a new shape  
from PIL import Image  
# load the image  
image = Image.open('opera_house.jpg')  
# report the size of the image  
print(image.size)  
# resize image and ignore original aspect ratio  
img_resized = image.resize((200,200))  
# report the size of the thumbnail  
print(img_resized.size)  
# show the image  
img_resized.show()
```

Listing 4.16: Example of resizing an image and forcing an aspect ratio.

Running the example loads the image, reports the shape of the image, then resizes it to have a width and height of 200 pixels.

```
(640, 360)  
(200, 200)
```

Listing 4.17: Example output from resizing an image and forcing an aspect ratio.

The size of the image is shown and we can see that the wide photograph has been compressed into a square, although all of the features are still quite visible and obvious. Standard resampling algorithms are used to invent or remove pixels when resizing, and you can specify a technique, although default is a bicubic resampling algorithm that suits most general applications.

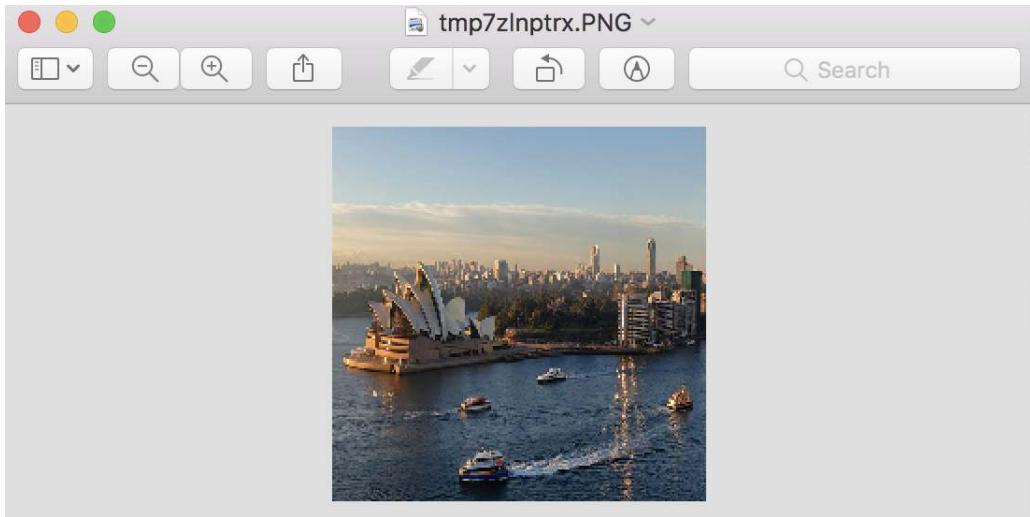


Figure 4.5: Resized Photograph That Does Not Preserve the Original Aspect Ratio.

4.7 How to Flip, Rotate, and Crop Images

Simple image manipulation can be used to create new versions of images that, in turn, can provide a richer training dataset when modeling. Generally, this is referred to as data augmentation and may involve creating flipped, rotated, cropped, or other modified versions of the original images with the hope that the algorithm will learn to extract the same features from the image data regardless of where they might appear. You may want to implement your own data augmentation schemes, in which case you need to know how to perform basic manipulations of your image data.

4.7.1 Flip Image

An image can be flipped by calling the `flip()` function and passing in a method such as `FLIP_LEFT_RIGHT` for a horizontal flip or `FLIP_TOP_BOTTOM` for a vertical flip. Other flips are also available. The example below creates both horizontal and vertical flipped versions of the image.

```
# create flipped versions of an image
from PIL import Image
from matplotlib import pyplot
# load image
image = Image.open('opera_house.jpg')
# horizontal flip
hoz_flip = image.transpose(Image.FLIP_LEFT_RIGHT)
# vertical flip
ver_flip = image.transpose(Image.FLIP_TOP_BOTTOM)
# plot all three images using matplotlib
pyplot.subplot(311)
pyplot.imshow(image)
pyplot.subplot(312)
pyplot.imshow(hoz_flip)
pyplot.subplot(313)
pyplot.imshow(ver_flip)
```

```
pyplot.show()
```

Listing 4.18: Example of creating flipped versions of an image.

Running the example loads the photograph and creates horizontally and vertically flipped versions of the photograph, then plots all three versions as subplots using Matplotlib. You will note that the `imshow()` function can plot the `Image` object directly without having to convert it to a NumPy array.

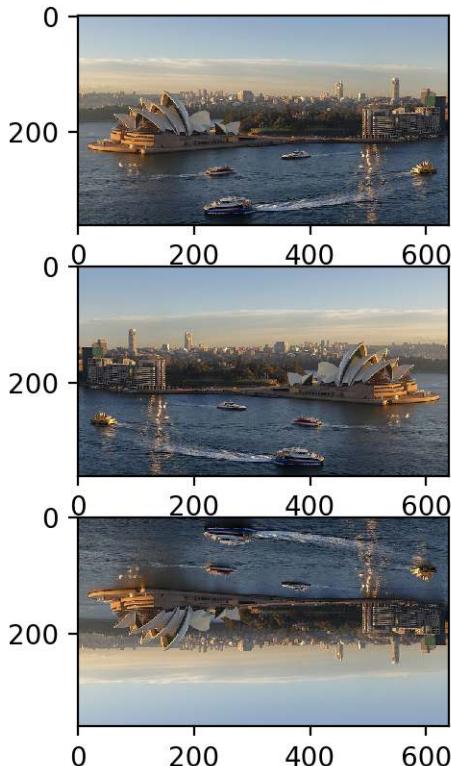


Figure 4.6: Plot of Original, Horizontal, and Vertical Flipped Versions of a Photograph.

4.7.2 Rotate Image

An image can be rotated using the `rotate()` function and passing in the angle for the rotation. The function offers additional control such as whether or not to expand the dimensions of the image to fit the rotated pixel values (default is to clip to the same size), where to center the rotation of the image (default is the center), and the fill color for pixels outside of the image (default is black). The example below creates a few rotated versions of the image.

```
# create rotated versions of an image
from PIL import Image
from matplotlib import pyplot
# load image
image = Image.open('opera_house.jpg')
```

```
# plot original image
pyplot.subplot(311)
pyplot.imshow(image)
# rotate 45 degrees
pyplot.subplot(312)
pyplot.imshow(image.rotate(45))
# rotate 90 degrees
pyplot.subplot(313)
pyplot.imshow(image.rotate(90))
pyplot.show()
```

Listing 4.19: Example of creating rotated versions of an image.

Running the example plots the original photograph, then a version of the photograph rotated 45 degrees, and another rotated 90 degrees. You can see that in both rotations, the pixels are clipped to the original dimensions of the image and that the empty pixels are filled with black color.

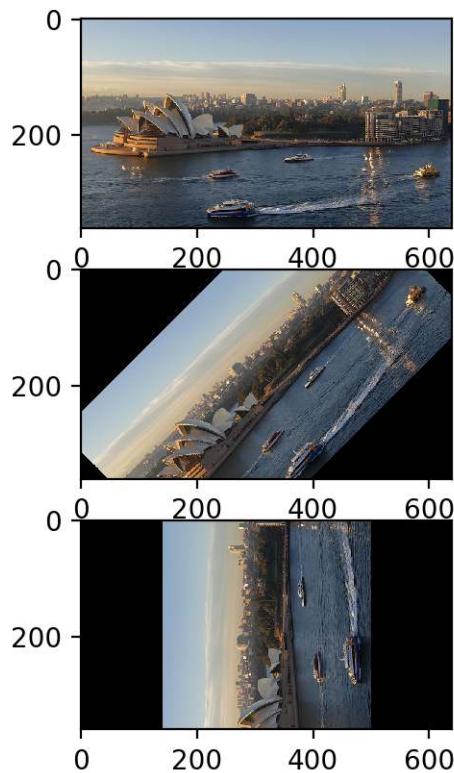


Figure 4.7: Plot of Original and Rotated Version of a Photograph.

4.7.3 Cropped Image

An image can be cropped: that is, a piece can be cut out to create a new image, using the `crop()` function. The `crop` function takes a tuple argument that defines the two x and y coordinates of

the box to crop out of the image. For example, if the image is 2,000 by 2,000 pixels, we can clip out a 100 by 100 pixel box in the image. The example below demonstrates how to create a new image as a crop from a loaded image.

```
# example of cropping an image
from PIL import Image
# load image
image = Image.open('opera_house.jpg')
# create a cropped image
cropped = image.crop((100, 100, 200, 200))
# show cropped image
cropped.show()
```

Listing 4.20: Example of creating cropped versions of an image.

Running the example creates a cropped square image of 100 pixels starting at 100,100 and extending down and left to 200,200. The cropped square is then displayed.

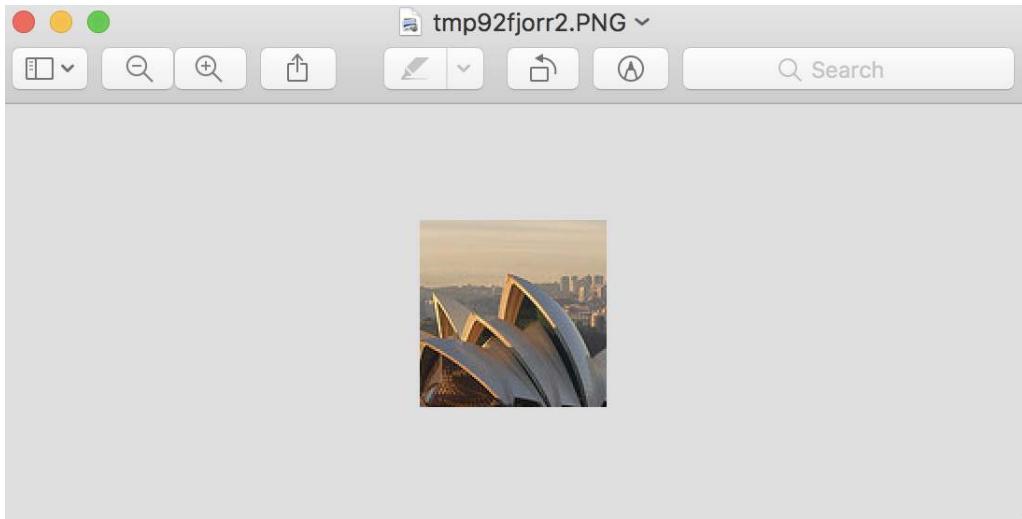


Figure 4.8: Example of a Cropped Version of a Photograph.

4.8 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Your Own Images.** Experiment with Pillow functions for reading and manipulating images with your own image data.
- **More Transforms.** Review the Pillow API documentation and experiment with additional image manipulation functions.
- **Image Pre-processing.** Write a function to create augmented versions of an image ready for use with a deep learning neural network.

If you explore any of these extensions, I'd love to know.

4.9 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- Pillow Homepage.
<https://python-pillow.org/>
- Pillow Installation Instructions.
<https://pillow.readthedocs.io/en/stable/installation.html>
- Pillow (PIL Fork) API Documentation.
<https://pillow.readthedocs.io/en/5.3.x/>
- Pillow Handbook Tutorial.
<https://pillow.readthedocs.io/en/stable/handbook/tutorial.html>
- Pillow GitHub Project.
<https://github.com/python-pillow/Pillow>
- Python Imaging Library (PIL) Homepage.
<http://www.pythonware.com/products/pil/>
- Python Imaging Library, Wikipedia.
https://en.wikipedia.org/wiki/Python_Imaging_Library
- Matplotlib: Image tutorial.
https://matplotlib.org/users/image_tutorial.html

4.10 Summary

In this tutorial, you discovered how to load and manipulate image data using the Pillow Python library. Specifically, you learned:

- How to install the Pillow library and confirm it is working correctly.
- How to load images from file, convert loaded images to NumPy arrays, and save images in new formats.
- How to perform basic transforms to image data such as resize, flips, rotations, and cropping.

4.10.1 Next

In the next section, you will discover how to manually scale pixel values prior to modeling.

Chapter 5

How to Manually Scale Image Pixel Data

Images are comprised of matrices of pixel values. Black and white images are single matrix of pixels, whereas color images have a separate array of pixel values for each color channel, such as red, green, and blue. Pixel values are often unsigned integers in the range between 0 and 255. Although these pixel values can be presented directly to neural network models in their raw format, this can result in challenges during modeling, such as in slower than expected training of the model.

Instead, there can be great benefit in preparing the image pixel values prior to modeling, such as simply scaling pixel values to the range 0-1 to centering and even standardizing the values. In this tutorial, you will discover how to prepare image data for modeling with deep learning neural networks. After completing this tutorial, you will know:

- How to normalize pixel values to a range between zero and one.
- How to center pixel values both globally across channels and locally per channel.
- How to standardize pixel values and how to shift standardized pixel values to the positive domain.

Let's get started.

5.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Sample Image
2. Normalize Pixel Values
3. Center Pixel Values
4. Standardize Pixel Values

5.2 Sample Image

We need a sample image for testing in this tutorial. We will use a photograph of the Sydney Harbor Bridge taken by *Bernard Spragg. NZ*¹ and released under a permissive license.



Figure 5.1: Photograph of the Sydney Harbor Bridge.

Download the photograph and place it into your current working directory with the filename `sydney_bridge.jpg`.

- Download Photo ([sydney_bridge.jpg](#)).

The example below will load the image, display some properties about the loaded image, then show the image. This example and the rest of the tutorial assumes that you have the Pillow Python library installed.

```
# load and show an image with Pillow
from PIL import Image
# load the image
image = Image.open('sydney_bridge.jpg')
# summarize some details about the image
print(image.format)
print(image.mode)
print(image.size)
# show the image
image.show()
```

Listing 5.1: Example of loading and displaying an image with PIL.

¹<https://www.flickr.com/photos/volvob12b/27318376851/>

Running the example reports the format of the image, which is JPEG, and the mode, which is RGB for the three color channels. Next, the size of the image is reported, showing 640 pixels in width and 374 pixels in height.

```
JPEG  
RGB  
(640, 374)
```

Listing 5.2: Example output from loading and displaying an image with PIL.



Figure 5.2: The Sydney Harbor Bridge Photograph Loaded From File.

5.3 Normalize Pixel Values

For most image data, the pixel values are integers with values between 0 and 255. Neural networks process inputs using small weight values, and inputs with large integer values can disrupt or slow down the learning process. As such it is good practice to normalize the pixel values so that each pixel value has a value between 0 and 1. It is valid for images to have pixel values in the range 0-1 and images can be viewed normally.

This can be achieved by dividing all pixels values by the largest pixel value; that is 255. This is performed across all channels, regardless of the actual range of pixel values that are present in the image. The example below loads the image and converts it into a NumPy array. The data type of the array is reported and the minimum and maximum pixel values across all three channels are then printed. Next, the array is converted to the float data type before the pixel values are normalized and the new range of pixel values is reported.

```
# example of pixel normalization
from numpy import asarray
from PIL import Image
# load image
image = Image.open('sydney_bridge.jpg')
pixels = asarray(image)
# confirm pixel range is 0-255
print('Data Type: %s' % pixels.dtype)
print('Min: %.3f, Max: %.3f' % (pixels.min(), pixels.max()))
# convert from integers to floats
pixels = pixels.astype('float32')
# normalize to the range 0-1
pixels /= 255.0
# confirm the normalization
print('Min: %.3f, Max: %.3f' % (pixels.min(), pixels.max()))
```

Listing 5.3: Example of normalizing pixel values.

Running the example prints the data type of the NumPy array of pixel values, which we can see is an 8-bit unsigned integer.

The min and maximum pixel values are printed, showing the expected 0 and 255 respectively. The pixel values are normalized and the new minimum and maximum of 0.0 and 1.0 are then reported.

```
Data Type: uint8
Min: 0.000, Max: 255.000
Min: 0.000, Max: 1.000
```

Listing 5.4: Example output from normalizing pixel values.

Normalization is a good default data preparation that can be performed if you are in doubt as to the type of data preparation to perform. It can be performed per image and does not require the calculation of statistics across the training dataset, as the range of pixel values is a domain standard.

5.4 Center Pixel Values

A popular data preparation technique for image data is to subtract the mean value from the pixel values. This approach is called centering, as the distribution of the pixel values is centered on the value of zero. Centering can be performed before or after normalization. Centering the pixels then normalizing will mean that the pixel values will be centered close to 0.5 and be in the range 0-1. Centering after normalization will mean that the pixels will have positive and negative values, in which case images will not display correctly (e.g. pixels are expected to have value in the range 0-255 or 0-1). Centering after normalization might be preferred, although it might be worth testing both approaches.

Centering requires that a mean pixel value be calculated prior to subtracting it from the pixel values. There are multiple ways that the mean can be calculated; for example:

- Per image.
- Per minibatch of images (under stochastic gradient descent).

- Per training dataset.

The mean can be calculated for all pixels in the image, referred to as a global centering, or it can be calculated for each channel in the case of color images, referred to as local centering.

- **Global Centering:** Calculating and subtracting the mean pixel value across color channels.
- **Local Centering:** Calculating and subtracting the mean pixel value per color channel.

Per-image global centering is common because it is trivial to implement. Also common is per minibatch global or local centering for the same reason: it is fast and easy to implement. In some cases, per-channel means are pre-calculated across an entire training dataset. In this case, the image means must be stored and used both during training and any inference with the trained models in the future. For models trained on images centered using these means that may be used for transfer learning on new tasks, it can be beneficial or even required to normalize images for the new task using the same means. Let's look at a few examples.

5.4.1 Global Centering

The example below calculates a global mean across all three color channels in the loaded image, then centers the pixel values using the global mean.

```
# example of global centering (subtract mean)
from numpy import asarray
from PIL import Image
# load image
image = Image.open('sydney_bridge.jpg')
pixels = asarray(image)
# convert from integers to floats
pixels = pixels.astype('float32')
# calculate global mean
mean = pixels.mean()
print('Mean: %.3f' % mean)
print('Min: %.3f, Max: %.3f' % (pixels.min(), pixels.max()))
# global centering of pixels
pixels = pixels - mean
# confirm it had the desired effect
mean = pixels.mean()
print('Mean: %.3f' % mean)
print('Min: %.3f, Max: %.3f' % (pixels.min(), pixels.max()))
```

Listing 5.5: Example of global centering pixel values.

Running the example, we can see that the mean pixel value is about 152. Once centered, we can confirm that the new mean for the pixel values is 0.0 and that the new data range is negative and positive around this mean.

```
Mean: 152.149
Min: 0.000, Max: 255.000
Mean: -0.000
Min: -152.149, Max: 102.851
```

Listing 5.6: Example output from global centering pixel values.

5.4.2 Local Centering

The example below calculates the mean for each color channel in the loaded image, then centers the pixel values for each channel separately. Note that NumPy allows us to specify the dimensions over which a statistic like the mean, min, and max are calculated via the `axis` argument. In this example, we set this to `(0,1)` for the width and height dimensions, which leaves the third dimension or channels. The result is one mean, min, or max for each of the three channel arrays.

Also note that when we calculate the mean that we specify the `dtype` as '`float64`'; this is required as it will cause all sub-operations of the mean, such as the sum, to be performed with 64-bit precision. Without this, the sum will be performed at lower resolution and the resulting mean will be wrong given the accumulated errors in the loss of precision, in turn meaning the mean of the centered pixel values for each channel will not be zero (or a very small number close to zero).

```
# example of per-channel centering (subtract mean)
from numpy import asarray
from PIL import Image
# load image
image = Image.open('sydney_bridge.jpg')
pixels = asarray(image)
# convert from integers to floats
pixels = pixels.astype('float32')
# calculate per-channel means and standard deviations
means = pixels.mean(axis=(0,1), dtype='float64')
print('Means: %s' % means)
print('Mins: %s, Maxs: %s' % (pixels.min(axis=(0,1)), pixels.max(axis=(0,1))))
# per-channel centering of pixels
pixels -= means
# confirm it had the desired effect
means = pixels.mean(axis=(0,1), dtype='float64')
print('Means: %s' % means)
print('Mins: %s, Maxs: %s' % (pixels.min(axis=(0,1)), pixels.max(axis=(0,1))))
```

Listing 5.7: Example of local centering pixel values.

Running the example first reports the mean pixels values for each channel, as well as the min and max values for each channel. The pixel values are centered, then the new means and min/max pixel values across each channel are reported. We can see that the new mean pixel values are very small numbers close to zero and the values are negative and positive values centered on zero.

```
Means: [148.61581718 150.64154412 157.18977691]
Mins: [0. 0. 0.], Maxs: [255. 255. 255.]
Means: [1.14413078e-06 1.61369515e-06 1.37722619e-06]
Mins: [-148.61581 -150.64154 -157.18977], Maxs: [106.384186 104.35846 97.81023 ]
```

Listing 5.8: Example output from local centering pixel values.

5.5 Standardize Pixel Values

The distribution of pixel values often follows a Normal or Gaussian distribution, e.g. bell shape. This distribution may be present per image, per minibatch of images, or across the

training dataset and globally or per channel. As such, there may be benefit in transforming the distribution of pixel values to be a standard Gaussian: that is both centering the pixel values on zero and normalizing the values by the standard deviation. The result is a standard Gaussian of pixel values with a mean of 0.0 and a standard deviation of 1.0.

As with centering, the operation can be performed per image, per minibatch, and across the entire training dataset, and it can be performed globally across channels or locally per channel. Standardization may be preferred to normalization and centering alone and it results in both zero-centered values and small input values, roughly in the range -3 to 3, depending on the specifics of the dataset. For consistency of the input data, it may make more sense to standardize images per-channel using statistics calculated per minibatch or across the training dataset, if possible. Let's look at some examples.

5.5.1 Global Standardization

The example below calculates the mean and standard deviations across all color channels in the loaded image, then uses these values to standardize the pixel values.

```
# example of global pixel standardization
from numpy import asarray
from PIL import Image
# load image
image = Image.open('sydney_bridge.jpg')
pixels = asarray(image)
# convert from integers to floats
pixels = pixels.astype('float32')
# calculate global mean and standard deviation
mean, std = pixels.mean(), pixels.std()
print('Mean: %.3f, Standard Deviation: %.3f' % (mean, std))
# global standardization of pixels
pixels = (pixels - mean) / std
# confirm it had the desired effect
mean, std = pixels.mean(), pixels.std()
print('Mean: %.3f, Standard Deviation: %.3f' % (mean, std))
```

Listing 5.9: Example of the global standardizing of pixel values.

Running the example first calculates the global mean and standard deviation pixel values, standardizes the pixel values, then confirms the transform by reporting the new global mean and standard deviation of 0.0 and 1.0 respectively.

```
Mean: 152.149, Standard Deviation: 70.642
Mean: -0.000, Standard Deviation: 1.000
```

Listing 5.10: Example output from the global standardizing of pixel values.

5.5.2 Positive Global Standardization

There may be a desire to maintain the pixel values in the positive domain, perhaps so the images can be visualized or perhaps for the benefit of a chosen activation function in the model. A popular way of achieving this is to clip the standardized pixel values to the range [-1, 1] and then rescale the values from [-1,1] to [0,1]. The example below updates the global standardization example to demonstrate this additional rescaling.

```
# example of global pixel standardization shifted to positive domain
from numpy import asarray
from numpy import clip
from PIL import Image
# load image
image = Image.open('sydney_bridge.jpg')
pixels = asarray(image)
# convert from integers to floats
pixels = pixels.astype('float32')
# calculate global mean and standard deviation
mean, std = pixels.mean(), pixels.std()
print('Mean: %.3f, Standard Deviation: %.3f' % (mean, std))
# global standardization of pixels
pixels = (pixels - mean) / std
# clip pixel values to [-1,1]
pixels = clip(pixels, -1.0, 1.0)
# shift from [-1,1] to [0,1] with 0.5 mean
pixels = (pixels + 1.0) / 2.0
# confirm it had the desired effect
mean, std = pixels.mean(), pixels.std()
print('Mean: %.3f, Standard Deviation: %.3f' % (mean, std))
print('Min: %.3f, Max: %.3f' % (pixels.min(), pixels.max()))
```

Listing 5.11: Example of the positive global standardizing of pixel values.

Running the example first reports the global mean and standard deviation pixel values; the pixels are standardized then rescaled. Next, the new mean and standard deviation are reported of about 0.5 and 0.3 respectively and the new minimum and maximum values are confirmed of 0.0 and 1.0.

```
Mean: 152.149, Standard Deviation: 70.642
Mean: 0.510, Standard Deviation: 0.388
Min: 0.000, Max: 1.000
```

Listing 5.12: Example output from the positive global standardizing of pixel values.

5.5.3 Local Standardization

The example below calculates the mean and standard deviation of the loaded image per-channel, then uses these statistics to standardize the pixels separately in each channel.

```
# example of per-channel pixel standardization
from numpy import asarray
from PIL import Image
# load image
image = Image.open('sydney_bridge.jpg')
pixels = asarray(image)
# convert from integers to floats
pixels = pixels.astype('float32')
# calculate per-channel means and standard deviations
means = pixels.mean(axis=(0,1), dtype='float64')
stds = pixels.std(axis=(0,1), dtype='float64')
print('Means: %s, Stds: %s' % (means, stds))
# per-channel standardization of pixels
```

```

pixels = (pixels - means) / stds
# confirm it had the desired effect
means = pixels.mean(axis=(0,1), dtype='float64')
stds = pixels.std(axis=(0,1), dtype='float64')
print('Means: %s, Stds: %s' % (means, stds))

```

Listing 5.13: Example of the local standardizing of pixel values.

Running the example first calculates and reports the means and standard deviation of the pixel values in each channel. The pixel values are then standardized and statistics are re-calculated, confirming the new zero-mean and unit standard deviation.

```

Means: [148.61581718 150.64154412 157.18977691], Stds: [70.21666738 70.6718887 70.75185228]
Means: [ 6.26286458e-14 -4.40909176e-14 -8.38046276e-13], Stds: [1. 1. 1.]

```

Listing 5.14: Example output from the local standardizing of pixel values.

5.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Develop Function.** Develop a function to scale a provided image, using arguments to choose the type of preparation to perform,
- **Projection Methods.** Investigate and implement data preparation methods that remove linear correlations from the pixel data, such as PCA and ZCA.
- **Dataset Statistics.** Select and update one of the centering or standardization examples to calculate statistics across an entire training dataset, then apply those statistics when preparing image data for training or inference.

If you explore any of these extensions, I'd love to know.

5.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

5.7.1 API

- `numpy.mean` API.
<https://docs.scipy.org/doc/numpy/reference/generated/numpy.mean.html>
- `numpy.mean` along multiple axis gives wrong result for large arrays, Issue.
<https://github.com/numpy/numpy/issues/8869>

5.7.2 Articles

- Data Preprocessing, CS231n Convolutional Neural Networks for Visual Recognition.
<http://cs231n.github.io/neural-networks-2/>
- Why normalize images by subtracting dataset's image mean, instead of the current image mean in deep learning?
<https://goo.gl/vGNexL>
- What are some ways of pre-processing images before applying convolutional neural networks for the task of image classification?
<https://goo.gl/B3EBo2>
- Confused about the image preprocessing in classification, Pytorch Issue.
<https://goo.gl/JNcjPw>

5.8 Summary

In this tutorial, you discovered how to prepare image data for modeling with deep learning neural networks. Specifically, you learned:

- How to normalize pixel values to a range between zero and one.
- How to center pixel values both globally across channels and locally per channel.
- How to standardize pixel values and how to shift standardized pixel values to the positive domain.

5.8.1 Next

In the next section, you will discover how to load and handle image data using the Keras API.

Chapter 6

How to Load and Manipulate Images with Keras

The Keras deep learning library provides a sophisticated API for loading, preparing, and augmenting image data. Also included in the API are some undocumented functions that allow you to quickly and easily load, convert, and save image files. These functions can be convenient when getting started on a computer vision deep learning project, allowing you to use the same Keras API initially to inspect and handle image data and later to model it. In this tutorial, you will discover how to use the basic image handling functions provided by the Keras API. After completing this tutorial, you will know:

- How to load and display an image using the Keras API.
- How to convert a loaded image to a NumPy array and back to PIL format using the Keras API.
- How to convert a loaded image to grayscale and save it to a new file using the Keras API.

Let's get started.

6.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Test Image
2. Keras Image Processing API
3. How to Load an Image With Keras
4. How to Convert an Image With Keras
5. How to Save an Image With Keras

6.2 Test Image

The first step is to select a test image to use in this tutorial. We will use a photograph of Bondi Beach, Sydney, taken by Isabell Schulz¹, released under a permissive creative commons license.



Figure 6.1: Photograph of Bondi Beach, Sydney.

Download the image and place it into your current working directory with the filename `bondi_beach.jpg`.

- Download Photo (`bondi_beach.jpg`).²

6.3 Keras Image Processing API

The Keras deep learning library provides utilities for working with image data. The main API is the `ImageDataGenerator` class that combines data loading, preparation, and augmentation. We will not cover the `ImageDataGenerator` class in this tutorial. Instead, we will take a closer look at a few less-documented or undocumented functions that may be useful when working with image data and modeling with the Keras API. Specifically, Keras provides functions for loading, converting, and saving image data. The functions are in the `utils.py` function and exposed via the `image.py` module. These functions can be useful convenience functions when getting started on a new deep learning computer vision project or when you need to inspect specific images.

¹<https://www.flickr.com/photos/isapisa/45545118405/>

²https://machinelearningmastery.com/wp-content/uploads/2019/01/bondi_beach.jpg

Some of these functions are demonstrated when working with pre-trained models in the Applications section of the API documentation. All image handling in Keras requires that the Pillow library is installed. Let's take a closer look at each of these functions in turn.

6.4 How to Load an Image with Keras

Keras provides the `load_img()` function for loading an image from file as a PIL image object. The example below loads the Bondi Beach photograph from file as a PIL image and reports details about the loaded image.

```
# example of loading an image with the Keras API
from keras.preprocessing.image import load_img
# load the image
img = load_img('bondi_beach.jpg')
# report details about the image
print(type(img))
print(img.format)
print(img.mode)
print(img.size)
# show the image
img.show()
```

Listing 6.1: Example of loading an image with Keras.

Running the example loads the image and reports details about the loaded image. We can confirm that the image was loaded as a PIL image in JPEG format with RGB channels and the size of 640 by 427 pixels.

```
<class 'PIL.JpegImagePlugin.JpegImageFile'>
JPEG
RGB
(640, 427)
```

Listing 6.2: Example output from loading an image with Keras.

The loaded image is then displayed using the default application on the workstation, in this case, the Preview application on macOS.

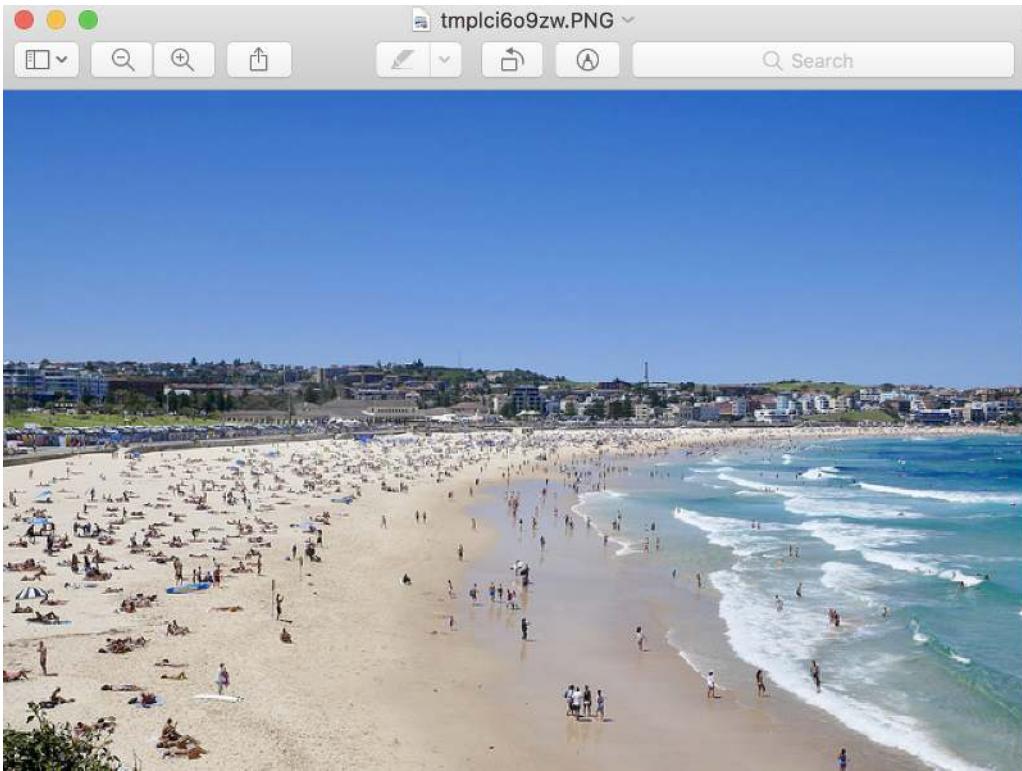


Figure 6.2: Example of Displaying a PIL image using the Default Application.

The `load_img()` function provides additional arguments that may be useful when loading the image, such as ‘`grayscale`’ that allows the image to be loaded in grayscale (defaults to `False`), `color_mode` that allows the image mode or channel format to be specified (defaults to `rgb`), and `target_size` that allows a tuple of (height, width) to be specified, resizing the image automatically after being loaded.

6.5 How to Convert an Image With Keras

Keras provides the `img_to_array()` function for converting a loaded image in PIL format into a NumPy array for use with deep learning models. The API also provides the `array_to_img()` function that can be used for converting a NumPy array of pixel data into a PIL image. This can be useful if the pixel data is modified in array format because it can be saved or viewed. The example below loads the test image, converts it to a NumPy array, and then converts it back into a PIL image.

```
# example of converting an image with the Keras API
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import array_to_img
# load the image
img = load_img('bondi_beach.jpg')
print(type(img))
# convert to numpy array
img_array = img_to_array(img)
print(img_array.dtype)
```

```

print(img_array.shape)
# convert back to image
img_pil = array_to_img(img_array)
print(type(img))

```

Listing 6.3: Example of converting an image to an array with Keras.

Running the example first loads the photograph in PIL format, then converts the image to a NumPy array and reports the data type and shape. We can see that the pixel values are converted from unsigned integers to 32-bit floating point values, and in this case, converted to the array format [height, width, channels]. Finally, the image is converted back into PIL format.

```

<class 'PIL.JpegImagePlugin.JpegImageFile'>
float32
(427, 640, 3)
<class 'PIL.JpegImagePlugin.JpegImageFile'>

```

Listing 6.4: Example output from converting an image to an array with Keras.

6.6 How to Save an Image With Keras

The Keras API also provides the `save_img()` function to save an image to file. The function takes the path to save the image, and the image data in NumPy array format. The file format is inferred from the filename, but can also be specified via the `file_format` argument. This can be useful if you have manipulated image pixel data, such as scaling, and wish to save the image for later use. The example below loads the photograph image in grayscale format, converts it to a NumPy array, and saves it to a new file name.

```

# example of saving an image with the Keras API
from keras.preprocessing.image import load_img
from keras.preprocessing.image import save_img
from keras.preprocessing.image import img_to_array
# load image as as grayscale
img = load_img('bondi_beach.jpg', color_mode='grayscale')
# convert image to a numpy array
img_array = img_to_array(img)
# save the image with a new filename
save_img('bondi_beach_grayscale.jpg', img_array)
# load the image to confirm it was saved correctly
img = load_img('bondi_beach_grayscale.jpg')
print(type(img))
print(img.format)
print(img.mode)
print(img.size)
img.show()

```

Listing 6.5: Example of saving an image with Keras.

Running the example first loads the image and forces the color channels to be grayscale. The image is then converted to a NumPy array and saved to the new filename `bondi_beach_grayscale.jpg` in the current working directory. To confirm that the file was saved correctly, it is loaded again as a PIL image and details of the image are reported.

```
<class 'PIL.Image.Image'>
None
RGB
(640, 427)
```

Listing 6.6: Example output from converting an image to an array with Keras.

The loaded grayscale image is then displayed using the default image preview application on the workstation, which in macOS is the Preview application.



Figure 6.3: Example of Saved Grayscale Image Shown Using the Default Image Viewing Application.

6.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Your Own Image.** Experiment loading and handling your own images using the Keras API.
- **Test Arguments.** Experiment with the arguments of the `load_img` function, including the `interpolation` method and `target_size`.
- **Convenience Function.** Develop a function to make it easier to load an image and convert it to a NumPy array.

If you explore any of these extensions, I'd love to know.

6.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

6.8.1 API

- Pillow Library.
<https://python-pillow.org/>
- Keras Image Preprocessing API.
<https://keras.io/preprocessing/image/>
- Keras Applications API.
<https://keras.io/applications/>
- Keras image.py Source Code.
<https://github.com/keras-team/keras/blob/master/keras/preprocessing/image.py>
- Keras utils.py Source Code.
https://github.com/keras-team/keras-preprocessing/blob/master/keras_preprocessing/image/utils.py

6.9 Summary

In this tutorial, you discovered how to use the basic image handling functions provided by the Keras API. Specifically, you learned:

- How to load and display an image using the Keras API.
- How to convert a loaded image to a NumPy array and back to PIL format using the Keras API.
- How to convert a loaded image to grayscale and save it to a new file using the Keras API.

6.9.1 Next

In the next section, you will discover how to scale image pixel data using the Keras API.

Chapter 7

How to Scale Image Pixel Data with Keras

The pixel values in images must be scaled prior to providing the images as input to a deep learning neural network model during the training or evaluation of the model. Traditionally, the images would have to be scaled prior to the development of the model and stored in memory or on disk in the scaled format. An alternative approach is to scale the images using a preferred scaling technique just-in-time during the training or model evaluation process. Keras supports this type of data preparation for image data via the `ImageDataGenerator` class and API.

In this tutorial, you will discover how to use the `ImageDataGenerator` class to scale pixel data just-in-time when fitting and evaluating deep learning neural network models. After completing this tutorial, you will know:

- How to configure and use the `ImageDataGenerator` class for train, validation, and test datasets of images.
- How to use the `ImageDataGenerator` to normalize pixel values for a train and test image dataset.
- How to use the `ImageDataGenerator` to center and standardize pixel values for a train and test image dataset.

Let's get started.

7.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. MNIST Handwritten Image Classification Dataset.
2. `ImageDataGenerator` Class for Pixel Scaling
3. How to Normalize Images With `ImageDataGenerator`
4. How to Center Images With `ImageDataGenerator`
5. How to Standardize Images With `ImageDataGenerator`

7.2 MNIST Handwritten Image Classification Dataset

Before we dive into the usage of the `ImageDataGenerator` class for preparing image data, we must select an image dataset on which to test the generator. The MNIST problem, is an image classification problem comprised of 70,000 images of handwritten digits. The goal of the problem is to classify a given image of a handwritten digit as an integer from 0 to 9. As such, it is a multiclass image classification problem.

This dataset is provided as part of the Keras library and can be automatically downloaded (if needed) and loaded into memory by a call to the `keras.datasets.mnist.load_data()` function. The function returns two tuples: one for the training inputs and outputs and one for the test inputs and outputs. Note, if this is the first time using the MNIST dataset in Keras, the dataset will be downloaded and placed in the `.keras/datasets/` directory in your home directory. The dataset is only 11 megabytes and will download very quickly.

```
# example of loading the MNIST dataset
from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
...
```

Listing 7.1: Example of loading the MNIST dataset.

We can load the MNIST dataset and summarize the dataset. The complete example is listed below.

```
# load and summarize the MNIST dataset
from keras.datasets import mnist
# load dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
# summarize dataset shape
print('Train', train_images.shape, train_labels.shape)
print('Test', (test_images.shape, test_labels.shape))
# summarize pixel values
print('Train', train_images.min(), train_images.max(), train_images.mean(),
      train_images.std())
print('Test', test_images.min(), test_images.max(), test_images.mean(), test_images.std())
```

Listing 7.2: Example of loading and summarizing the MNIST dataset.

Running the example first loads the dataset into memory. Then the shape of the train and test datasets is reported. We can see that all images are 28 by 28 pixels with a single channel for black-and-white images. There are 60,000 images for the training dataset and 10,000 for the test dataset. We can also see that pixel values are integer values between 0 and 255 and that the mean and standard deviation of the pixel values are similar between the two datasets.

```
Train (60000, 28, 28) (60000,)
Test ((10000, 28, 28), (10000,))
Train 0 255 33.318421449829934 78.56748998339798
Test 0 255 33.791224489795916 79.17246322228644
```

Listing 7.3: Example output from loading and summarizing the MNIST dataset.

We will use this dataset to explore different pixel scaling methods using the `ImageDataGenerator` class in Keras.

7.3 ImageDataGenerator Class for Pixel Scaling

The `ImageDataGenerator` class in Keras provides a suite of techniques for scaling pixel values in your image dataset prior to modeling. The class will wrap your image dataset, then when requested, it will return images in batches to the algorithm during training, validation, or evaluation and apply the scaling operations just-in-time. This provides an efficient and convenient approach to scaling image data when modeling with neural networks.

The usage of the `ImageDataGenerator` class is as follows.

1. Load your dataset.
2. Configure the `ImageDataGenerator` (e.g. construct an instance).
3. Calculate image statistics (e.g. call the `fit()` function).
4. Use the generator to fit the model (e.g. pass the instance to the `fit_generator()` function).
5. Use the generator to evaluate the model (e.g. pass the instance to the `evaluate_generator()` function).

The `ImageDataGenerator` class supports a number of pixel scaling methods, as well as a range of data augmentation techniques. We will focus on the pixel scaling techniques in this chapter and leave the data augmentation methods to a later discussion (Chapter 9). The three main types of pixel scaling techniques supported by the `ImageDataGenerator` class are as follows:

- **Pixel Normalization:** scale pixel values to the range 0-1.
- **Pixel Centering:** scale pixel values to have a zero mean.
- **Pixel Standardization:** scale pixel values to have a zero mean and unit variance.

Pixel standardization is supported at two levels: either per-image (called sample-wise) or per-dataset (called feature-wise). Specifically, just the mean, or the mean and standard deviation statistics required to standardize pixel values can be calculated from the pixel values in each image only (sample-wise) or across the entire training dataset (feature-wise). Other pixel scaling methods are supported, such as ZCA, brightening, and more, but we will focus on these three most common methods. The choice of pixel scaling is selected by specifying arguments to the `ImageDataGenerator` class when an instance is constructed; for example:

```
...
# create data generator
datagen = ImageDataGenerator()
```

Listing 7.4: Example of creating an image data generator.

Next, if the chosen scaling method requires that statistics be calculated across the training dataset, then these statistics can be calculated and stored by calling the `fit()` function. When evaluating and selecting a model, it is common to calculate these statistics on the training dataset and then apply them to the validation and test datasets.

```
...
# calculate scaling statistics on the training dataset
datagen.fit(trainX)
```

Listing 7.5: Example of fitting a model directly on the training dataset.

Once prepared, the data generator can be used to fit a neural network model by calling the `flow()` function to retrieve an iterator that returns batches of samples and passing it to the `fit_generator()` function.

```
...
# get batch iterator
train_iterator = datagen.flow(trainX, trainy)
# fit model
model.fit_generator(train_iterator, ...)
```

Listing 7.6: Example of creating a data iterator and fitting a model with it.

If a validation dataset is required, a separate batch iterator can be created from the same data generator that will perform the same pixel scaling operations and use any required statistics calculated on the training dataset.

```
...
# get batch iterator for training
train_iterator = datagen.flow(trainX, trainy)
# get batch iterator for validation
val_iterator = datagen.flow(valX, valy)
# fit model
model.fit_generator(train_iterator, validation_data=val_iterator, ...)
```

Listing 7.7: Example of creating a data iterator for train and validation datasets.

Once fit, the model can be evaluated by creating a batch iterator for the test dataset and calling the `evaluate_generator()` function on the model. Again, the same pixel scaling operations will be performed and any statistics calculated on the training dataset will be used, if needed.

```
...
# get batch iterator for testing
test_iterator = datagen.flow(testX, testy)
# evaluate model loss on test dataset
loss = model.evaluate_generator(test_iterator, ...)
```

Listing 7.8: Example of evaluating a model with a test data iterator.

Now that we are familiar with how to use the `ImageDataGenerator` class for scaling pixel values, let's look at some specific examples.

7.4 How to Normalize Images With `ImageDataGenerator`

The `ImageDataGenerator` class can be used to rescale pixel values from the range of 0-255 to the range 0-1 preferred for neural network models. Scaling data to the range of 0-1 is traditionally referred to as normalization. This can be achieved by setting the `rescale` argument to a ratio by which each pixel can be multiplied to achieve the desired range. In this case, the ratio is $\frac{1}{255}$ or about 0.0039. For example:

```
...
# create generator (1.0/255.0 = 0.003921568627451)
datagen = ImageDataGenerator(rescale=1.0/255.0)
```

Listing 7.9: Example of creating an image data generator for normalization.

The `ImageDataGenerator` does not need to be fit in this case because there are no global statistics like mean and standard deviation that need to be calculated. Next, iterators can be created using the generator for both the train and test datasets. We will use a batch size of 64. This means that each of the train and test datasets of images are divided into groups of 64 images that will then be scaled when returned from the iterator. We can see how many batches there will be in one epoch, e.g. one pass through the training dataset, by printing the length of each iterator.

```
...
# prepare an iterators to scale images
train_iterator = datagen.flow(trainX, trainY, batch_size=64)
test_iterator = datagen.flow(testX, testY, batch_size=64)
print('Batches train=%d, test=%d' % (len(train_iterator), len(test_iterator)))
```

Listing 7.10: Example of creating data iterators for normalization.

We can then confirm that the pixel normalization has been performed as expected by retrieving the first batch of scaled images and inspecting the min and max pixel values.

```
...
# confirm the scaling works
batchX, batchy = train_iterator.next()
print('Batch shape=%s, min=%f, max=%f' % (batchX.shape, batchX.min(), batchX.max()))
```

Listing 7.11: Example of getting one batch of data with the data iterator.

We can tie all of this together; the complete example is listed below.

```
# example of normalizing a image dataset
from keras.datasets import mnist
from keras.preprocessing.image import ImageDataGenerator
# load dataset
(trainX, trainY), (testX, testY) = mnist.load_data()
# reshape dataset to have a single channel
width, height, channels = trainX.shape[1], trainX.shape[2], 1
trainX = trainX.reshape((trainX.shape[0], width, height, channels))
testX = testX.reshape((testX.shape[0], width, height, channels))
# confirm scale of pixels
print('Train min=%f, max=%f' % (trainX.min(), trainX.max()))
print('Test min=%f, max=%f' % (testX.min(), testX.max()))
# create generator (1.0/255.0 = 0.003921568627451)
datagen = ImageDataGenerator(rescale=1.0/255.0)
# Note: there is no need to fit the generator in this case
# prepare a iterators to scale images
train_iterator = datagen.flow(trainX, trainY, batch_size=64)
test_iterator = datagen.flow(testX, testY, batch_size=64)
print('Batches train=%d, test=%d' % (len(train_iterator), len(test_iterator)))
# confirm the scaling works
batchX, batchy = train_iterator.next()
print('Batch shape=%s, min=%f, max=%f' % (batchX.shape, batchX.min(), batchX.max()))
```

Listing 7.12: Example normalizing pixel values on the MNIST dataset.

Running the example first reports the minimum and maximum pixel value for the train and test datasets. The MNIST dataset only has a single channel because the images are black and white (grayscale), but if the images were color, the min and max pixel values would be calculated across all channels in all images in the training dataset, i.e. there would not be a separate mean value for each channel.

The `ImageDataGenerator` does not need to be fit on the training dataset as there is nothing that needs to be calculated, we have provided the scale factor directly. A single batch of normalized images is retrieved and we can confirm that the min and max pixel values are zero and one respectively.

```
Using TensorFlow backend.
Train min=0.000, max=255.000
Test min=0.000, max=255.000
Batches train=938, test=157
Batch shape=(64, 28, 28, 1), min=0.000, max=1.000
```

Listing 7.13: Example output from normalizing pixel values on the MNIST dataset.

7.5 How to Center Images With *ImageDataGenerator*

Another popular pixel scaling method is to calculate the mean pixel value across the entire training dataset, then subtract it from each image. This is called centering and has the effect of centering the distribution of pixel values on zero: that is, the mean pixel value for centered images will be zero. The `ImageDataGenerator` class refers to centering that uses the mean calculated on the training dataset as feature-wise centering. It requires that the statistic is calculated on the training dataset prior to scaling.

```
...
# create generator that centers pixel values
datagen = ImageDataGenerator(featurewise_center=True)
# calculate the mean on the training dataset
datagen.fit(trainX)
```

Listing 7.14: Example of creating an image data generator for feature-wise centering.

It is different to calculating of the mean pixel value for each image, which Keras refers to as sample-wise centering and does not require any statistics to be calculated on the training dataset.

```
...
# create generator that centers pixel values
datagen = ImageDataGenerator(samplewise_center=True)
# calculate the mean on the training dataset
datagen.fit(trainX)
```

Listing 7.15: Example of creating an image data generator for sample-wise centering.

We will demonstrate feature-wise centering in this section. Once the statistic is calculated on the training dataset, we can confirm the value by accessing and printing it; for example:

```
...
# print the mean calculated on the training dataset.
print(datagen.mean)
```

Listing 7.16: Example of summarizing mean pixel values calculated on the training dataset.

We can also confirm that the scaling procedure has had the desired effect by calculating the mean of a batch of images returned from the batch iterator. We would expect the mean to be a small value close to zero, but not zero because of the small number of images in the batch.

```
...
# get a batch
batchX, batchy = iterator.next()
# mean pixel value in the batch
print(batchX.shape, batchX.mean())
```

Listing 7.17: Example of getting one batch of centered images.

A better check would be to set the batch size to the size of the training dataset (e.g. 60,000 samples), retrieve one batch, then calculate the mean. It should be a very small value close to zero.

```
...
# try to flow the entire training dataset
iterator = datagen.flow(trainX, trainy, batch_size=len(trainX), shuffle=False)
# get a batch
batchX, batchy = iterator.next()
# mean pixel value in the batch
print(batchX.shape, batchX.mean())
```

Listing 7.18: Example of checking the mean pixel values of one batch of images.

```
# example of centering a image dataset
from keras.datasets import mnist
from keras.preprocessing.image import ImageDataGenerator
# load dataset
(trainX, trainy), (testX, testy) = mnist.load_data()
# reshape dataset to have a single channel
width, height, channels = trainX.shape[1], trainX.shape[2], 1
trainX = trainX.reshape((trainX.shape[0], width, height, channels))
testX = testX.reshape((testX.shape[0], width, height, channels))
# report per-image mean
print('Means train=% .3f, test=% .3f' % (trainX.mean(), testX.mean()))
# create generator that centers pixel values
datagen = ImageDataGenerator(featurewise_center=True)
# calculate the mean on the training dataset
datagen.fit(trainX)
print('Data Generator Mean: % .3f' % datagen.mean())
# demonstrate effect on a single batch of samples
iterator = datagen.flow(trainX, trainy, batch_size=64)
# get a batch
batchX, batchy = iterator.next()
# mean pixel value in the batch
print(batchX.shape, batchX.mean())
# demonstrate effect on entire training dataset
iterator = datagen.flow(trainX, trainy, batch_size=len(trainX), shuffle=False)
```

```
# get a batch
batchX, batchy = iterator.next()
# mean pixel value in the batch
print(batchX.shape, batchX.mean())
```

Listing 7.19: Example feature-wise centering on the MNIST dataset.

Running the example first reports the mean pixel value for the train and test datasets. The MNIST dataset only has a single channel because the images are black and white (grayscale), but if the images were color, the mean pixel values would be calculated across all channels in all images in the training dataset, i.e. there would not be a separate mean value for each channel. The *ImageDataGenerator* is fit on the training dataset and we can confirm that the mean pixel value matches our own manual calculation. A single batch of centered images is retrieved and we can confirm that the mean pixel value is a small-ish value close to zero. The test is repeated using the entire training dataset as the batch size, and in this case, the mean pixel value for the scaled dataset is a number very close to zero, confirming that centering is having the desired effect. **Note:** your statistics may vary slightly due to a different random batch of images being selected.

```
Means train=33.318, test=33.791
Data Generator Mean: 33.318
(64, 28, 28, 1) 0.09971977
(60000, 28, 28, 1) -1.9512918e-05
```

Listing 7.20: Example output from feature-wise centering on the MNIST dataset.

7.6 How to Standardize Images With *ImageDataGenerator*

Standardization is a data scaling technique that assumes that the distribution of the data is Gaussian and shifts the distribution of the data to have a mean of zero and a standard deviation of one. Data with this distribution is referred to as a standard Gaussian. It can be beneficial when training neural networks as the dataset sums to zero and the inputs are small values in the rough range of about -3.0 to 3.0 (e.g. 99.7 of the values will fall within three standard deviations of the mean). Standardization of images is achieved by subtracting the mean pixel value and dividing the result by the standard deviation of the pixel values. The mean and standard deviation statistics can be calculated on the training dataset, and as discussed in the previous section, Keras refers to this as feature-wise.

```
...
# feature-wise generator
datagen = ImageDataGenerator(featurewise_center=True, featurewise_std_normalization=True)
# calculate mean and standard deviation on the training dataset
datagen.fit(trainX)
```

Listing 7.21: Example of creating an image data generator for feature-wise standardization.

The statistics can also be calculated then used to standardize each image separately, and Keras refers to this as sample-wise standardization.

```
...
# sample-wise standardization
datagen = ImageDataGenerator(samplewise_center=True, samplewise_std_normalization=True)
```

```
# calculate mean and standard deviation on the training dataset
datagen.fit(trainX)
```

Listing 7.22: Example of creating an image data generator for sample-wise standardization.

We will demonstrate the feature-wise approach to image standardization in this section. The effect will be batches of images with an approximate mean of zero and a standard deviation of one. As with the previous section, we can confirm this with some simple experiments. The complete example is listed below.

```
# example of standardizing a image dataset
from keras.datasets import mnist
from keras.preprocessing.image import ImageDataGenerator
# load dataset
(trainX, trainy), (testX, testy) = mnist.load_data()
# reshape dataset to have a single channel
width, height, channels = trainX.shape[1], trainX.shape[2], 1
trainX = trainX.reshape((trainX.shape[0], width, height, channels))
testX = testX.reshape((testX.shape[0], width, height, channels))
# report pixel means and standard deviations
print('Statistics train=% .3f (% .3f), test=% .3f (% .3f)' % (trainX.mean(), trainX.std(),
    testX.mean(), testX.std()))
# create generator that centers pixel values
datagen = ImageDataGenerator(featurewise_center=True, featurewise_std_normalization=True)
# calculate the mean on the training dataset
datagen.fit(trainX)
print('Data Generator mean=% .3f, std=% .3f' % (datagen.mean, datagen.std))
# demonstrate effect on a single batch of samples
iterator = datagen.flow(trainX, trainy, batch_size=64)
# get a batch
batchX, batchy = iterator.next()
# pixel stats in the batch
print(batchX.shape, batchX.mean(), batchX.std())
# demonstrate effect on entire training dataset
iterator = datagen.flow(trainX, trainy, batch_size=len(trainX), shuffle=False)
# get a batch
batchX, batchy = iterator.next()
# pixel stats in the batch
print(batchX.shape, batchX.mean(), batchX.std())
```

Listing 7.23: Example feature-wise standardization on the MNIST dataset.

Running the example first reports the mean and standard deviation of pixel values in the train and test datasets. The data generator is then configured for feature-wise standardization and the statistics are calculated on the training dataset, matching what we would expect when the statistics are calculated manually. A single batch of 64 standardized images is then retrieved and we can confirm that the mean and standard deviation of this small sample is close to the expected standard Gaussian. The test is then repeated on the entire training dataset and we can confirm that the mean is indeed a very small value close to 0.0 and the standard deviation is a value very close to 1.0. **Note:** your statistics may vary slightly due to a different random batch of images being selected.

```
Statistics train=33.318 (78.567), test=33.791 (79.172)
Data Generator mean=33.318, std=78.567
(64, 28, 28, 1) 0.010656365 1.0107679
```

```
(60000, 28, 28, 1) -3.4560264e-07 0.9999998
```

Listing 7.24: Example output from feature-wise standardization on the MNIST dataset.

7.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Color.** Update an example to use an image dataset with color images and confirm that scaling is performed across the entire image rather than per-channel.
- **Sample-Wise.** Demonstrate an example of sample-wise centering or standardization of pixel images.
- **ZCA Whitening.** Demonstrate an example of using the ZCA approach to image data preparation.

If you explore any of these extensions, I'd love to know.

7.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

7.8.1 API

- MNIST database of handwritten digits, Keras API.
<https://keras.io/datasets/#mnist-database-of-handwritten-digits>
- Image Preprocessing Keras API.
<https://keras.io/preprocessing/image/>
- Sequential Model Keras API.
<https://keras.io/models/sequential/>

7.8.2 Articles

- MNIST database, Wikipedia.
https://en.wikipedia.org/wiki/MNIST_database
- 68-95-99.7 rule, Wikipedia.
https://en.wikipedia.org/wiki/68%2595%2599.7_rule

7.9 Summary

In this tutorial, you discovered how to use the `ImageDataGenerator` class to scale pixel data just-in-time when fitting and evaluating deep learning neural network models. Specifically, you learned:

- How to configure and use the `ImageDataGenerator` class for train, validation, and test datasets of images.
- How to use the `ImageDataGenerator` to normalize pixel values for a train and test image dataset.
- How to use the `ImageDataGenerator` to center and standardize pixel values for a train and test image dataset.

7.9.1 Next

In the next section, you will discover how to load large image datasets from directories and files.

Chapter 8

How to Load Large Datasets From Directories with Keras

There are conventions for storing and structuring your image dataset on disk in order to make it fast and efficient to load and when training and evaluating deep learning models. Once structured, you can use tools like the `ImageDataGenerator` class in the Keras deep learning library to automatically load your train, test, and validation datasets. In addition, the generator will progressively load the images in your dataset (e.g. just-in-time), allowing you to work with both small and very large datasets containing thousands or millions of images that may not fit into system memory. In this tutorial, you will discover how to structure an image dataset and how to load it progressively when fitting and evaluating a deep learning model. After completing this tutorial, you will know:

- How to organize train, validation, and test image datasets into a consistent directory structure.
- How to use the `ImageDataGenerator` class to progressively load the images for a given dataset.
- How to use a prepared data generator to train, evaluate, and make predictions with a deep learning model.

Let's get started.

8.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Dataset Directory Structure
2. Example Dataset Structure
3. How to Progressively Load Images

8.2 Dataset Directory Structure

There is a standard way to lay out your image data for modeling. After you have collected your images, you must sort them first by dataset, such as train, test, and validation, and second by their class. For example, imagine an image classification problem where we wish to classify photos of cars based on their color, e.g. red cars, blue cars, etc. First, we have a `data/` directory where we will store all of the image data. Next, we will have a `data/train/` directory for the training dataset and a `data/test/` for the holdout test dataset. We may also have a `data/validation/` for a validation dataset during training. So far, we have:

```
data/
data/train/
data/test/
data/validation/
```

Listing 8.1: Example directory structure for image datasets.

Under each of the dataset directories, we will have subdirectories, one for each class where the actual image files will be placed. For example, if we have a binary classification task for classifying photos of cars as either a red car or a blue car, we would have two classes, `red` and `blue`, and therefore two class directories under each dataset directory. For example:

```
data/
data/train/
data/train/red/
data/train/blue/
data/test/
data/test/red/
data/test/blue/
data/validation/
data/validation/red/
data/validation/blue/
```

Listing 8.2: Example directory structure with classes for image datasets.

Images of red cars would then be placed in the appropriate class directory.

For example:

```
data/train/red/car01.jpg
data/train/red/car02.jpg
data/train/red/car03.jpg
...
data/train/blue/car01.jpg
data/train/blue/car02.jpg
data/train/blue/car03.jpg
...
```

Listing 8.3: Example directory structure with classes for image files.

Remember, we are not placing the same files under the `red/` and `blue/` directories; instead, there are different photos of red cars and blue cars respectively. Also recall that we require different photos in the train, test, and validation datasets. The filenames used for the actual images often do not matter as we will load all images with given file extensions. A good naming convention, if you have the ability to rename files consistently, is to use some name followed by a number with zero padding, e.g. `image0001.jpg` if you have thousands of images for a class.

8.3 Example Dataset Structure

We can make the image dataset structure concrete with an example. Imagine we are classifying photographs of cars, as we discussed in the previous section. Specifically, a binary classification problem with red cars and blue cars. We must create the directory structure outlined in the previous section, specifically:

```
data/
data/train/
data/train/red/
data/train/blue/
data/test/
data/test/red/
data/test/blue/
data/validation/
data/validation/red/
data/validation/blue/
```

Listing 8.4: Example of proposed directory structure for the image dataset.

Let's actually create these directories. We can also put some photos in the directories. To keep things simple, we will use two photographs of cars released under a permissive license. Specifically, a photo of a red car by Dennis Jarvis¹, and a photo of a blue car by Bill Smith². Download both photos to your current working director with the filenames `red_car_01.jpg` and `blue_car_01.jpg` respectively.

- Download Red Car Photo (`red_car_01.jpg`).³
- Download Blue Car Photo (`blue_car_01.jpg`).⁴



Figure 8.1: Photograph of a Red Car.

¹<https://www.flickr.com/photos/archer10/7296236992/>

²<https://www.flickr.com/photos/byzantiumbooks/25158850275/>

³https://machinelearningmastery.com/wp-content/uploads/2019/01/red_car_01.jpg

⁴https://machinelearningmastery.com/wp-content/uploads/2019/01/blue_car_01.jpg



Figure 8.2: Photograph of a Blue Car.

We must have different photos for each of the train, test, and validation datasets. In the interest of keeping this tutorial focused, we will re-use the same image files in each of the three datasets but pretend they are different photographs. Place copies of the `red_car_01.jpg` file in `data/train/red/`, `data/test/red/`, and `data/validation/red/` directories. Now place copies of the `blue_car_01.jpg` file in `data/train/blue/`, `data/test/blue/`, and `data/validation/blue/` directories. We now have a very basic dataset layout that looks like the following (output from the `tree` command):

```
data
└── test
    ├── blue
    │   └── blue_car_01.jpg
    └── red
        └── red_car_01.jpg
└── train
    ├── blue
    │   └── blue_car_01.jpg
    └── red
        └── red_car_01.jpg
└── validation
    ├── blue
    │   └── blue_car_01.jpg
    └── red
        └── red_car_01.jpg
```

Listing 8.5: Tree structure of the proposed dataset.

Below is a screenshot of the directory structure, taken from the Finder window on macOS.

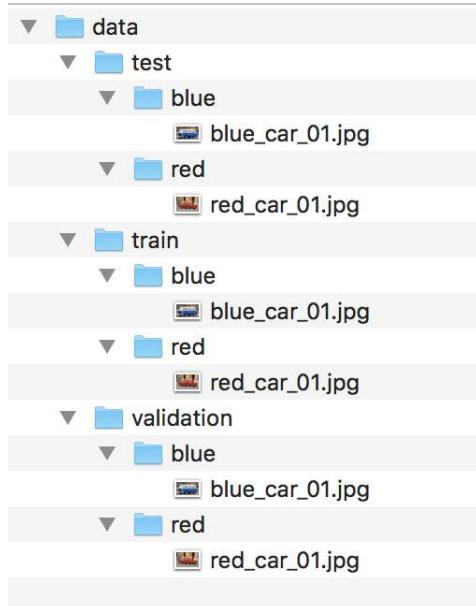


Figure 8.3: Screenshot of Image Dataset Directory and File Structure.

Now that we have a basic directory structure, let's practice loading image data from file for use with modeling.

8.4 How to Progressively Load Images

It is possible to write code to manually load image data and return data ready for modeling. This would include walking the directory structure for a dataset, loading image data, and returning the input (pixel arrays) and output (class integer). Thankfully, we don't need to write this code. Instead, we can use the `ImageDataGenerator` class provided by Keras. The main benefit of using this class to load the data is that images are loaded for a single dataset in batches, meaning that it can be used for loading both small datasets as well as very large image datasets with thousands or millions of images.

Instead of loading all images into memory, it will load just enough images into memory for the current and perhaps the next few mini-batches when training and evaluating a deep learning model. I refer to this as progressive loading (or lazy loading), as the dataset is progressively loaded from file, retrieving just enough data for what is needed immediately. Two additional benefits of the using the `ImageDataGenerator` class is that it can also automatically scale pixel values of images and it can automatically generate augmented versions of images. We will leave these topics for discussion in another tutorial (see Chapter 9) and instead focus on how to use the `ImageDataGenerator` class to load image data from file. The pattern for using the `ImageDataGenerator` class is used as follows:

1. Construct and configure an instance of the `ImageDataGenerator` class.
2. Retrieve an iterator by calling the `flow_from_directory()` function.
3. Use the iterator in the training or evaluation of a model.

Let's take a closer look at each step. The constructor for the `ImageDataGenerator` contains many arguments to specify how to manipulate the image data after it is loaded, including pixel scaling and data augmentation. We do not need any of these features at this stage, so configuring the `ImageDataGenerator` is easy.

```
...
# create a data generator
datagen = ImageDataGenerator()
```

Listing 8.6: Example of creating an image data generator.

Next, an iterator is required to progressively load images for a single dataset. This requires calling the `flow_from_directory()` function and specifying the dataset directory, such as the train, test, or validation directory. The function also allows you to configure more details related to the loading of images. Of note is the `target_size` argument that allows you to load all images to a specific size, which is often required when modeling. The function defaults to square images with the size (256, 256).

The function also allows you to specify the type of classification task via the `class_mode` argument, specifically whether it is ‘binary’ or a multiclass classification ‘categorical’. The default `batch_size` is 32, which means that 32 randomly selected images from across the classes in the dataset will be returned in each batch when training. Larger or smaller batches may be desired. You may also want to return batches in a deterministic order when evaluating a model, which you can do by setting `shuffle` to `False`. The subdirectories of images, one for each class, are loaded by the `flow_from_directory()` function in alphabetical order and assigned an integer for each class. For example, the subdirectory `blue` comes before `red` alphabetically, therefore the class labels are assigned the integers: `blue=0, red=1`. This can be changed via the `classes` argument in calling `flow_from_directory()` when training the model.

There are many other options, and I encourage you to review the API documentation. We can use the same `ImageDataGenerator` to prepare separate iterators for separate dataset directories. This is useful if we would like the same pixel scaling applied to multiple datasets (e.g. train, test, etc.).

```
...
# load and iterate training dataset
train_it = datagen.flow_from_directory('data/train/', class_mode='binary', batch_size=64)
# load and iterate validation dataset
val_it = datagen.flow_from_directory('data/validation/', class_mode='binary', batch_size=64)
# load and iterate test dataset
test_it = datagen.flow_from_directory('data/test/', class_mode='binary', batch_size=64)
```

Listing 8.7: Example of creating dataset iterators from an image data generator.

Once the iterators have been prepared, we can use them when fitting and evaluating a deep learning model. For example, fitting a model with a data generator can be achieved by calling the `fit_generator()` function on the model and passing the training iterator (`train_it`). The validation iterator (`val_it`) can be specified when calling this function via the `validation_data` argument. The `steps_per_epoch` argument must be specified for the training iterator in order to define how many batches of images defines a single epoch.

For example, if you have 1,000 images in the training dataset (across all classes) and a batch size of 64, then the `steps_per_epoch` would be about 16, or $\frac{1000}{64}$. Similarly, if a validation iterator is applied, then the `validation_steps` argument must also be specified to indicate the number of batches in the validation dataset defining one epoch.

```

...
# define model
model = ...
# fit model
model.fit_generator(train_it, steps_per_epoch=16, validation_data=val_it,
validation_steps=8)

```

Listing 8.8: Example of fitting a model with a data iterator.

Once the model is fit, it can be evaluated on a test dataset using the `evaluate_generator()` function and passing in the test iterator (`test_it`). The `steps` argument defines the number of batches of samples to step through when evaluating the model before stopping.

```

...
# evaluate model
loss = model.evaluate_generator(test_it, steps=24)

```

Listing 8.9: Example of evaluating a model with a data iterator.

Finally, if you want to use your fit model for making predictions on a very large dataset, you can create an iterator for that dataset as well (e.g. `predict_it`) and call the `predict_generator()` function on the model.

```

...
# make a prediction
yhat = model.predict_generator(predict_it, steps=24)

```

Listing 8.10: Example of making a prediction with a model using a data iterator.

Let's use our small dataset defined in the previous section to demonstrate how to define an `ImageDataGenerator` instance and prepare the dataset iterators. A complete example is listed below.

```

# example of progressively loading images from file
from keras.preprocessing.image import ImageDataGenerator
# create generator
datagen = ImageDataGenerator()
# prepare an iterators for each dataset
train_it = datagen.flow_from_directory('data/train/', class_mode='binary')
val_it = datagen.flow_from_directory('data/validation/', class_mode='binary')
test_it = datagen.flow_from_directory('data/test/', class_mode='binary')
# confirm the iterator works
batchX, batchy = train_it.next()
print('Batch shape=%s, min=%f, max=%f' % (batchX.shape, batchX.min(), batchX.max()))

```

Listing 8.11: Example of loading a dataset from directory with Keras.

Running the example first creates an instance of the `ImageDataGenerator` with all of the default configuration. Next, three iterators are created, one for each of the train, validation, and test binary classification datasets. As each iterator is created, we can see debug messages reporting the number of images and classes discovered and prepared. Finally, we test out the train iterator that would be used to fit a model. The first batch of images is retrieved and we can confirm that the batch contains two images, as only two images were available. We can also confirm that the images were loaded and forced to the square dimensions of 256 rows and 256 columns of pixels and the pixel data was not scaled and remains in the range [0, 255].

```
Found 2 images belonging to 2 classes.  
Found 2 images belonging to 2 classes.  
Found 2 images belonging to 2 classes.  
Batch shape=(2, 256, 256, 3), min=0.000, max=255.000
```

Listing 8.12: Example output loading a dataset from directory with Keras.

8.5 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Your Own Images.** Design a small dataset with just one or a few images in each class and practice loading it using the Keras API.
- **Iterate and Plot.** Experiment with the image dataset iterator by retrieving and plotting batches of images from a dataset.
- **Apply to Dataset.** Locate a small image dataset and organize into appropriate directories and load and iterate over it using the Keras API

If you explore any of these extensions, I'd love to know.

8.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

8.6.1 API

- Image Preprocessing Keras API.
<https://keras.io/preprocessing/image/>
- Sequential Model API.
<https://keras.io/models/sequential/>

8.6.2 Articles

- Building powerful image classification models using very little data, Keras Blog.
<https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html>

8.7 Summary

In this tutorial, you discovered how to structure an image dataset and how to load it progressively when fitting and evaluating a deep learning model. Specifically, you learned:

- How to organize train, validation, and test image datasets into a consistent directory structure.

- How to use the `ImageDataGenerator` class to progressively load the images for a given dataset.
- How to use a prepared data generator to train, evaluate, and make predictions with a deep learning model.

8.7.1 Next

In the next section, you will discover how to use image data augmentation with the Keras API.

Chapter 9

How to Use Image Data Augmentation in Keras

Image data augmentation is a technique that can be used to artificially expand the size of a training dataset by creating modified versions of images in the dataset. Training deep learning neural network models on more data can result in more skillful models, and the augmentation techniques can create variations of the images that can improve the ability of the fit models to generalize what they have learned to new images. The Keras deep learning neural network library provides the capability to fit models using image data augmentation via the `ImageDataGenerator` class. In this tutorial, you will discover how to use image data augmentation when training deep learning neural networks.

After completing this tutorial, you will know:

- Image data augmentation is used to expand the training dataset in order to improve the performance and ability of the model to generalize.
- Image data augmentation is supported in the Keras deep learning library via the `ImageDataGenerator` class.
- How to use shift, flip, brightness, and zoom image data augmentation.

Let's get started.

9.1 Tutorial Overview

This tutorial is divided into eight parts; they are:

1. Image Data Augmentation
2. Sample Image
3. Image Augmentation With `ImageDataGenerator`
4. Horizontal and Vertical Shift Augmentation
5. Horizontal and Vertical Flip Augmentation

6. Random Rotation Augmentation
7. Random Brightness Augmentation
8. Random Zoom Augmentation

9.2 Image Data Augmentation

The performance of deep learning neural networks often improves with the amount of data available. Data augmentation is a technique to artificially create new training data from existing training data. This is done by applying domain-specific techniques to examples from the training data that create new and different training examples. Image data augmentation is perhaps the most well-known type of data augmentation and involves creating transformed versions of images in the training dataset that belong to the same class as the original image. Transforms include a range of operations from the field of image manipulation, such as shifts, flips, zooms, and much more.

The intent is to expand the training dataset with new plausible examples. This means, variations of the training set images that are likely to be seen by the model. For example, a horizontal flip of a picture of a cat may make sense, because the photo could have been taken from the left or right. A vertical flip of the photo of a cat does not make sense and would probably not be appropriate given that the model is very unlikely to see a photo of an upside down cat. As such, it is clear that the choice of the specific data augmentation techniques used for a training dataset must be chosen carefully and within the context of the training dataset and knowledge of the problem domain. In addition, it can be useful to experiment with data augmentation methods in isolation and in concert to see if they result in a measurable improvement to model performance, perhaps with a small prototype dataset, model, and training run.

Modern deep learning algorithms, such as the convolutional neural network, or CNN, can learn features that are invariant to their location in the image. Nevertheless, augmentation can further aid in this transform-invariant approach to learning and can aid the model in learning features that are also invariant to transforms such as left-to-right to top-to-bottom ordering, light levels in photographs, and more. Image data augmentation is typically only applied to the training dataset, and not to the validation or test dataset. This is different from data preparation such as image resizing and pixel scaling; they must be performed consistently across all datasets that interact with the model.

9.3 Sample Image

We need a sample image to demonstrate standard data augmentation techniques. In this tutorial, we will use a photograph of a bird by *AndYaDontStop*¹, released under a permissive license. Download the image and save it in your current working directory with the filename `bird.jpg`.

¹<https://www.flickr.com/photos/thenovys/3854468621/>



Figure 9.1: Photograph of a Bird.

Download the image and place it into your current working directory with the filename `bird.jpg`.

- Download Photo (`bird.jpg`).²

9.4 Image Augmentation With `ImageDataGenerator`

The Keras deep learning library provides the ability to use data augmentation automatically when training a model. This is achieved by using the `ImageDataGenerator` class. First, the class must be instantiated and the configuration for the types of data augmentation are specified by arguments to the class constructor. A range of techniques are supported, as well as pixel scaling methods. We will focus on five main types of data augmentation techniques for image data; specifically:

- Image shifts via the `width_shift_range` and `height_shift_range` arguments.
- Image flips via the `horizontal_flip` and `vertical_flip` arguments.
- Image rotations via the `rotation_range` argument
- Image brightness via the `brightness_range` argument.
- Image zoom via the `zoom_range` argument.

²<https://machinelearningmastery.com/wp-content/uploads/2019/01/bird.jpg>

For example, an instance of the `ImageDataGenerator` class can be constructed.

```
...  
# create data generator  
datagen = ImageDataGenerator()
```

Listing 9.1: Example of creating an image data generator.

Once constructed, an iterator can be created for an image dataset. The iterator will return one batch of augmented images for each iteration. An iterator can be created from an image dataset loaded in memory via the `flow()` function; for example:

```
...  
# load image dataset  
X, y = ...  
# create iterator  
it = datagen.flow(X, y)
```

Listing 9.2: Example of creating a dataset iterator from a dataset.

Alternately, an iterator can be created for an image dataset located on disk in a specified directory, where images in that directory are organized into subdirectories according to their class.

```
...  
# create iterator  
it = datagen.flow_from_directory(X, y, ...)
```

Listing 9.3: Example of creating a dataset iterator from a directory.

Once the iterator is created, it can be used to train a neural network model by calling the `fit_generator()` function. The `steps_per_epoch` argument must specify the number of batches of samples comprising one epoch. For example, if your original dataset has 10,000 images and your batch size is 32, then a reasonable value for `steps_per_epoch` when fitting a model on the augmented data might be `ceil(10,000/32)`, or 313 batches.

```
# define model  
model = ...  
# fit model on the augmented dataset  
model.fit_generator(it, steps_per_epoch=313, ...)
```

Listing 9.4: Example of fitting a model using a dataset iterator.

The images in the dataset are not used directly. Instead, only augmented images are provided to the model. Because the augmentations are performed randomly, this allows both modified images and close facsimiles of the original images (e.g. almost no augmentation) to be generated and used during training. A data generator can also be used to specify the validation dataset and the test dataset. Often, a separate `ImageDataGenerator` instance is used that may have the same pixel scaling configuration (not covered in this tutorial) as the `ImageDataGenerator` instance used for the training dataset, but would not use data augmentation. This is because data augmentation is only used as a technique for artificially extending the training dataset in order to improve model performance on an unaugmented dataset.

Now that we are familiar with how to use the `ImageDataGenerator`, let's look at some specific data augmentation techniques for image data. We will demonstrate each technique standalone by reviewing examples of images after they have been augmented. This is a good

practice and is recommended when configuring your data augmentation. It is also common to use a range of augmentation techniques at the same time when training. We have isolated the techniques to one per section for demonstration purposes only.

9.5 Horizontal and Vertical Shift Augmentation

A shift to an image means moving all pixels of the image in one direction, such as horizontally or vertically, while keeping the image dimensions the same. This means that some of the pixels will be clipped off the image and there will be a region of the image where new pixel values will have to be specified. The `width_shift_range` and `height_shift_range` arguments to the `ImageDataGenerator` constructor control the amount of horizontal and vertical shift respectively. These arguments can specify a floating point value that indicates the percentage (between 0 and 1) of the width or height of the image to shift. Alternately, a number of pixels can be specified to shift the image.

Specifically, a value in the range between no shift and the percentage or number of pixels will be sampled for each image and the shift performed, e.g. `[0, value]`. Alternately, you can specify a tuple or array of the min and max range from which the shift will be shifted; for example: `[-100, 100]` or `[-0.5, 0.5]`. The example below demonstrates a horizontal shift with the `width_shift_range` argument between `[-200,200]` pixels and generates a plot of generated images to demonstrate the effect.

```
# example of horizontal shift image augmentation
from numpy import expand_dims
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
# load the image
img = load_img('bird.jpg')
# convert to numpy array
data = img_to_array(img)
# expand dimension to one sample
samples = expand_dims(data, 0)
# create image data augmentation generator
datagen = ImageDataGenerator(width_shift_range=[-200,200])
# prepare iterator
it = datagen.flow(samples, batch_size=1)
# generate samples and plot
for i in range(9):
    # define subplot
    pyplot.subplot(330 + 1 + i)
    # generate batch of images
    batch = it.next()
    # convert to unsigned integers for viewing
    image = batch[0].astype('uint8')
    # plot raw pixel data
    pyplot.imshow(image)
    # show the figure
pyplot.show()
```

Listing 9.5: Example of a horizontal shift image data augmentation with Keras.

Running the example creates the instance of `ImageDataGenerator` configured for image augmentation, then creates the iterator. The iterator is then called nine times in a loop and each augmented image is plotted. We can see in the plot of the result that a range of different randomly selected positive and negative horizontal shifts was performed and the pixel values at the edge of the image are duplicated to fill in the empty part of the image created by the shift.

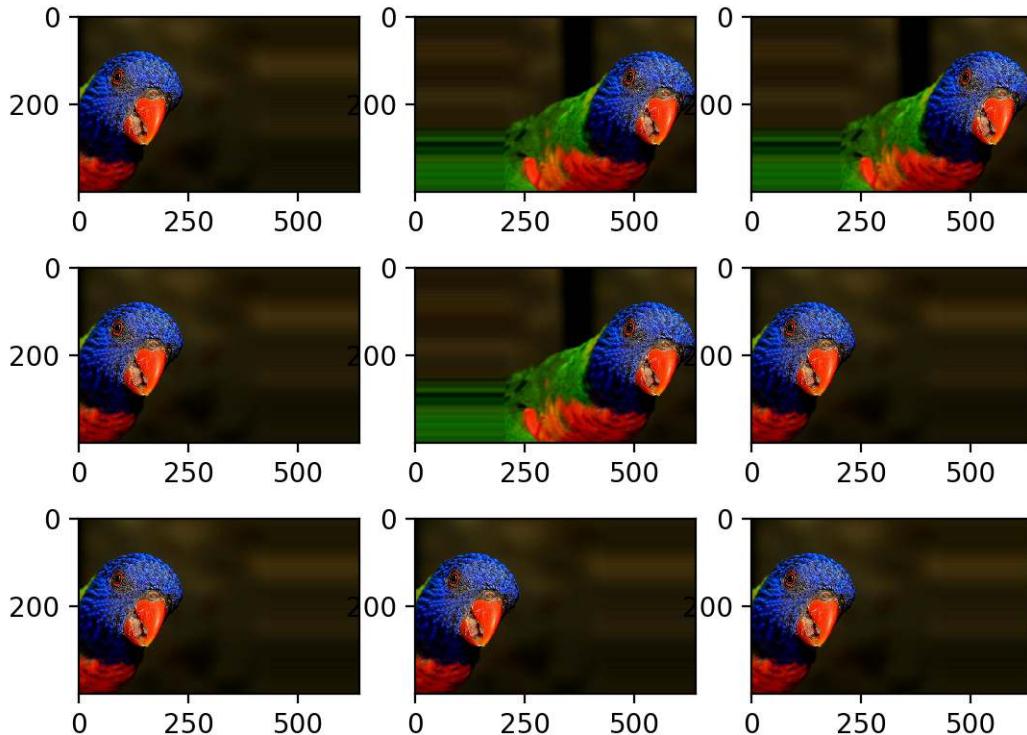


Figure 9.2: Plot of Augmented Generated With a Random Horizontal Shift.

Below is the same example updated to perform vertical shifts of the image via the `height_shift_range` argument, in this case specifying the percentage of the image to shift as 0.5 the height of the image.

```
# example of vertical shift image augmentation
from numpy import expand_dims
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
# load the image
img = load_img('bird.jpg')
# convert to numpy array
data = img_to_array(img)
# expand dimension to one sample
samples = expand_dims(data, 0)
```

```
# create image data augmentation generator
datagen = ImageDataGenerator(height_shift_range=0.5)
# prepare iterator
it = datagen.flow(samples, batch_size=1)
# generate samples and plot
for i in range(9):
    # define subplot
    pyplot.subplot(330 + 1 + i)
    # generate batch of images
    batch = it.next()
    # convert to unsigned integers for viewing
    image = batch[0].astype('uint8')
    # plot raw pixel data
    pyplot.imshow(image)
# show the figure
pyplot.show()
```

Listing 9.6: Example of a vertical shift image data augmentation with Keras.

Running the example creates a plot of images augmented with random positive and negative vertical shifts. We can see that both horizontal and vertical positive and negative shifts probably make sense for the chosen photograph, but in some cases, the replicated pixels at the edge of the image may not make sense to a model.

Note that other fill modes can be specified via `fill_mode` argument.

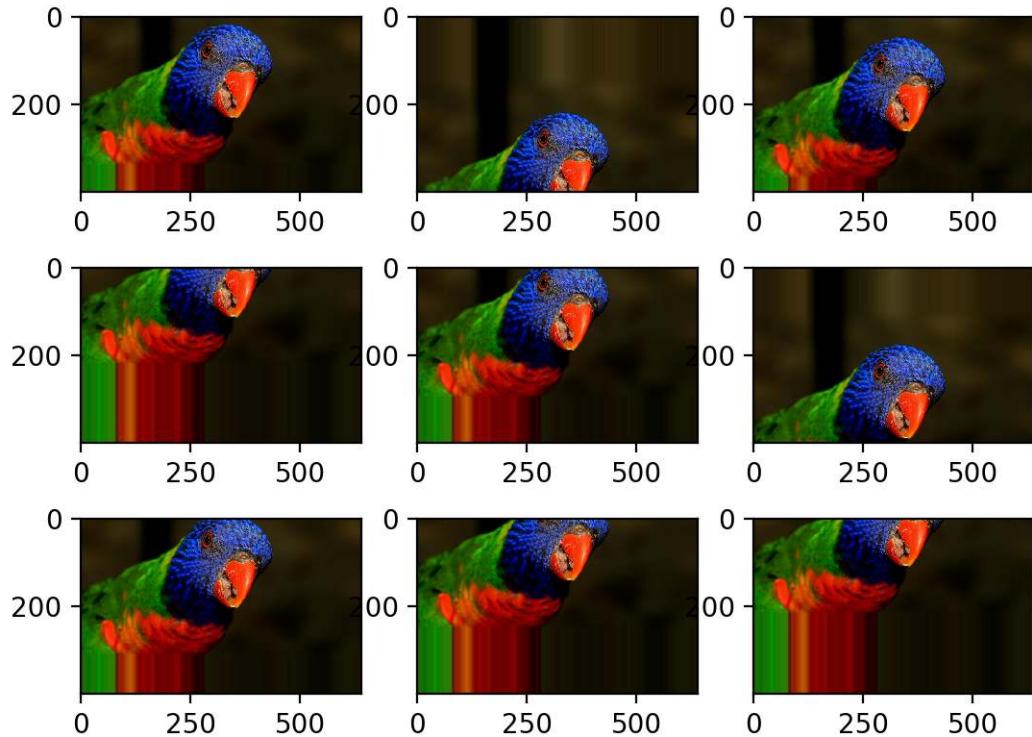


Figure 9.3: Plot of Augmented Images With a Random Vertical Shift.

9.6 Horizontal and Vertical Flip Augmentation

An image flip means reversing the rows or columns of pixels in the case of a vertical or horizontal flip respectively. The flip augmentation is specified by a boolean `horizontal_flip` or `vertical_flip` argument to the `ImageDataGenerator` class constructor. For photographs like the bird photograph used in this tutorial, horizontal flips may make sense, but vertical flips would not. For other types of images, such as aerial photographs, cosmology photographs, and microscopic photographs, perhaps vertical flips make sense. The example below demonstrates augmenting the chosen photograph with horizontal flips via the `horizontal_flip` argument.

```
# example of horizontal flip image augmentation
from numpy import expand_dims
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
# load the image
img = load_img('bird.jpg')
# convert to numpy array
data = img_to_array(img)
# expand dimension to one sample
```

```

samples = expand_dims(data, 0)
# create image data augmentation generator
datagen = ImageDataGenerator(horizontal_flip=True)
# prepare iterator
it = datagen.flow(samples, batch_size=1)
# generate samples and plot
for i in range(9):
    # define subplot
    pyplot.subplot(330 + 1 + i)
    # generate batch of images
    batch = it.next()
    # convert to unsigned integers for viewing
    image = batch[0].astype('uint8')
    # plot raw pixel data
    pyplot.imshow(image)
# show the figure
pyplot.show()

```

Listing 9.7: Example of a horizontal flip image data augmentation with Keras.

Running the example creates a plot of nine augmented images. We can see that the horizontal flip is applied randomly to some images and not others.

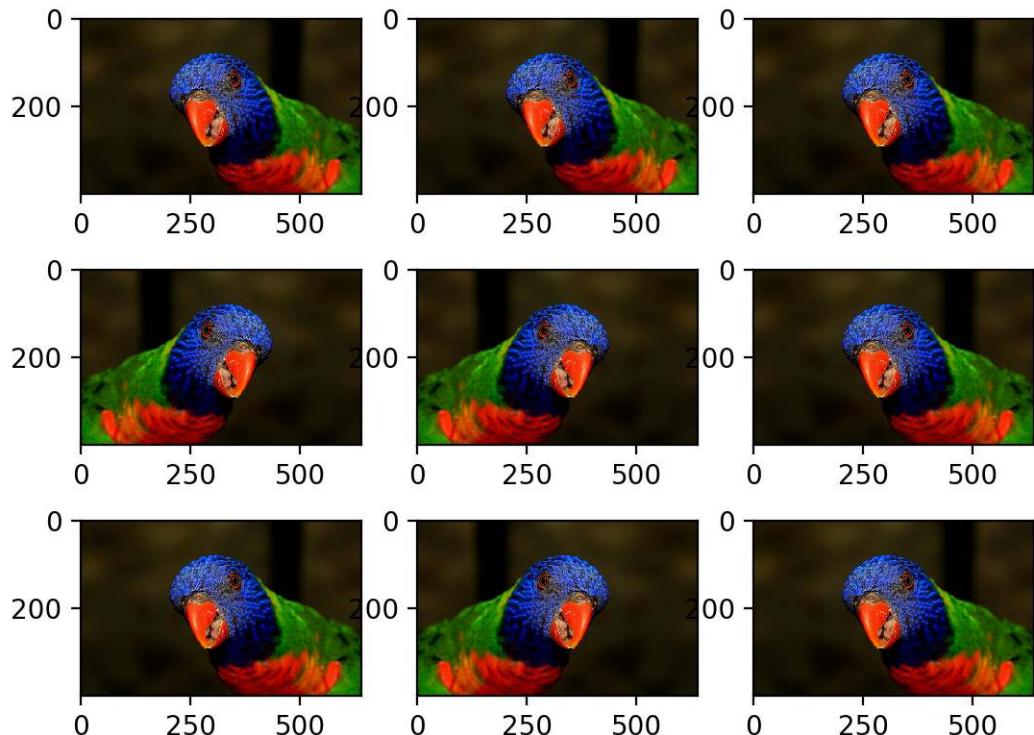


Figure 9.4: Plot of Augmented Images With a Random Horizontal Flip.

9.7 Random Rotation Augmentation

A rotation augmentation randomly rotates the image clockwise by a given number of degrees from 0 to 360. The rotation will likely rotate pixels out of the image frame and leave areas of the frame with no pixel data that must be filled in. The example below demonstrates random rotations via the `rotation_range` argument, with rotations to the image between 0 and 90 degrees.

```
# example of random rotation image augmentation
from numpy import expand_dims
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
# load the image
img = load_img('bird.jpg')
# convert to numpy array
data = img_to_array(img)
# expand dimension to one sample
samples = expand_dims(data, 0)
# create image data augmentation generator
datagen = ImageDataGenerator(rotation_range=90)
# prepare iterator
it = datagen.flow(samples, batch_size=1)
# generate samples and plot
for i in range(9):
    # define subplot
    pyplot.subplot(330 + 1 + i)
    # generate batch of images
    batch = it.next()
    # convert to unsigned integers for viewing
    image = batch[0].astype('uint8')
    # plot raw pixel data
    pyplot.imshow(image)
# show the figure
pyplot.show()
```

Listing 9.8: Example of a rotation image data augmentation with Keras.

Running the example generates examples of the rotated image, showing in some cases pixels rotated out of the frame and the nearest-neighbor fill.

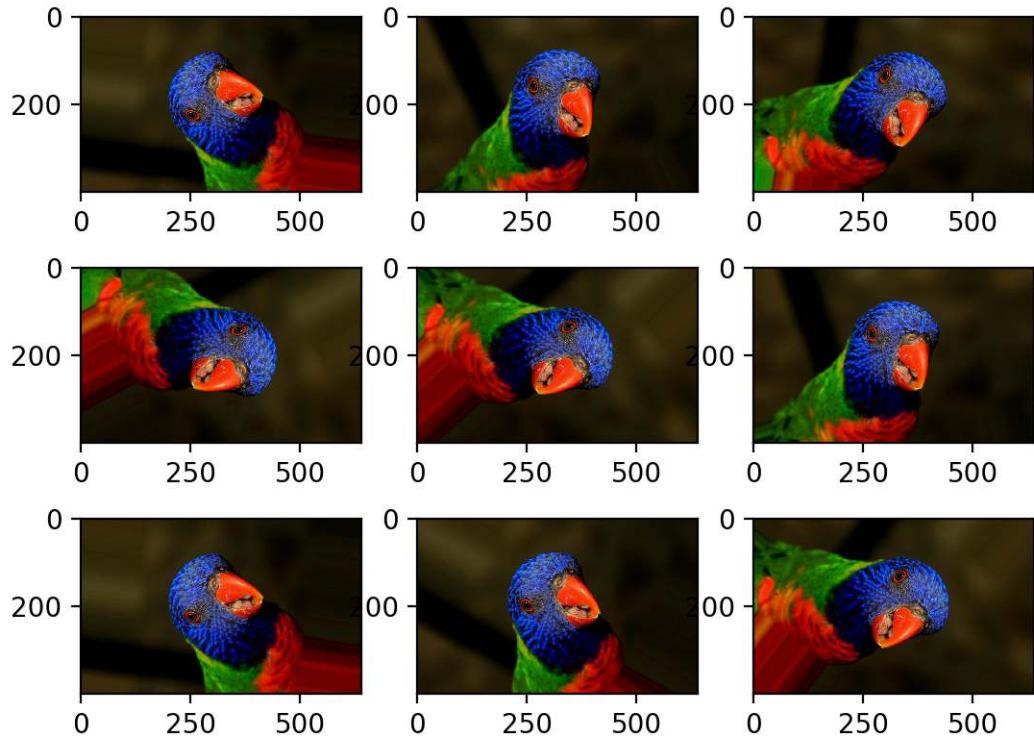


Figure 9.5: Plot of Images Generated With a Random Rotation Augmentation.

9.8 Random Brightness Augmentation

The brightness of the image can be augmented by either randomly darkening images, brightening images, or both. The intent is to allow a model to generalize across images trained on different lighting levels. This can be achieved by specifying the `brightness_range` argument to the `ImageDataGenerator()` constructor that specifies min and max range as a float representing a percentage for selecting a darkening or brightening amount. Values less than 1.0 darken the image, e.g. [0.5, 1.0], whereas values larger than 1.0 brighten the image, e.g. [1.0, 1.5], where 1.0 has no effect on brightness. The example below demonstrates a brightness image augmentation, allowing the generator to randomly darken the image between 1.0 (no change) and 0.2 or 20%.

```
# example of brightening image augmentation
from numpy import expand_dims
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
# load the image
img = load_img('bird.jpg')
# convert to numpy array
data = img_to_array(img)
```

```
# expand dimension to one sample
samples = expand_dims(data, 0)
# create image data augmentation generator
datagen = ImageDataGenerator(brightness_range=[0.2,1.0])
# prepare iterator
it = datagen.flow(samples, batch_size=1)
# generate samples and plot
for i in range(9):
    # define subplot
    pyplot.subplot(330 + 1 + i)
    # generate batch of images
    batch = it.next()
    # convert to unsigned integers for viewing
    image = batch[0].astype('uint8')
    # plot raw pixel data
    pyplot.imshow(image)
# show the figure
pyplot.show()
```

Listing 9.9: Example of a brightness image data augmentation with Keras.

Running the example shows the augmented images with varying amounts of darkening applied.

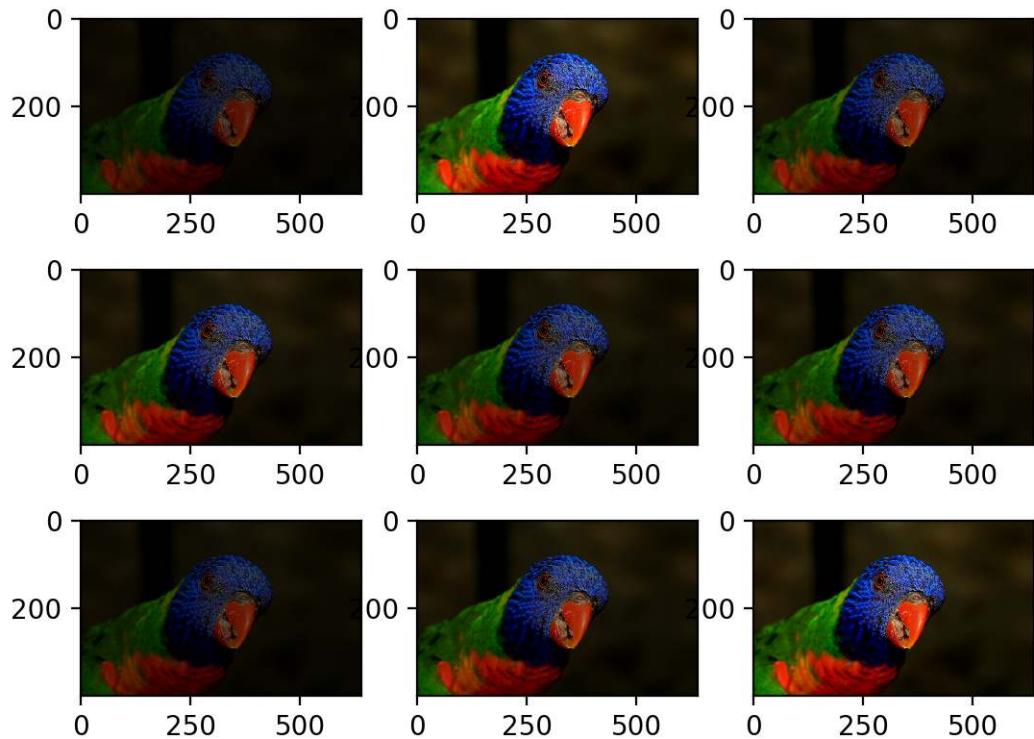


Figure 9.6: Plot of Images Generated With a Random Brightness Augmentation.

9.9 Random Zoom Augmentation

A zoom augmentation randomly zooms the image and either adds new pixel values around the image or interpolates pixel values respectively. Image zooming can be configured by the `zoom_range` argument to the `ImageDataGenerator` constructor. You can specify the percentage of the zoom as a single float or a range as an array or tuple. If a float is specified, then the range for the zoom will be $[1\text{-value}, 1+\text{value}]$. For example, if you specify 0.3, then the range will be $[0.7, 1.3]$, or between 70% (zoom in) and 130% (zoom out). The zoom amount is uniformly randomly sampled from the zoom region for each dimension (width, height) separately.

The zoom may not feel intuitive. Note that zoom values less than 1.0 will zoom the image in, e.g. $[0.5, 0.5]$ makes the object in the image 50% larger or closer, and values larger than 1.0 will zoom the image out by 50%, e.g. $[1.5, 1.5]$ makes the object in the image smaller or further away. A zoom of $[1.0, 1.0]$ has no effect. The example below demonstrates zooming the image in, e.g. making the object in the photograph larger.

```
# example of zoom image augmentation
from numpy import expand_dims
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
# load the image
img = load_img('bird.jpg')
# convert to numpy array
data = img_to_array(img)
# expand dimension to one sample
samples = expand_dims(data, 0)
# create image data augmentation generator
datagen = ImageDataGenerator(zoom_range=[0.5,1.0])
# prepare iterator
it = datagen.flow(samples, batch_size=1)
# generate samples and plot
for i in range(9):
    # define subplot
    pyplot.subplot(330 + 1 + i)
    # generate batch of images
    batch = it.next()
    # convert to unsigned integers for viewing
    image = batch[0].astype('uint8')
    # plot raw pixel data
    pyplot.imshow(image)
# show the figure
pyplot.show()
```

Listing 9.10: Example of a zoom image data augmentation with Keras.

Running the example generates examples of the zoomed image, showing a random zoom in that is different on both the width and height dimensions that also randomly changes the aspect ratio of the object in the image.

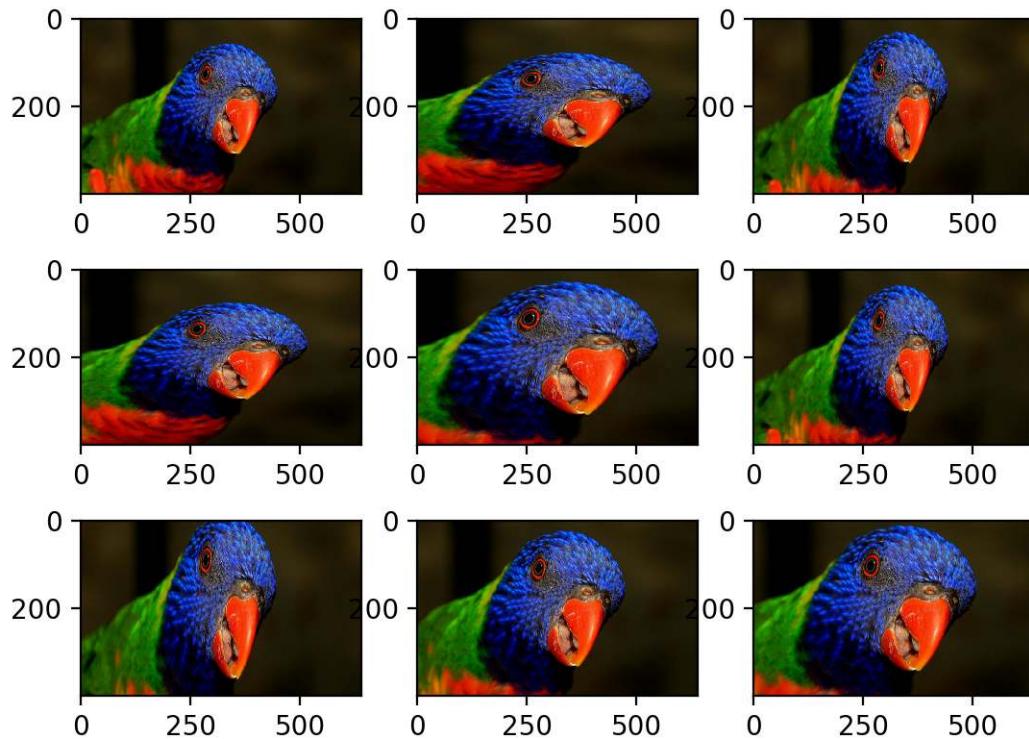


Figure 9.7: Plot of Images Generated With a Random Zoom Augmentation.

9.10 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Your Own Image.** Experiment with data augmentation on your own image, including varying the augmentation parameters.
- **Save Images.** Experiment with configuring the image data generator to save augmented images to file.
- **Other Augmentations.** Experiment with other augmentation methods not covered in this tutorial, such as shear.

If you explore any of these extensions, I'd love to know.

9.11 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

9.11.1 API

- Image Preprocessing Keras API.

<https://keras.io/preprocessing/image/>

- Keras Image Preprocessing Code.

https://github.com/keras-team/keras-preprocessing/blob/master/keras_preprocessing/image/affine_transformations.py

- Sequential Model API.

<https://keras.io/models/sequential/>

9.11.2 Articles

- Building powerful image classification models using very little data, Keras Blog.

<https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html>

9.12 Summary

In this tutorial, you discovered how to use image data augmentation when training deep learning neural networks. Specifically, you learned:

- Image data augmentation is used to expand the training dataset in order to improve the performance and ability of the model to generalize.
- Image data augmentation is supported in the Keras deep learning library via the `ImageDataGenerator` class.
- How to use shift, flip, brightness, and zoom image data augmentation.

9.12.1 Next

This was the last tutorial in the data preparation part. In the next part, you will discover the details of the layers of a convolutional neural network.

Part III

Convolutions and Pooling

Overview

In this part you will discover the basics of convolutional and pooling layers, the two main elements that make up convolutional neural network models. After reading the chapters in this part, you will know:

- The difference between channels-first and channels-last image formats and how to configure Keras for each (Chapter [10](#)).
- How convolutional filters are applied to input images to create output feature maps (Chapter [11](#)).
- The effect of filter size, the effect of the stride size, and the problem of border effects and how they can be addressed with padding (Chapter [12](#)).
- How pooling layers are used to downsample feature maps and the effect of average, max, and global pooling (Chapter [13](#)).

Chapter 10

How to Use Different Color Channel Ordering Formats

Color images have height, width, and color channel dimensions. When represented as three-dimensional arrays, the channel dimension for the image data is last by default, but may be moved to be the first dimension, often for performance-tuning reasons. The use of these two *channel ordering formats* and preparing data to meet a specific preferred channel ordering can be confusing to beginners. In this tutorial, you will discover channel ordering formats, how to prepare and manipulate image data to meet formats, and how to configure the Keras deep learning library for different channel orderings. After completing this tutorial, you will know:

- The three-dimensional array structure of images and the channels first and channels last array formats.
- How to add a channels dimension and how to convert images between the channel formats.
- How the Keras deep learning library manages a preferred channel ordering and how to change and query this preference.

Let's get started.

10.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Images as 3D Arrays
2. Manipulating Image Channels
3. Keras Channel Ordering

10.2 Images as 3D Arrays

An image can be stored as a three-dimensional array in memory. Typically, the image format has one dimension for rows (height), one for columns (width) and one for channels. If the image

is black and white (grayscale), the channels dimension may not be explicitly present, e.g. there is one unsigned integer pixel value for each (row, column) coordinate in the image. Colored images typically have three channels. A given pixel value at the (row, column) coordinate will have red, green, and blue components. Deep learning neural networks require that image data be provided as three-dimensional arrays.

This applies even if your image is grayscale. In this case, the additional dimension for the single color channel must be added. There are two ways to represent the image data as a three dimensional array. The first involves having the channels as the last or third dimension in the array. This is called *channels last*. The second involves having the channels as the first dimension in the array, called *channels first*.

- **Channels Last.** Image data is represented in a three-dimensional array where the last channel represents the color channels, e.g. [rows] [cols] [channels].
- **Channels First.** Image data is represented in a three-dimensional array where the first channel represents the color channels, e.g. [channels] [rows] [cols].

Some image processing and deep learning libraries prefer channels first ordering, and some prefer channels last. As such, it is important to be familiar with the two approaches to representing images.

10.3 Manipulating Image Channels

You may need to change or manipulate the image channels or channel ordering. This can be achieved easily using the NumPy Python library. Let's look at some examples. In this tutorial, we will use a photograph taken by Larry Koester¹, some rights reserved, of the Phillip Island Penguin Parade.

¹<https://www.flickr.com/photos/larrywkoester/45435718605/>



Figure 10.1: Photograph of the Phillip Island Penguin Parade.

Download the photograph and place it into your current working directory with the filename `penguin_parade.jpg`.

- Download Photo (`penguin_parade.jpg`).²

10.3.1 How to Add a Channel to a Grayscale Image

Grayscale images are loaded as a two-dimensional array. Before they can be used for modeling, you may have to add an explicit channel dimension to the image. This does not add new data; instead, it changes the array data structure to have an additional third axis with one dimension to hold the grayscale pixel values. For example, a grayscale image with the dimensions `[rows] [cols]` can be changed to `[rows] [cols] [channels]` or `[channels] [rows] [cols]` where the new `[channels]` axis has one dimension.

This can be achieved using the `expand_dims()` NumPy function. The `axis` argument allows you to specify whether the new dimension will be added to the first, e.g. first for channels first or last for channels last. The example below loads the Penguin Parade photograph as a grayscale image using the Pillow library and demonstrates how to add a channel dimension.

```
# example of expanding dimensions
from numpy import expand_dims
from numpy import asarray
from PIL import Image
# load the image
```

²https://machinelearningmastery.com/wp-content/uploads/2019/01/penguin_arade.jpg

```



```

Listing 10.1: Example of adding channels to a grayscale image.

Running the example first loads the photograph using the Pillow library, then converts it to a grayscale image. The image object is converted to a NumPy array and we confirm the shape of the array is two dimensional, specifically (424, 640).

The `expand_dims()` function is then used to add a channel via `axis=0` to the front of the array and the change is confirmed with the shape (1, 424, 640). The same function is then used to add a channel to the end or third dimension of the array with `axis=2` and the change is confirmed with the shape (424, 640, 1).

```
(424, 640)
(1, 424, 640)
(424, 640, 1)
```

Listing 10.2: Example output from adding channels to a grayscale image.

Another popular alternative to expanding the dimensions of an array is to use the `reshape()` NumPy function and specify a tuple with the new shape; for example:

```

...
# add a channels dimension
data = data.reshape((424, 640, 1))

```

Listing 10.3: Example of using the `reshape()` function to add a channels dimension.

10.3.2 How to Change Image Channel Ordering

After a color image is loaded as a three-dimensional array, the channel ordering can be changed. This can be achieved using the `moveaxis()` NumPy function. It allows you to specify the index of the source axis and the destination axis. This function can be used to change an array in channel last format such, as [rows] [cols] [channels] to channels first format, such as [channels] [rows] [cols], or the reverse. The example below loads the Penguin Parade photograph in channel last format and uses the `moveaxis()` function change it to channels first format.

```

# change image from channels last to channels first format
from numpy import moveaxis
from numpy import asarray
from PIL import Image
# load the color image


```

```
# convert to numpy array
data = asarray(img)
print(data.shape)
# change channels last to channels first format
data = moveaxis(data, 2, 0)
print(data.shape)
# change channels first to channels last format
data = moveaxis(data, 0, 2)
print(data.shape)
```

Listing 10.4: Example of changing the channel ordering of an image.

Running the example first loads the photograph using the Pillow library and converts it to a NumPy array confirming that the image was loaded in channels last format with the shape (424, 640, 3). The `moveaxis()` function is then used to move the channels axis from position 2 to position 0 and the result is confirmed showing channels first format (3, 424, 640). This is then reversed, moving the channels in position 0 to position 2 again.

```
(424, 640, 3)
(3, 424, 640)
(424, 640, 3)
```

Listing 10.5: Example output from changing the channel ordering of an image.

10.4 Keras Channel Ordering

The Keras deep learning library is agnostic to how you wish to represent images in either channel first or last format, but the preference must be specified and adhered to when using the library. Keras wraps a number of mathematical libraries, and each has a preferred channel ordering. The three main libraries that Keras may wrap and their preferred channel ordering are listed below:

- **TensorFlow**: Channels last order.
- **Theano**: Channels first order.
- **CNTK**: Channels last order.

By default, Keras is configured to use TensorFlow and the channel ordering is also by default channels last. You can use either channel ordering with any library and the Keras library. Some libraries claim that the preferred channel ordering can result in a large difference in performance. For example, use of the MXNet mathematical library as the backend for Keras recommends using the channels first ordering for better performance.

We strongly recommend changing the `image_data_format` to `channels_first`. MXNet is significantly faster on `channels_first` data.

— *Performance Tuning Keras with MXNet Backend*, Apache MXNet, MXNet Project.

10.4.1 Default Channel Ordering

The library and preferred channel ordering are listed in the Keras configuration file, stored in your home directory under `/.keras/keras.json`. The preferred channel ordering is stored in the `image_data_format` configuration setting and can be set as either `channels_last` or `channels_first`. For example, below is the contents of a `keras.json` configuration file. In it, you can see that the system is configured to use ‘‘`tensorflow`’’ and `channels_last` order.

```
{
  "image_data_format": "channels_last",
  "backend": "tensorflow",
  "epsilon": 1e-07,
  "floatx": "float32"
}
```

Listing 10.6: Example of Keras configuration file showing channel ordering.

Based on your preferred channel ordering, you will have to prepare your image data to match the preferred ordering. Specifically, this will include tasks such as:

- Resizing or expanding the dimensions of any training, validation, and test data to meet the expectation.
- Specifying the expected input shape of samples when defining models (e.g. `input_shape=(28, 28, 1)`).

10.4.2 Model-Specific Channel Ordering

In addition, those neural network layers that are designed to work with images, such as `Conv2D`, also provide an argument called `data_format` that allows you to specify the channel ordering. For example:

```
...
# configure channel ordering in layer
model.add(Conv2D(..., data_format='channels_first'))
```

Listing 10.7: Example of configuring the channel ordering for a layer.

By default, this will use the preferred ordering specified in the `image_data_format` value of the Keras configuration file. Nevertheless, you can change the channel order for a given model, and in turn, the datasets and input shape would also have to be changed to use the new channel ordering for the model. This can be useful when loading a model used for transfer learning that has a channel ordering different to your preferred channel ordering.

10.4.3 Query Channel Ordering

You can confirm your current preferred channel ordering by printing the result of the `image_data_format()` function. The example below demonstrates.

```
...
# configure channel ordering in layer
model.add(Conv2D(..., data_format='channels_first'))
```

Listing 10.8: Example of configuring the channel ordering for a layer.

```
# show preferred channel order
from keras import backend
print(backend.image_data_format())
```

Listing 10.9: Example of showing the currently configured channel ordering.

Running the example prints your preferred channel ordering as configured in your Keras configuration file. In this case, the channels last format is used.

```
channels_last
```

Listing 10.10: Example output from showing the currently configured channel ordering.

Accessing this property can be helpful if you want to automatically construct models or prepare data differently depending on the systems preferred channel ordering; for example:

```
...
# conditional logic based on channel ordering
if backend.image_data_format() == 'channels_last':
    ...
else:
    ...
```

Listing 10.11: Example of conditional logic based on channel ordering.

10.4.4 Force Channel Ordering

Finally, the channel ordering can be forced for a specific program. This can be achieved by calling the `set_image_data_format()` function on the Keras backend to either ‘`channels_first`’ (Theano) for channel-first ordering, or ‘`channels_last`’ (TensorFlow) for channel-last ordering. This can be useful if you want a program or model to operate consistently regardless of Keras default channel ordering configuration.

```
# force a channel ordering
from keras import backend
# force channels-first ordering
backend.set_image_data_format('channels_first')
print(backend.image_data_format())
# force channels-last ordering
backend.set_image_data_format('channels_last')
print(backend.image_data_format())
```

Listing 10.12: Example of forcing a channel ordering.

Running the example first forces channels-first ordering, then channels-last ordering, confirming each configuration by printing the channel ordering mode after the change.

```
channels_first
channels_last
```

Listing 10.13: Example output from forcing a channel ordering.

10.5 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Change Default.** Change the default channel ordering for your workstation and confirm the change had the desired effect.
- **Model Channel Ordering.** Develop a small CNN model that uses channels-first and channels-last ordering and review the model summary.
- **Convenience Function.** Develop a function that will appropriately change the structure of an image dataset based on the configured channel ordering.

If you explore any of these extensions, I'd love to know.

10.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

10.6.1 APIs

- `numpy.expand_dims` API.
https://docs.scipy.org/doc/numpy/reference/generated/numpy.expand_dims.html
- `numpy.reshape` API.
<https://docs.scipy.org/doc/numpy/reference/generated/numpy.reshape.html>
- `numpy.moveaxis` API.
<https://docs.scipy.org/doc/numpy/reference/generated/numpy.moveaxis.html>
- Keras Backend API.
<https://keras.io/backend/>
- Keras Convolutional Layers API.
<https://keras.io/layers/convolutional/>

10.6.2 Articles

- Performance Tuning - Keras with MXNet Backend, MXNet Project.
<https://github.com/awslabs/keras-apache-mxnet/wiki/Performance-Tuning---Keras-with-MXNet-Backend>

10.7 Summary

In this tutorial, you discovered channel ordering formats, how to prepare and manipulate image data to meet formats, and how to configure the Keras deep learning library for different channel orderings. Specifically, you learned:

- The three-dimensional array structure of images and the channels first and channels last array formats.
- How to add a channels dimension and how to convert images between the channel formats.
- How the Keras deep learning library manages a preferred channel ordering and how to change and query this preference.

10.7.1 Next

In the next section, you will discover how convolutional layers work in a convolutional neural network.

Chapter 11

How Convolutional Layers Work

Convolution and the convolutional layer are the major building blocks used in convolutional neural networks. A convolution is the simple application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected feature in an input, such as an image. The innovation of convolutional neural networks is the ability to automatically learn a large number of filters in parallel specific to a training dataset under the constraints of a specific predictive modeling problem, such as image classification. The result is that highly specific features can be detected anywhere on input images. In this tutorial, you will discover how convolutions work in the convolutional neural network. After completing this tutorial, you will know:

- Convolutional neural networks apply a filter to an input to create a feature map that summarizes the presence of detected features in the input.
- Filters can be handcrafted, such as line detectors, but the innovation of convolutional neural networks is to learn the filters during training in the context of a specific prediction problem.
- How to calculate the feature map for one- and two-dimensional convolutional layers in a convolutional neural network.

Let's get started.

11.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Convolution in Convolutional Neural Networks
2. Convolution in Computer Vision
3. Power of Learned Filters
4. Worked Example of Convolutional Layers

11.2 Convolution in Convolutional Neural Networks

The convolutional neural network, or CNN for short, is a specialized type of neural network model designed for working with two-dimensional image data, although they can be used with one-dimensional and three-dimensional data. Central to the convolutional neural network is the convolutional layer that gives the network its name. This layer performs an operation called a *convolution*. In the context of a convolutional neural network, a convolution is a linear operation that involves the multiplication of a set of weights with the input, much like a traditional neural network. Given that the technique was designed for two-dimensional input, the multiplication is performed between an array of input data and a two-dimensional array of weights, called a filter or a kernel.

The filter is smaller than the input data and the type of multiplication applied between a filter-sized patch of the input and the filter is a dot product. A dot product is the element-wise multiplication between the filter-sized patch of the input and filter, which is then summed, always resulting in a single value. Because it results in a single value, the operation is often referred to as the *scalar product*. Using a filter smaller than the input is intentional as it allows the same filter (set of weights) to be multiplied by the input array multiple times at different locations on the input. Specifically, the filter is applied systematically to each overlapping filter-sized patch of the input data, left to right, top to bottom.

This systematic application of the same filter across an image is a powerful idea. If the filter is designed to detect a specific type of feature in the input, then the application of that filter systematically across the entire input image allows the filter an opportunity to discover that feature anywhere in the image. This capability is commonly referred to as translation invariance, e.g. the general interest in whether the feature is present rather than where it was present.

Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is. For example, when determining whether an image contains a face, we need not know the location of the eyes with pixel-perfect accuracy, we just need to know that there is an eye on the left side of the face and an eye on the right side of the face.

— Page 342, *Deep Learning*, 2016.

The output from multiplying the filter with a filter-sized-patch of the input array one time is a single value. As the filter is applied systematically across the input array, the result is a two-dimensional array comprised of output values that represent a filtering of the input. As such, the two-dimensional output array from this operation is called a *feature map*. Once a feature map is created, we can pass each value in the feature map through a nonlinearity, such as a ReLU, much like we do for the outputs of a fully connected layer.

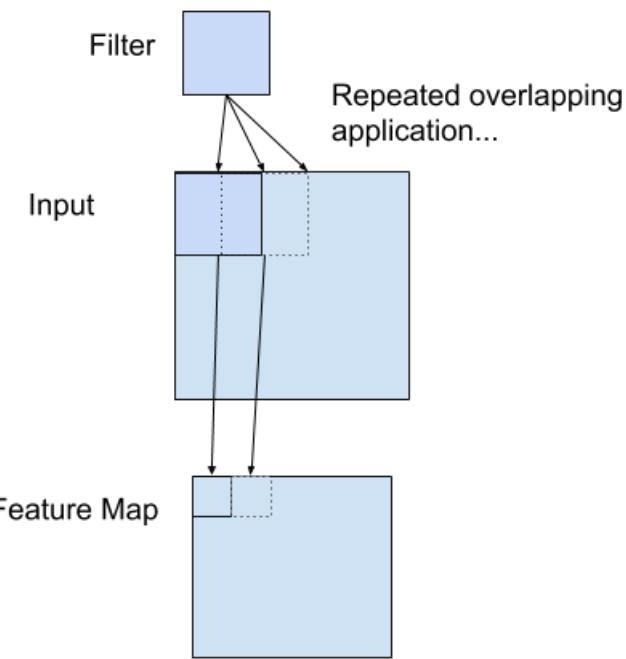


Figure 11.1: Example of a Filter Applied to a Two-Dimensional Input to Create a Feature Map.

If you come from a digital signal processing field or related area of mathematics, you may understand the convolution operation on a matrix as something different. Specifically, the filter (kernel) is flipped prior to being applied to the input. Technically, the convolution as described in the use of convolutional neural networks is actually a *cross-correlation*. Nevertheless, in deep learning, it is referred to as a *convolution* operation.

Many machine learning libraries implement cross-correlation but call it convolution.

— Page 333, *Deep Learning*, 2016.

In summary, we have an input, such as an image of pixel values, and we have a filter, which is a set of weights, and the filter is systematically applied to the input data to create a feature map.

11.3 Convolution in Computer Vision

The idea of applying the convolutional operation to image data is not new or unique to convolutional neural networks; it is a common technique used in computer vision. Historically, filters were designed by hand by computer vision experts, which were then applied to an image to result in a feature map or output from applying the filter that makes the analysis of the image easier in some way. For example, below is a hand crafted 3×3 element filter for detecting vertical lines:

0.0, 1.0, 0.0
0.0, 1.0, 0.0
0.0, 1.0, 0.0

Listing 11.1: Example of a vertical line filter.

Applying this filter to an image will result in a feature map that only contains vertical lines. It is a vertical line detector. You can see this from the weight values in the filter; any pixels values in the center vertical line will be positively activated and any on either side will be negatively activated. Dragging this filter systematically across pixel values in an image can only highlight vertical line pixels. A horizontal line detector could also be created and also applied to the image, for example:

```
0.0, 0.0, 0.0
1.0, 1.0, 1.0
0.0, 0.0, 0.0
```

Listing 11.2: Example of a horizontal line filter.

Combining the results from both filters, e.g. combining both feature maps, will result in all of the lines in an image being highlighted. A suite of tens or even hundreds of other small filters can be designed to detect other features in the image. The innovation of using the convolution operation in a neural network is that the values of the filter are weights to be learned during the training of the network. The network will learn what types of features to extract from the input. Specifically, training under stochastic gradient descent, the network is forced to learn to extract features from the image that minimize the loss for the specific task the network is being trained to solve, e.g. extract features that are the most useful for classifying images as dogs or cats. In this context, you can see that this is a powerful idea.

11.4 Power of Learned Filters

Learning a single filter specific to a machine learning task is a powerful technique. Yet, convolutional neural networks achieve much more in practice.

11.4.1 Multiple Filters

Convolutional neural networks do not learn a single filter; they, in fact, learn multiple features in parallel for a given input. For example, it is common for a convolutional layer to learn from 32 to 512 filters in parallel for a given input. This gives the model 32, or even 512, different ways of extracting features from an input, or many different ways of both *learning to see* and after training, many different ways of *seeing* the input data. This diversity allows specialization, e.g. not just lines, but the specific lines seen in your specific training data.

11.4.2 Multiple Channels

Color images have multiple channels, typically one for each color channel, such as red, green, and blue. From a data perspective, that means that a single image provided as input to the model is, in fact, three two-dimensional images. A filter must always have the same number of channels as the input, often referred to as *depth*. If an input image has 3 channels (e.g. a depth of 3), then a filter applied to that image must also have 3 channels (e.g. a depth of 3). In this case, a 3×3 filter would in fact be $3 \times 3 \times 3$ or [3, 3, 3] for rows, columns, and depth.

Regardless of the depth of the input and depth of the filter, the filter is applied to the input using a dot product operation which results in a single value.

This means that if a convolutional layer has 32 filters, these 32 filters are not just two-dimensional for the two-dimensional image input, but are also three-dimensional, having specific filter weights for each of the three channels. Yet, each filter results in a single two-dimensional feature map. Which means that the depth of the output of applying the convolutional layer with 32 filters is 32 for the 32 feature maps created.

11.4.3 Multiple Layers

Convolutional layers are not only applied to input data, e.g. raw pixel values, but they can also be applied to the output of other layers. The stacking of convolutional layers allows a hierarchical decomposition of the input. Consider that the filters that operate directly on the raw pixel values will learn to extract low-level features, such as lines. The filters that operate on the output of the first line layers may extract features that are combinations of lower-level features, such as features that comprise multiple lines to express shapes. This process continues until very deep layers are extracting faces, animals, houses, and so on. This is exactly what we see in practice. The abstraction of features to high and higher orders as the depth of the network is increased.

11.5 Worked Example of Convolutional Layers

The Keras deep learning library provides a suite of convolutional layers. We can better understand the convolution operation by looking at some worked examples with contrived data and handcrafted filters. In this section, we'll look at both a one-dimensional convolutional layer and a two-dimensional convolutional layer example to both make the convolution operation concrete and provide a worked example of using the Keras layers.

11.5.1 Example of 1D Convolutional Layer

We can define a one-dimensional input that has eight elements; two in the middle with the value 1.0, and three on either side with the value of 0.0.

```
[0, 0, 0, 1, 1, 0, 0, 0]
```

Listing 11.3: Example of a one-dimensional sequence.

The input to Keras must be three dimensional for a 1D convolutional layer. The first dimension refers to each input sample; in this case, we only have one sample. The second dimension refers to the length of each sample; in this case, the length is eight. The third dimension refers to the number of channels in each sample; in this case, we only have a single channel. Therefore, the shape of the input array will be [1, 8, 1].

```
...
# define input data
data = asarray([0, 0, 0, 1, 1, 0, 0, 0])
data = data.reshape(1, 8, 1)
```

Listing 11.4: Example of defining a one-dimensional sequences as a sample.

We will define a model that expects input samples to have the shape [8, 1]. The model will have a single filter with the shape of 3, or three elements wide. Keras refers to the shape of the filter as the `kernel_size` (the required second argument to the layer).

```
...
# create model
model = Sequential()
model.add(Conv1D(1, 3, input_shape=(8, 1)))
```

Listing 11.5: Example of a model with a Conv1D layer and a single filter.

By default, the filters in a convolutional layer are initialized with random weights. In this contrived example, we will manually specify the weights for the single filter. We will define a filter that is capable of detecting bumps, that is a high input value surrounded by low input values, as we defined in our input example. The three element filter we will define looks as follows:

```
[0, 1, 0]
```

Listing 11.6: Example of a bump detection one-dimensional filter.

Each filter also has a bias input value that also requires a weight that we will set to zero. Therefore, we can force the weights of our one-dimensional convolutional layer to use our handcrafted filter as follows:

```
...
# define a vertical line detector
weights = [asarray([[0],[1],[0]]), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
```

Listing 11.7: Example of a hard coding a bump detection one-dimensional filter.

The weights must be specified in a three-dimensional structure, in terms of rows, columns, and channels. The filter has a single row, three columns, and one channel. We can retrieve the weights and confirm that they were set correctly.

```
...
# confirm they were stored
print(model.get_weights())
```

Listing 11.8: Example of a confirming the weights in a hard coded filter.

Finally, we can apply the single filter to our input data. We can achieve this by calling the `predict()` function on the model. This will return the feature map directly: that is the output of applying the filter systematically across the input sequence.

```
...
# apply filter to input data
yhat = model.predict(data)
print(yhat)
```

Listing 11.9: Example of applying a filter to the input sample.

Tying all of this together, the complete example is listed below.

```
# example of calculation 1d convolutions
from numpy import asarray
from keras.models import Sequential
from keras.layers import Conv1D
# define input data
data = asarray([0, 0, 0, 1, 1, 0, 0, 0])
data = data.reshape(1, 8, 1)
# create model
model = Sequential()
model.add(Conv1D(1, 3, input_shape=(8, 1)))
# define a vertical line detector
weights = [asarray([[0],[1],[0]]), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
# confirm they were stored
print(model.get_weights())
# apply filter to input data
yhat = model.predict(data)
print(yhat)
```

Listing 11.10: Example of applying a hard-coded filter to an input sequence.

Running the example first prints the weights of the network; that is the confirmation that our handcrafted filter was set in the model as we expected. Next, the filter is applied to the input pattern and the feature map is calculated and displayed. We can see from the values of the feature map that the bump was detected correctly.

```
[array([[[0.],
       [[1.],
        [[0.]], dtype=float32), array([0.], dtype=float32)

[[[0.]
 [0.]
 [1.]
 [1.]
 [0.]
 [0.]]]
```

Listing 11.11: Example output from applying a hard-coded filter to an input sequence.

Let's take a closer look at what happened. Recall that the input is an eight element vector with the values: $[0, 0, 0, 1, 1, 0, 0, 0]$. First, the three-element filter $[0, 1, 0]$ was applied to the first three inputs of the input $[0, 0, 0]$ by calculating the dot product (\cdot operator), which resulted in a single output value in the feature map of zero. Recall that a dot product is the sum of the element-wise multiplications, or here it is $(0 \times 0) + (1 \times 0) + (0 \times 0) = 0$. In NumPy, this can be implemented manually as:

```
# manually apply a 1d filter
from numpy import asarray
print(asarray([0, 1, 0]).dot(asarray([0, 0, 0])))
```

Listing 11.12: Example of manually applying a one-dimensional filter.

In our manual example, this is as follows:

$$[0, 1, 0] \cdot [0, 0, 0] = 0 \quad (11.1)$$

The filter was then moved along one element of the input sequence and the process was repeated; specifically, the same filter was applied to the input sequence at indexes 1, 2, and 3, which also resulted in a zero output in the feature map.

$$[0, 1, 0] \cdot [0, 0, 1] = 0 \quad (11.2)$$

We are being systematic, so again, the filter is moved along one more element of the input and applied to the input at indexes 2, 3, and 4. This time the output is a value of one in the feature map. We detected the feature and activated appropriately.

$$[0, 1, 0] \cdot [0, 1, 1] = 1 \quad (11.3)$$

The process is repeated until we calculate the entire feature map. Note that the feature map has six elements, whereas our input has eight elements. This is an artefact of how the filter was applied to the input sequence. For the filter to be applied to elements 7 and 8, there would have to be an element 9. For the filter to be applied to element 8, there would have to be elements 9 and 10. There are other ways to apply the filter to the input sequence that changes the shape of the resulting feature map, such as padding, but we will not discuss these methods in this tutorial. You can imagine that with different inputs, we may detect the feature with more or less intensity, and with different weights in the filter, that we would detect different features in the input sequence.

11.5.2 Example of 2D Convolutional Layer

We can expand the bump detection example in the previous section to a vertical line detector in a two-dimensional image. Again, we can constrain the input, in this case to a square 8×8 pixel input image with a single channel (e.g. grayscale) with a single vertical line in the middle.

```
[0, 0, 0, 1, 1, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 0, 0]
```

Listing 11.13: Example of a two-dimensional image with a vertical line.

The input to a `Conv2D` layer must be four-dimensional. The first dimension defines the samples; in this case, there is only a single sample. The second dimension defines the number of rows; in this case, eight. The third dimension defines the number of columns, again eight in this case, and finally the number of channels, which is one in this case. Therefore, the input must have the four-dimensional shape `[samples, columns, rows, channels]` or `[1, 8, 8, 1]` in this case.

```
...
# define input data
data = [[0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
```

```
[0, 0, 0, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 0, 0, 0],
[0, 0, 0, 1, 1, 0, 0, 0]
data = asarray(data)
data = data.reshape(1, 8, 8, 1)
```

Listing 11.14: Example of defining the image as a sample.

We will define the Conv2D layer with a single filter as we did in the previous section with the Conv1D example. The filter will be two-dimensional and square with the shape 3×3 . The layer will expect input samples to have the shape [columns, rows, channels] or [8,8,1].

```
...
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), input_shape=(8, 8, 1)))
```

Listing 11.15: Example of defining a model with a Conv2D layer and a single filter.

We will define a vertical line detector filter to detect the single vertical line in our input data. The filter looks as follows:

```
0, 1, 0
0, 1, 0
0, 1, 0
```

Listing 11.16: Example of a 3×3 vertical line detecting filter.

We can implement this as follows:

```
...
# define a vertical line detector
detector = [[[0],[1],[0]],
            [[0],[1],[0]],
            [[0],[1],[0]]]
weights = [asarray(detector), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
# confirm they were stored
print(model.get_weights())
```

Listing 11.17: Example of manually defining the weights for the vertical line detecting filter.

Finally, we will apply the filter to the input image, which will result in a feature map that we would expect to show the detection of the vertical line in the input image.

```
...
# apply filter to input data
yhat = model.predict(data)
```

Listing 11.18: Example of applying a filter to the input sample.

The shape of the feature map output will be four-dimensional with the shape [batch, rows, columns, filters]. We will be performing a single batch and we have a single filter (one filter and one input channel), therefore the output or feature map shape is [1, 6, 6, 1]. We can pretty-print the content of the single feature map as follows:

```

...
for r in range(yhat.shape[1]):
    # print each column in the row
    print([yhat[0,r,c,0] for c in range(yhat.shape[2])])

```

Listing 11.19: Example of summarizing the output from applying the filter.

Tying all of this together, the complete example is listed below.

```

# example of calculation 2d convolutions
from numpy import asarray
from keras.models import Sequential
from keras.layers import Conv2D
# define input data
data = [[0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0]]
data = asarray(data)
data = data.reshape(1, 8, 8, 1)
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), input_shape=(8, 8, 1)))
# define a vertical line detector
detector = [[[0],[1],[0]],
            [[0],[1],[0]],
            [[0],[1],[0]]]
weights = [asarray(detector), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
# confirm they were stored
print(model.get_weights())
# apply filter to input data
yhat = model.predict(data)
for r in range(yhat.shape[1]):
    # print each column in the row
    print([yhat[0,r,c,0] for c in range(yhat.shape[2])])

```

Listing 11.20: Example of manually applying a two-dimensional filter.

Running the example first confirms that the handcrafted filter was correctly defined in the layer weights. Next, the calculated feature map is printed. We can see from the scale of the numbers that indeed the filter has detected the single vertical line with strong activation in the middle of the feature map.

```
[array([[[0.],
       [[1.],
        [[0.]]],
      [[[0.],
        [[1.],
         [[0.]]],
       [[[0.]]],
```

```
[[1.],
 [[0.]]], dtype=float32), array([0.], dtype=float32)

[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
```

Listing 11.21: Example output from manually applying a two-dimensional filter.

Let's take a closer look at what was calculated. First, the filter was applied to the top left corner of the image, or an image patch of 3×3 elements. Technically, the image patch is three dimensional with a single channel, and the filter has the same dimensions. We cannot implement this in NumPy using the `dot()` function, instead, we must use the `tensordot()` function so we can appropriately sum across all dimensions, for example:

```
# example of manually applying a 2d filter.
from numpy import asarray
from numpy import tensordot
m1 = asarray([[0, 1, 0],
              [0, 1, 0],
              [0, 1, 0]])
m2 = asarray([[0, 0, 0],
              [0, 0, 0],
              [0, 0, 0]])
print(tensordot(m1, m2))
```

Listing 11.22: Example of manually applying a two-dimensional filter.

This calculation results in a single output value of 0.0, e.g., the feature was not detected. This gives us the first element in the top-left corner of the feature map. Manually, this would be as follows:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = 0 \quad (11.4)$$

The filter is moved along one column to the left and the process is repeated. Again, the feature is not detected.

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} = 0 \quad (11.5)$$

One more move to the left to the next column and the feature is detected for the first time, resulting in a strong activation.

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} = 3 \quad (11.6)$$

This process is repeated until the right edge of the filter rests against the right edge or final column of the input image. This gives the last element in the first full row of the feature map.

```
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
```

Listing 11.23: Example output from manually calculating a two-dimensional filter.

The filter then moves down one row and back to the first column and the process is repeated from left to right to give the second row of the feature map. And on until the bottom of the filter rests on the bottom or last row of the input image. Again, as with the previous section, we can see that the feature map is a 6×6 matrix, smaller than the 8×8 input image because of the limitations of how the filter can be applied to the input image.

11.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Horizontal Line Filter.** Manually calculate the effect of a horizontal line detector filter on an input image with a horizontal line.
- **Filter Size.** Update the example to use a smaller or larger filter size and review the effect.
- **Custom Filter.** Update the examples to use your own custom filter design and a test image to demonstrate its capability.

If you explore any of these extensions, I'd love to know.

11.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

11.7.1 Books

- Chapter 9: Convolutional Networks, *Deep Learning*, 2016.
<https://amzn.to/2Dl124s>
- Chapter 5: Deep Learning for Computer Vision, *Deep Learning with Python*, 2017.
<https://amzn.to/2Dnshvc>

11.7.2 API

- Keras Convolutional Layers API.
<https://keras.io/layers/convolutional/>
- `numpy.asarray` API.
<https://docs.scipy.org/doc/numpy/reference/generated/numpy.asarray.html>

11.8 Summary

In this tutorial, you discovered how convolutions work in the convolutional neural network. Specifically, you learned:

- Convolutional neural networks apply a filter to an input to create a feature map that summarizes the presence of detected features in the input.
- Filters can be handcrafted, such as line detectors, but the innovation of convolutional neural networks is to learn the filters during training in the context of a specific prediction problem.
- How to calculate the feature map for one- and two-dimensional convolutional layers in a convolutional neural network.

11.8.1 Next

In the next section, you will discover how the number of filters, padding and stride impact the processing of image data by convolutional layers.

Chapter 12

How to Use Filter Size, Padding, and Stride

The convolutional layer in convolutional neural networks systematically applies filters to an input and creates output feature maps. Although the convolutional layer is very simple, it is capable of achieving sophisticated and impressive results. Nevertheless, it can be challenging to develop an intuition for how the shape of the filters impacts the shape of the output feature map and how related configuration hyperparameters such as padding and stride should be configured. In this tutorial, you will discover an intuition for filter size, the need for padding, and stride in convolutional neural networks. After completing this tutorial, you will know:

- How filter size or kernel size impacts the shape of the output feature map.
- How the filter size creates a border effect in the feature map and how it can be overcome with padding.
- How the stride of the filter on the input image can be used to downsample the size of the output feature map.

Let's get started.

12.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Convolutional Layer
2. Problem of Border Effects
3. Effect of Filter Size (Kernel Size)
4. Fix the Border Effect Problem With Padding
5. Downsample Input With Stride

12.2 Convolutional Layer

In a convolutional neural network, a convolutional layer is responsible for the systematic application of one or more filters to an input. The multiplication of the filter to the input image results in a single output. The input is typically three-dimensional images (e.g. rows, columns and channels), and in turn, the filters are also three-dimensional with the same number of channels and fewer rows and columns than the input image. As such, the filter is repeatedly applied to each part of the input image, resulting in a two-dimensional output map of activations, called a feature map. Keras provides an implementation of the convolutional layer called a `Conv2D` layer.

It requires that you specify the expected shape of the input images in terms of rows (height), columns (width), and channels (depth) or `[rows, columns, channels]`. The filter contains the weights that must be learned during the training of the layer. The filter weights represent the structure or feature that the filter will detect and the strength of the activation indicates the degree to which the feature was detected. The layer requires that both the number of filters be specified and that the shape of the filters be specified. We can demonstrate this with a small example (intentionally based on the example from Chapter 11). In this example, we define a single input image or sample that has one channel and is an eight pixel by eight pixel square with all 0 values and a two-pixel wide vertical line in the center.

```
...
# define input data
data = [[0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0]]
data = asarray(data)
data = data.reshape(1, 8, 8, 1)
```

Listing 12.1: Example of defining the image as a sample.

Next, we can define a model that expects input samples to have the shape `(8, 8, 1)` and has a single hidden convolutional layer with a single filter with the shape of three pixels by three pixels.

```
...
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), input_shape=(8, 8, 1)))
# summarize model
model.summary()
```

Listing 12.2: Example of defining a model with a `Conv2D` layer and a single filter.

The filter is initialized with random weights as part of the initialization of the model. We will overwrite the random weights and hard code our own 3×3 filter that will detect vertical lines. That is, the filter will strongly activate when it detects a vertical line and weakly activate when it does not. We expect that by applying this filter across the input image, the output feature map will show that the vertical line was detected.

```

...
# define a vertical line detector
detector = [[[0],[[1],[[0]]],  

            [[0],[[1],[[0]]],  

             [[0],[[1],[[0]]]]  

weights = [asarray(detector), asarray([0.0])]  

# store the weights in the model  

model.set_weights(weights)

```

Listing 12.3: Example of manually defining the weights for the vertical line detecting filter.

Next, we can apply the filter to our input image by calling the `predict()` function on the model.

```

...
# apply filter to input data
yhat = model.predict(data)

```

Listing 12.4: Example of applying a filter to the input sample.

The result is a four-dimensional output with one batch, a given number of rows and columns, and one filter, or `[batch, rows, columns, filters]`. We can print the activations in the single feature map to confirm that the line was detected.

```

...
# enumerate rows
for r in range(yhat.shape[1]):  

    # print each column in the row
    print([yhat[0,r,c,0] for c in range(yhat.shape[2])])

```

Listing 12.5: Example of summarizing the output from applying the filter.

Tying all of this together, the complete example is listed below.

```

# example of using a single convolutional layer
from numpy import asarray
from keras.models import Sequential
from keras.layers import Conv2D
# define input data
data = [[0, 0, 0, 1, 1, 0, 0, 0],  

        [0, 0, 0, 1, 1, 0, 0, 0],  

        [0, 0, 0, 1, 1, 0, 0, 0],  

        [0, 0, 0, 1, 1, 0, 0, 0],  

        [0, 0, 0, 1, 1, 0, 0, 0],  

        [0, 0, 0, 1, 1, 0, 0, 0],  

        [0, 0, 0, 1, 1, 0, 0, 0],  

        [0, 0, 0, 1, 1, 0, 0, 0]]  

data = asarray(data)
data = data.reshape(1, 8, 8, 1)
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), input_shape=(8, 8, 1)))
# summarize model
model.summary()
# define a vertical line detector
detector = [[[0],[[1],[[0]]],  

            [[0],[[1],[[0]]],  

             [[0],[[1],[[0]]]]]

```

```

[[[0]], [[1]], [[0]]]
weights = [asarray(detector), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
# apply filter to input data
yhat = model.predict(data)
# enumerate rows
for r in range(yhat.shape[1]):
    # print each column in the row
    print([yhat[0,r,c,0] for c in range(yhat.shape[2])])

```

Listing 12.6: Example of manually applying a two-dimensional filter.

Running the example first summarizes the structure of the model. Of note is that the single hidden convolutional layer will take the 8×8 pixel input image and will produce a feature map with the dimensions of 6×6 . We will go into why this is the case in the next section. We can also see that the layer has 10 parameters, that is nine weights for the filter (3×3) and one weight for the bias. Finally, the feature map is printed. We can see from reviewing the numbers in the 6×6 matrix that indeed the manually specified filter detected the vertical line in the middle of our input image.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 6, 6, 1)	10

Total params: 10
Trainable params: 10
Non-trainable params: 0

```

[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]

```

Listing 12.7: Example output from manually applying a two-dimensional filter.

12.3 Problem of Border Effects

In the previous section, we defined a single filter with the size of three pixels high and three pixels wide (rows, columns). We saw that the application of the 3×3 filter, referred to as the kernel size in Keras, to the 8×8 input image resulted in a feature map with the size of 6×6 . That is, the input image with 64 pixels was reduced to a feature map with 36 pixels. Where did the other 28 pixels go?

The filter is applied systematically to the input image. It starts at the top left corner of the image and is moved from left to right one pixel column at a time until the edge of the filter reaches the edge of the image. For a 3×3 pixel filter applied to a 8×8 input image, we can see that it can only be applied six times, resulting in the width of six in the output feature

map. For example, let's work through each of the first six patches of the input image (left) dot product (. operator) the filter (right):

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} = 0 \quad (12.1)$$

Moved right one pixel:

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} = 0 \quad (12.2)$$

Moved right one pixel:

$$\begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} = 3 \quad (12.3)$$

Moved right one pixel:

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} = 3 \quad (12.4)$$

Moved right one pixel:

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} = 0 \quad (12.5)$$

Moved right one pixel:

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} = 0 \quad (12.6)$$

That gives us the first row and each column of the output feature map:

```
0.0, 0.0, 3.0, 3.0, 0.0, 0.0
```

Listing 12.8: Example output from manually calculating a two-dimensional filter.

The reduction in the size of the input to the feature map is referred to as border effects. It is caused by the interaction of the filter with the border of the image. This is often not a problem for large images and small filters but can be a problem with small images. It can also become a problem once a number of convolutional layers are stacked. For example, below is the same model updated to have two stacked convolutional layers. This means that a 3×3 filter is applied to the 8×8 input image to result in a 6×6 feature map as in the previous section. A 3×3 filter is then applied to the 6×6 feature map.

```
# example of stacked convolutional layers
from keras.models import Sequential
from keras.layers import Conv2D
```

```
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), input_shape=(8, 8, 1)))
model.add(Conv2D(1, (3,3)))
# summarize model
model.summary()
```

Listing 12.9: Example of summarizing a model with stacked Conv2D layers.

Running the example summarizes the shape of the output from each layer. We can see that the application of filters to the feature map output of the first layer, in turn, results in a 4×4 feature map. This can become a problem as we develop very deep convolutional neural network models with tens or hundreds of layers. We will simply run out of data in our feature maps upon which to operate.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 6, 6, 1)	10
conv2d_2 (Conv2D)	(None, 4, 4, 1)	10

Total params: 20
Trainable params: 20
Non-trainable params: 0

Listing 12.10: Example output from summarizing a model with stacked Conv2D layers.

12.4 Effect of Filter Size (Kernel Size)

Different sized filters will detect different sized features in the input image and, in turn, will result in differently sized feature maps. It is common to use 3×3 sized filters, and perhaps 5×5 or even 7×7 sized filters, for larger input images. For example, below is an example of the model with a single filter updated to use a filter size of 5×5 pixels.

```
# example of a convolutional layer
from keras.models import Sequential
from keras.layers import Conv2D
# create model
model = Sequential()
model.add(Conv2D(1, (5,5), input_shape=(8, 8, 1)))
# summarize model
model.summary()
```

Listing 12.11: Example of summarizing a model with a 5×5 kernel.

Running the example demonstrates that the 5×5 filter can only be applied to the 8×8 input image 4 times, resulting in a 4×4 feature map output.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 4, 4, 1)	26

```
=====
Total params: 26
Trainable params: 26
Non-trainable params: 0
=====
```

Listing 12.12: Example output from summarizing a model with a 5×5 kernel.

It may help to further develop the intuition of the relationship between filter size and the output feature map to look at two extreme cases. The first is a filter with the size of 1×1 pixels (for more on 1×1 filters, see Chapter 16).

```
# example of a convolutional layer
from keras.models import Sequential
from keras.layers import Conv2D
# create model
model = Sequential()
model.add(Conv2D(1, (1,1), input_shape=(8, 8, 1)))
# summarize model
model.summary()
```

Listing 12.13: Example of summarizing a model with a 1×1 kernel.

Running the example demonstrates that the output feature map has the same size as the input, specifically 8×8 . This is because the filter only has a single weight (and a bias).

```
=====
Layer (type)          Output Shape         Param #
=====
conv2d_1 (Conv2D)     (None, 8, 8, 1)      2
=====
Total params: 2
Trainable params: 2
Non-trainable params: 0
=====
```

Listing 12.14: Example output from summarizing a model with a 1×1 kernel.

The other extreme is a filter with the same size as the input, in this case, 8×8 pixels.

```
# example of a convolutional layer
from keras.models import Sequential
from keras.layers import Conv2D
# create model
model = Sequential()
model.add(Conv2D(1, (8,8), input_shape=(8, 8, 1)))
# summarize model
model.summary()
```

Listing 12.15: Example of summarizing a model with a 8×8 kernel.

Running the example, we can see that, as you might expect, there is one filter weight for each pixel in the input image ($64 + 1$ for the bias) and that the output is a feature map with a single pixel.

```
=====
Layer (type)          Output Shape         Param #
=====
```

```
conv2d_1 (Conv2D)           (None, 1, 1, 1)      65
=====
Total params: 65
Trainable params: 65
Non-trainable params: 0
=====
```

Listing 12.16: Example output from summarizing a model with a 8×8 kernel.

Now that we are familiar with the effect of filter sizes on the size of the resulting feature map, let's look at how we can stop losing pixels.

12.5 Fix the Border Effect Problem With Padding

By default, a filter starts at the left of the image with the left-hand side of the filter sitting on the far left pixels of the image. The filter is then stepped across the image one column at a time until the right-hand side of the filter is sitting on the far right pixels of the image. An alternative approach to applying a filter to an image is to ensure that each pixel in the image is given an opportunity to be at the center of the filter. By default, this is not the case, as the pixels on the edge of the input are only ever exposed to the edge of the filter. By starting the filter outside the frame of the image, it gives the pixels on the border of the image more of an opportunity for interacting with the filter, more of an opportunity for features to be detected by the filter, and in turn, an output feature map that has the same shape as the input image.

For example, in the case of applying a 3×3 filter to the 8×8 input image, we can add a border of one pixel around the outside of the image. This has the effect of artificially creating a 10×10 input image. When the 3×3 filter is applied, it results in an 8×8 feature map. The added pixel values could have the value zero value that has no effect with the dot product operation when the filter is applied.

$$\begin{pmatrix} x & x & x \\ x & 0 & 0 \\ x & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} = 0 \quad (12.7)$$

The addition of pixels to the edge of the image is called padding. In Keras, this is specified via the `padding` argument on the `Conv2D` layer, which has the default value of '`valid`' (no padding). This means that the filter is applied only to valid ways to the input. The `padding` value of '`same`' calculates and adds the padding required to the input image (or feature map) to ensure that the output has the same shape as the input. The example below adds padding to the convolutional layer in our worked example.

```
# example a convolutional layer with padding
from keras.models import Sequential
from keras.layers import Conv2D
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), padding='same', input_shape=(8, 8, 1)))
# summarize model
model.summary()
```

Listing 12.17: Example of summarizing a model with padding.

Running the example demonstrates that the shape of the output feature map is the same as the input image: that the padding had the desired effect.

```

Layer (type)          Output Shape         Param #
=====
conv2d_1 (Conv2D)    (None, 8, 8, 1)      10
=====
Total params: 10
Trainable params: 10
Non-trainable params: 0
=====
```

Listing 12.18: Example output from summarizing a model with padding.

The addition of padding allows the development of very deep models in such a way that the feature maps do not dwindle away to nothing. The example below demonstrates this with three stacked convolutional layers.

```

# example a deep cnn with padding
from keras.models import Sequential
from keras.layers import Conv2D
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), padding='same', input_shape=(8, 8, 1)))
model.add(Conv2D(1, (3,3), padding='same'))
model.add(Conv2D(1, (3,3), padding='same'))
# summarize model
model.summary()
```

Listing 12.19: Example of summarizing a stacked model with padding.

Running the example, we can see that with the addition of padding, the shape of the output feature maps remains fixed at 8×8 even three layers deep.

```

Layer (type)          Output Shape         Param #
=====
conv2d_1 (Conv2D)    (None, 8, 8, 1)      10
=====
conv2d_2 (Conv2D)    (None, 8, 8, 1)      10
=====
conv2d_3 (Conv2D)    (None, 8, 8, 1)      10
=====
Total params: 30
Trainable params: 30
Non-trainable params: 0
=====
```

Listing 12.20: Example output from summarizing a stacked model with padding.

12.6 Downsample Input With Stride

The filter is moved across the image left to right, top to bottom, with a one-pixel column change on the horizontal movements, then a one-pixel row change on the vertical movements. The

amount of movement between applications of the filter to the input image is referred to as the stride, and it is almost always symmetrical in height and width dimensions. The default stride or strides in two dimensions is $(1, 1)$ for the height and the width movement, performed when needed. And this default works well in most cases. The stride can be changed, which has an effect both on how the filter is applied to the image and, in turn, the size of the resulting feature map.

For example, the stride can be changed to $(2, 2)$. This has the effect of moving the filter two pixels left for each horizontal movement of the filter and two pixels down for each vertical movement of the filter when creating the feature map. We can demonstrate this with an example using the 8×8 image with a vertical line (left) dot product (. operator) with the vertical line filter (right) with a stride of two pixels:

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} = 0 \quad (12.8)$$

Moved right two pixels:

$$\begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} = 3 \quad (12.9)$$

Moved right two pixels:

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} = 0 \quad (12.10)$$

We can see that there are only three valid applications of the 3×3 filters to the 8×8 input image with a stride of two. This will be the same in the vertical dimension. This has the effect of applying the filter in such a way that the normal feature map output (6×6) is down-sampled so that the size of each dimension is reduced by half (3×3), resulting in $\frac{1}{4}$ the number of pixels (36 pixels down to 9). The stride can be specified in Keras on the `Conv2D` layer via the `stride` argument and specified as a tuple with height and width. The example demonstrates the application of our manual vertical line filter on the 8×8 input image with a convolutional layer that has a stride of two.

```
# example of vertical line filter with a stride of 2
from numpy import asarray
from keras.models import Sequential
from keras.layers import Conv2D
# define input data
data = [[0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0]]
data = asarray(data)
data = data.reshape(1, 8, 8, 1)
```

```

# create model
model = Sequential()
model.add(Conv2D(1, (3,3), strides=(2, 2), input_shape=(8, 8, 1)))
# summarize model
model.summary()
# define a vertical line detector
detector = [[[[0]],[[1]],[[0]]],
            [[[0]],[[1]],[[0]]],
            [[[0]],[[1]],[[0]]]]
weights = [asarray(detector), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
# apply filter to input data
yhat = model.predict(data)
# enumerate rows
for r in range(yhat.shape[1]):
    # print each column in the row
    print([yhat[0,r,c,0] for c in range(yhat.shape[2])])

```

Listing 12.21: Example of applying a manual filter with a larger stride.

Running the example, we can see from the summary of the model that the shape of the output feature map will be 3×3 . Applying the handcrafted filter to the input image and printing the resulting activation feature map, we can see that, indeed, the filter still detected the vertical line, and can represent this finding with less information. Downsampling may be desirable in some cases where deeper knowledge of the filters used in the model or of the model architecture allows for some compression in the resulting feature maps.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 3, 3, 1)	10

Total params: 10
Trainable params: 10
Non-trainable params: 0

[0.0, 3.0, 0.0]
[0.0, 3.0, 0.0]
[0.0, 3.0, 0.0]

Listing 12.22: Example output from applying a manual filter with a larger stride.

12.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Padding Size.** Calculate the amount of padding required with different filter sizes for a fixed sized image input.
- **Stride Size.** Calculate the stride ranges for different filter sizes to ensure that there is always some overlap when processing an input image.

- **Padding and Stride.** Experiment with combinations of padding and differently sized strides and review the effect on summary of the model.

If you explore any of these extensions, I'd love to know.

12.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

12.8.1 Books

- Chapter 9: Convolutional Networks, *Deep Learning*, 2016.
<https://amzn.to/2Dl124s>
- Chapter 5: Deep Learning for Computer Vision, *Deep Learning with Python*, 2017.
<https://amzn.to/2Dnshvc>

12.8.2 API

- Keras Convolutional Layers API.
<https://keras.io/layers/convolutional/>

12.9 Summary

In this tutorial, you discovered an intuition for filter size, the need for padding, and stride in convolutional neural networks. Specifically, you learned:

- How filter size or kernel size impacts the shape of the output feature map.
- How the filter size creates a border effect in the feature map and how it can be overcome with padding.
- How the stride of the filter on the input image can be used to downsample the size of the output feature map.

12.9.1 Next

In the next section, you will discover how pooling layers work in convolutional neural networks.

Chapter 13

How Pooling Layers Work

Convolutional layers in a convolutional neural network summarize the presence of features in an input image. A problem with the output feature maps is that they are sensitive to the location of the features in the input. One approach to address this sensitivity is to downsample the feature maps. This has the effect of making the resulting downsampled feature maps more robust to changes in the position of the feature in the image, referred to by the technical phrase *local translation invariance*. Pooling layers provide an approach to down sampling feature maps by summarizing the presence of features in patches of the feature map. Two common pooling methods are average pooling and max pooling that summarize the average presence of a feature and the most activated presence of a feature respectively. In this tutorial, you will discover how the pooling operation works and how to implement it in convolutional neural networks. After completing this tutorial, you will know:

- Pooling can be used to downsample the detection of features in feature maps.
- How to calculate and implement average and maximum pooling in a convolutional neural network.
- How to use global pooling in a convolutional neural network.

Let's get started.

13.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Pooling Layers
2. Detecting Vertical Lines
3. Average Pooling Layers
4. Max Pooling Layers
5. Global Pooling Layers

13.2 Pooling Layers

Convolutional layers in a convolutional neural network systematically apply learned filters to input images in order to create feature maps that summarize the presence of those features in the input. Convolutional layers prove very effective, and stacking convolutional layers in deep models allows layers close to the input to learn low-level features (e.g. lines) and layers deeper in the model to learn high-order or more abstract features, like shapes or specific objects. A limitation of the feature map output of convolutional layers is that they record the precise position of features in the input. This means that small movements in the position of the feature in the input image will result in a different feature map. This can happen with re-cropping, rotation, shifting, and other minor changes to the input image.

A common approach to addressing this problem from signal processing is called down sampling. This is where a lower resolution version of an input signal is created that still contains the large or important structural elements, without the fine detail that may not be as useful to the task. Down sampling can be achieved with convolutional layers by changing the stride of the convolution across the image. A more robust and common approach is to use a pooling layer. A pooling layer is a new layer added after the convolutional layer. Specifically, after a nonlinearity (e.g. ReLU) has been applied to the feature maps output by a convolutional layer; for example the layers in a model may look as follows:

1. Input Image.
2. Convolutional Layer.
3. Nonlinearity.
4. Pooling Layer.

The addition of a pooling layer after the convolutional layer is a common pattern used for ordering layers within a convolutional neural network that may be repeated one or more times in a given model. The pooling layer operates upon each feature map separately to create a new set of the same number of pooled feature maps. Pooling involves selecting a pooling operation, much like a filter to be applied to feature maps. The size of the pooling operation or filter is smaller than the size of the feature map; specifically, it is almost always 2×2 pixels applied with a stride of 2 pixels.

This means that the pooling layer will always reduce the size of each feature map by a factor of 2, e.g. each dimension is halved, reducing the number of pixels or values in each feature map to one quarter the size. For example, a pooling layer applied to a feature map of 6×6 (36 pixels) will result in an output pooled feature map of 3×3 (9 pixels). The pooling operation is specified, rather than learned. Two common functions used in the pooling operation are:

- **Average Pooling:** Calculate the average value for each patch on the feature map.
- **Maximum Pooling** (or Max Pooling): Calculate the maximum value for each patch of the feature map.

The result of using a pooling layer and creating downsampled or pooled feature maps is a summarized version of the features detected in the input. They are useful as small changes

in the location of the feature in the input detected by the convolutional layer will result in a pooled feature map with the feature in the same location. This capability added by pooling is called the model's invariance to local translation.

In all cases, pooling helps to make the representation become approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change.

— Page 342, *Deep Learning*, 2016.

Now that we are familiar with the need and benefit of pooling layers, let's look at some specific examples.

13.3 Detecting Vertical Lines

Before we look at some examples of pooling layers and their effects, let's develop a small example of an input image and convolutional layer to which we can later add and evaluate pooling layers (intentionally based on the example from Chapter 11). In this example, we define a single input image or sample that has one channel and is an 8 pixel by 8 pixel square with all 0 values and a two-pixel wide vertical line in the center.

```
...
# define input data
data = [[0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0]]
data = asarray(data)
data = data.reshape(1, 8, 8, 1)
```

Listing 13.1: Example of defining the image as a sample.

Next, we can define a model that expects input samples to have the shape (8, 8, 1) and has a single hidden convolutional layer with a single filter with the shape of 3 pixels by 3 pixels. A rectified linear activation function, or ReLU for short, is then applied to each value in the feature map. This is a simple and effective nonlinearity, that in this case will not change the values in the feature map, but is present because we will later add subsequent pooling layers and pooling is added after the nonlinearity applied to the feature maps, e.g. a best practice.

```
...
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), activation='relu', input_shape=(8, 8, 1)))
# summarize model
model.summary()
```

Listing 13.2: Example of defining a model with a Conv2D layer and a single filter.

The filter is initialized with random weights as part of the initialization of the model. Instead, we will hard code our own 3×3 filter that will detect vertical lines. That is, the filter will strongly activate when it detects a vertical line and weakly activate when it does not. We expect that by applying this filter across the input image that the output feature map will show that the vertical line was detected.

```
...
# define a vertical line detector
detector = [[[0],[1],[0]],
            [[0],[1],[0]],
            [[0],[1],[0]]]
weights = [asarray(detector), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
```

Listing 13.3: Example of manually defining the weights for the vertical line detecting filter.

Next, we can apply the filter to our input image by calling the `predict()` function on the model.

```
...
# apply filter to input data
yhat = model.predict(data)
```

Listing 13.4: Example of applying a filter to the input sample.

The result is a four-dimensional output with one batch, a given number of rows and columns, and one filter, or [batch, rows, columns, filters]. We can print the activations in the single feature map to confirm that the line was detected.

```
...
# enumerate rows
for r in range(yhat.shape[1]):
    # print each column in the row
    print([yhat[0,r,c,0] for c in range(yhat.shape[2])])
```

Listing 13.5: Example of summarizing the output from applying the filter.

Tying all of this together, the complete example is listed below.

```
# example of vertical line detection with a convolutional layer
from numpy import asarray
from keras.models import Sequential
from keras.layers import Conv2D
# define input data
data = [[0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0]]
data = asarray(data)
data = data.reshape(1, 8, 8, 1)
# create model
model = Sequential()
```

```

model.add(Conv2D(1, (3,3), activation='relu', input_shape=(8, 8, 1)))
# summarize model
model.summary()
# define a vertical line detector
detector = [[[0],[1],[0]],
            [[0],[1],[0]],
            [[0],[1],[0]]]
weights = [asarray(detector), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
# apply filter to input data
yhat = model.predict(data)
# enumerate rows
for r in range(yhat.shape[1]):
    # print each column in the row
    print([yhat[0,r,c,0] for c in range(yhat.shape[2])])

```

Listing 13.6: Example of manually applying a two-dimensional filter.

Running the example first summarizes the structure of the model. Of note is that the single hidden convolutional layer will take the 8×8 pixel input image and will produce a feature map with the dimensions of 6×6 . We can also see that the layer has 10 parameters: that is nine weights for the filter (3×3) and one weight for the bias. Finally, the single feature map is printed. We can see from reviewing the numbers in the 6×6 matrix that indeed the manually specified filter detected the vertical line in the middle of our input image.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 6, 6, 1)	10

Total params: 10
Trainable params: 10
Non-trainable params: 0

[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]

Listing 13.7: Example output from manually applying a two-dimensional filter.

We can now look at some common approaches to pooling and how they impact the output feature maps.

13.4 Average Pooling Layer

On two-dimensional feature maps, pooling is typically applied in 2×2 patches of the feature map with a stride of $(2,2)$. Average pooling involves calculating the average for each patch of the feature map. This means that each 2×2 square of the feature map is downsampled to the average value in the square. For example, the output of the line detector convolutional filter

in the previous section was a 6×6 feature map. We can look at applying the average pooling operation to the first line of patches of that feature map manually. The first *line* of pooling input (first two rows and six columns) of the output feature map were as follows:

```
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
```

Listing 13.8: Example activation from the feature map.

The first pooling operation is applied as follows:

$$\text{average} \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} = 0 \quad (13.1)$$

Given the stride of two, the operation is moved along two columns to the left and the average is calculated:

$$\text{average} \begin{pmatrix} 3 & 3 \\ 3 & 3 \end{pmatrix} = 3 \quad (13.2)$$

Again, the operation is moved along two columns to the left and the average is calculated:

$$\text{average} \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} = 0 \quad (13.3)$$

That's it for the first line of pooling operations. The result is the first line of the average pooling operation:

```
[0.0, 3.0, 0.0]
...
```

Listing 13.9: Example output manually calculating average pooling.

Given the (2,2) stride, the operation would then be moved down two rows and back to the first column and the process continued. Because the downsampling operation halves each dimension, we will expect the output of pooling applied to the 6×6 feature map to be a new 3×3 feature map. Given the horizontal symmetry of the feature map input, we would expect each row to have the same average pooling values. Therefore, we would expect the resulting average pooling of the detected line feature map from the previous section to look as follows:

```
[0.0, 3.0, 0.0]
[0.0, 3.0, 0.0]
[0.0, 3.0, 0.0]
```

Listing 13.10: Example output manually calculating average pooling to the entire feature map.

We can confirm this by updating the example from the previous section to use average pooling. This can be achieved in Keras by using the `AveragePooling2D` layer. The default `pool_size` (e.g. like the kernel size or filter size) of the layer is (2,2) and the default strides is None, which in this case means using the `pool_size` as the strides, which will be (2,2).

```
...
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), activation='relu', input_shape=(8, 8, 1)))
model.add(AveragePooling2D())
```

Listing 13.11: Example of defining a model with one filter and average pooling.

The complete example with average pooling is listed below.

```
# example of average pooling
from numpy import asarray
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import AveragePooling2D
# define input data
data = [[0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0]]
data = asarray(data)
data = data.reshape(1, 8, 8, 1)
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), activation='relu', input_shape=(8, 8, 1)))
model.add(AveragePooling2D())
# summarize model
model.summary()
# define a vertical line detector
detector = [[[0],[1],[0]],
             [[0],[1],[0]],
             [[0],[1],[0]]]
weights = [asarray(detector), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
# apply filter to input data
yhat = model.predict(data)
# enumerate rows
for r in range(yhat.shape[1]):
    # print each column in the row
    print([yhat[0,r,c,0] for c in range(yhat.shape[2])])
```

Listing 13.12: Example of manually applying a two-dimensional filter with average pooling.

Running the example first summarizes the model. We can see from the model summary that the input to the pooling layer will be a single feature map with the shape (6,6) and that the output of the average pooling layer will be a single feature map with each dimension halved, with the shape (3,3). Applying the average pooling results in a new feature map that still detects the line, although in a downsampled manner, exactly as we expected from calculating the operation manually.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 6, 6, 1)	10
average_pooling2d_1 (Average)	(None, 3, 3, 1)	0

Total params: 10
Trainable params: 10

```
Non-trainable params: 0
-----
[0.0, 3.0, 0.0]
[0.0, 3.0, 0.0]
[0.0, 3.0, 0.0]
```

Listing 13.13: Example output from manually applying a two-dimensional filter with average pooling.

Average pooling works well, although it is more common to use max pooling.

13.5 Max Pooling Layer

Maximum pooling, or max pooling, is a pooling operation that calculates the maximum, or largest, value in each patch of each feature map. The results are downsampled or pooled feature maps that highlight the most present feature in the patch, not the average presence of the features, as in the case of average pooling. This has been found to work better in practice than average pooling for computer vision tasks like image classification.

In a nutshell, the reason is that features tend to encode the spatial presence of some pattern or concept over the different tiles of the feature map (hence, the term feature map), and it's more informative to look at the maximal presence of different features than at their average presence.

— Page 129, *Deep Learning with Python*, 2017.

We can make the max pooling operation concrete by again applying it to the output feature map of the line detector convolutional operation and manually calculate the first row of the pooled feature map. The first *line* of pooling input (first two rows and six columns) of the output feature map were as follows:

```
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
```

Listing 13.14: Example activation from the feature map.

The first max pooling operation is applied as follows:

$$\max \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} = 0 \quad (13.4)$$

Given the stride of two, the operation is moved along two columns to the left and the max is calculated:

$$\max \begin{pmatrix} 3 & 3 \\ 3 & 3 \end{pmatrix} = 3 \quad (13.5)$$

Again, the operation is moved along two columns to the left and the max is calculated:

$$\max \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} = 0 \quad (13.6)$$

That's it for the first line of pooling operations. The result is the first line of the max pooling operation:

```
[0.0, 3.0, 0.0]
...

```

Listing 13.15: Example output manually calculating max pooling.

Again, given the horizontal symmetry of the feature map provided for pooling, we would expect the pooled feature map to look as follows:

```
[0.0, 3.0, 0.0]
[0.0, 3.0, 0.0]
[0.0, 3.0, 0.0]
```

Listing 13.16: Example output manually calculating max pooling to the entire feature map.

It just so happens that the chosen line detector image and feature map produce the same output when downsampled with average pooling and maximum pooling. The maximum pooling operation can be added to the worked example by adding the `MaxPooling2D` layer provided by the Keras API.

```
...
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), activation='relu', input_shape=(8, 8, 1)))
model.add(MaxPooling2D())
```

Listing 13.17: Example of defining a model with one filter and max pooling.

The complete example of vertical line detection with max pooling is listed below.

```
# example of max pooling
from numpy import asarray
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
# define input data
data = [[0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0]]
data = asarray(data)
data = data.reshape(1, 8, 8, 1)
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), activation='relu', input_shape=(8, 8, 1)))
model.add(MaxPooling2D())
# summarize model
model.summary()
# define a vertical line detector
detector = [[[0],[1],[0]],
            [[0],[1],[0]],
            [[0],[1],[0]]]
```

```

weights = [asarray(detector), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
# apply filter to input data
yhat = model.predict(data)
# enumerate rows
for r in range(yhat.shape[1]):
    # print each column in the row
    print([yhat[0,r,c,0] for c in range(yhat.shape[2])])

```

Listing 13.18: Example of manually applying a two-dimensional filter with max pooling.

Running the example first summarizes the model. We can see, as we might expect by now, that the output of the max pooling layer will be a single feature map with each dimension halved, with the shape (3,3). Applying the max pooling results in a new feature map that still detects the line, although in a downsampled manner.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 6, 6, 1)	10
max_pooling2d_1 (MaxPooling2D)	(None, 3, 3, 1)	0
<hr/>		
Total params:	10	
Trainable params:	10	
Non-trainable params:	0	
<hr/>		
[0.0, 3.0, 0.0]		
[0.0, 3.0, 0.0]		
[0.0, 3.0, 0.0]		

Listing 13.19: Example output from manually applying a two-dimensional filter with max pooling.

13.6 Global Pooling Layers

There is another type of pooling that is sometimes used called global pooling. Instead of down sampling patches of the input feature map, global pooling downsamples the entire feature map to a single value. This would be the same as setting the `pool_size` to the size of the input feature map. Global pooling can be used in a model to aggressively summarize the presence of a feature in an image. It is also sometimes used in models as an alternative to using a fully connected layer to transition from feature maps to an output prediction for the model. Both global average pooling and global max pooling are supported by Keras via the `GlobalAveragePooling2D` and `GlobalMaxPooling2D` classes respectively. For example, we can add global max pooling to the convolutional model used for vertical line detection.

```

...
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), activation='relu', input_shape=(8, 8, 1)))
model.add(GlobalMaxPooling2D())

```

Listing 13.20: Example of defining a model with one filter and global max pooling.

The outcome will be a single value that will summarize the strongest activation or presence of the vertical line in the input image. The complete code listing is provided below.

```
# example of using global max pooling
from numpy import asarray
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import GlobalMaxPooling2D
# define input data
data = [[0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 0, 0, 0]]
data = asarray(data)
data = data.reshape(1, 8, 8, 1)
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), activation='relu', input_shape=(8, 8, 1)))
model.add(GlobalMaxPooling2D())
# summarize model
model.summary()
# define a vertical line detector
detector = [[[0],[1],[0]],
            [[0],[1],[0]],
            [[0],[1],[0]]]
weights = [asarray(detector), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
# apply filter to input data
yhat = model.predict(data)
# show result
print(yhat)
```

Listing 13.21: Example of manually applying a two-dimensional filter with global max pooling.

Running the example first summarizes the model. We can see that, as expected, the output of the global pooling layer is a single value that summarizes the presence of the feature in the single feature map. Next, the output of the model is printed showing the effect of global max pooling on the feature map, printing the single largest activation.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 6, 6, 1)	10
global_max_pooling2d_1 (Glob)	(None, 1)	0

Total params: 10
Trainable params: 10

```
| Non-trainable params: 0  
|-----  
|[ [3.] ]|
```

Listing 13.22: Example output from manually applying a two-dimensional filter with global max pooling.

13.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Global Max Pooling.** Develop a small model to demonstrate global max pooling.
- **Stride Size.** Experiment with different stride sizes with a pooling layer and review the effect on the model summary.
- **Padding.** Experiment with padding in a pooling layer and review the effect on the model summary.

If you explore any of these extensions, I'd love to know.

13.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

13.8.1 Books

- Chapter 9: Convolutional Networks, *Deep Learning*, 2016.
<https://amzn.to/2Dl124s>
- Chapter 5: Deep Learning for Computer Vision, *Deep Learning with Python*, 2017.
<https://amzn.to/2Dnshvc>

13.8.2 API

- Keras Convolutional Layers API.
<https://keras.io/layers/convolutional/>
- Keras Pooling Layers API.
<https://keras.io/layers/pooling/>

13.9 Summary

In this tutorial, you discovered how the pooling operation works and how to implement it in convolutional neural networks. Specifically, you learned:

- Pooling can be used to downsample the detection of features in feature maps.
- How to calculate and implement average and maximum pooling in a convolutional neural network.
- How to use global pooling in a convolutional neural network.

13.9.1 Next

This was the final tutorial in the layers part. In the next part, you will discover the convolutional neural network model architectural innovations used in modern deep learning.

Part IV

Convolutional Neural Networks

Overview

In this part you will discover the architectural innovations in the development and use of convolutional neural networks. After reading the chapters in this part, you will know:

- The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) and the deep learning-based innovations it has helped to bring about (Chapter 14).
- The key architectural milestones in the development and use of convolutional neural networks over the last two decades (Chapter 15).
- How a 1×1 convolutional layer can be used to both increase and decrease the complexity in a convolutional neural network (Chapter 16).
- How to implement three architectural innovations in Keras from scratch, namely VGG Blocks, the Inception Module and the ResNet Module (Chapter 17).
- How to use pre-trained models for classification and transfer learning in Keras (Chapter 18).

Chapter 14

ImageNet, ILSVRC, and Milestone Architectures

The rise in popularity and use of deep learning neural network techniques can be traced back to the innovations in the application of convolutional neural networks to image classification tasks. Some of the most important innovations have sprung from submissions by academics and industry leaders to the ImageNet Large Scale Visual Recognition Challenge, or ILSVRC. The ILSVRC is an annual computer vision competition developed upon a subset of a publicly available computer vision dataset called ImageNet. As such, the tasks and even the challenge itself is often referred to as the ImageNet Competition. In this tutorial, you will discover the ImageNet dataset, the ILSVRC, and the key milestones in image classification that have resulted from the competitions. After reading this tutorial, you will know:

- The ImageNet dataset is a very large collection of human annotated photographs designed by academics for developing computer vision algorithms.
- The ImageNet Large Scale Visual Recognition Challenge, or ILSVRC, is an annual competition that uses subsets from the ImageNet dataset and is designed to foster the development and benchmarking of state-of-the-art algorithms.
- The ILSVRC tasks have led to milestone model architectures and techniques in the intersection of computer vision and deep learning.

Let's get started.

14.1 Overview

This tutorial is divided into three parts; they are:

1. ImageNet Dataset
2. ImageNet Large Scale Visual Recognition Challenge
3. Deep Learning Milestones From ILSVRC

14.2 ImageNet Dataset

ImageNet is a large dataset of annotated photographs intended for computer vision research. The goal of developing the dataset was to provide a resource to promote the research and development of improved methods for computer vision.

We believe that a large-scale ontology of images is a critical resource for developing advanced, large-scale content-based image search and image understanding algorithms, as well as for providing critical training and benchmarking data for such algorithms.

— *ImageNet: A Large-Scale Hierarchical Image Database*, 2009.

Based on statistics about the dataset recorded on the ImageNet homepage, there are a little more than 14 million images in the dataset, a little more than 21 thousand groups or classes (called *synsets*), and a little more than 1 million images that have bounding box annotations (e.g. boxes around identified objects in the images). The photographs were annotated by humans using crowdsourcing platforms such as Amazon’s Mechanical Turk. The project to develop and maintain the dataset was organized and executed by a collaboration between academics at Princeton, Stanford, and other American universities. The project does not own the photographs that make up the images; instead, they are owned by the copyright holders. As such, the dataset is not distributed directly; URLs are provided to the images included in the dataset.

14.3 ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

The ImageNet Large Scale Visual Recognition Challenge, or ILSVRC for short, is an annual computer vision competition held in which challenge tasks use subsets of the ImageNet dataset. The goal of the challenge was to both promote the development of better computer vision techniques and to benchmark the state-of-the-art. The annual challenge focuses on multiple tasks for *image classification* that includes both assigning a class label to an image based on the main object in the photograph and *object detection* that involves localizing objects within the photograph.

ILSVRC annotations fall into one of two categories: (1) image-level annotation of a binary label for the presence or absence of an object class in the image, [...] and (2) object-level annotation of a tight bounding box and class label around an object instance in the image

— *ImageNet Large Scale Visual Recognition Challenge*, 2015.

The general challenge tasks for most years are as follows:

- **Image classification:** Predict the classes of objects present in an image.
- **Single-object localization:** Image classification + draw a bounding box around one example of each object present.

- **Object detection:** Image classification + draw a bounding box around each object present.

More recently, and given the great success in the development of techniques for still photographs, the challenge tasks are changing to more difficult tasks such as labeling videos. The datasets comprised approximately 1 million images and 1,000 object classes. The datasets used in challenge tasks are sometimes varied (depending on the task) and were released publicly to promote widespread participation from academia and industry.

For each annual challenge, an annotated training dataset was released, along with an unannotated test dataset for which annotations had to be made and submitted to a server for evaluation. Typically, the training dataset was comprised of 1 million images, with 50,000 for a validation dataset and 150,000 for a test dataset.

The publicly released dataset contains a set of manually annotated training images. A set of test images is also released, with the manual annotations withheld. Participants train their algorithms using the training images and then automatically annotate the test images. These predicted annotations are submitted to the evaluation server. Results of the evaluation are revealed at the end of the competition period

— *ImageNet Large Scale Visual Recognition Challenge*, 2015.

Results were presented at an annual workshop at a computer vision conference to promote the sharing and distribution of successful techniques. The datasets are still available for each annual challenge, although you must register.

14.4 Deep Learning Milestones From ILSVRC

Researchers working on ILSVRC tasks have pushed back the frontier of computer vision research and the methods and papers that describe them are milestones in the fields of computer vision, deep learning, and more broadly in artificial intelligence. The pace of improvement in the first five years of the ILSVRC was dramatic, perhaps even shocking to the field of computer vision. Success has primarily been achieved by large (deep) convolutional neural networks (CNNs) on graphical processing unit (GPU) hardware, which sparked an interest in deep learning that extended beyond the field out into the mainstream.

State-of-the-art accuracy has improved significantly from ILSVRC2010 to ILSVRC2014, showcasing the massive progress that has been made in large-scale object recognition over the past five years

— *ImageNet Large Scale Visual Recognition Challenge*, 2015.

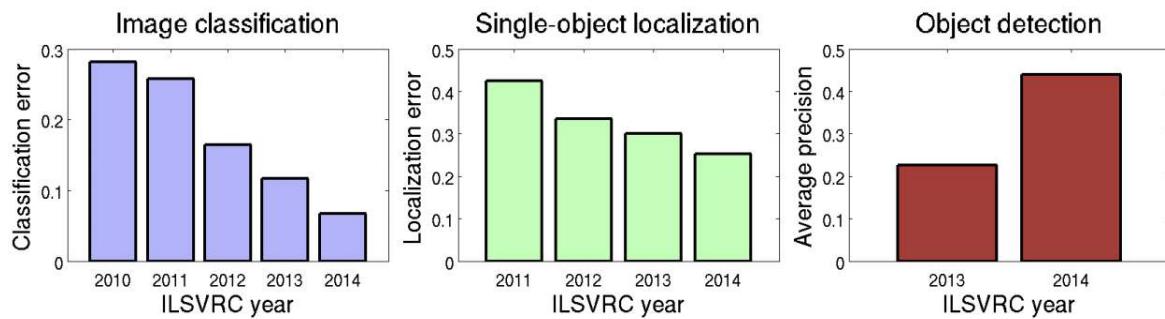


Figure 14.1: Summary of the Improvement on ILSVRC Tasks Over the First Five Years of the Competition. Taken from ImageNet Large Scale Visual Recognition Challenge, 2015.

There has been widespread participation in the ILSVRC over the years with many important developments and an enormous number of academic publications. Picking out milestones from so much work is a challenge in and of itself. Nevertheless, there are techniques, often named for their parent university, research group, or company that stand out and have become staples in the intersecting fields of deep learning and computer vision. The papers that describe the methods have become required reading and the techniques used by the models have become heuristics when using the general techniques in practice.

In this section, we will highlight some of these milestone techniques proposed as part of ILSVRC in which they were introduced and the papers that describe them. The focus will be on image classification tasks.

14.4.1 ILSVRC-2012

AlexNet (SuperVision)

Alex Krizhevsky, et al. from the University of Toronto in their 2012 paper titled *ImageNet Classification with Deep Convolutional Neural Networks* developed a convolutional neural network that achieved top results on the ILSVRC-2010 and ILSVRC-2012 image classification tasks. These results sparked interest in deep learning in computer vision.

... we trained one of the largest convolutional neural networks to date on the subsets of ImageNet used in the ILSVRC-2010 and ILSVRC-2012 competitions and achieved by far the best results ever reported on these datasets.

— *ImageNet Classification with Deep Convolutional Neural Networks*, 2012.

14.4.2 ILSVRC-2013

ZFNet (Clarifai)

Matthew Zeiler and Rob Fergus proposed a variation of AlexNet generally referred to as ZFNet in their 2013 paper titled *Visualizing and Understanding Convolutional Networks*, a variation of which won the ILSVRC-2013 image classification task.

14.4.3 ILSVRC-2014

Inception (GoogLeNet)

Christian Szegedy, et al. from Google achieved top results for object detection with their GoogLeNet model that made use of the inception module and architecture. This approach was described in their 2014 paper titled *Going Deeper with Convolutions*.

We propose a deep convolutional neural network architecture codenamed Inception, which was responsible for setting the new state-of-the-art for classification and detection in the ImageNet Large-Scale Visual Recognition Challenge 2014 (ILSVRC14).

— *Going Deeper with Convolutions*, 2014.

VGG

Karen Simonyan and Andrew Zisserman from the Oxford Vision Geometry Group (VGG) achieved top results for image classification and localization with their VGG model. Their approach is described in their 2015 paper titled *Very Deep Convolutional Networks for Large-Scale Image Recognition*.

... we come up with significantly more accurate ConvNet architectures, which not only achieve the state-of-the-art accuracy on ILSVRC classification and localisation tasks, but are also applicable to other image recognition datasets, where they achieve excellent performance even when used as a part of a relatively simple pipelines

— *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2015.

14.4.4 ILSVRC-2015

ResNet (MSRA)

Kaiming He, et al. from Microsoft Research achieved top results for object detection and object detection with localization tasks with their Residual Network or ResNet described in their 2015 paper titled *Deep Residual Learning for Image Recognition*.

An ensemble of these residual nets achieves 3.57% error on the ImageNet test set. This result won the 1st place on the ILSVRC 2015 classification task.

— *Deep Residual Learning for Image Recognition*, 2015.

14.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

14.5.1 Papers

- *ImageNet: A Large-Scale Hierarchical Image Database*, 2009.
<https://ieeexplore.ieee.org/document/5206848>
- *ImageNet Large Scale Visual Recognition Challenge*, 2015.
<https://link.springer.com/article/10.1007/s11263-015-0816-y>
- *ImageNet Classification with Deep Convolutional Neural Networks*, 2012.
<https://dl.acm.org/citation.cfm?id=3065386>
- *Visualizing and Understanding Convolutional Networks*, 2013.
<https://arxiv.org/abs/1311.2901>
- *Going Deeper with Convolutions*, 2014.
<https://arxiv.org/abs/1409.4842>
- *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2015.
<https://arxiv.org/abs/1409.1556>
- *Deep Residual Learning for Image Recognition*, 2015.
<https://arxiv.org/abs/1512.03385>

14.5.2 Articles

- ImageNet, Wikipedia.
<https://en.wikipedia.org/wiki/ImageNet>
- ImageNet Homepage.
<http://www.image-net.org/>
- Large Scale Visual Recognition Challenge (ILSVRC) Homepage.
<http://www.image-net.org/challenges/LSVRC/>
- Stanford Vision Lab.
<http://vision.stanford.edu/>

14.6 Summary

In this tutorial, you discovered the ImageNet dataset, the ILSVRC competitions, and the key milestones in image classification that have resulted from the competitions. Specifically, you learned:

- The ImageNet dataset is a very large collection of human annotated photographs designed by academics for developing computer vision algorithms.
- The ImageNet Large Scale Visual Recognition Challenge, or ILSVRC, is an annual competition that uses subsets from the ImageNet dataset and is designed to foster the development and benchmarking of state-of-the-art algorithms.
- The ILSVRC tasks have led to milestone model architectures and techniques in the intersection of computer vision and deep learning.

14.6.1 Next

In the next section, you will discover the details of the model architectural innovations used in convolutional neural networks.

Chapter 15

How Milestone Model Architectural Innovations Work

Convolutional neural networks are comprised of two very simple elements, namely convolutional layers and pooling layers. Although simple, there are near-infinite ways to arrange these layers for a given computer vision problem. Fortunately, there are both common patterns for configuring these layers and architectural innovations that you can use in order to develop very deep convolutional neural networks. Studying these architectural design decisions developed for state-of-the-art image classification tasks can provide both a rationale and intuition for how to use these designs when designing your own deep convolutional neural network models. In this tutorial, you will discover the key architectural milestones for the use of convolutional neural networks for challenging image classification problems. After completing this tutorial, you will know:

- How to pattern the number of filters and filter sizes when implementing convolutional neural networks.
- How to arrange convolutional and pooling layers in a uniform pattern to develop well-performing models.
- How to use the inception module and residual module to develop much deeper convolutional networks.

Let's get started.

15.1 Tutorial Overview

This tutorial is divided into six parts; they are:

1. Architectural Design for CNNs
2. LeNet-5
3. AlexNet
4. VGG

5. Inception and GoogLeNet
6. Residual Network or ResNet

15.2 Architectural Design for CNNs

The elements of a convolutional neural network, such as convolutional and pooling layers, are relatively straightforward to understand. The challenging part of using convolutional neural networks in practice is how to design model architectures that best use these simple elements. A useful approach to learning how to design effective convolutional neural network architectures is to study successful applications. This is particularly straightforward to do because of the intense study and application of CNNs through 2012 to 2016 for the ImageNet Large Scale Visual Recognition Challenge, or ILSVRC. This challenge resulted in both the rapid advancement in the state-of-the-art for very difficult computer vision tasks and the development of general innovations in the architecture of convolutional neural network models.

We will begin with LeNet-5 that is often described as the first successful and important application of CNNs prior to the ILSVRC, then look at four different winning architectural innovations for CNNs developed for the ILSVRC, namely, AlexNet, VGG, Inception, and ResNet. By understanding these milestone models and their architecture or architectural innovations from a high-level, you will develop both an appreciation for the use of these architectural elements in modern applications of CNNs in computer vision, and be able to identify and choose architecture elements that may be useful in the design of your own models.

15.3 LeNet-5

Perhaps the first widely known and successful application of convolutional neural networks was LeNet-5, described by Yann LeCun, et al. in their 1998 paper titled *Gradient-Based Learning Applied to Document Recognition*. The system was developed for use in a handwritten character recognition problem and demonstrated on the MNIST standard dataset, achieving approximately 99.2% classification accuracy (or a 0.8% error rate). The network was then described as the central technique in a broader system referred to as Graph Transformer Networks.

It is a long paper, and perhaps the best part to focus on is Section II. B. that describes the LeNet-5 architecture. In that section, the paper describes the network as having seven layers with input grayscale images having the shape 32×32 , the size of images in the MNIST dataset. The model proposes a pattern of a convolutional layer followed by an average pooling layer, referred to as a subsampling layer. This pattern is repeated two and a half times before the output feature maps are flattened and fed to a number of fully connected layers for interpretation and a final prediction. A picture of the network architecture is provided in the paper and reproduced below.

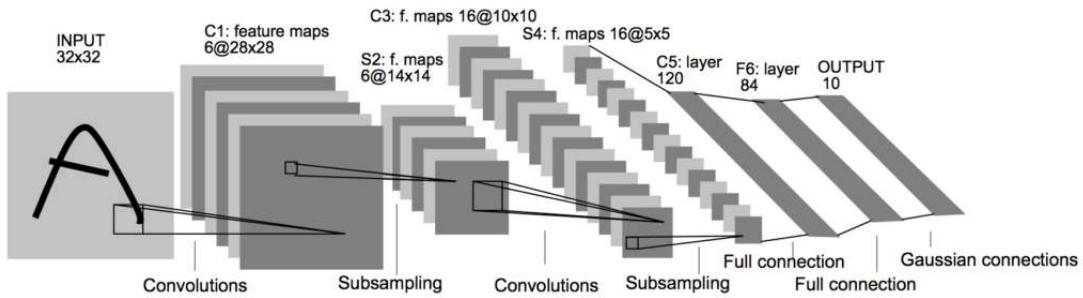


Figure 15.1: Architecture of the LeNet-5 Convolutional Neural Network for Handwritten Character Recognition (taken from the 1998 paper).

The pattern of blocks of convolutional layers and pooling layers (referred to as subsampling) grouped together and repeated remains a common pattern in designing and using convolutional neural networks today, more than twenty years later. Interestingly, the architecture uses a small number of filters with a modest size as the first hidden layer, specifically 6 filters each with the size of 5×5 pixels. After pooling, another convolutional layer has many more filters, again with the same size, specifically 16 filters with a size of 5×5 pixels, again followed by pooling. In the repetition of these two blocks of convolution and pooling layers, the trend is an increase in the number of filters.

Compared to modern applications, the number of filters is also small, but the trend of increasing the number of filters with the depth of the network also remains a common pattern in modern usage of the technique. The flattening of the feature maps and interpretation and classification of the extracted features by fully connected layers also remains a common pattern today. In modern terminology, the final section of the architecture is often referred to as the classifier, whereas the convolutional and pooling layers earlier in the model are referred to as the feature extractor.

We can summarize the key aspects of the architecture relevant in modern models as follows:

- Fixed-sized input images.
- Group convolutional and pooling layers into blocks.
- Repetition of convolutional-pooling blocks in the architecture.
- Increase in the number of filters with the depth of the network.
- Distinct feature extraction and classifier parts of the architecture.

15.4 AlexNet

The work that perhaps could be credited with sparking renewed interest in neural networks and the beginning of the dominance of deep learning in many computer vision applications was the 2012 paper by Alex Krizhevsky, et al. titled *ImageNet Classification with Deep Convolutional Neural Networks*. The paper describes a model later referred to as *AlexNet* designed to address the ImageNet Large Scale Visual Recognition Challenge or ILSVRC-2010 competition for classifying photographs of objects into one of 1,000 different categories.

The ILSVRC was a competition, designed to spur innovation in the field of computer vision. Before the development of AlexNet, the task was thought very difficult and far beyond the capability of modern computer vision methods. AlexNet successfully demonstrated the capability of the convolutional neural network model in the domain, and kindled a fire that resulted in many more improvements and innovations, many demonstrated on the same ILSVRC task in subsequent years. More broadly, the paper showed that it is possible to develop deep and effective end-to-end models for a challenging problem without using unsupervised pre-training techniques that were popular at the time.

Important in the design of AlexNet was a suite of methods that were new or successful, but not widely adopted at the time. Now, they have become requirements when using CNNs for image classification. AlexNet made use of the rectified linear activation function, or ReLU, as the nonlinearity after each convolutional layer, instead of S-shaped functions such as the logistic or Tanh that were common up until that point. Also, a softmax activation function was used in the output layer, now a staple for multiclass classification with neural networks.

The average pooling used in LeNet-5 was replaced with a max pooling method, although in this case, overlapping pooling was found to outperform non-overlapping pooling that is commonly used today (e.g. stride of pooling operation is the same size as the pooling operation, e.g. 2 by 2 pixels). To address overfitting, the newly proposed dropout method was used between the fully connected layers of the classifier part of the model to improve generalization error. The architecture of AlexNet is deep and extends upon some of the patterns established with LeNet-5. The image below, taken from the paper, summarizes the model architecture, in this case, split into two pipelines to train on the GPU hardware of the time.

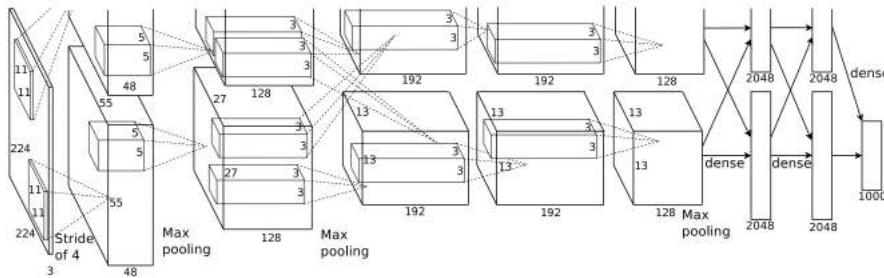


Figure 15.2: Architecture of the AlexNet Convolutional Neural Network for Object Photo Classification (taken from the 2012 paper where the diagram is also clipped or cut-off).

The model has five convolutional layers in the feature extraction part of the model and three fully connected layers in the classifier part of the model. Input images were fixed to the size 224×224 with three color channels. In terms of the number of filters used in each convolutional layer, the pattern of increasing the number of filters with depth seen in LeNet was mostly adhered to, in this case, the sizes: 96, 256, 384, 384, and 256. Similarly, the pattern of decreasing the size of the filter (kernel) with depth was used, starting from the smaller size of 11×11 and decreasing to 5×5 , and then to 3×3 in the deeper layers. Use of small filters such as 5×5 and 3×3 is now the norm.

The pattern of a convolutional layer followed by pooling layer was used at the start and end of the feature detection part of the model. Interestingly, a pattern of a convolutional layer followed immediately by a second convolutional layer was used. This pattern too has become a modern standard. The model was trained with data augmentation, artificially increasing the

size of the training dataset and giving the model more of an opportunity to learn the same features in different orientations. We can summarize the key aspects of the architecture relevant in modern models as follows:

- Use of the ReLU activation function after convolutional layers and softmax for the output layer.
- Use of Max Pooling instead of Average Pooling.
- Use of Dropout regularization between the fully connected layers.
- Pattern of a convolutional layer fed directly to another convolutional layer.
- Use of Data Augmentation.

15.5 VGG

The development of deep convolutional neural networks for computer vision tasks appeared to be a little bit of a dark art after AlexNet. An important work that sought to standardize architecture design for deep convolutional networks and developed much deeper and better performing models in the process was the 2014 paper titled *Very Deep Convolutional Networks for Large-Scale Image Recognition* by Karen Simonyan and Andrew Zisserman. Their architecture is generally referred to as VGG after the name of their lab, the Visual Geometry Group at Oxford. Their model was developed and demonstrated on the same ILSVRC competition, in this case, the ILSVRC-2014 version of the challenge. The first important difference that has become a de facto standard is the use of a large number of small filters. Specifically, filters with the size 3×3 and 1×1 with the stride of one, different from the large sized filters in LeNet-5 and the smaller but still relatively large filters and large stride of four in AlexNet.

Max pooling layers are used after most, but not all, convolutional layers, learning from the example in AlexNet, yet all pooling is performed with the size 2×2 and the same stride, that too has become a de facto standard. Specifically, the VGG networks use examples of two, three, and even four convolutional layers stacked together before a max pooling layer is used. The rationale was that stacked convolutional layers with smaller filters approximate the effect of one convolutional layer with a larger sized filter, e.g. three stacked convolutional layers with 3×3 filters approximates one convolutional layer with a 7×7 filter. Another important difference is the very large number of filters used. The number of filters increases with the depth of the model, although starts at a relatively large number of 64 and increases through 128, 256, and 512 filters at the end of the feature extraction part of the model.

A number of variants of the architecture were developed and evaluated, although two are referred to most commonly given their performance and depth. They are named for the number of layers: they are the VGG-16 and the VGG-19 for 16 and 19 learned layers respectively. Below is a table taken from the paper; note the two far right columns indicating the configuration (number of filters) used in the VGG-16 and VGG-19 versions of the architecture.

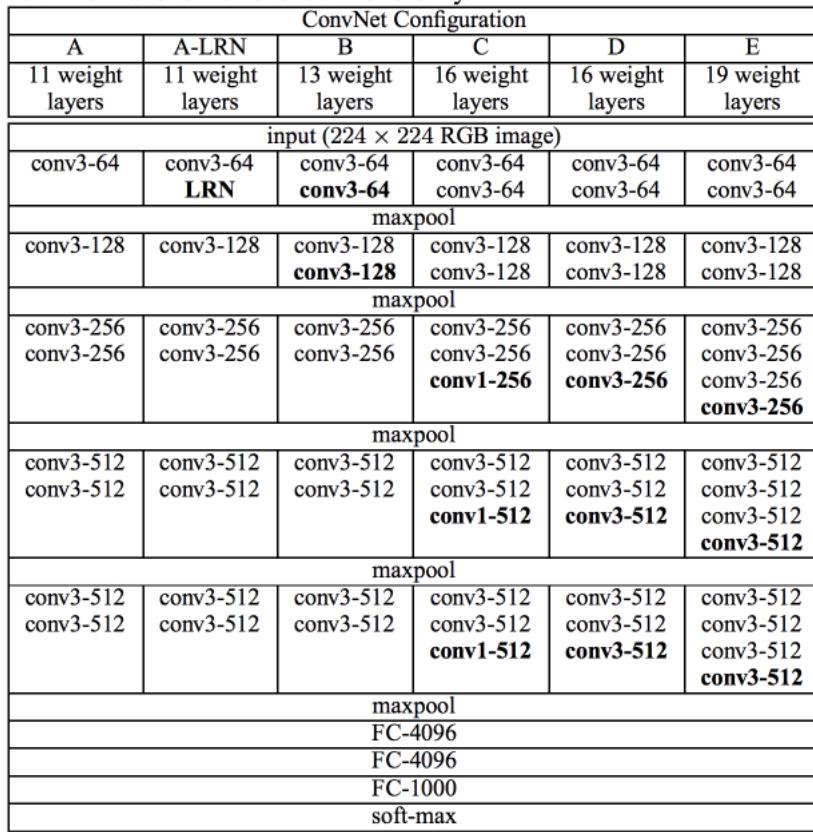


Figure 15.3: Architecture of the VGG Convolutional Neural Network for Object Photo Classification (taken from the 2014 paper).

The design decisions in the VGG models have become the starting point for simple and direct use of convolutional neural networks in general. Finally, the VGG work was among the first to release the valuable model weights under a permissive license that led to a trend among deep learning computer vision researchers. This, in turn, has led to the heavy use of pre-trained models like VGG in transfer learning as a starting point on new computer vision tasks. We can summarize the key aspects of the architecture relevant in modern models as follows:

- Use of very small convolutional filters, e.g. 3×3 and 1×1 with a stride of one.
- Use of max pooling with a size of 2×2 and a stride of the same dimensions.
- The importance of stacking convolutional layers together before using a pooling layer to define a block.
- Dramatic repetition of the convolutional-pooling block pattern.
- Development of very deep (16 and 19 layer) models.

We will explore how to implement VGG blocks in Keras in Chapter [17](#).

15.6 Inception and GoogLeNet

Important innovations in the use of convolutional layers were proposed in the 2015 paper by Christian Szegedy, et al. titled *Going Deeper with Convolutions*. In the paper, the authors propose an architecture referred to as inception (or inception v1 to differentiate it from extensions) and a specific model called GoogLeNet that achieved top results in the 2014 version of the ILSVRC challenge. The key innovation on the inception models is called the inception module. This is a block of parallel convolutional layers with different sized filters (e.g. 1×1 , 3×3 , 5×5) and a 3×3 max pooling layer, the results of which are then concatenated. Below is an example of the inception module taken from the paper.

A problem with a naive implementation of the inception model is that the number of filters (depth or channels) begins to build up fast, especially when inception modules are stacked. Performing convolutions with larger filter sizes (e.g. 3 and 5) can be computationally expensive on a large number of filters. To address this, 1×1 convolutional layers are used to reduce the number of filters in the inception model. Specifically before the 3×3 and 5×5 convolutional layers and after the pooling layer. The image below taken from the paper shows this change to the inception module.

A second important design decision in the inception model was connecting the output at different points in the model. This was achieved by creating small off-shoot output networks from the main network that were trained to make a prediction. The intent was to provide an additional error signal from the classification task at different points of the deep model in order to address the vanishing gradients problem. These small output networks were then removed after training. Below shows a rotated version (left-to-right for input-to-output) of the architecture of the GoogLeNet model taken from the paper using the Inception modules from the input on the left to the output classification on the right and the two additional output networks that were only used during training.

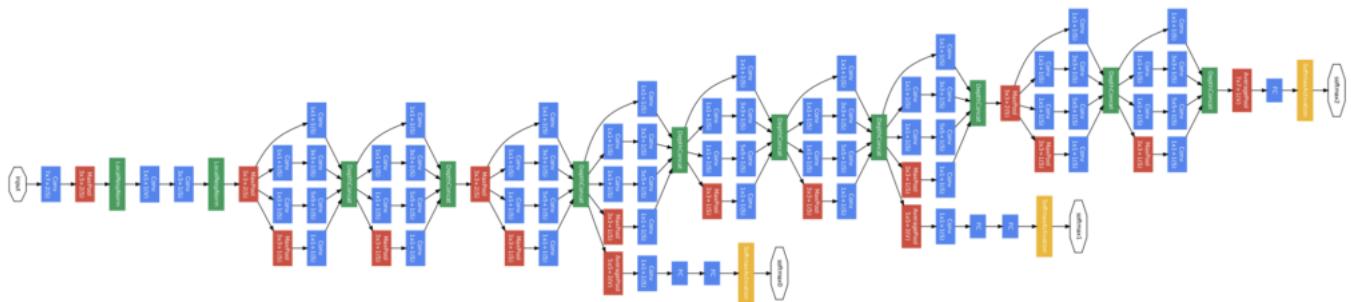


Figure 15.4: Architecture of the GoogLeNet Model Used During Training for Object Photo Classification (taken from the 2015 paper).

Interestingly, overlapping max pooling was used and a large average pooling operation was used at the end of the feature extraction part of the model prior to the classifier part of the model. We can summarize the key aspects of the architecture relevant in modern models as follows:

- Development and repetition of the Inception module.

- Heavy use of the 1×1 convolution to reduce the number of channels.
- Use of error feedback at multiple points in the network.
- Development of very deep (22-layer) models.
- Use of global average pooling for the output of the model.

We will explore how to implement inception modules in Keras in Chapter [17](#).

15.7 Residual Network or ResNet

A final important innovation in convolutional neural nets that we will review was proposed by Kaiming He, et al. in their 2016 paper titled *Deep Residual Learning for Image Recognition*. In the paper, the authors proposed a very deep model called a Residual Network, or ResNet for short, an example of which achieved success on the 2015 version of the ILSVRC challenge. Their model had an impressive 152 layers. Key to the model design is the idea of residual blocks that make use of shortcut connections. These are simply connections in the network architecture where the input is kept as-is (not weighted) and passed on to a deeper layer, e.g. skipping the next layer.

A residual block is a pattern of two convolutional layers with ReLU activation where the output of the block is combined with the input to the block, e.g. the shortcut connection. A projected version of the input used via 1×1 if the shape of the input to the block is different to the output of the block, so-called 1×1 convolutions. These are referred to as projected shortcut connections, compared to the unweighted or identity shortcut connections. The authors start with what they call a plain network, which is a VGG-inspired deep convolutional neural network with small filters (3×3), grouped convolutional layers followed with no pooling in between, and an average pooling at the end of the feature detector part of the model prior to the fully connected output layer with a softmax activation function.

The plain network is modified to become a residual network by adding shortcut connections in order to define residual blocks. Typically the shape of the input for the shortcut connection is the same size as the output of the residual block. The image below was taken from the paper and from left to right compares the architecture of a VGG model, a plain convolutional model, and a version of the plain convolutional with residual modules, called a residual network.

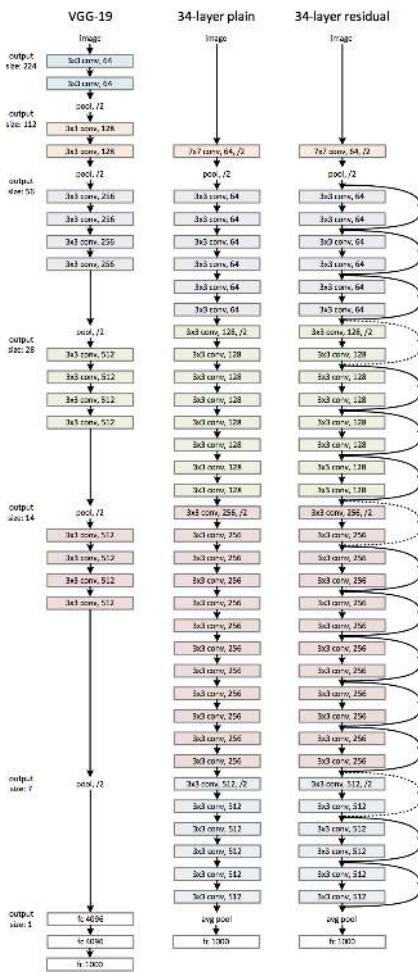


Figure 15.5: Architecture of the Residual Network for Object Photo Classification (taken from the 2016 paper).

We can summarize the key aspects of the architecture relevant in modern models as follows:

- Use of shortcut connections.
 - Development and repetition of the residual blocks.
 - Development of very deep (152-layer) models.

We will explore how to implement residual block in Keras in Chapter 17.

15.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

15.8.1 Papers

- *Gradient-based Learning Applied To Document Recognition*, 1998.
<https://ieeexplore.ieee.org/abstract/document/726791>

- *ImageNet Classification with Deep Convolutional Neural Networks*, 2012.
<https://dl.acm.org/citation.cfm?id=3065386>
- *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2014.
<https://arxiv.org/abs/1409.1556>
- *Going Deeper with Convolutions*, 2015.
<https://arxiv.org/abs/1409.4842>
- *Deep Residual Learning for Image Recognition*, 2016.
<https://arxiv.org/abs/1512.03385>

15.8.2 API

- Keras Applications API.
<https://keras.io/applications/>

15.9 Summary

In this tutorial, you discovered the key architectural milestones for the use of convolutional neural networks for challenging image classification. Specifically, you learned:

- How to pattern the number of filters and filter sizes when implementing convolutional neural networks.
- How to arrange convolutional and pooling layers in a uniform pattern to develop well-performing models.
- How to use the inception module and residual module to develop much deeper convolutional networks.

15.9.1 Next

In the next section, you will discover the details and power of the 1×1 convolution and the different ways in which it can be used.

Chapter 16

How to Use 1×1 Convolutions to Manage Model Complexity

Pooling can be used to downsample the content of feature maps, reducing their width and height whilst maintaining their salient features. A problem with deep convolutional neural networks is that the number of feature maps often increases with the depth of the network. This problem can result in a dramatic increase in the number of parameters and computation required when larger filter sizes are used, such as 5×5 and 7×7 .

To address this problem, a 1×1 convolutional layer can be used that offers a channel-wise pooling, often called feature map pooling or a projection layer. This simple technique can be used for dimensionality reduction, decreasing the number of feature maps whilst retaining their salient features. It can also be used directly to create a one-to-one projection of the feature maps to pool features across channels or to increase the number of feature maps, such as after traditional pooling layers. In this tutorial, you will discover how to use 1×1 filters to control the number of feature maps in a convolutional neural network. After completing this tutorial, you will know:

- The 1×1 filter can be used to create a linear projection of a stack of feature maps.
- The projection created by a 1×1 filter can act like channel-wise pooling and be used for dimensionality reduction.
- The projection created by a 1×1 filter can also be used directly or be used to increase the number of feature maps in a model.

Let's get started.

16.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Convolutions Over Channels
2. Problem of Too Many Feature Maps
3. Downsample Feature Maps With 1×1 Filters

4. Examples of How to Use 1×1 Convolutions
5. Examples of 1×1 Filters in CNN Model Architectures

16.2 Convolutions Over Channels

Recall that a convolutional operation is a linear application of a smaller filter to a larger input that results in an output feature map. A filter applied to an input image or input feature map always results in a single number. The systematic left-to-right and top-to-bottom application of the filter to the input results in a two-dimensional feature map. One filter creates one corresponding feature map. A filter must have the same depth or number of channels as the input, yet, regardless of the depth of the input and the filter, the resulting output is a single number and one filter creates a feature map with a single channel. Let's make this concrete with some examples:

- If the input has one channel such as a grayscale image, then a 3×3 filter will be applied in $3 \times 3 \times 1$ blocks.
- If the input image has three channels for red, green, and blue, then a 3×3 filter will be applied in $3 \times 3 \times 3$ blocks.
- If the input is a block of feature maps from another convolutional or pooling layer and has the depth of 64, then the 3×3 filter will be applied in $3 \times 3 \times 64$ blocks to create the single values to make up the single output feature map.

The depth of the output of one convolutional layer is only defined by the number of parallel filters applied to the input.

16.3 Problem of Too Many Feature Maps

The depth of the input or number of filters used in convolutional layers often increases with the depth of the network, resulting in an increase in the number of resulting feature maps. It is a common model design pattern. Further, some network architectures, such as the inception architecture, may also concatenate the output feature maps from multiple convolutional layers, which may also dramatically increase the depth of the input to subsequent convolutional layers.

A large number of feature maps in a convolutional neural network can cause a problem as a convolutional operation must be performed down through the depth of the input. This is a particular problem if the convolutional operation being performed is relatively large, such as 5×5 or 7×7 pixels, as it can result in considerably more parameters (weights) and, in turn, computation to perform the convolutional operations (large space and time complexity). Pooling layers are designed to downscale feature maps and systematically halve the width and height of feature maps in the network. Nevertheless, pooling layers do not change the number of filters in the model, the depth, or number of channels. Deep convolutional neural networks require a corresponding pooling type of layer that can downsample or reduce the depth or number of feature maps.

16.4 Downsample Feature Maps With 1×1 Filters

The solution is to use a 1×1 filter to downsample the depth or number of feature maps. A 1×1 filter will only have a single parameter or weight for each channel in the input, and like the application of any filter results in a single output value. This structure allows the 1×1 filter to act like a single neuron with an input from the same position across each of the feature maps in the input. This single neuron can then be applied systematically with a stride of one, left-to-right and top-to-bottom without any need for padding, resulting in a feature map with the same width and height as the input.

The 1×1 filter is so simple that it does not involve any neighboring pixels in the input; it may not be considered a convolutional operation. Instead, it is a linear weighting or projection of the input. Further, a nonlinearity is used as with other convolutional layers, allowing the projection to perform non-trivial computation on the input feature maps. This simple 1×1 filter provides a way to usefully summarize the input feature maps. The use of multiple 1×1 filters, in turn, allows the tuning of the number of summaries of the input feature maps to create, effectively allowing the depth of the feature maps to be increased or decreased as needed.

A convolutional layer with a 1×1 filter can, therefore, be used at any point in a convolutional neural network to control the number of feature maps. As such, it is often referred to as a projection operation or projection layer, or even a feature map or channel pooling layer. Now that we know that we can control the number of feature maps with 1×1 filters, let's make it concrete with some examples.

16.5 Examples of How to Use 1×1 Convolutions

We can make the use of a 1×1 filter concrete with some examples. Consider that we have a convolutional neural network that expected color images input with the square shape of $256 \times 256 \times 3$ pixels. These images then pass through a first hidden layer with 512 filters, each with the size of 3×3 with the same padding, followed by a ReLU activation function. The example below demonstrates this simple model.

```
# example of simple cnn model
from keras.models import Sequential
from keras.layers import Conv2D
# create model
model = Sequential()
model.add(Conv2D(512, (3,3), padding='same', activation='relu', input_shape=(256, 256, 3)))
# summarize model
model.summary()
```

Listing 16.1: Example of defining and summarizing a model with a normal filter.

Running the example creates the model and summarizes the model architecture. There are no surprises; the output of the first hidden layer is a block of feature maps with the three-dimensional shape of $256 \times 256 \times 512$.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 256, 256, 512)	14336

```
Total params: 14,336
Trainable params: 14,336
Non-trainable params: 0
```

Listing 16.2: Example output from defining and summarizing a model with a normal filter.

16.5.1 Example of Projecting Feature Maps

A 1×1 filter can be used to create a projection of the feature maps. The number of feature maps created will be the same number and the effect may be a refinement of the features already extracted. This is often called channel-wise pooling, as opposed to traditional feature-wise pooling on each channel. It can be implemented as follows:

```
model.add(Conv2D(512, (1,1), activation='relu'))
```

Listing 16.3: Example of a convolutional layer with 1×1 filters and projecting feature maps.

We can see that we use the same number of features and still follow the application of the filter with a rectified linear activation function. The complete example is listed below.

```
# example of a 1x1 filter for projection
from keras.models import Sequential
from keras.layers import Conv2D
# create model
model = Sequential()
model.add(Conv2D(512, (3,3), padding='same', activation='relu', input_shape=(256, 256, 3)))
model.add(Conv2D(512, (1,1), activation='relu'))
# summarize model
model.summary()
```

Listing 16.4: Example of summarizing a model with a 1×1 filters for projecting feature maps.

Running the example creates the model and summarizes the architecture. We can see that no change is made to the width or height of the feature maps, and by design, the number of feature maps is kept constant with a simple projection operation applied (e.g. a possible simplification of the feature maps).

```
-----
Layer (type)          Output Shape         Param #
-----
conv2d_1 (Conv2D)      (None, 256, 256, 512)  14336
-----
conv2d_2 (Conv2D)      (None, 256, 256, 512)  262656
-----
Total params: 276,992
Trainable params: 276,992
Non-trainable params: 0
```

Listing 16.5: Example output from summarizing a model with a 1×1 filters for projecting feature maps.

16.5.2 Example of Decreasing Feature Maps

The 1×1 filter can be used to decrease the number of feature maps. This is the most common application of this type of filter and in this way, the layer is often called a feature map pooling layer. In this example, we can decrease the depth (or channels) from 512 to 64. This might be useful if the subsequent layer we were going to add to our model would be another convolutional layer with 7×7 filters. These filters would only be applied at a depth of 64 rather than 512.

```
model.add(Conv2D(64, (1,1), activation='relu'))
```

Listing 16.6: Example of a convolutional layer with 1×1 filters that creates fewer feature maps.

The composition of the 64 feature maps is not the same as the original 512, but contains a useful summary of dimensionality reduction that captures the salient features, such that the 7×7 operation may have a similar effect on the 64 feature maps as it might have on the original 512. Further, a 7×7 convolutional layer with 64 filters itself applied to the 512 feature maps output by the first hidden layer would result in approximately one million parameters (activations). If the 1×1 filter is used to reduce the number of feature maps to 64 first, then the number of parameters required for the 7×7 layer is only approximately 200,000, an enormous difference. The complete example of using a 1×1 filter for dimensionality reduction is listed below.

```
# example of a 1x1 filter for dimensionality reduction
from keras.models import Sequential
from keras.layers import Conv2D
# create model
model = Sequential()
model.add(Conv2D(512, (3,3), padding='same', activation='relu', input_shape=(256, 256, 3)))
model.add(Conv2D(64, (1,1), activation='relu'))
# summarize model
model.summary()
```

Listing 16.7: Example of summarizing a model with a 1×1 filters for decreasing feature maps.

Running the example creates the model and summarizes its structure. We can see that the width and height of the feature maps are unchanged, yet the number of feature maps was reduced from 512 to 64.

```
-----
Layer (type)          Output Shape         Param #
-----
conv2d_1 (Conv2D)     (None, 256, 256, 512) 14336
-----
conv2d_2 (Conv2D)     (None, 256, 256, 64)   32832
-----
Total params: 47,168
Trainable params: 47,168
Non-trainable params: 0
-----
```

Listing 16.8: Example output from summarizing a model with a 1×1 filters for decreasing feature maps.

16.5.3 Example of Increasing Feature Maps

The 1×1 filter can be used to increase the number of feature maps. This is a common operation used after a pooling layer prior to applying another convolutional layer. The projection effect of the filter can be applied as many times as needed to the input, allowing the number of feature maps to be scaled up and yet have a composition that captures the salient features of the original. We can increase the number of feature maps from 512 input from the first hidden layer to double the size at 1,024 feature maps.

```
model.add(Conv2D(1024, (1,1), activation='relu'))
```

Listing 16.9: Example of a convolutional layer with 1×1 filters and more feature maps.

The complete example is listed below.

```
# example of a 1x1 filter to increase dimensionality
from keras.models import Sequential
from keras.layers import Conv2D
# create model
model = Sequential()
model.add(Conv2D(512, (3,3), padding='same', activation='relu', input_shape=(256, 256, 3)))
model.add(Conv2D(1024, (1,1), activation='relu'))
# summarize model
model.summary()
```

Listing 16.10: Example of summarizing a model with a 1×1 filters for increasing feature maps.

Running the example creates the model and summarizes its structure. We can see that the width and height of the feature maps are unchanged and that the number of feature maps was increased from 512 to double the size at 1,024.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 256, 256, 512)	14336
conv2d_2 (Conv2D)	(None, 256, 256, 1024)	525312

Total params: 539,648
 Trainable params: 539,648
 Non-trainable params: 0

Listing 16.11: Example output from summarizing a model with a 1×1 filters for increasing feature maps.

Now that we are familiar with how to use 1×1 filters, let's look at some examples where they have been used in the architecture of convolutional neural network models.

16.6 Examples of 1x1 Filters in CNN Model Architectures

In this section, we will highlight some important examples where 1×1 filters have been used as key elements in modern convolutional neural network model architectures.

16.6.1 Network in Network

The 1×1 filter was perhaps first described and popularized in the 2013 paper by Min Lin, et al. in their paper titled *Network In Network*. In the paper, the authors propose the need for a *MLP convolutional layer* and the need for cross-channel pooling to promote learning across channels.

This cascaded cross channel parametric pooling structure allows complex and learnable interactions of cross channel information.

— *Network In Network*, 2013.

They describe a 1×1 convolutional layer as a specific implementation of cross-channel parametric pooling, which, in effect, is exactly what a 1×1 filter achieves.

Each pooling layer performs weighted linear recombination on the input feature maps, which then go through a rectifier linear unit. [...] The cross channel parametric pooling layer is also equivalent to a convolution layer with 1×1 convolution kernel.

— *Network In Network*, 2013.

16.6.2 Inception Architecture

The 1×1 filter was used explicitly for dimensionality reduction and for increasing the dimensionality of feature maps after pooling in the design of the inception module, used in the GoogLeNet model by Christian Szegedy, et al. in their 2014 paper titled *Going Deeper with Convolutions* (introduced in Chapter 15). The paper describes an *inception module* where an input block of feature maps is processed in parallel by different convolutional layers each with differently sized filters, where a 1×1 size filter is one of the layers used.

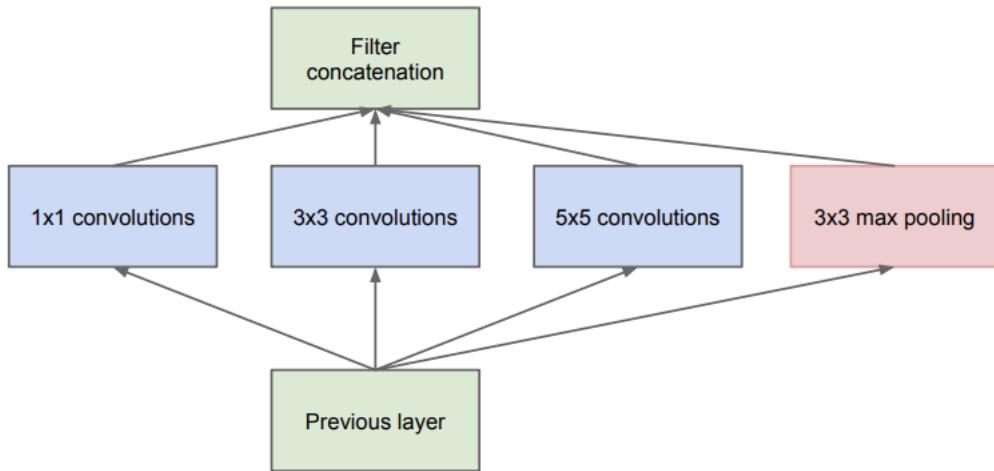


Figure 16.1: Example of the Naive Inception Module. Taken from *Going Deeper with Convolutions*, 2014.

The output of the parallel layers are then stacked, channel-wise, resulting in very deep stacks of convolutional layers to be processed by subsequent inception modules.

The merging of the output of the pooling layer with the outputs of convolutional layers would lead to an inevitable increase in the number of outputs from stage to stage. Even while this architecture might cover the optimal sparse structure, it would do it very inefficiently, leading to a computational blow up within a few stages.

— *Going Deeper with Convolutions*, 2014.

The inception module is then redesigned to use 1×1 filters to reduce the number of feature maps prior to parallel convolutional layers with 5×5 and 7×7 sized filters.

This leads to the second idea of the proposed architecture: judiciously applying dimension reductions and projections wherever the computational requirements would increase too much otherwise. [...] That is, 1×1 convolutions are used to compute reductions before the expensive 3×3 and 5×5 convolutions. Besides being used as reductions, they also include the use of rectified linear activation which makes them dual-purpose

— *Going Deeper with Convolutions*, 2014.

The 1×1 filter is also used to increase the number of feature maps after pooling, artificially creating more projections of the downsampled feature map content.

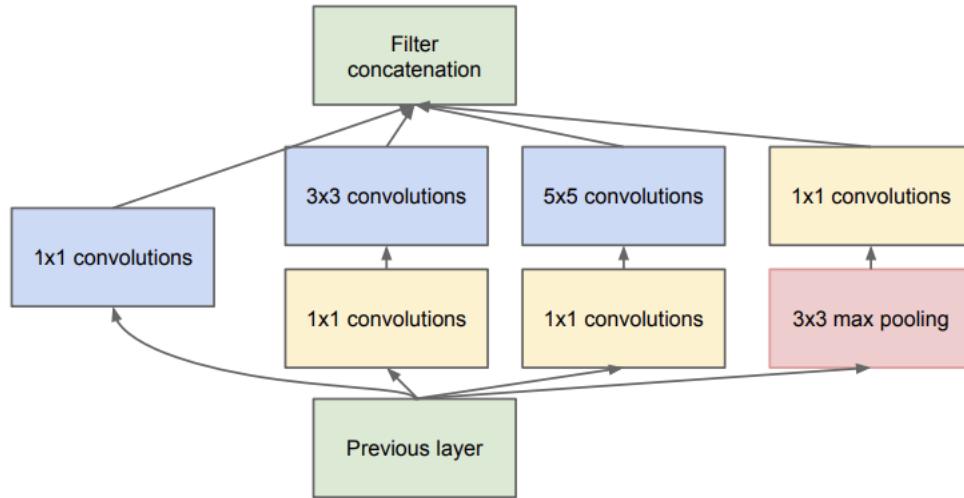


Figure 16.2: Example of the Inception Module With Dimensionality Reduction. Taken from *Going Deeper with Convolutions*, 2014.

16.6.3 Residual Architecture

The 1×1 filter was used as a projection technique to match the number of filters of input to the output of residual modules in the design of the residual network by Kaiming He, et al. in their 2015 paper titled *Deep Residual Learning for Image Recognition* (introduced in Chapter 15). The authors describe an architecture comprised of *residual modules* where the

input to a module is added to the output of the module in what is referred to as a shortcut connection. Because the input is added to the output of the module, the dimensionality must match in terms of width, height, and depth. Width and height can be maintained via padding, although a 1×1 filter is used to change the depth of the input as needed so that it can be added with the output of the module. This type of connection is referred to as a projection shortcut connection. Further, the residual modules use a bottleneck design with 1×1 filters to reduce the number of feature maps for computational efficiency reasons.

The three layers are 1×1 , 3×3 , and 1×1 convolutions, where the 1×1 layers are responsible for reducing and then increasing (restoring) dimensions, leaving the 3×3 layer a bottleneck with smaller input/output dimensions.

— Deep Residual Learning for Image Recognition, 2015.

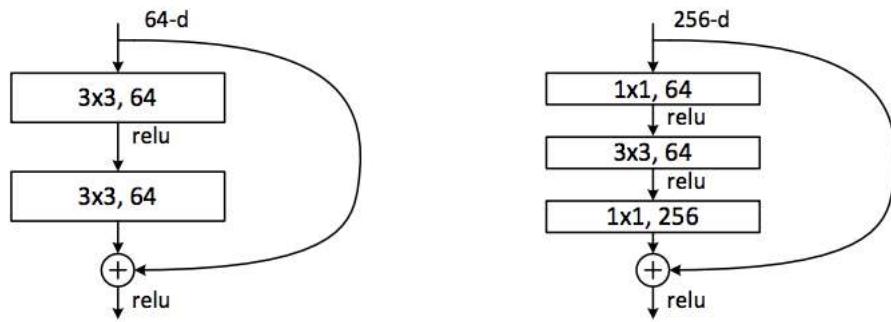


Figure 16.3: Example of a Normal and Bottleneck Residual Modules With Shortcut Connections. Taken from Deep Residual Learning for Image Recognition, 2015..

16.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- The 1×1 filter can be used to create a linear projection of a stack of feature maps.
- The projection created by a 1×1 filter can act like channel-wise pooling and be used for dimensionality reduction.
- The projection created by a 1×1 filter can also be used directly or be used to increase the number of feature maps in a model.

If you explore any of these extensions, I'd love to know.

16.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

16.8.1 Papers

- *Network In Network*, 2013.
<https://arxiv.org/abs/1312.4400>
- *Going Deeper with Convolutions*, 2014.
<https://arxiv.org/abs/1409.4842>
- *Deep Residual Learning for Image Recognition*, 2015.
<https://arxiv.org/abs/1512.03385>

16.8.2 Articles

- One by One [1×1] Convolution — counter-intuitively useful, 2016.
<https://iamaaditya.github.io/2016/03/one-by-one-convolution/>
- Yann LeCun on No Fully Connected Layers in CNN, 2015.
<https://www.facebook.com/yann.lecun/posts/10152820758292143>
- Networks in Networks and 1×1 Convolutions, YouTube.
<https://www.youtube.com/watch?v=c1RBQzKsDCk>

16.9 Summary

In this tutorial, you discovered how to use 1×1 filters to control the number of feature maps in a convolutional neural network. Specifically, you learned:

- The 1×1 filter can be used to create a linear projection of a stack of feature maps.
- The projection created by a 1×1 can act like channel-wise pooling and be used for dimensionality reduction.
- The projection created by a 1×1 can also be used directly or be used to increase the number of feature maps in a model.

16.9.1 Next

In the next section, you will discover how to implement three major model architectural innovations from scratch.

Chapter 17

How To Implement Model Architecture Innovations

There are discrete architectural elements from milestone models that you can use in the design of your own convolutional neural networks. Specifically, models that have achieved state-of-the-art results for tasks like image classification use discrete architecture elements repeated multiple times, such as the VGG block in the VGG models, the inception module in the GoogLeNet, and the residual module in the ResNet. Once you are able to implement parameterized versions of these architecture elements, you can use them in the design of your own models for computer vision and other applications. In this tutorial, you will discover how to implement the key architecture elements from milestone convolutional neural network models, from scratch. After completing this tutorial, you will know:

- How to implement a VGG module used in the VGG-16 and VGG-19 convolutional neural network models.
- How to implement the naive and optimized inception module used in the GoogLeNet model.
- How to implement the identity residual module used in the ResNet model.

Let's get started.

17.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. How to implement VGG Blocks
2. How to implement the Inception Module
3. How to implement the Residual Module

17.2 How to implement VGG Blocks

The VGG convolutional neural network architecture, named for the Visual Geometry Group at Oxford, was an important milestone in the use of deep learning methods for computer vision (introduced in Chapter 15). The architecture was described in the 2014 paper titled *Very Deep Convolutional Networks for Large-Scale Image Recognition* by Karen Simonyan and Andrew Zisserman and achieved top results in the LSVRC-2014 computer vision competition. The key innovation in this architecture was the definition and repetition of what we will refer to as VGG-blocks. These are groups of convolutional layers that use small filters (e.g. 3×3 pixels) followed by a max pooling layer.

The image is passed through a stack of convolutional (conv.) layers, where we use filters with a very small receptive field: 3×3 (which is the smallest size to capture the notion of left/right, up/down, center). [...] Max-pooling is performed over a 2×2 pixel window, with stride 2.

— *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2014.

A convolutional neural network with VGG-blocks is a sensible starting point when developing a new model from scratch as it is easy to understand, easy to implement, and very effective at extracting features from images. We can generalize the specification of a VGG-block as one or more convolutional layers with the same number of filters and a filter size of 3×3 , a stride of 1×1 , same padding so the output size is the same as the input size for each filter, and the use of a rectified linear activation function. These layers are then followed by a max pooling layer with a size of 2×2 and a stride of the same dimensions. We can define a function to create a VGG-block using the Keras functional API with a given number of convolutional layers and with a given number of filters per layer.

```
# function for creating a vgg block
def vgg_block(layer_in, n_filters, n_conv):
    # add convolutional layers
    for _ in range(n_conv):
        layer_in = Conv2D(n_filters, (3,3), padding='same', activation='relu')(layer_in)
    # add max pooling layer
    layer_in = MaxPooling2D((2,2), strides=(2,2))(layer_in)
    return layer_in
```

Listing 17.1: Example of a function for defining a VGG block.

To use the function, one would pass in the layer prior to the block and receive the layer for the end of the block that can be used to integrate into the model. For example, the first layer might be an input layer which could be passed into the function as an argument. The function then returns a reference to the final layer in the block, the pooling layer, that could be connected to a flatten layer and subsequent dense layers for making a classification prediction. We can demonstrate how to use this function by defining a small model that expects square color images as input and adds a single VGG block to the model with two convolutional layers, each with 64 filters.

```
# Example of creating a CNN model with a VGG block
from keras.models import Model
from keras.layers import Input
```

```

from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.utils import plot_model

# function for creating a vgg block
def vgg_block(layer_in, n_filters, n_conv):
    # add convolutional layers
    for _ in range(n_conv):
        layer_in = Conv2D(n_filters, (3,3), padding='same', activation='relu')(layer_in)
    # add max pooling layer
    layer_in = MaxPooling2D((2,2), strides=(2,2))(layer_in)
    return layer_in

# define model input
visible = Input(shape=(256, 256, 3))
# add vgg module
layer = vgg_block(visible, 64, 2)
# create model
model = Model(inputs=visible, outputs=layer)
# summarize model
model.summary()
# plot model architecture
plot_model(model, show_shapes=True, to_file='vgg_block.png')

```

Listing 17.2: Example of creating and summarizing a model with one VGG block.

Running the example creates the model and summarizes the structure. We can see that, as intended, the model added a single VGG block with two convolutional layers each with 64 filters, followed by a max pooling layer.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 256, 256, 3)	0
conv2d_1 (Conv2D)	(None, 256, 256, 64)	1792
conv2d_2 (Conv2D)	(None, 256, 256, 64)	36928
max_pooling2d_1 (MaxPooling2D)	(None, 128, 128, 64)	0
Total params:	38,720	
Trainable params:	38,720	
Non-trainable params:	0	

Listing 17.3: Example output from creating and summarizing a model with one VGG block.

A plot is also created of the model architecture that may help to make the model layout more concrete. Note, creating the plot assumes that you have `pydot` and `pygraphviz` installed. If this is not the case, you can comment out the import statement and call to the `plot_model()` function in the example.

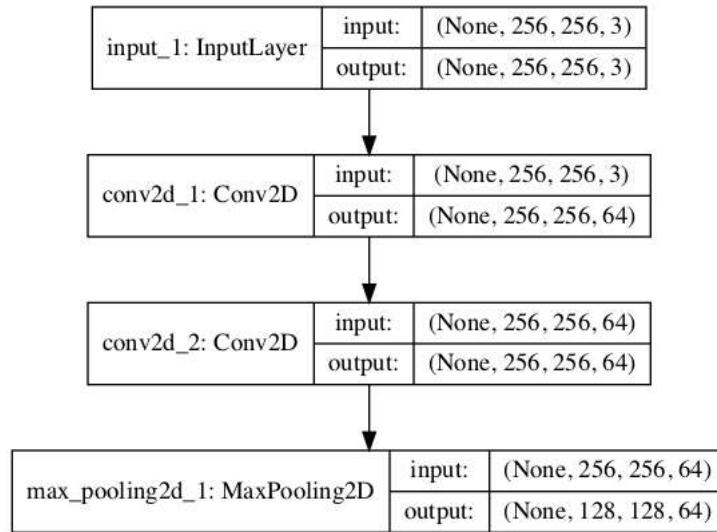


Figure 17.1: Plot of Convolutional Neural Network Architecture With a VGG Block.

Using VGG blocks in your own models should be common because they are so simple and effective. We can expand the example and demonstrate a single model that has three VGG blocks, the first two blocks have two convolutional layers with 64 and 128 filters respectively, the third block has four convolutional layers with 256 filters. This is a common usage of VGG blocks where the number of filters is increased with the depth of the model. The complete code listing is provided below.

```
# Example of creating a CNN model with many VGG blocks
from keras.models import Model
from keras.layers import Input
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.utils import plot_model

# function for creating a vgg block
def vgg_block(layer_in, n_filters, n_conv):
    # add convolutional layers
    for _ in range(n_conv):
        layer_in = Conv2D(n_filters, (3,3), padding='same', activation='relu')(layer_in)
    # add max pooling layer
    layer_in = MaxPooling2D((2,2), strides=(2,2))(layer_in)
    return layer_in

# define model input
visible = Input(shape=(256, 256, 3))
# add vgg module
layer = vgg_block(visible, 64, 2)
# add vgg module
layer = vgg_block(layer, 128, 2)
# add vgg module
layer = vgg_block(layer, 256, 4)
# create model
model = Model(inputs=visible, outputs=layer)
# summarize model
model.summary()
```

```
# plot model architecture
plot_model(model, show_shapes=True, to_file='multiple_vgg_blocks.png')
```

Listing 17.4: Example of creating and summarizing a model with many VGG blocks.

Again, running the example summarizes the model architecture and we can clearly see the pattern of VGG blocks.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 256, 256, 3)	0
conv2d_1 (Conv2D)	(None, 256, 256, 64)	1792
conv2d_2 (Conv2D)	(None, 256, 256, 64)	36928
max_pooling2d_1 (MaxPooling2D)	(None, 128, 128, 64)	0
conv2d_3 (Conv2D)	(None, 128, 128, 128)	73856
conv2d_4 (Conv2D)	(None, 128, 128, 128)	147584
max_pooling2d_2 (MaxPooling2D)	(None, 64, 64, 128)	0
conv2d_5 (Conv2D)	(None, 64, 64, 256)	295168
conv2d_6 (Conv2D)	(None, 64, 64, 256)	590080
conv2d_7 (Conv2D)	(None, 64, 64, 256)	590080
conv2d_8 (Conv2D)	(None, 64, 64, 256)	590080
max_pooling2d_3 (MaxPooling2D)	(None, 32, 32, 256)	0
Total params:	2,325,568	
Trainable params:	2,325,568	
Non-trainable params:	0	

Listing 17.5: Example output from creating and summarizing a model with many VGG blocks.

A plot of the model architecture is created providing a different perspective on the same linear progression of layers.

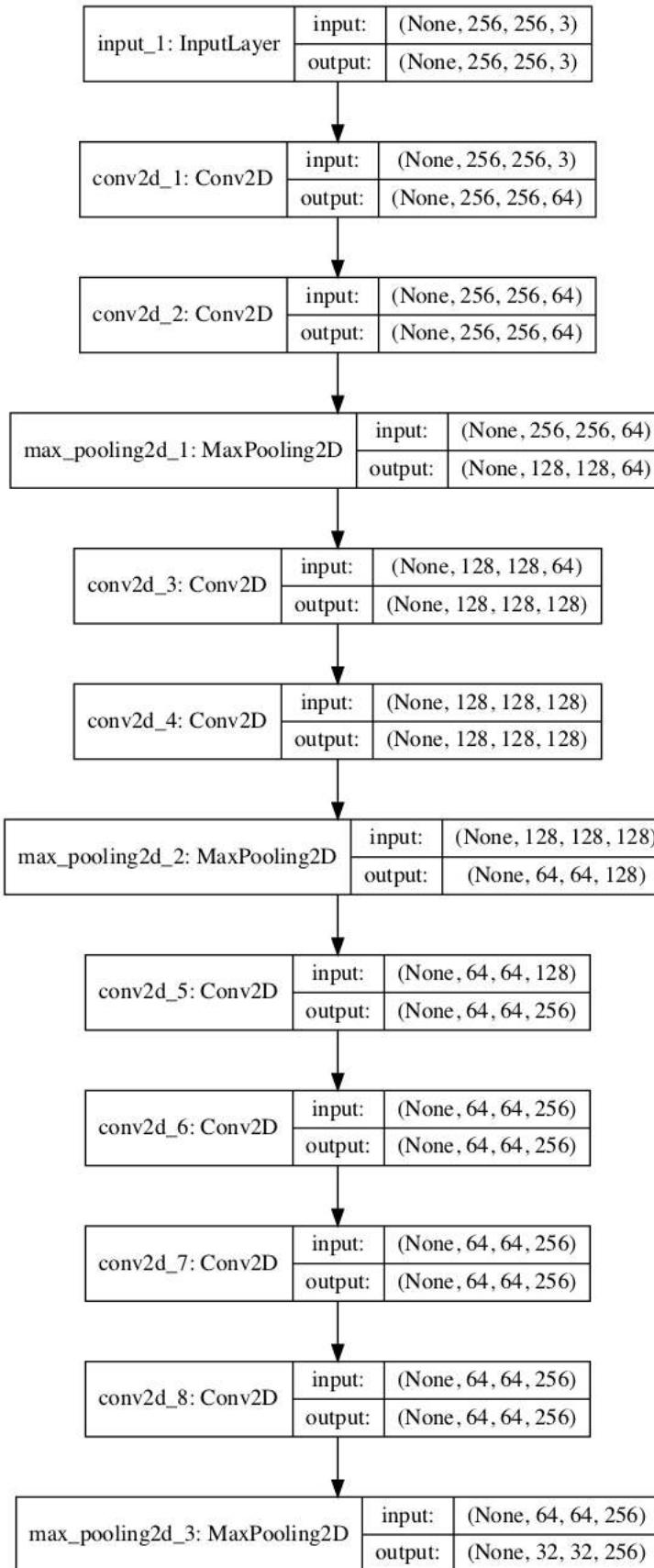


Figure 17.2: Plot of Convolutional Neural Network Architecture With Multiple VGG Blocks.

17.3 How to Implement the Inception Module

The inception module was described and used in the GoogLeNet model in the 2015 paper by Christian Szegedy, et al. titled *Going Deeper with Convolutions* (introduced in Chapter 15). Like the VGG model, the GoogLeNet model achieved top results in the 2014 version of the ILSVRC challenge. The key innovation on the inception model is called the inception module. This is a block of parallel convolutional layers with different sized filters (e.g. 1×1 , 3×3 , 5×5) and a 3×3 max pooling layer, the results of which are then concatenated.

In order to avoid patch-alignment issues, current incarnations of the Inception architecture are restricted to filter sizes 1×1 , 3×3 and 5×5 ; this decision was based more on convenience rather than necessity. [...] Additionally, since pooling operations have been essential for the success of current convolutional networks, it suggests that adding an alternative parallel pooling path in each such stage should have additional beneficial effect, too

— *Going Deeper with Convolutions*, 2015.

This is a very simple and powerful architectural unit that allows the model to learn not only parallel filters of the same size, but parallel filters of differing sizes, allowing learning at multiple scales. We can implement an inception module directly using the Keras functional API. The function below will create a single inception module with a specified number of filters for each of the parallel convolutional layers. From the GoogLeNet architecture described in the paper, it does not appear to use a systematic number of filters for parallel convolutional layers as the model is highly optimized. As such, we can parameterize the module definition so that we can specify the number of filters to use in each of the 1×1 , 3×3 , and 5×5 filters.

```
# function for creating a naive inception block
def inception_module(layer_in, f1, f2, f3):
    # 1x1 conv
    conv1 = Conv2D(f1, (1,1), padding='same', activation='relu')(layer_in)
    # 3x3 conv
    conv3 = Conv2D(f2, (3,3), padding='same', activation='relu')(layer_in)
    # 5x5 conv
    conv5 = Conv2D(f3, (5,5), padding='same', activation='relu')(layer_in)
    # 3x3 max pooling
    pool = MaxPooling2D((3,3), strides=(1,1), padding='same')(layer_in)
    # concatenate filters, assumes filters/channels last
    layer_out = concatenate([conv1, conv3, conv5, pool], axis=-1)
    return layer_out
```

Listing 17.6: Example of a function for defining a naive inception module.

To use the function, provide the reference to the prior layer as input, the number of filters, and it will return a reference to the concatenated filters layer that you can then connect to more inception modules or a submodel for making a prediction. We can demonstrate how to use this function by creating a model with a single inception module. In this case, the number of filters is based on *inception (3a)* from Table 1 in the paper. The complete example is listed below.

```
# example of creating a CNN with an inception module
from keras.models import Model
from keras.layers import Input
```

```

from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers.merge import concatenate
from keras.utils import plot_model

# function for creating a naive inception block
def naive_inception_module(layer_in, f1, f2, f3):
    # 1x1 conv
    conv1 = Conv2D(f1, (1,1), padding='same', activation='relu')(layer_in)
    # 3x3 conv
    conv3 = Conv2D(f2, (3,3), padding='same', activation='relu')(layer_in)
    # 5x5 conv
    conv5 = Conv2D(f3, (5,5), padding='same', activation='relu')(layer_in)
    # 3x3 max pooling
    pool = MaxPooling2D((3,3), strides=(1,1), padding='same')(layer_in)
    # concatenate filters, assumes filters/channels last
    layer_out = concatenate([conv1, conv3, conv5, pool], axis=-1)
    return layer_out

# define model input
visible = Input(shape=(256, 256, 3))
# add inception module
layer = naive_inception_module(visible, 64, 128, 32)
# create model
model = Model(inputs=visible, outputs=layer)
# summarize model
model.summary()
# plot model architecture
plot_model(model, show_shapes=True, to_file='naive_inception_module.png')

```

Listing 17.7: Example of creating and summarizing a model with a naive inception module.

Running the example creates the model and summarizes the layers. We know the convolutional and pooling layers are parallel, but this summary does not capture the structure easily.

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 256, 256, 3)	0	
conv2d_1 (Conv2D)	(None, 256, 256, 64)	256	input_1[0][0]
conv2d_2 (Conv2D)	(None, 256, 256, 128)	3584	input_1[0][0]
conv2d_3 (Conv2D)	(None, 256, 256, 32)	2432	input_1[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 256, 256, 3)	0	input_1[0][0]
concatenate_1 (Concatenate)	(None, 256, 256, 227)	0	conv2d_1[0][0] conv2d_2[0][0] conv2d_3[0][0] max_pooling2d_1[0][0]
<hr/>			
Total params: 6,272			
Trainable params: 6,272			

```
Non-trainable params: 0
```

Listing 17.8: Example output from creating and summarizing a model with a naive inception module.

A plot of the model architecture is also created that helps to clearly see the parallel structure of the module as well as the matching shapes of the output of each element of the module that allows their direct concatenation by the third dimension (filters or channels).

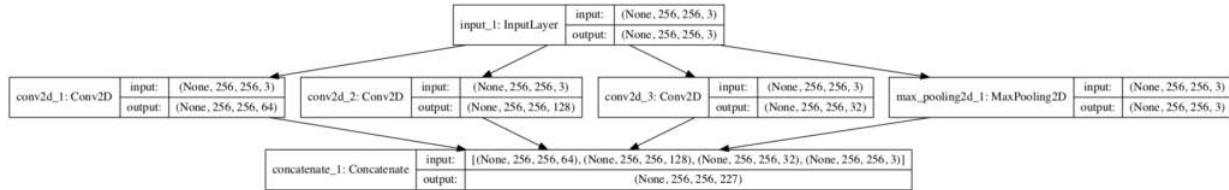


Figure 17.3: Plot of Convolutional Neural Network Architecture With a Naive Inception Module.

The version of the inception module that we have implemented is called the naive inception module. A modification to the module was made in order to reduce the amount of computation required. Specifically, 1×1 convolutional layers were added to reduce the number of filters before the 3×3 and 5×5 convolutional layers, and to increase the number of filters after the pooling layer.

This leads to the second idea of the Inception architecture: judiciously reducing dimension wherever the computational requirements would increase too much otherwise. [...] That is, 1×1 convolutions are used to compute reductions before the expensive 3×3 and 5×5 convolutions. Besides being used as reductions, they also include the use of rectified linear activation making them dual-purpose

— *Going Deeper with Convolutions*, 2015.

If you intend to use many inception modules in your model, you may require this computational performance-based modification. The function below implements this optimization improvement with parameterization so that you can control the amount of reduction in the number of filters prior to the 3×3 and 5×5 convolutional layers and the number of increased filters after max pooling.

```
# function for creating a projected inception module
def inception_module(layer_in, f1, f2_in, f2_out, f3_in, f3_out, f4_out):
    # 1x1 conv
    conv1 = Conv2D(f1, (1,1), padding='same', activation='relu')(layer_in)
    # 3x3 conv
    conv3 = Conv2D(f2_in, (1,1), padding='same', activation='relu')(layer_in)
    conv3 = Conv2D(f2_out, (3,3), padding='same', activation='relu')(conv3)
    # 5x5 conv
    conv5 = Conv2D(f3_in, (1,1), padding='same', activation='relu')(layer_in)
    conv5 = Conv2D(f3_out, (5,5), padding='same', activation='relu')(conv5)
    # 3x3 max pooling
    pool = MaxPooling2D((3,3), strides=(1,1), padding='same')(layer_in)
    pool = Conv2D(f4_out, (1,1), padding='same', activation='relu')(pool)
```

```
# concatenate filters, assumes filters/channels last
layer_out = concatenate([conv1, conv3, conv5, pool], axis=-1)
return layer_out
```

Listing 17.9: Example of a function for defining an efficient inception module.

We can create a model with two of these optimized inception modules to get a concrete idea of how the architecture looks in practice. In this case, the number of filter configurations are based on *inception (3a)* and *inception (3b)* from Table 1 in the paper. The complete example is listed below.

```
# example of creating a CNN with an efficient inception module
from keras.models import Model
from keras.layers import Input
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers.merge import concatenate
from keras.utils import plot_model

# function for creating a projected inception module
def inception_module(layer_in, f1, f2_in, f2_out, f3_in, f3_out, f4_out):
    # 1x1 conv
    conv1 = Conv2D(f1, (1,1), padding='same', activation='relu')(layer_in)
    # 3x3 conv
    conv3 = Conv2D(f2_in, (1,1), padding='same', activation='relu')(layer_in)
    conv3 = Conv2D(f2_out, (3,3), padding='same', activation='relu')(conv3)
    # 5x5 conv
    conv5 = Conv2D(f3_in, (1,1), padding='same', activation='relu')(layer_in)
    conv5 = Conv2D(f3_out, (5,5), padding='same', activation='relu')(conv5)
    # 3x3 max pooling
    pool = MaxPooling2D((3,3), strides=(1,1), padding='same')(layer_in)
    pool = Conv2D(f4_out, (1,1), padding='same', activation='relu')(pool)
    # concatenate filters, assumes filters/channels last
    layer_out = concatenate([conv1, conv3, conv5, pool], axis=-1)
    return layer_out

# define model input
visible = Input(shape=(256, 256, 3))
# add inception block 1
layer = inception_module(visible, 64, 96, 128, 16, 32, 32)
# add inception block 1
layer = inception_module(layer, 128, 128, 192, 32, 96, 64)
# create model
model = Model(inputs=visible, outputs=layer)
# summarize model
model.summary()
# plot model architecture
plot_model(model, show_shapes=True, to_file='inception_module.png')
```

Listing 17.10: Example of creating and summarizing a model with an optimized inception module.

Running the example creates a linear summary of the layers that does not really help to understand what is going on. The output is omitted here for brevity. A plot of the model architecture is created that does make the layout of each module clear and how the first model feeds the second module. Note that the first 1×1 convolution in each inception module is on

the far right for space reasons, but besides that, the other layers are organized left to right within each module.

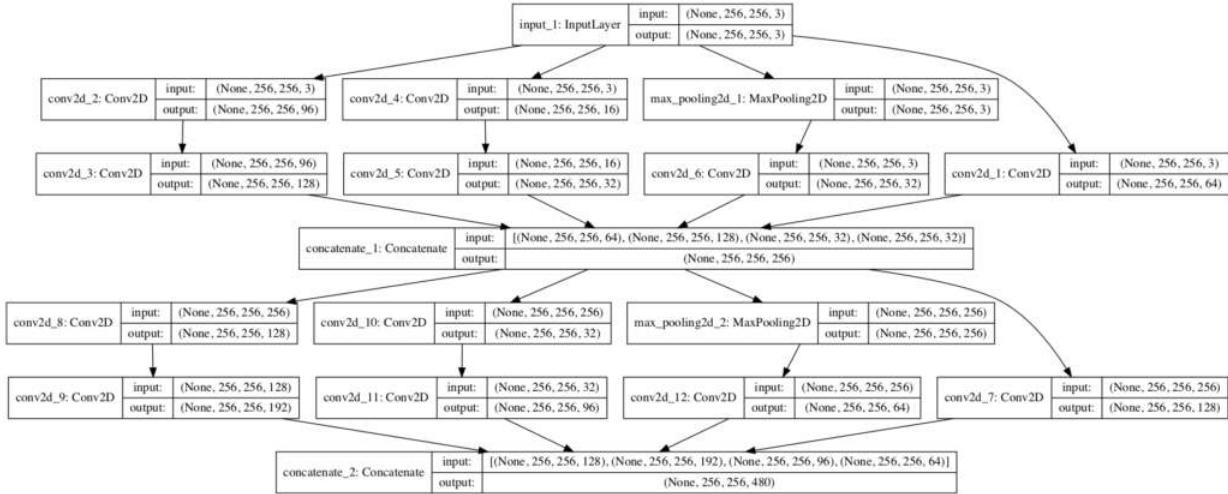


Figure 17.4: Plot of Convolutional Neural Network Architecture With a Efficient Inception Module.

17.4 How to Implement the Residual Module

The Residual Network, or ResNet, architecture for convolutional neural networks was proposed by Kaiming He, et al. in their 2016 paper titled *Deep Residual Learning for Image Recognition*, which achieved success on the 2015 version of the ILSVRC challenge (introduced in Chapter 15). A key innovation in the ResNet was the residual module. The residual module, specifically the identity residual model, is a block of two convolutional layers with the same number of filters and a small filter size where the output of the second layer is added with the input to the first convolutional layer. Drawn as a graph, the input to the module is added to the output of the module and is called a shortcut connection. We can implement this directly in Keras using the functional API and the `add()` merge function.

```
# function for creating an identity residual module
def residual_module(layer_in, n_filters):
    # conv1
    conv1 = Conv2D(n_filters, (3,3), padding='same', activation='relu',
                  kernel_initializer='he_normal')(layer_in)
    # conv2
    conv2 = Conv2D(n_filters, (3,3), padding='same', activation='linear',
                  kernel_initializer='he_normal')(conv1)
    # add filters, assumes filters/channels last
    layer_out = add([conv2, layer_in])
    # activation function
    layer_out = Activation('relu')(layer_out)
    return layer_out
```

Listing 17.11: Example of a function for defining an identity residual module.

A limitation with this direct implementation is that if the number of filters in the input layer does not match the number of filters in the last convolutional layer of the module (defined

by `n_filters`), then we will get an error. One solution is to use a 1×1 convolution layer, often referred to as a projection layer, to either increase the number of filters for the input layer or reduce the number of filters for the last convolutional layer in the module. The former solution makes more sense, and is the approach proposed in the paper, referred to as a projection shortcut.

When the dimensions increase [...], we consider two options: (A) The shortcut still performs identity mapping, with extra zero entries padded for increasing dimensions. This option introduces no extra parameter; (B) The projection shortcut [...] is used to match dimensions (done by 1×1 convolutions).

— Deep Residual Learning for Image Recognition, 2015.

Below is an updated version of the function that will use the identity if possible, otherwise a projection of the number of filters in the input does not match the `n_filters` argument.

```
# function for creating an identity or projection residual module
def residual_module(layer_in, n_filters):
    merge_input = layer_in
    # check if the number of filters needs to be increase, assumes channels last format
    if layer_in.shape[-1] != n_filters:
        merge_input = Conv2D(n_filters, (1,1), padding='same', activation='relu',
                             kernel_initializer='he_normal')(layer_in)
    # conv1
    conv1 = Conv2D(n_filters, (3,3), padding='same', activation='relu',
                   kernel_initializer='he_normal')(layer_in)
    # conv2
    conv2 = Conv2D(n_filters, (3,3), padding='same', activation='linear',
                   kernel_initializer='he_normal')(conv1)
    # add filters, assumes filters/channels last
    layer_out = add([conv2, merge_input])
    # activation function
    layer_out = Activation('relu')(layer_out)
    return layer_out
```

Listing 17.12: Example of a function for defining an projection residual module.

We can demonstrate the usage of this module in a simple model. The complete example is listed below.

```
# example of a CNN model with an identity or projection residual module
from keras.models import Model
from keras.layers import Input
from keras.layers import Activation
from keras.layers import Conv2D
from keras.layers import add
from keras.utils import plot_model

# function for creating an identity or projection residual module
def residual_module(layer_in, n_filters):
    merge_input = layer_in
    # check if the number of filters needs to be increase, assumes channels last format
    if layer_in.shape[-1] != n_filters:
        merge_input = Conv2D(n_filters, (1,1), padding='same', activation='relu',
                             kernel_initializer='he_normal')(layer_in)
```

```

# conv1
conv1 = Conv2D(n_filters, (3,3), padding='same', activation='relu',
    kernel_initializer='he_normal')(layer_in)
# conv2
conv2 = Conv2D(n_filters, (3,3), padding='same', activation='linear',
    kernel_initializer='he_normal')(conv1)
# add filters, assumes filters/channels last
layer_out = add([conv2, merge_input])
# activation function
layer_out = Activation('relu')(layer_out)
return layer_out

# define model input
visible = Input(shape=(256, 256, 3))
# add vgg module
layer = residual_module(visible, 64)
# create model
model = Model(inputs=visible, outputs=layer)
# summarize model
model.summary()
# plot model architecture
plot_model(model, show_shapes=True, to_file='residual_module.png')

```

Listing 17.13: Example of creating and summarizing a model with a residual module.

Running the example first creates the model then prints a summary of the layers. Because the module is linear, this summary is helpful to see what is going on.

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 256, 256, 3)	0	
conv2d_2 (Conv2D)	(None, 256, 256, 64)	1792	input_1[0] [0]
conv2d_3 (Conv2D)	(None, 256, 256, 64)	36928	conv2d_2[0] [0]
conv2d_1 (Conv2D)	(None, 256, 256, 64)	256	input_1[0] [0]
add_1 (Add)	(None, 256, 256, 64)	0	conv2d_3[0] [0] conv2d_1[0] [0]
activation_1 (Activation)	(None, 256, 256, 64)	0	add_1[0] [0]

Total params: 38,976
Trainable params: 38,976
Non-trainable params: 0

Listing 17.14: Example output from creating and summarizing a model with a residual module.

A plot of the model architecture is also created. We can see the module with the inflation of the number of filters in the input and the addition of the two elements at the end of the module.

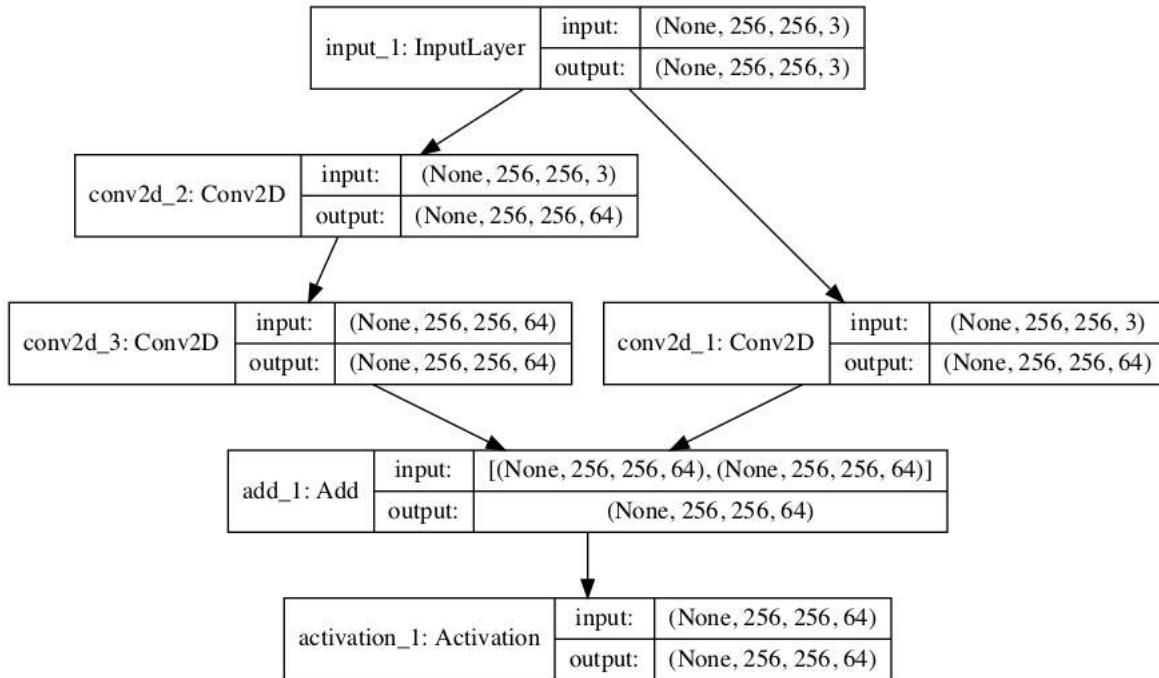


Figure 17.5: Plot of Convolutional Neural Network Architecture With an Residual Module.

The paper describes other types of residual connections such as bottlenecks. These are left as an exercise for the reader that can be implemented easily by updating the `residual_module()` function.

17.5 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **New Configuration.** Select a module and experiment with different configurations and review the resulting model summary.
- **Code Review.** Review the Keras implementation for one of the modules and list any differences in the implementation or resulting model summary.
- **Hybrid.** Develop a hybrid module using elements of the inception module and the residual net module.

If you explore any of these extensions, I'd love to know.

17.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

17.6.1 Papers

- *ImageNet Classification with Deep Convolutional Neural Networks*, 2012.
<https://dl.acm.org/citation.cfm?id=3065386>
- *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2014.
<https://arxiv.org/abs/1409.1556>
- *Going Deeper with Convolutions*, 2015.
<https://arxiv.org/abs/1409.4842>
- *Deep Residual Learning for Image Recognition*, 2016.
<https://arxiv.org/abs/1512.03385>

17.6.2 API

- Keras Functional API.
<https://keras.io/models/model/>
- Keras Applications API.
<https://keras.io/applications/>
- Keras Applications Source Code.
<https://github.com/keras-team/keras-applications>

17.7 Summary

In this tutorial, you discovered how to implement key architecture elements from milestone convolutional neural network models, from scratch. Specifically, you learned:

- How to implement a VGG module used in the VGG-16 and VGG-19 convolutional neural network models.
- How to implement the naive and optimized inception module used in the GoogLeNet model.
- How to implement the identity residual module used in the ResNet model.

17.7.1 Next

In the next section, you will discover the value of pre-trained models and their wide use in transfer learning.

Chapter 18

How to Use Pre-Trained Models and Transfer Learning

Deep convolutional neural network models may take days or even weeks to train on very large datasets. A way to short-cut this process is to re-use the model weights from pre-trained models that were developed for standard computer vision benchmark datasets, such as the ImageNet image recognition tasks. Top performing models can be downloaded and used directly, or integrated into a new model for your own computer vision problems. In this tutorial, you will discover how to use transfer learning when developing convolutional neural networks for computer vision applications. After reading this tutorial, you will know:

- Transfer learning involves using models trained on one problem as a starting point on a related problem.
- Transfer learning is flexible, allowing the use of pre-trained models directly, as feature extraction preprocessing, and integrated into entirely new models.
- Keras provides convenient access to many top performing models on the ImageNet image recognition tasks such as VGG, Inception, and ResNet.

Let's get started.

18.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. What Is Transfer Learning?
2. Transfer Learning for Image Recognition
3. How to Use Pre-Trained Models
4. Models for Transfer Learning
5. Examples of Using Pre-Trained Models

18.2 What Is Transfer Learning?

Transfer learning generally refers to a process where a model trained on one problem is used in some way on a second, related problem. In deep learning, transfer learning is a technique whereby a neural network model is first trained on a problem similar to the problem that is being solved. One or more layers from the trained model are then used in a new model trained on the problem of interest.

This is typically understood in a supervised learning context, where the input is the same but the target may be of a different nature. For example, we may learn about one set of visual categories, such as cats and dogs, in the first setting, then learn about a different set of visual categories, such as ants and wasps, in the second setting.

— Page 536, *Deep Learning*, 2016.

Transfer learning has the benefit of decreasing the training time for a neural network model and can result in lower generalization error. The weights in re-used layers may be used as the starting point for the training process and adapted in response to the new problem. This usage treats transfer learning as a type of weight initialization scheme. This may be useful when the first related problem has a lot more labeled data than the problem of interest and the similarity in the structure of the problem may be useful in both contexts.

... the objective is to take advantage of data from the first setting to extract information that may be useful when learning or even when directly making predictions in the second setting.

— Page 538, *Deep Learning*, 2016.

18.3 Transfer Learning for Image Recognition

A range of high-performing models have been developed for image classification and demonstrated on the annual ImageNet Large Scale Visual Recognition Challenge, or ILSVRC. This challenge, often referred to simply as ImageNet, given the source of the image used in the competition, has resulted in a number of innovations in the architecture and training of convolutional neural networks. In addition, many of the models used in the competitions have been released under a permissive license. These models can be used as the basis for transfer learning in computer vision applications. This is desirable for a number of reasons, not least:

- **Useful Learned Features:** The models have learned how to detect generic features from photographs, given that they were trained on more than 1,000,000 images for 1,000 categories.
- **State-of-the-Art Performance:** The models achieved state-of-the-art performance and remain effective on the specific image recognition task for which they were developed.
- **Easily Accessible:** The model weights are provided as free downloadable files and many libraries provide convenient APIs to download and use the models directly.

The model weights can be downloaded and used in the same model architecture using a range of different deep learning libraries, including Keras.

18.4 How to Use Pre-Trained Models

The use of a pre-trained model is limited only by your creativity. For example, a model may be downloaded and used as-is, such as embedded into an application and used to classify new photographs. Alternately, models may be downloaded and used as feature extraction models. Here, the output of the model from a layer prior to the output layer of the model is used as input to a new classifier model. Recall that convolutional layers closer to the input layer of the model learn low-level features such as lines, that layers in the middle of the layer learn complex abstract features that combine the lower level features extracted from the input, and layers closer to the output interpret the extracted features in the context of a classification task.

Armed with this understanding, a level of detail for feature extraction from an existing pre-trained model can be chosen. For example, if a new task is quite different from classifying objects in photographs (e.g. different to ImageNet), then perhaps the output of the pre-trained model after the first few layers would be appropriate. If a new task is quite similar to the task of classifying objects in photographs, then perhaps the output from layers much deeper in the model can be used, or even the output of the fully connected layer prior to the output layer can be used.

The pre-trained model can be used as a separate feature extraction program, in which case input can be pre-processed by the model or portion of the model to a given an output (e.g. vector of numbers) for each input image, that can then used as input when training a new model. Alternately, the pre-trained model or desired portion of the model can be integrated directly into a new neural network model. In this usage, the weights of the pre-trained can be frozen so that they are not updated as the new model is trained. Alternately, the weights may be updated during the training of the new model, perhaps with a lower learning rate, allowing the pre-trained model to act like a weight initialization scheme when training the new model. We can summarize some of these usage patterns as follows:

- **Classifier:** The pre-trained model is used directly to classify new images.
- **Standalone Feature Extractor:** The pre-trained model, or some portion of the model, is used to pre-process images and extract relevant features.
- **Integrated Feature Extractor:** The pre-trained model, or some portion of the model, is integrated into a new model, but layers of the pre-trained model are frozen during training.
- **Weight Initialization:** The pre-trained model, or some portion of the model, is integrated into a new model, and the layers of the pre-trained model are trained in concert with the new model.

Each approach can be effective and save significant time in developing and training a deep convolutional neural network model. It may not be clear as to which usage of the pre-trained model may yield the best results on your new computer vision task, therefore some experimentation may be required.

18.5 Models for Transfer Learning

There are perhaps a dozen or more top-performing models for image recognition that can be downloaded and used as the basis for image recognition and related computer vision tasks. Perhaps three of the more popular models are as follows:

- VGG (e.g. VGG16 or VGG19).
- GoogLeNet (e.g. InceptionV3).
- Residual Network (e.g. ResNet50).

These models are both widely used for transfer learning both because of their performance, but also because they were examples that introduced specific architectural innovations, namely consistent and repeating structures (VGG), inception modules (GoogLeNet), and residual modules (ResNet). Keras provides access to a number of top-performing pre-trained models that were developed for image recognition tasks. They are available via the Applications API, and include functions to load a model with or without the pre-trained weights, and prepare data in a way that a given model may expect (e.g. scaling of size and pixel values).

The first time a pre-trained model is loaded, Keras will download the required model weights, which may take some time given the speed of your internet connection. Weights are stored in the `.keras/models/` directory under your home directory and will be loaded from this location the next time that they are used. When loading a given model, the `include_top` argument can be set to `False`, in which case the fully-connected output layers of the model used to make predictions is not loaded, allowing a new output layer to be added and trained. For example:

```
...
# load model without output layer
model = VGG16(include_top=False)
```

Listing 18.1: Example of a loading a model without the top.

Additionally, when the `include_top` argument is `False`, the `input_tensor` argument must be specified, allowing the expected fixed-sized input of the model to be changed. For example:

```
...
# load model and specify a new input shape for images
new_input = Input(shape=(640, 480, 3))
model = VGG16(include_top=False, input_tensor=new_input)
```

Listing 18.2: Example of a loading a model without the top and defining the input shape.

A model without a top will output activations from the last convolutional or pooling layer directly. One approach to summarizing these activations for their use in a classifier or as a feature vector representation of input is to add a global pooling layer, such as a max global pooling or average global pooling. The result is a vector that can be used as a feature descriptor for an input (e.g. a vector of *extracted features*). Keras provides this capability directly via the `pooling` argument that can be set to '`avg`' or '`max`'. For example:

```
...
# load model and specify a new input shape for images and avg pooling output
new_input = Input(shape=(640, 480, 3))
model = VGG16(include_top=False, input_tensor=new_input, pooling='avg')
```

Listing 18.3: Example of a loading a model without the top and using average pooling.

Images can be prepared for a given model using the `preprocess_input()` function; e.g., pixel scaling is performed in a way that was performed to images in the training dataset when the model was developed. For example:

```
...
# prepare an image
from keras.applications.vgg16 import preprocess_input
images = ...
prepared_images = preprocess_input(images)
```

Listing 18.4: Example of using model-specific data preparation.

Finally, you may wish to use a model architecture on your dataset, but not use the pre-trained weights, and instead initialize the model with random weights and train the model from scratch. This can be achieved by setting the `weights` argument to `None` instead of the default '`imagenet`'. Additionally, the `classes` argument can be set to define the number of classes in your dataset, which will then be configured for you in the output layer of the model. For example:

```
...
# define a new model with random weights and 10 classes
new_input = Input(shape=(640, 480, 3))
model = VGG16(weights=None, input_tensor=new_input, classes=10)
```

Listing 18.5: Example of not using pre-trained model weights.

Now that we are familiar with the API, let's take a look at loading three models using the Keras Applications API.

18.5.1 Load the VGG16 Pre-trained Model

The VGG16 model was developed by the Visual Graphics Group (VGG) at Oxford and was described in the 2014 paper titled `Very Deep Convolutional Networks for Large-Scale Image Recognition`. By default, the model expects color input images to be rescaled to the size of 224×224 squares. The model can be loaded as follows:

```
# example of loading the vgg16 model
from keras.applications.vgg16 import VGG16
# load model
model = VGG16()
# summarize the model
model.summary()
```

Listing 18.6: Example of loading and summarizing a VGG-16 model.

Running the example will load the VGG16 model and download the model weights if required. The model can then be used directly to classify a photograph into one of 1,000 classes. In this case, the model architecture is summarized to confirm that it was loaded correctly.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792

```

block1_conv2 (Conv2D)      (None, 224, 224, 64) 36928
-----
block1_pool (MaxPooling2D) (None, 112, 112, 64) 0
-----
block2_conv1 (Conv2D)      (None, 112, 112, 128) 73856
-----
block2_conv2 (Conv2D)      (None, 112, 112, 128) 147584
-----
block2_pool (MaxPooling2D) (None, 56, 56, 128) 0
-----
block3_conv1 (Conv2D)      (None, 56, 56, 256) 295168
-----
block3_conv2 (Conv2D)      (None, 56, 56, 256) 590080
-----
block3_conv3 (Conv2D)      (None, 56, 56, 256) 590080
-----
block3_pool (MaxPooling2D) (None, 28, 28, 256) 0
-----
block4_conv1 (Conv2D)      (None, 28, 28, 512) 1180160
-----
block4_conv2 (Conv2D)      (None, 28, 28, 512) 2359808
-----
block4_conv3 (Conv2D)      (None, 28, 28, 512) 2359808
-----
block4_pool (MaxPooling2D) (None, 14, 14, 512) 0
-----
block5_conv1 (Conv2D)      (None, 14, 14, 512) 2359808
-----
block5_conv2 (Conv2D)      (None, 14, 14, 512) 2359808
-----
block5_conv3 (Conv2D)      (None, 14, 14, 512) 2359808
-----
block5_pool (MaxPooling2D) (None, 7, 7, 512) 0
-----
flatten (Flatten)         (None, 25088) 0
-----
fc1 (Dense)               (None, 4096) 102764544
-----
fc2 (Dense)               (None, 4096) 16781312
-----
predictions (Dense)        (None, 1000) 4097000
=====

Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
-----
```

Listing 18.7: Example output from loading and summarizing a VGG-16 model.

18.5.2 Load the InceptionV3 Pre-Trained Model

The InceptionV3 is the third iteration of the inception architecture, first developed for the GoogLeNet model. This model was developed by researchers at Google and described in the 2015

paper titled [Rethinking the Inception Architecture for Computer Vision](#). The model expects color images to have the square shape 299×299 . The model can be loaded as follows:

```
# example of loading the inception v3 model
from keras.applications.inception_v3 import InceptionV3
# load model
model = InceptionV3()
# summarize the model
model.summary()
```

Listing 18.8: Example of loading and summarizing a Inception model.

Running the example will load the model, downloading the weights if required, and then summarize the model architecture to confirm it was loaded correctly. The output is omitted in this case for brevity, as it is a deep model with many layers.

18.5.3 Load the ResNet50 Pre-trained Model

The Residual Network, or ResNet for short, is a model that makes use of the residual module involving shortcut connections. It was developed by researchers at Microsoft and described in the 2015 paper titled [Deep Residual Learning for Image Recognition](#). The model expects color images to have the square shape 224×224 .

```
# example of loading the resnet50 model
from keras.applications.resnet50 import ResNet50
# load model
model = ResNet50()
# summarize the model
model.summary()
```

Listing 18.9: Example of loading and summarizing a ResNet model.

Running the example will load the model, downloading the weights if required, and then summarize the model architecture to confirm it was loaded correctly. The output is omitted in this case for brevity, as it is a deep model.

18.6 Examples of Using Pre-Trained Models

Now that we are familiar with how to load pre-trained models in Keras, let's look at some examples of how they might be used in practice. In these examples, we will work with the VGG16 model as it is a relatively straightforward model to use and a simple model architecture to understand. We also need a photograph to work with in these examples. Below is a photograph of a dog, taken by Justin Morgan¹ and made available under a permissive license.

¹<https://www.flickr.com/photos/jmorgan/5164287/>



Figure 18.1: Photograph of a Dog.

Download the photograph and place it in your current working directory with the filename `dog.jpg`.

- Download the Photograph of a Dog (`dog.jpg`).²

18.6.1 Pre-Trained Model as Classifier

A pre-trained model can be used directly to classify new photographs as one of the 1,000 known classes in the image classification task in the ILSVRC. We will use the VGG16 model to classify new images. First, the photograph needs to be loaded and reshaped to a 224×224 square, expected by the model, and the pixel values scaled in the way expected by the model. The model operates on an array of samples, therefore the dimensions of a loaded image need to be expanded by 1, for one image with 224×224 pixels and three channels.

```
# load an image from file
image = load_img('dog.jpg', target_size=(224, 224))
# convert the image pixels to a numpy array
image = img_to_array(image)
# reshape data for the model
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
# prepare the image for the VGG model
image = preprocess_input(image)
```

Listing 18.10: Example of preparing the image for input to the vgg model.

Next, the model can be loaded and a prediction made. This means that a predicted probability of the photo belonging to each of the 1,000 classes is made. In this example, we are only concerned with the most likely class, so we can decode the predictions and retrieve the label or name of the class with the highest probability.

²<https://machinelearningmastery.com/wp-content/uploads/2019/02/dog.jpg>

```
# predict the probability across all output classes
yhat = model.predict(image)
# convert the probabilities to class labels
label = decode_predictions(yhat)
# retrieve the most likely result, e.g. highest probability
label = label[0][0]
```

Listing 18.11: Example of making a prediction with the vgg model.

Tying all of this together, the complete example below loads a new photograph and predicts the most likely class.

```
# example of using a pre-trained model as a classifier
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.applications.vgg16 import decode_predictions
from keras.applications.vgg16 import VGG16
# load an image from file
image = load_img('dog.jpg', target_size=(224, 224))
# convert the image pixels to a numpy array
image = img_to_array(image)
# reshape data for the model
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
# prepare the image for the VGG model
image = preprocess_input(image)
# load the model
model = VGG16()
# predict the probability across all output classes
yhat = model.predict(image)
# convert the probabilities to class labels
label = decode_predictions(yhat)
# retrieve the most likely result, e.g. highest probability
label = label[0][0]
# print the classification
print('%s (%.2f%%)' % (label[1], label[2]*100))
```

Listing 18.12: Example of classifying a photograph using a pre-trained classifier model.

Running the example predicts more than just dog; it also predicts the specific breed of Doberman with a probability of 33.59%, which may, in fact, be correct.

```
Doberman (33.59%)
```

Listing 18.13: Example output from classifying a photograph using a pre-trained classifier model.

18.6.2 Pre-Trained Model as Feature Extractor Preprocessor

The pre-trained model may be used as a standalone program to extract features from new photographs. Specifically, the extracted features of a photograph may be a vector of numbers that the model will use to describe the specific features in a photograph. These features can then be used as input in the development of a new model. The last few layers of the VGG16 model are fully connected layers prior to the output layer. These layers will provide a complex

set of features to describe a given input image and may provide useful input when training a new model for image classification or related computer vision task.

The image can be loaded and prepared for the model, as we did before in the previous example. We will load the model with the classifier output part of the model, but manually remove the final output layer. This means that the second last fully connected layer with 4,096 nodes will be the new output layer.

```
# load model
model = VGG16()
# remove the output layer
model = Model(inputs=model.inputs, outputs=model.layers[-2].output)
```

Listing 18.14: Example of removing the output layer of a pre-trained model.

This vector of 4,096 numbers will be used to represent the complex features of a given input image that can then be saved to file to be loaded later and used as input to train a new model. We can save it as a pickle file.

```
# get extracted features
features = model.predict(image)
print(features.shape)
# save to file
dump(features, open('dog.pkl', 'wb'))
```

Listing 18.15: Example of saving a predicted feature vector to file.

Tying all of this together, the complete example of using the model as a standalone feature extraction model is listed below.

```
# example of using the vgg16 model as a feature extraction model
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.applications.vgg16 import VGG16
from keras.models import Model
from pickle import dump
# load an image from file
image = load_img('dog.jpg', target_size=(224, 224))
# convert the image pixels to a numpy array
image = img_to_array(image)
# reshape data for the model
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
# prepare the image for the VGG model
image = preprocess_input(image)
# load model
model = VGG16()
# remove the output layer
model = Model(inputs=model.inputs, outputs=model.layers[-2].output)
# get extracted features
features = model.predict(image)
print(features.shape)
# save to file
dump(features, open('dog.pkl', 'wb'))
```

Listing 18.16: Example of a standalone pre-trained feature extractor model.

Running the example loads the photograph, then prepares the model as a feature extraction model. The features are extracted from the loaded photo and the shape of the feature vector is printed, showing it has 4,096 numbers. This feature vector is then saved to a new file `dog.pkl` in the current working directory.

```
(1, 4096)
```

Listing 18.17: Example output from a standalone pre-trained feature extractor model.

This process could be repeated for each photo in a new training dataset.

18.6.3 Pre-Trained Model as Feature Extractor in Model

We can use some or all of the layers in a pre-trained model as a feature extraction component of a new model directly. This can be achieved by loading the model, then simply adding new layers. This may involve adding new convolutional and pooling layers to expand upon the feature extraction capabilities of the model or adding new fully connected classifier type layers to learn how to interpret the extracted features on a new dataset, or some combination. For example, we can load the VGG16 models without the classifier part of the model by specifying the `include_top` argument to `False`, and specify the preferred shape of the images in our new dataset as 300×300 .

```
# load model without classifier layers
model = VGG16(include_top=False, input_shape=(300, 300, 3))
```

Listing 18.18: Example of loading the pre-trained model without the classifier output model.

We can then use the Keras functional API to add a new `Flatten` layer after the last pooling layer in the VGG16 model, then define a new classifier model with a `Dense` fully connected layer and an output layer that will predict the probability for 10 classes.

```
# add new classifier layers
flat1 = Flatten()(model.layers[-1].output)
class1 = Dense(1024, activation='relu')(flat1)
output = Dense(10, activation='softmax')(class1)
# define new model
model = Model(inputs=model.inputs, outputs=output)
```

Listing 18.19: Example of adding new classifier output layers to the pre-trained model.

An alternative approach to adding a `Flatten` layer would be to define the VGG16 model with an average pooling layer, and then add fully connected layers. Perhaps try both approaches on your application and see which results in the best performance. The weights of the VGG16 model and the weights for the new model will all be trained together on the new dataset. The complete example is listed below.

```
# example of tending the vgg16 model
from keras.applications.vgg16 import VGG16
from keras.models import Model
from keras.layers import Dense
from keras.layers import Flatten
# load model without classifier layers
model = VGG16(include_top=False, input_shape=(300, 300, 3))
# add new classifier layers
flat1 = Flatten()(model.layers[-1].output)
```

```

class1 = Dense(1024, activation='relu')(flat1)
output = Dense(10, activation='softmax')(class1)
# define new model
model = Model(inputs=model.inputs, outputs=output)
# summarize
model.summary()
# ...

```

Listing 18.20: Example of a using a pre-trained model with new classifier output layers.

Running the example defines the new model ready for training and summarizes the model architecture. We can see that we have flattened the output of the last pooling layer and added our new fully connected layers.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 300, 300, 3)	0
block1_conv1 (Conv2D)	(None, 300, 300, 64)	1792
block1_conv2 (Conv2D)	(None, 300, 300, 64)	36928
block1_pool (MaxPooling2D)	(None, 150, 150, 64)	0
block2_conv1 (Conv2D)	(None, 150, 150, 128)	73856
block2_conv2 (Conv2D)	(None, 150, 150, 128)	147584
block2_pool (MaxPooling2D)	(None, 75, 75, 128)	0
block3_conv1 (Conv2D)	(None, 75, 75, 256)	295168
block3_conv2 (Conv2D)	(None, 75, 75, 256)	590080
block3_conv3 (Conv2D)	(None, 75, 75, 256)	590080
block3_pool (MaxPooling2D)	(None, 37, 37, 256)	0
block4_conv1 (Conv2D)	(None, 37, 37, 512)	1180160
block4_conv2 (Conv2D)	(None, 37, 37, 512)	2359808
block4_conv3 (Conv2D)	(None, 37, 37, 512)	2359808
block4_pool (MaxPooling2D)	(None, 18, 18, 512)	0
block5_conv1 (Conv2D)	(None, 18, 18, 512)	2359808
block5_conv2 (Conv2D)	(None, 18, 18, 512)	2359808
block5_conv3 (Conv2D)	(None, 18, 18, 512)	2359808
block5_pool (MaxPooling2D)	(None, 9, 9, 512)	0
flatten_1 (Flatten)	(None, 41472)	0

```

-----  

dense_1 (Dense)           (None, 1024)      42468352  

-----  

dense_2 (Dense)           (None, 10)        10250  

=====  

Total params: 57,193,290  

Trainable params: 57,193,290  

Non-trainable params: 0  

-----
```

Listing 18.21: Example output from using a pre-trained model with new classifier output layers.

Alternately, we may wish to use the VGG16 model layers, but train the new layers of the model without updating the weights of the VGG16 layers. This will allow the new output layers to learn to interpret the learned features of the VGG16 model. This can be achieved by setting the `trainable` property on each of the layers in the loaded VGG model to `False` prior to training. For example:

```

# load model without classifier layers
model = VGG16(include_top=False, input_shape=(300, 300, 3))
# mark loaded layers as not trainable
for layer in model.layers:
    layer.trainable = False
...
```

Listing 18.22: Example of marking some model layers as non-trainable.

You can pick and choose which layers are trainable. For example, perhaps you want to retrain some of the convolutional layers deep in the model, but none of the layers earlier in the model. For example:

```

# load model without classifier layers
model = VGG16(include_top=False, input_shape=(300, 300, 3))
# mark some layers as not trainable
model.get_layer('block1_conv1').trainable = False
model.get_layer('block1_conv2').trainable = False
model.get_layer('block2_conv1').trainable = False
model.get_layer('block2_conv2').trainable = False
...
```

Listing 18.23: Example of marking some model layers as non-trainable by name.

18.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Your Own Image.** Use the pre-trained VGG model to classifying a photograph of your own.
- **Alternate Model.** Experiment classifying photos using a different pre-trained model such as a ResNet.
- **Develop an SVM.** Use a pre-trained model to create feature vectors for all images in a small image classification dataset and fit an SVM using the embeddings as the input instead of the images.

If you explore any of these extensions, I'd love to know.

18.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

18.8.1 Books

- *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>

18.8.2 Papers

- *A Survey on Transfer Learning*, 2010.
<https://ieeexplore.ieee.org/document/5288526>
- *How Transferable Are Features In Deep Neural Networks?*, 2014.
<https://arxiv.org/abs/1411.1792>
- *CNN Features Off-the-shelf: An Astounding Baseline For Recognition*, 2014.
<https://arxiv.org/abs/1403.6382>

18.8.3 APIs

- Keras Applications API.
<https://keras.io/applications/>

18.8.4 Articles

- Transfer Learning, Wikipedia.
https://en.wikipedia.org/wiki/Transfer_learning
- Transfer Learning — Machine Learning's Next Frontier, 2017.
<http://ruder.io/transfer-learning/>

18.9 Summary

In this tutorial, you discovered how to use transfer learning when developing convolutional neural networks for computer vision applications. Specifically, you learned:

- Transfer learning involves using models trained on one problem as a starting point on a related problem.
- Transfer learning is flexible, allowing the use of pre-trained models directly as feature extraction preprocessing and integrated into entirely new models.
- Keras provides convenient access to many top performing models on the ImageNet image recognition tasks such as VGG, Inception, and ResNet.

18.9.1 Next

This was the final tutorial in this part on convolutional neural network model architectures. In the next part you will discover how to develop models for image classification.

Part V

Image Classification

Overview

In this part you will discover how to develop deep convolutional neural networks for a range of image classification tasks. After reading the chapters in this part, you will know:

- How to develop a convolutional neural network from scratch for clothing classification (Chapter [19](#)).
- How to develop a model from scratch with data augmentation for object photograph classification (Chapter [20](#)).
- How to develop a deep convolutional neural network model to classify photographs of dogs and cats (Chapter [21](#)).
- How to develop a deep convolutional neural network model to label satellite photographs of the Amazon rainforest (Chapter [22](#)).

Chapter 19

How to Classify Black and White Photos of Clothing

The Fashion-MNIST clothing classification problem is a new standard dataset used in computer vision and deep learning. Although the dataset is relatively simple, it can be used as the basis for learning and practicing how to develop, evaluate, and use deep convolutional neural networks for image classification from scratch. This includes how to develop a robust test harness for estimating the performance of the model, how to explore improvements to the model, and how to save the model and later load it to make predictions on new data. In this tutorial, you will discover how to develop a convolutional neural network for clothing classification from scratch. After completing this tutorial, you will know:

- How to develop a test harness to develop a robust evaluation of a model and establish a baseline of performance for a classification task.
- How to explore extensions to a baseline model to improve learning and model capacity.
- How to develop a finalized model, evaluate the performance of the final model, and use it to make predictions on new images.

Let's get started.

19.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Fashion-MNIST Clothing Classification
2. Model Evaluation Methodology
3. How to Develop a Baseline Model
4. How to Develop an Improved Model
5. How to Finalize the Model and Make Predictions

19.2 Fashion-MNIST Clothing Classification

The Fashion-MNIST dataset is proposed as a more challenging replacement dataset for the MNIST dataset. It is a dataset comprised of 60,000 small square 28×28 pixel grayscale images of items of 10 types of clothing, such as shoes, t-shirts, dresses, and more. The mapping of all 0-9 integers to class labels is listed below.

- 0: T-shirt/top
- 1: Trouser
- 2: Pullover
- 3: Dress
- 4: Coat
- 5: Sandal
- 6: Shirt
- 7: Sneaker
- 8: Bag
- 9: Ankle boot

It is a more challenging classification problem than MNIST and top results are achieved by deep learning convolutional neural networks with a classification accuracy of about 90% to 95% on the hold out test dataset. The example below loads the Fashion-MNIST dataset using the Keras API and creates a plot of the first nine images in the training dataset.

```
# example of loading the fashion mnist dataset
from matplotlib import pyplot
from keras.datasets import fashion_mnist
# load dataset
(trainX, trainy), (testX, testy) = fashion_mnist.load_data()
# summarize loaded dataset
print('Train: X=%s, y=%s' % (trainX.shape, trainy.shape))
print('Test: X=%s, y=%s' % (testX.shape, testy.shape))
# plot first few images
for i in range(9):
    # define subplot
    pyplot.subplot(330 + 1 + i)
    # plot raw pixel data
    pyplot.imshow(trainX[i], cmap=pyplot.get_cmap('gray'))
# show the figure
pyplot.show()
```

Listing 19.1: Example of loading and summarizing the Fashion-MNIST dataset.

Running the example loads the Fashion-MNIST train and test dataset and prints their shape. We can see that there are 60,000 examples in the training dataset and 10,000 in the test dataset and that images are indeed square with 28×28 pixels.

```
Train: X=(60000, 28, 28), y=(60000,)
Test: X=(10000, 28, 28), y=(10000,)
```

Listing 19.2: Example output from loading and summarizing the Fashion-MNIST dataset.

A plot of the first nine images in the dataset is also created showing that indeed the images are grayscale photographs of items of clothing.

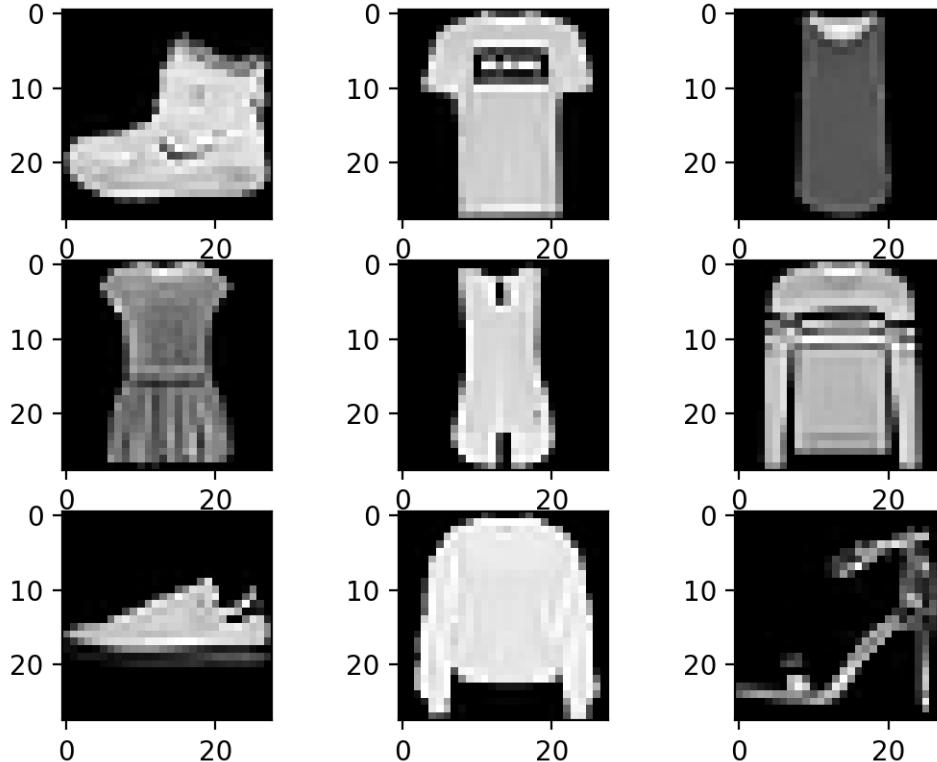


Figure 19.1: Plot of a Subset of Images From the Fashion-MNIST Dataset.

19.3 Model Evaluation Methodology

The Fashion-MNIST dataset was developed as a response to the wide use of the MNIST dataset, that has been effectively *solved* given the use of modern convolutional neural networks. Fashion-MNIST was proposed to be a replacement for MNIST, and although it has not been solved, it is possible to routinely achieve error rates of 10% or less. Like MNIST, it can be a useful starting point for developing and practicing a methodology for solving image classification using convolutional neural networks. Instead of reviewing the literature on well-performing models on the dataset, we can develop a new model from scratch.

The dataset already has a well-defined train and test dataset that we can use. In order to estimate the performance of a model for a given training run, we can further split the training set

into a train and validation dataset. Performance on the train and validation dataset over each run can then be plotted to provide learning curves and insight into how well a model is learning the problem. The Keras API supports this by specifying the `validation_data` argument to the `model.fit()` function when training the model, that will, in turn, return an object that describes model performance for the chosen loss and metrics on each training epoch.

```
...
# record model performance on a validation dataset during training
history = model.fit(..., validation_data=(valX, valY))
```

Listing 19.3: Example of fitting a model with a validation dataset.

In order to estimate the performance of a model on the problem in general, we can use k -fold cross-validation, perhaps 5-fold cross-validation. This will give some account of the model's variance with both respect to differences in the training and test datasets and the stochastic nature of the learning algorithm. The performance of a model can be taken as the mean performance across k -folds, given with the standard deviation, that could be used to estimate a confidence interval if desired. We can use the `KFold` class from the scikit-learn API to implement the k -fold cross-validation evaluation of a given neural network model. There are many ways to achieve this, although we can choose a flexible approach where the `KFold` is only used to specify the row indexes used for each split.

```
...
# example of k-fold cv for a neural net
data = ...
# prepare cross validation
kfold = KFold(5, shuffle=True, random_state=1)
# enumerate splits
for train_ix, test_ix in kfold.split(data):
    model = ...
    ...
```

Listing 19.4: Example of evaluating a model with k -fold cross-validation.

We will hold back the actual test dataset and use it as an evaluation of our final model.

19.4 How to Develop a Baseline Model

The first step is to develop a baseline model. This is critical as it both involves developing the infrastructure for the test harness so that any model we design can be evaluated on the dataset, and it establishes a baseline in model performance on the problem, by which all improvements can be compared. The design of the test harness is modular, and we can develop a separate function for each piece. This allows a given aspect of the test harness to be modified or inter-changed, if we desire, separately from the rest. We can develop this test harness with five key elements. They are the loading of the dataset, the preparation of the dataset, the definition of the model, the evaluation of the model, and the presentation of results.

19.4.1 Load Dataset

We know some things about the dataset. For example, we know that the images are all pre-segmented (e.g. each image contains a single item of clothing), that the images all have the

same square size of 28×28 pixels, and that the images are grayscale. Therefore, we can load the images and reshape the data arrays to have a single color channel.

```
...
# load dataset
(trainX, trainY), (testX, testY) = fashion_mnist.load_data()
# reshape dataset to have a single channel
trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
testX = testX.reshape((testX.shape[0], 28, 28, 1))
```

Listing 19.5: Example of adding a channels dimension to the loaded dataset.

We also know that there are 10 classes and that classes are represented as unique integers. We can, therefore, use a one hot encoding for the class element of each sample, transforming the integer into a 10 element binary vector with a 1 for the index of the class value. We can achieve this with the `to_categorical()` utility function.

```
...
# one hot encode target values
trainY = to_categorical(trainY)
testY = to_categorical(testY)
```

Listing 19.6: Example of one hot encoding the target variable.

The `load_dataset()` function implements these behaviors and can be used to load the dataset.

```
# load train and test dataset
def load_dataset():
    # load dataset
    (trainX, trainY), (testX, testY) = fashion_mnist.load_data()
    # reshape dataset to have a single channel
    trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
    testX = testX.reshape((testX.shape[0], 28, 28, 1))
    # one hot encode target values
    trainY = to_categorical(trainY)
    testY = to_categorical(testY)
    return trainX, trainY, testX, testY
```

Listing 19.7: Example of a function for loading the dataset.

19.4.2 Prepare Pixel Data

We know that the pixel values for each image in the dataset are unsigned integers in the range between black and white, or 0 and 255. We do not know the best way to scale the pixel values for modeling, but we know that some scaling will be required. A good starting point is to normalize the pixel values of grayscale images, e.g. rescale them to the range [0,1]. This involves first converting the data type from unsigned integers to floats, then dividing the pixel values by the maximum value.

```
...
# convert from integers to floats
train_norm = train.astype('float32')
test_norm = test.astype('float32')
# normalize to range 0-1
```

```
train_norm = train_norm / 255.0
test_norm = test_norm / 255.0
```

Listing 19.8: Example of normalizing the pixel values.

The `prep_pixels()` function below implements these behaviors and is provided with the pixel values for both the train and test datasets that will need to be scaled.

```
# scale pixels
def prep_pixels(train, test):
    # convert from integers to floats
    train_norm = train.astype('float32')
    test_norm = test.astype('float32')
    # normalize to range 0-1
    train_norm = train_norm / 255.0
    test_norm = test_norm / 255.0
    # return normalized images
    return train_norm, test_norm
```

Listing 19.9: Example of a function for scaling the pixel values.

This function must be called to prepare the pixel values prior to any modeling.

19.4.3 Define Model

Next, we need to define a baseline convolutional neural network model for the problem. The model has two main aspects: the feature extraction front end comprised of convolutional and pooling layers, and the classifier backend that will make a prediction. For the convolutional front end, we can start with a single convolutional layer with a small filter size (3,3) and a modest number of filters (32) followed by a max pooling layer. The filter maps can then be flattened to provide features to the classifier.

Given that the problem is a multiclass classification, we know that we will require an output layer with 10 nodes in order to predict the probability distribution of an image belonging to each of the 10 classes. This will also require the use of a softmax activation function. Between the feature extractor and the output layer, we can add a dense layer to interpret the features, in this case with 100 nodes. All layers will use the ReLU activation function and the He weight initialization scheme, both best practices.

We will use a conservative configuration for the stochastic gradient descent optimizer with a learning rate of 0.01 and a momentum of 0.9. The categorical cross-entropy loss function will be optimized, suitable for multiclass classification, and we will monitor the classification accuracy metric, which is appropriate given we have the same number of examples in each of the 10 classes. The `define_model()` function below will define and return this model.

```
# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    input_shape=(28, 28, 1)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(10, activation='softmax'))
    # compile model
```

```

opt = SGD(lr=0.01, momentum=0.9)
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
return model

```

Listing 19.10: Example of a function for defining the model.

19.4.4 Evaluate Model

After the model is defined, we need to evaluate it. The model will be evaluated using 5-fold cross-validation. The value of `k=5` was chosen to provide a baseline for both repeated evaluation and to not be too large as to require a long running time. Each test set will be 20% of the training dataset, or about 12,000 examples, close to the size of the actual test set for this problem. The training dataset is shuffled prior to being split and the sample shuffling is performed each time so that any model we evaluate will have the same train and test datasets in each fold, providing an apples-to-apples comparison.

We will train the baseline model for a modest 10 training epochs with a default batch size of 32 examples. The test set for each fold will be used to evaluate the model both during each epoch of the training run, so we can later create learning curves, and at the end of the run, so we can estimate the performance of the model. As such, we will keep track of the resulting history from each run, as well as the classification accuracy of the fold. The `evaluate_model()` function below implements these behaviors, taking the training dataset as arguments and returning a list of accuracy scores and training histories that can be later summarized.

```

# evaluate a model using k-fold cross-validation
def evaluate_model(dataX, dataY, n_folds=5):
    scores, histories = list(), list()
    # prepare cross validation
    kfold = KFold(n_folds, shuffle=True, random_state=1)
    # enumerate splits
    for train_ix, test_ix in kfold.split(dataX):
        # define model
        model = define_model()
        # select rows for train and test
        trainX, trainY, testX, testY = dataX[train_ix], dataY[train_ix], dataX[test_ix],
            dataY[test_ix]
        # fit model
        history = model.fit(trainX, trainY, epochs=10, batch_size=32, validation_data=(testX,
            testY), verbose=0)
        # evaluate model
        _, acc = model.evaluate(testX, testY, verbose=0)
        print('> %.3f' % (acc * 100.0))
        # append scores
        scores.append(acc)
        histories.append(history)
    return scores, histories

```

Listing 19.11: Example of a function for evaluating the performance of a model.

19.4.5 Present Results

Once the model has been evaluated, we can present the results. There are two key aspects to present: the diagnostics of the learning behavior of the model during training and the estimation

of the model performance. These can be implemented using separate functions. First, the diagnostics involve creating a line plot showing model performance on the train and test set during each fold of the k -fold cross-validation. These plots are valuable for getting an idea of whether a model is overfitting, underfitting, or has a good fit for the dataset. We will create a single figure with two subplots, one for loss and one for accuracy. Blue lines will indicate model performance on the training dataset and orange lines will indicate performance on the hold out test dataset. The `summarize_diagnostics()` function below creates and shows this plot given the collected training histories.

```
# plot diagnostic learning curves
def summarize_diagnostics(histories):
    for i in range(len(histories)):
        # plot loss
        pyplot.subplot(211)
        pyplot.title('Cross Entropy Loss')
        pyplot.plot(histories[i].history['loss'], color='blue', label='train')
        pyplot.plot(histories[i].history['val_loss'], color='orange', label='test')
        # plot accuracy
        pyplot.subplot(212)
        pyplot.title('Classification Accuracy')
        pyplot.plot(histories[i].history['accuracy'], color='blue', label='train')
        pyplot.plot(histories[i].history['val_accuracy'], color='orange', label='test')
    pyplot.show()
```

Listing 19.12: Example of a function for plotting learning curves.

Next, the classification accuracy scores collected during each fold can be summarized by calculating the mean and standard deviation. This provides an estimate of the average expected performance of the model trained on this dataset, with an estimate of the average variance in the mean. We will also summarize the distribution of scores by creating and showing a box and whisker plot. The `summarize_performance()` function below implements this for a given list of scores collected during model evaluation.

```
# summarize model performance
def summarize_performance(scores):
    # print summary
    print('Accuracy: mean=% .3f std=% .3f, n=%d' % (mean(scores)*100, std(scores)*100,
                                                       len(scores)))
    # box and whisker plots of results
    pyplot.boxplot(scores)
    pyplot.show()
```

Listing 19.13: Example of a function for summarizing model performance.

19.4.6 Complete Example

We need a function that will drive the test harness. This involves calling all of the defined functions.

```
# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # prepare pixel data
```

```

trainX, testX = prep_pixels(trainX, testX)
# evaluate model
scores, histories = evaluate_model(model, trainX, trainY)
# learning curves
summarize_diagnostics(histories)
# summarize estimated performance
summarize_performance(scores)

```

Listing 19.14: Example of a function for driving the test harness.

We now have everything we need; the complete code example for a baseline convolutional neural network model on the MNIST dataset is listed below.

```

# baseline cnn model for fashion mnist
from numpy import mean
from numpy import std
from matplotlib import pyplot
from sklearn.model_selection import KFold
from keras.datasets import fashion_mnist
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD

# load train and test dataset
def load_dataset():
    # load dataset
    (trainX, trainY), (testX, testY) = fashion_mnist.load_data()
    # reshape dataset to have a single channel
    trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
    testX = testX.reshape((testX.shape[0], 28, 28, 1))
    # one hot encode target values
    trainY = to_categorical(trainY)
    testY = to_categorical(testY)
    return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
    # convert from integers to floats
    train_norm = train.astype('float32')
    test_norm = test.astype('float32')
    # normalize to range 0-1
    train_norm = train_norm / 255.0
    test_norm = test_norm / 255.0
    # return normalized images
    return train_norm, test_norm

# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
        input_shape=(28, 28, 1)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())

```

```
model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(10, activation='softmax'))
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
return model

# evaluate a model using k-fold cross-validation
def evaluate_model(dataX, dataY, n_folds=5):
    scores, histories = list(), list()
    # prepare cross validation
    kfold = KFold(n_folds, shuffle=True, random_state=1)
    # enumerate splits
    for train_ix, test_ix in kfold.split(dataX):
        # define model
        model = define_model()
        # select rows for train and test
        trainX, trainY, testX, testY = dataX[train_ix], dataY[train_ix], dataX[test_ix],
            dataY[test_ix]
        # fit model
        history = model.fit(trainX, trainY, epochs=10, batch_size=32, validation_data=(testX,
            testY), verbose=0)
        # evaluate model
        _, acc = model.evaluate(testX, testY, verbose=0)
        print('> %.3f' % (acc * 100.0))
        # append scores
        scores.append(acc)
        histories.append(history)
    return scores, histories

# plot diagnostic learning curves
def summarize_diagnostics(histories):
    for i in range(len(histories)):
        # plot loss
        pyplot.subplot(211)
        pyplot.title('Cross Entropy Loss')
        pyplot.plot(histories[i].history['loss'], color='blue', label='train')
        pyplot.plot(histories[i].history['val_loss'], color='orange', label='test')
        # plot accuracy
        pyplot.subplot(212)
        pyplot.title('Classification Accuracy')
        pyplot.plot(histories[i].history['accuracy'], color='blue', label='train')
        pyplot.plot(histories[i].history['val_accuracy'], color='orange', label='test')
    pyplot.show()

# summarize model performance
def summarize_performance(scores):
    # print summary
    print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores)*100, std(scores)*100,
        len(scores)))
    # box and whisker plots of results
    pyplot.boxplot(scores)
    pyplot.show()

# run the test harness for evaluating a model
def run_test_harness():
```

```
# load dataset
trainX, trainY, testX, testY = load_dataset()
# prepare pixel data
trainX, testX = prep_pixels(trainX, testX)
# evaluate model
scores, histories = evaluate_model(trainX, trainY)
# learning curves
summarize_diagnostics(histories)
# summarize estimated performance
summarize_performance(scores)

# entry point, run the test harness
run_test_harness()
```

Listing 19.15: Example of defining and evaluating a baseline model on the dataset.

Running the example prints the classification accuracy for each fold of the cross-validation process. This is helpful to get an idea that the model evaluation is progressing. We can see that for each fold, the baseline model achieved an error rate below 10%, and in two cases 98% and 99% accuracy. These are good results.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
> 91.200
> 91.217
> 90.958
> 91.242
> 91.317
```

Listing 19.16: Example output from during the evaluation of each baseline model.

Next, a diagnostic plot is shown, giving insight into the learning behavior of the model across each fold. In this case, we can see that the model generally achieves a good fit, with train and test learning curves converging. There may be some signs of slight overfitting.

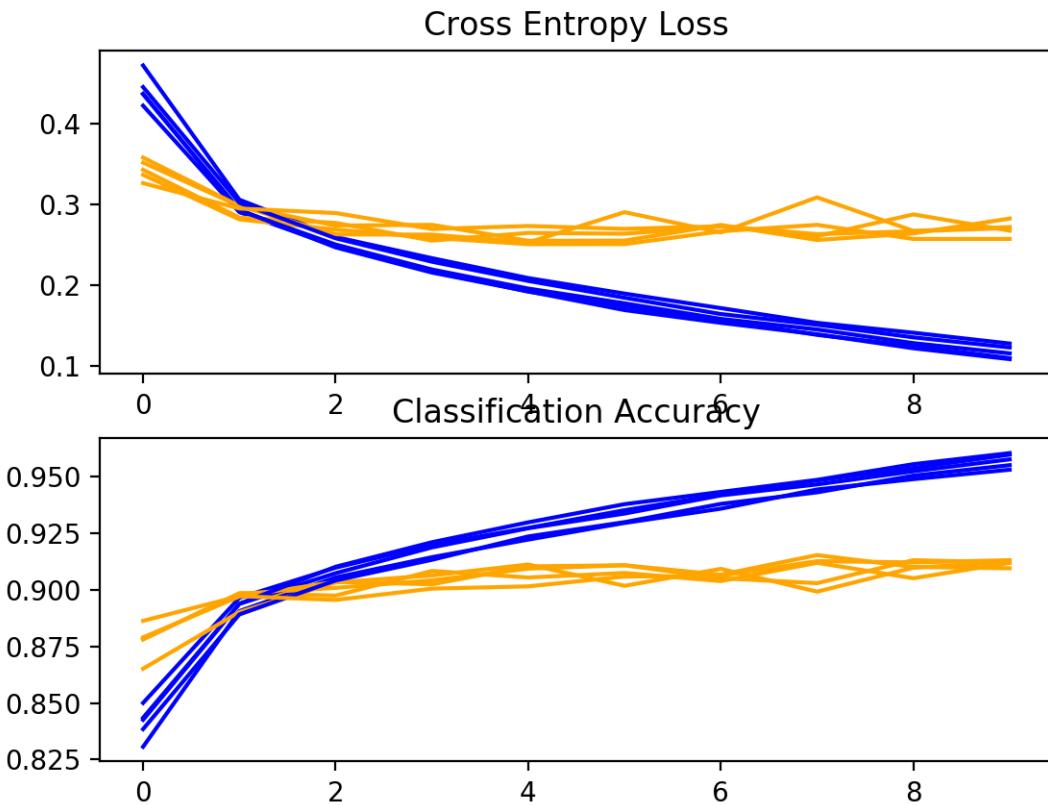


Figure 19.2: Loss and Accuracy Learning Curves for the Baseline Model on the Fashion-MNIST Dataset During k -Fold Cross-Validation.

Next, the summary of the model performance is calculated. We can see in this case, the model has an estimated skill of about 91%, which is impressive.

```
Accuracy: mean=91.187 std=0.121, n=5
```

Listing 19.17: Example output from the final evaluation of the baseline model.

Finally, a box and whisker plot is created to summarize the distribution of accuracy scores.

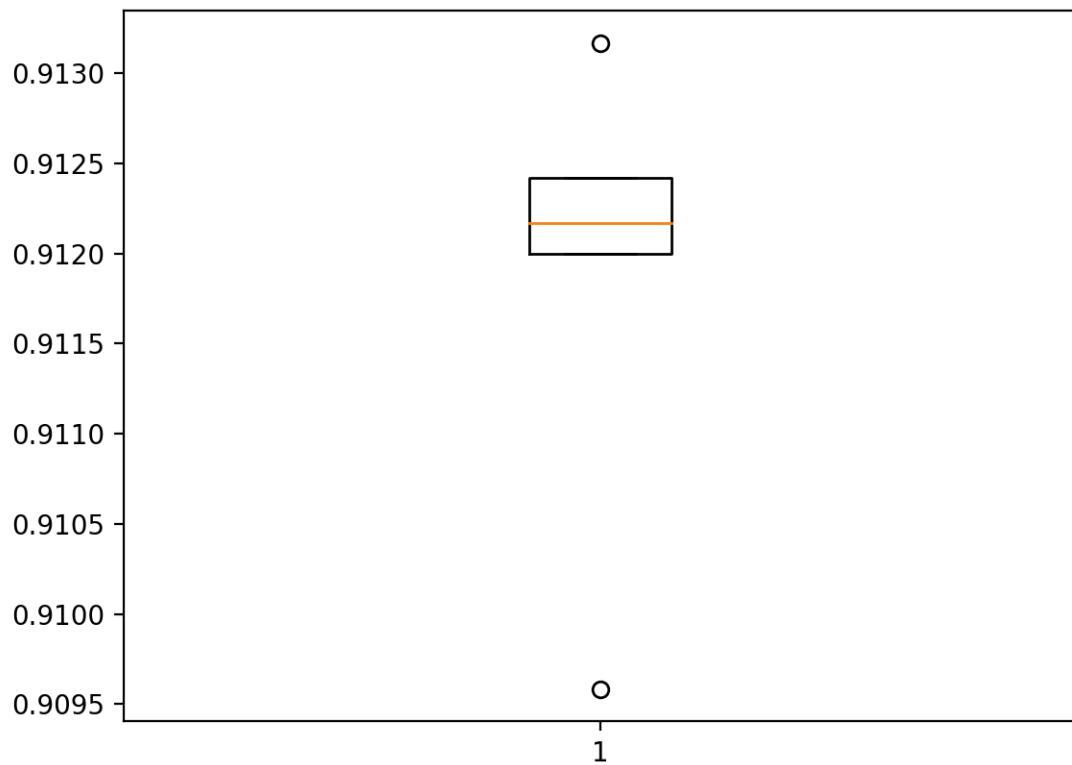


Figure 19.3: Box and Whisker Plot of Accuracy Scores for the Baseline Model on the Fashion-MNIST Dataset Evaluated Using k -Fold Cross-Validation.

As we would expect, the distribution spread across the low-nineties. We now have a robust test harness and a well-performing baseline model.

19.5 How to Develop an Improved Model

There are many ways that we might explore improvements to the baseline model. We will look at areas that often result in an improvement, so-called low-hanging fruit. The first will be a change to the convolutional operation to add padding and the second will build on this to increase the number of filters.

19.5.1 Padding Convolutions

Adding padding to the convolutional operation can often result in better model performance, as more of the input image or feature maps are given an opportunity to participate or contribute to the output. By default, the convolutional operation uses ‘`valid`’ padding, which means that convolutions are only applied where possible. This can be changed to ‘`same`’ padding so that zero values are added around the input such that the output has the same size as the input.

...

```
model.add(Conv2D(32, (3, 3), padding='same', activation='relu',
    kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
```

Listing 19.18: Example of padding convolutional layers.

The full code listing including the change to padding is provided below for completeness.

```
# model with padded convolutions for the fashion mnist dataset
from numpy import mean
from numpy import std
from matplotlib import pyplot
from sklearn.model_selection import KFold
from keras.datasets import fashion_mnist
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD

# load train and test dataset
def load_dataset():
    # load dataset
    (trainX, trainY), (testX, testY) = fashion_mnist.load_data()
    # reshape dataset to have a single channel
    trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
    testX = testX.reshape((testX.shape[0], 28, 28, 1))
    # one hot encode target values
    trainY = to_categorical(trainY)
    testY = to_categorical(testY)
    return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
    # convert from integers to floats
    train_norm = train.astype('float32')
    test_norm = test.astype('float32')
    # normalize to range 0-1
    train_norm = train_norm / 255.0
    test_norm = test_norm / 255.0
    # return normalized images
    return train_norm, test_norm

# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), padding='same', activation='relu',
        kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(10, activation='softmax'))
    # compile model
    opt = SGD(lr=0.01, momentum=0.9)
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

```
# evaluate a model using k-fold cross-validation
def evaluate_model(dataX, dataY, n_folds=5):
    scores, histories = list(), list()
    # prepare cross validation
    kfold = KFold(n_folds, shuffle=True, random_state=1)
    # enumerate splits
    for train_ix, test_ix in kfold.split(dataX):
        # define model
        model = define_model()
        # select rows for train and test
        trainX, trainY, testX, testY = dataX[train_ix], dataY[train_ix], dataX[test_ix],
            dataY[test_ix]
        # fit model
        history = model.fit(trainX, trainY, epochs=10, batch_size=32, validation_data=(testX,
            testY), verbose=0)
        # evaluate model
        _, acc = model.evaluate(testX, testY, verbose=0)
        print('> %.3f' % (acc * 100.0))
        # append scores
        scores.append(acc)
        histories.append(history)
    return scores, histories

# plot diagnostic learning curves
def summarize_diagnostics(histories):
    for i in range(len(histories)):
        # plot loss
        pyplot.subplot(211)
        pyplot.title('Cross Entropy Loss')
        pyplot.plot(histories[i].history['loss'], color='blue', label='train')
        pyplot.plot(histories[i].history['val_loss'], color='orange', label='test')
        # plot accuracy
        pyplot.subplot(212)
        pyplot.title('Classification Accuracy')
        pyplot.plot(histories[i].history['accuracy'], color='blue', label='train')
        pyplot.plot(histories[i].history['val_accuracy'], color='orange', label='test')
    pyplot.show()

# summarize model performance
def summarize_performance(scores):
    # print summary
    print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores)*100, std(scores)*100,
        len(scores)))
    # box and whisker plots of results
    pyplot.boxplot(scores)
    pyplot.show()

# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # prepare pixel data
    trainX, testX = prep_pixels(trainX, testX)
    # evaluate model
    scores, histories = evaluate_model(trainX, trainY)
```

```
# learning curves
summarize_diagnostics(histories)
# summarize estimated performance
summarize_performance(scores)

# entry point, run the test harness
run_test_harness()
```

Listing 19.19: Example of evaluating the baseline model with padded convolutional layers.

Running the example again reports model performance for each fold of the cross-validation process.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see perhaps a small improvement in model performance as compared to the baseline across the cross-validation folds.

```
> 90.875
> 91.442
> 91.242
> 91.275
> 91.450
```

Listing 19.20: Example output from during the evaluation of each model.

A plot of the learning curves is created. As with the baseline model, we may see some slight overfitting. This could be addressed perhaps with use of regularization or the training for fewer epochs.

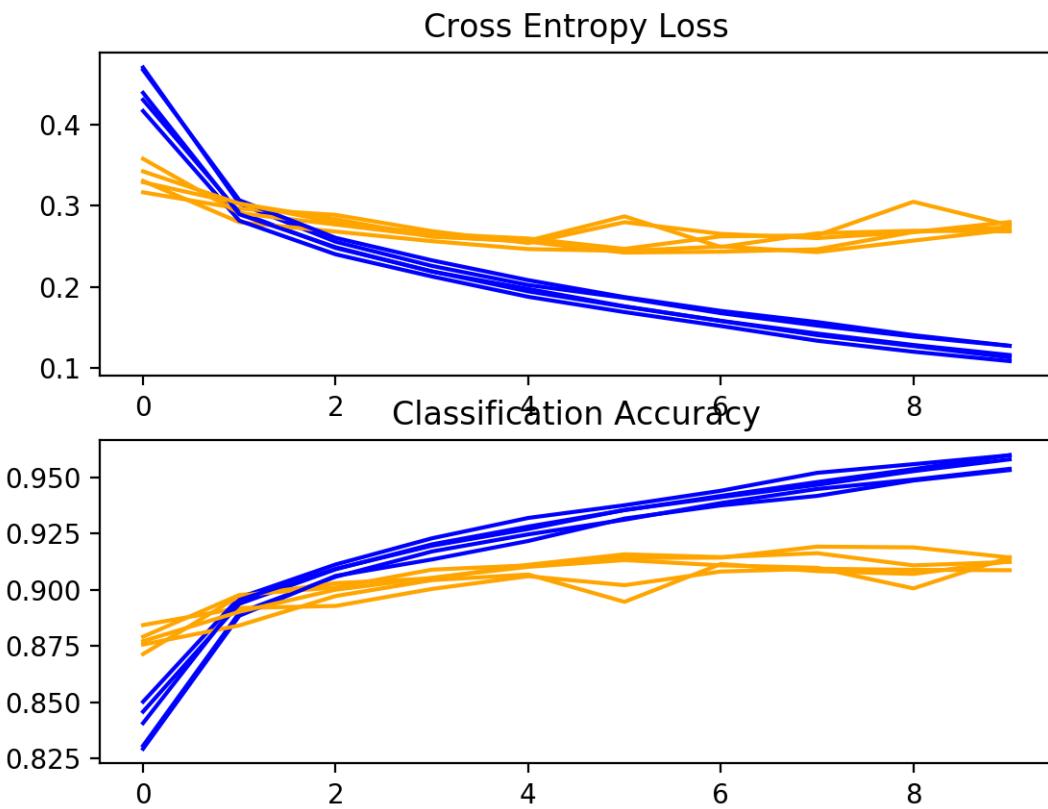


Figure 19.4: Loss and Accuracy Learning Curves for the Same Padding on the Fashion-MNIST Dataset During k -Fold Cross-Validation.

Next, the estimated performance of the model is presented, showing performance with a very slight increase in the mean accuracy of the model, 91.257% as compared to 91.187% with the baseline model. This may or may not be a real effect as it is within the bounds of the standard deviation. Perhaps more repeats of the experiment could tease out this fact.

```
Accuracy: mean=91.257 std=0.209, n=5
```

Listing 19.21: Example output from the final evaluation of the model.

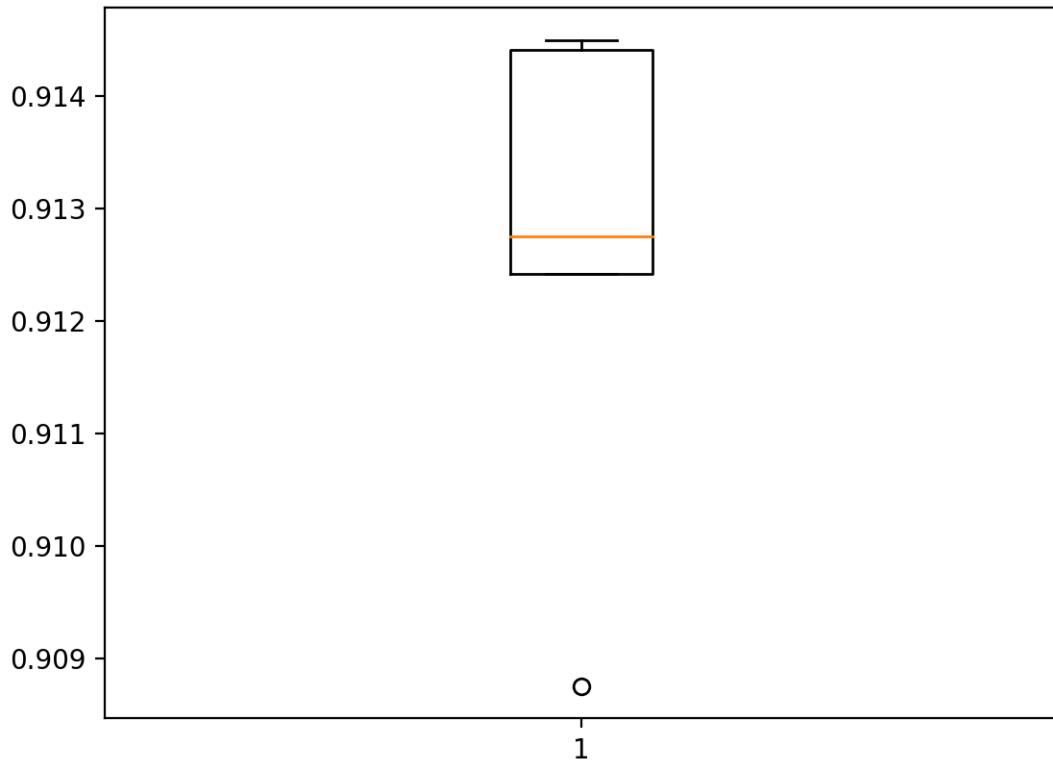


Figure 19.5: Box and Whisker Plot of Accuracy Scores for Same Padding on the Fashion-MNIST Dataset Evaluated Using k -Fold Cross-Validation.

19.5.2 Increasing Filters

An increase in the number of filters used in the convolutional layer can often improve performance, as it can provide more opportunity for extracting simple features from the input images. This is especially relevant when very small filters are used, such as 3×3 pixels. In this change, we can increase the number of filters in the convolutional layer from 32 to double that at 64. We will also build upon the possible improvement offered by using ‘same’ padding.

```
...
model.add(Conv2D(64, (3, 3), padding='same', activation='relu',
    kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
```

Listing 19.22: Example of increasing the number of filters.

The full code listing including the change to padding is provided below for completeness.

```
# model with double the filters for the fashion mnist dataset
from numpy import mean
from numpy import std
from matplotlib import pyplot
from sklearn.model_selection import KFold
from keras.datasets import fashion_mnist
```

```
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD

# load train and test dataset
def load_dataset():
    # load dataset
    (trainX, trainY), (testX, testY) = fashion_mnist.load_data()
    # reshape dataset to have a single channel
    trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
    testX = testX.reshape((testX.shape[0], 28, 28, 1))
    # one hot encode target values
    trainY = to_categorical(trainY)
    testY = to_categorical(testY)
    return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
    # convert from integers to floats
    train_norm = train.astype('float32')
    test_norm = test.astype('float32')
    # normalize to range 0-1
    train_norm = train_norm / 255.0
    test_norm = test_norm / 255.0
    # return normalized images
    return train_norm, test_norm

# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(64, (3, 3), padding='same', activation='relu',
        kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(10, activation='softmax'))
    # compile model
    opt = SGD(lr=0.01, momentum=0.9)
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    return model

# evaluate a model using k-fold cross-validation
def evaluate_model(dataX, dataY, n_folds=5):
    scores, histories = list(), list()
    # prepare cross validation
    kfold = KFold(n_folds, shuffle=True, random_state=1)
    # enumerate splits
    for train_ix, test_ix in kfold.split(dataX):
        # define model
        model = define_model()
        # select rows for train and test
        trainX, trainY, testX, testY = dataX[train_ix], dataY[train_ix],
```

```

    dataY[test_ix]
# fit model
history = model.fit(trainX, trainY, epochs=10, batch_size=32, validation_data=(testX,
    testY), verbose=0)
# evaluate model
_, acc = model.evaluate(testX, testY, verbose=0)
print('> %.3f' % (acc * 100.0))
# append scores
scores.append(acc)
histories.append(history)
return scores, histories

# plot diagnostic learning curves
def summarize_diagnostics(histories):
    for i in range(len(histories)):
        # plot loss
        pyplot.subplot(211)
        pyplot.title('Cross Entropy Loss')
        pyplot.plot(histories[i].history['loss'], color='blue', label='train')
        pyplot.plot(histories[i].history['val_loss'], color='orange', label='test')
        # plot accuracy
        pyplot.subplot(212)
        pyplot.title('Classification Accuracy')
        pyplot.plot(histories[i].history['accuracy'], color='blue', label='train')
        pyplot.plot(histories[i].history['val_accuracy'], color='orange', label='test')
    pyplot.show()

# summarize model performance
def summarize_performance(scores):
    # print summary
    print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores)*100, std(scores)*100,
        len(scores)))
    # box and whisker plots of results
    pyplot.boxplot(scores)
    pyplot.show()

# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # prepare pixel data
    trainX, testX = prep_pixels(trainX, testX)
    # evaluate model
    scores, histories = evaluate_model(trainX, trainY)
    # learning curves
    summarize_diagnostics(histories)
    # summarize estimated performance
    summarize_performance(scores)

# entry point, run the test harness
run_test_harness()

```

Listing 19.23: Example of evaluating the baseline model with padded convolutional layers and more filters.

Running the example reports model performance for each fold of the cross-validation process.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the per-fold scores may suggest some further improvement over the baseline and using same padding alone.

```
> 90.917
> 90.908
> 90.175
> 91.158
> 91.408
```

Listing 19.24: Example output from during the evaluation of each model.

A plot of the learning curves is created, in this case showing that the models still have a reasonable fit on the problem, with a small sign of some of the runs overfitting.

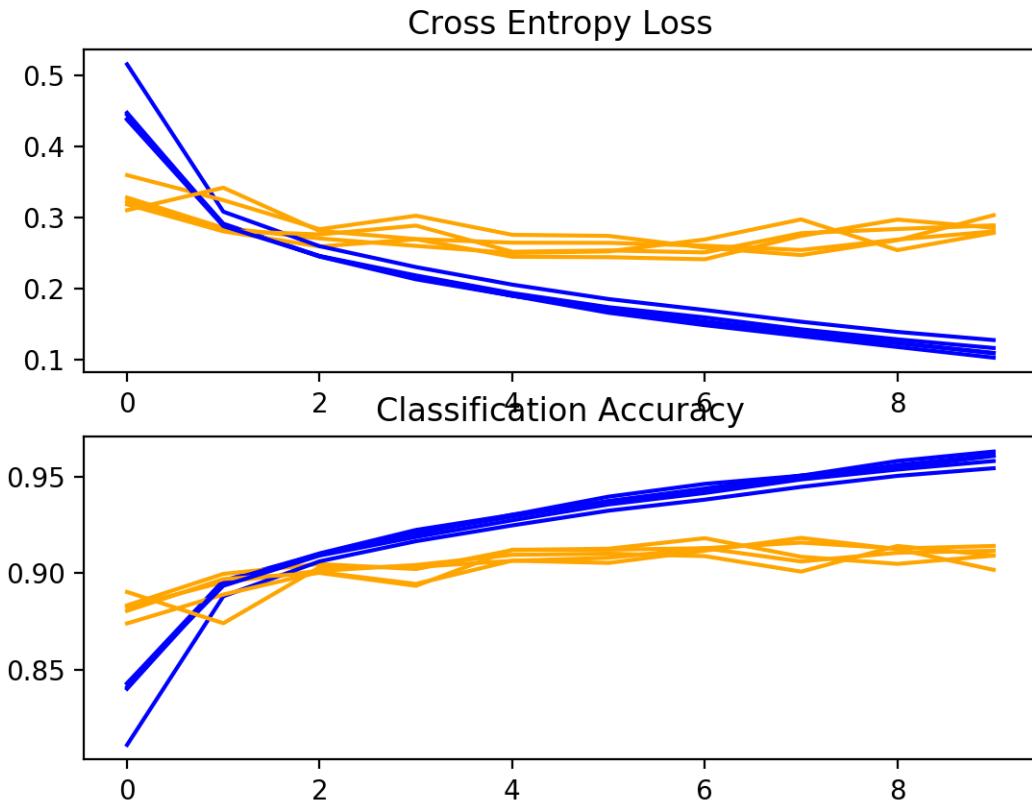


Figure 19.6: Loss and Accuracy Learning Curves for the More Filters and Padding on the Fashion-MNIST Dataset During k -Fold Cross-Validation.

Next, the estimated performance of the model is presented, showing a possible decrease in performance as compared to the baseline with padding from 90.913% to 91.257%. Again, the change is still within the bounds of the standard deviation, and it is not clear whether the effect is real.

```
Accuracy: mean=90.913 std=0.412, n=5
```

Listing 19.25: Example output from the final evaluation of the model.

19.6 How to Finalize the Model and Make Predictions

The process of model improvement may continue for as long as we have ideas and the time and resources to test them out. At some point, a final model configuration must be chosen and adopted. In this case, we will keep things simple and use the baseline model as the final model. First, we will finalize our model, by fitting a model on the entire training dataset and saving the model to file for later use. We will then load the model and evaluate its performance on the hold out test dataset, to get an idea of how well the chosen model actually performs in practice. Finally, we will use the saved model to make a prediction on a single image.

19.6.1 Save Final Model

A final model is typically fit on all available data, such as the combination of all train and test dataset. In this tutorial, we are intentionally holding back a test dataset so that we can estimate the performance of the final model, which can be a good idea in practice. As such, we will fit our model on the training dataset only.

```
...
# fit model
model.fit(trainX, trainY, epochs=10, batch_size=32, verbose=0)
```

Listing 19.26: Example of fitting the final model.

Once fit, we can save the final model to an h5 file by calling the `save()` function on the model and passing in the chosen filename.

```
...
# save model
model.save('final_model.h5')
```

Listing 19.27: Example of saving the final model.

Note: saving and loading a Keras model requires that the h5py library is installed on your workstation. The complete example of fitting the final model on the training dataset and saving it to file is listed below.

```
# save the final model to file
from keras.datasets import fashion_mnist
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD

# load train and test dataset
def load_dataset():
```

```

# load dataset
(trainX, trainY), (testX, testY) = fashion_mnist.load_data()
# reshape dataset to have a single channel
trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
testX = testX.reshape((testX.shape[0], 28, 28, 1))
# one hot encode target values
trainY = to_categorical(trainY)
testY = to_categorical(testY)
return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
    # convert from integers to floats
    train_norm = train.astype('float32')
    test_norm = test.astype('float32')
    # normalize to range 0-1
    train_norm = train_norm / 255.0
    test_norm = test_norm / 255.0
    # return normalized images
    return train_norm, test_norm

# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
        input_shape=(28, 28, 1)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(10, activation='softmax'))
    # compile model
    opt = SGD(lr=0.01, momentum=0.9)
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    return model

# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # prepare pixel data
    trainX, testX = prep_pixels(trainX, testX)
    # define model
    model = define_model()
    # fit model
    model.fit(trainX, trainY, epochs=10, batch_size=32, verbose=0)
    # save model
    model.save('final_model.h5')

    # entry point, run the test harness
run_test_harness()

```

Listing 19.28: Example of fitting and saving the final model.

After running this example, you will now have a 4.2-megabyte file with the name `final_model.h5` in your current working directory.

19.6.2 Evaluate Final Model

We can now load the final model and evaluate it on the hold out test dataset. This is something we might do if we were interested in presenting the performance of the chosen model to project stakeholders. The model can be loaded via the `load_model()` function. The complete example of loading the saved model and evaluating it on the test dataset is listed below.

```
# evaluate the deep model on the test dataset
from keras.datasets import fashion_mnist
from keras.models import load_model
from keras.utils import to_categorical

# load train and test dataset
def load_dataset():
    # load dataset
    (trainX, trainY), (testX, testY) = fashion_mnist.load_data()
    # reshape dataset to have a single channel
    trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
    testX = testX.reshape((testX.shape[0], 28, 28, 1))
    # one hot encode target values
    trainY = to_categorical(trainY)
    testY = to_categorical(testY)
    return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
    # convert from integers to floats
    train_norm = train.astype('float32')
    test_norm = test.astype('float32')
    # normalize to range 0-1
    train_norm = train_norm / 255.0
    test_norm = test_norm / 255.0
    # return normalized images
    return train_norm, test_norm

# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # prepare pixel data
    trainX, testX = prep_pixels(trainX, testX)
    # load model
    model = load_model('final_model.h5')
    # evaluate model on test dataset
    _, acc = model.evaluate(testX, testY, verbose=0)
    print('> %.3f' % (acc * 100.0))

# entry point, run the test harness
run_test_harness()
```

Listing 19.29: Example of loading and evaluating the final model.

Running the example loads the saved model and evaluates the model on the hold out test dataset. The classification accuracy for the model on the test dataset is calculated and printed.

Note: Your specific results may vary given the stochastic nature of the learning algorithm.

Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved an accuracy of 90.990%, or just less than 10% classification error, which is not bad.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
> 90.990
```

Listing 19.30: Example output from loading and evaluating the final model.

19.6.3 Make Prediction

We can use our saved model to make a prediction on new images. The model assumes that new images are grayscale, they have been segmented so that one image contains one centered piece of clothing on a black background, and that the size of the image is square with the size 28×28 pixels. Below is an image extracted from the MNIST test dataset.

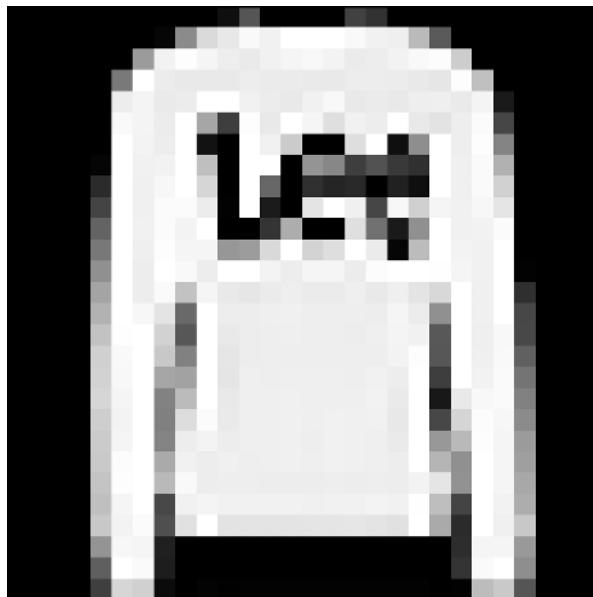


Figure 19.7: Sample Clothing (Pullover).

You can save it in your current working directory with the filename `sample_image.png`.

- Download Image (`sample_image.png`).¹

We will pretend this is an entirely new and unseen image, prepared in the required way, and see how we might use our saved model to predict the integer that the image represents. For this example, we expect class 2 for Pullover (also called a jumper). First, we can load the image, force it to be grayscale format, and force the size to be 28×28 pixels. The loaded image

¹https://machinelearningmastery.com/wp-content/uploads/2019/05/sample_image.png

can then be resized to have a single channel and represent a single sample in a dataset. The `load_image()` function implements this and will return the loaded image ready for classification. Importantly, the pixel values are prepared in the same way as the pixel values were prepared for the training dataset when fitting the final model, in this case, normalized.

```
# load and prepare the image
def load_image(filename):
    # load the image
    img = load_img(filename, grayscale=True, target_size=(28, 28))
    # convert to array
    img = img_to_array(img)
    # reshape into a single sample with 1 channel
    img = img.reshape(1, 28, 28, 1)
    # prepare pixel data
    img = img.astype('float32')
    img = img / 255.0
    return img
```

Listing 19.31: Example of a function for loading and preparing an image for prediction.

Next, we can load the model as in the previous section and call the `predict_classes()` function to predict the clothing in the image.

```
...
# predict the class
result = model.predict_classes(img)
```

Listing 19.32: Example of making a prediction with a prepared image.

The complete example is listed below.

```
# make a prediction for a new image.
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.models import load_model

# load and prepare the image
def load_image(filename):
    # load the image
    img = load_img(filename, grayscale=True, target_size=(28, 28))
    # convert to array
    img = img_to_array(img)
    # reshape into a single sample with 1 channel
    img = img.reshape(1, 28, 28, 1)
    # prepare pixel data
    img = img.astype('float32')
    img = img / 255.0
    return img

# load an image and predict the class
def run_example():
    # load the image
    img = load_image('sample_image.png')
    # load model
    model = load_model('final_model.h5')
    # predict the class
    result = model.predict_classes(img)
```

```
print(result[0])  
  
# entry point, run the example  
run_example()
```

Listing 19.33: Example of loading and making a prediction with the final model.

Running the example first loads and prepares the image, loads the model, and then correctly predicts that the loaded image represents a pullover or class 2.

```
2
```

Listing 19.34: Example output from loading and making a prediction with the final model.

19.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Regularization.** Explore how adding regularization impacts model performance as compared to the baseline model, such as weight decay, early stopping, and dropout.
- **Tune the Learning Rate.** Explore how different learning rates impact the model performance as compared to the baseline model, such as 0.001 and 0.0001.
- **Tune Model Depth.** Explore how adding more layers to the model impacts the model performance as compared to the baseline model, such as another block of convolutional and pooling layers or another dense layer in the classifier part of the model.

If you explore any of these extensions, I'd love to know.

19.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

19.8.1 APIs

- Keras Datasets API.
<https://keras.io/datasets/>
- Keras Datasets Code.
<https://github.com/keras-team/keras/tree/master/keras/datasets>
- `sklearn.model_selection.KFold` API.
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html

19.8.2 Articles

- Fashion-MNIST GitHub Repository.
<https://github.com/zalandoresearch/fashion-mnist>

19.9 Summary

In this tutorial, you discovered how to develop a convolutional neural network for clothing classification from scratch. Specifically, you learned:

- How to develop a test harness to develop a robust evaluation of a model and establish a baseline of performance for a classification task.
- How to explore extensions to a baseline model to improve learning and model capacity.
- How to develop a finalized model, evaluate the performance of the final model, and use it to make predictions on new images.

19.9.1 Next

In the next section, you will discover how to develop a deep convolutional neural network for classifying small photographs of objects.

Chapter 20

How to Classify Small Photos of Objects

The CIFAR-10 small photo classification problem is a standard dataset used in computer vision and deep learning. Although the dataset is effectively solved, it can be used as the basis for learning and practicing how to develop, evaluate, and use convolutional deep learning neural networks for image classification from scratch. This includes how to develop a robust test harness for estimating the performance of the model, how to explore improvements to the model, and how to save the model and later load it to make predictions on new data. In this tutorial, you will discover how to develop a convolutional neural network model from scratch for object photo classification. After completing this tutorial, you will know:

- How to develop a test harness to develop a robust evaluation of a model and establish a baseline of performance for a classification task.
- How to explore extensions to a baseline model to improve learning and model capacity.
- How to develop a finalized model, evaluate the performance of the final model, and use it to make predictions on new images.

Let's get started.

20.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. CIFAR-10 Photo Classification Dataset
2. Model Evaluation Test Harness
3. How to Develop a Baseline Model
4. How to Develop an Improved Model
5. How to Finalize the Model and Make Predictions

20.2 CIFAR-10 Photo Classification Dataset

CIFAR is an acronym that stands for the Canadian Institute For Advanced Research and the CIFAR-10 dataset was developed along with the CIFAR-100 dataset by researchers at the CIFAR institute. The dataset is comprised of 60,000 32×32 pixel color photographs of objects from 10 classes, such as frogs, birds, cats, ships, etc. The class labels and their standard associated integer values are listed below.

- 0: airplane
- 1: automobile
- 2: bird
- 3: cat
- 4: deer
- 5: dog
- 6: frog
- 7: horse
- 8: ship
- 9: truck

These are very small images, much smaller than a typical photograph, as the dataset was intended for computer vision research. CIFAR-10 is a well-understood dataset and widely used for benchmarking computer vision algorithms in the field of machine learning. The problem is *solved*. It is relatively straightforward to achieve 80% classification accuracy. Top performance on the problem is achieved by deep learning convolutional neural networks with a classification accuracy above 90% on the test dataset. The example below loads the CIFAR-10 dataset using the Keras API and creates a plot of the first nine images in the training dataset.

```
# example of loading the cifar10 dataset
from matplotlib import pyplot
from keras.datasets import cifar10
# load dataset
(trainX, trainy), (testX, testy) = cifar10.load_data()
# summarize loaded dataset
print('Train: X=%s, y=%s' % (trainX.shape, trainy.shape))
print('Test: X=%s, y=%s' % (testX.shape, testy.shape))
# plot first few images
for i in range(9):
    # define subplot
    pyplot.subplot(330 + 1 + i)
    # plot raw pixel data
    pyplot.imshow(trainX[i])
# show the figure
pyplot.show()
```

Listing 20.1: Example of loading and summarizing the CIFAR-10 dataset.

Running the example loads the CIFAR-10 train and test dataset and prints their shape. We can see that there are 50,000 examples in the training dataset and 10,000 in the test dataset and that images are indeed square with 32×32 pixels and color, with three channels.

```
Train: X=(50000, 32, 32, 3), y=(50000, 1)
Test: X=(10000, 32, 32, 3), y=(10000, 1)
```

Listing 20.2: Example output from loading and summarizing the CIFAR-10 dataset.

A plot of the first nine images in the dataset is also created. It is clear that the images are indeed very small compared to modern photographs; it can be challenging to see what exactly is represented in some of the images given the extremely low resolution. This low resolution is likely the cause of the limited performance that top-of-the-line algorithms are able to achieve on the dataset.

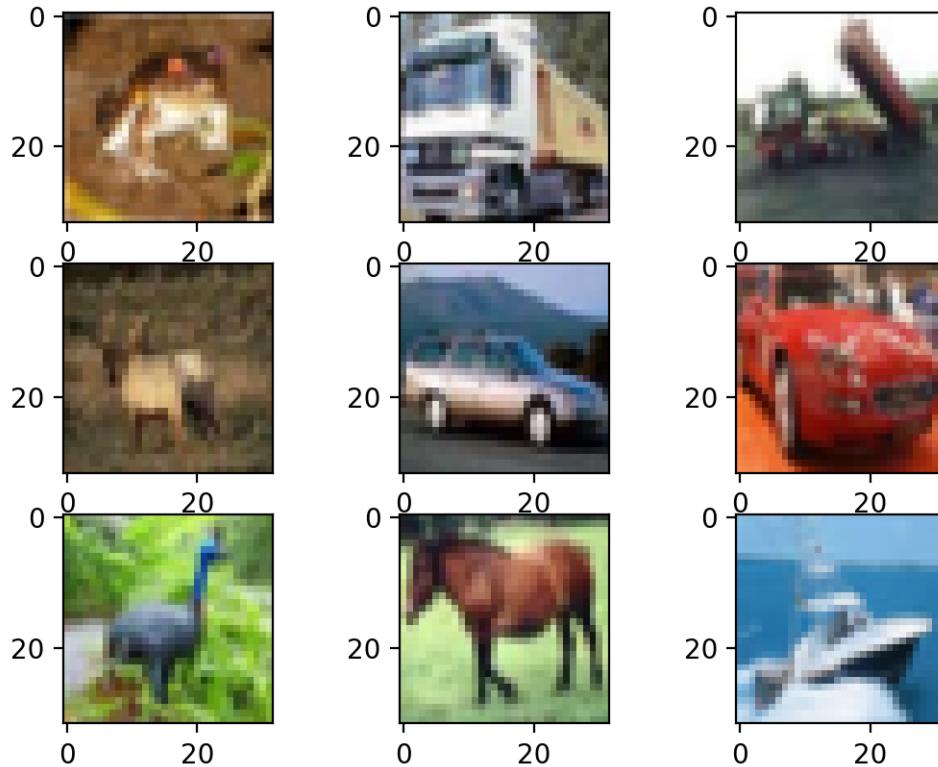


Figure 20.1: Plot of a Subset of Images From the CIFAR-10 Dataset.

20.3 Model Evaluation Test Harness

The CIFAR-10 dataset can be a useful starting point for developing and practicing a methodology for solving image classification problems using convolutional neural networks. Instead of reviewing the literature on well-performing models on the dataset, we can develop a new model from scratch. The dataset already has a well-defined train and test dataset that we will use. An

alternative might be to perform k -fold cross-validation with a $k=5$ or $k=10$. This is desirable if there are sufficient resources. In this case, and in the interest of ensuring the examples in this tutorial execute in a reasonable time, we will not use k -fold cross-validation.

The design of the test harness is modular, and we can develop a separate function for each piece. This allows a given aspect of the test harness to be modified or interchanged, if we desire, separately from the rest. We can develop this test harness with five key elements. They are the loading of the dataset, the preparation of the dataset, the definition of the model, the evaluation of the model, and the presentation of results.

20.3.1 Load Dataset

We know some things about the dataset. For example, we know that the images are all pre-segmented (e.g. each image contains a single object), that the images all have the same square size of 32×32 pixels, and that the images are color. Therefore, we can load the images and use them for modeling almost immediately.

```
...
# load dataset
(trainX, trainY), (testX, testY) = cifar10.load_data()
```

Listing 20.3: Example of loading the CIFAR-10 dataset.

We also know that there are 10 classes and that classes are represented as unique integers. We can, therefore, use a one hot encoding for the class element of each sample, transforming the integer into a 10 element binary vector with a 1 for the index of the class value. We can achieve this with the `to_categorical()` utility function.

```
...
# one hot encode target values
trainY = to_categorical(trainY)
testY = to_categorical(testY)
```

Listing 20.4: Example of one hot encoding the target variables.

The `load_dataset()` function implements these behaviors and can be used to load the dataset.

```
# load train and test dataset
def load_dataset():
    # load dataset
    (trainX, trainY), (testX, testY) = cifar10.load_data()
    # one hot encode target values
    trainY = to_categorical(trainY)
    testY = to_categorical(testY)
    return trainX, trainY, testX, testY
```

Listing 20.5: Example of a function for loading the CIFAR-10 dataset.

20.3.2 Prepare Pixel Data

We know that the pixel values for each image in the dataset are unsigned integers in the range between no color and full color, or 0 and 255. We do not know the best way to scale the pixel values for modeling, but we know that some scaling will be required. A good starting point is

to normalize the pixel values, e.g. rescale them to the range [0,1]. This involves first converting the data type from unsigned integers to floats, then dividing the pixel values by the maximum value.

```
...
# convert from integers to floats
train_norm = train.astype('float32')
test_norm = test.astype('float32')
# normalize to range 0-1
train_norm = train_norm / 255.0
test_norm = test_norm / 255.0
```

Listing 20.6: Example of normalizing pixel values.

The `prep_pixels()` function below implements these behaviors and is provided with the pixel values for both the train and test datasets that will need to be scaled.

```
# scale pixels
def prep_pixels(train, test):
    # convert from integers to floats
    train_norm = train.astype('float32')
    test_norm = test.astype('float32')
    # normalize to range 0-1
    train_norm = train_norm / 255.0
    test_norm = test_norm / 255.0
    # return normalized images
    return train_norm, test_norm
```

Listing 20.7: Example of scaling pixel values for the dataset.

This function must be called to prepare the pixel values prior to any modeling.

20.3.3 Define Model

Next, we need a way to define a neural network model. The `define_model()` function below will define and return this model and can be filled-in or replaced for a given model configuration that we wish to evaluate later.

```
# define cnn model
def define_model():
    model = Sequential()
    # ...
    return model
```

Listing 20.8: Example of a function for defining the model.

20.3.4 Evaluate Model

After the model is defined, we need to fit and evaluate it. Fitting the model will require that the number of training epochs and batch size to be specified. We will use a generic 100 training epochs for now and a modest batch size of 64. It is better to use a separate validation dataset, e.g. by splitting the train dataset into train and validation sets. We will not split the data in this case, and instead use the test dataset as a validation dataset to keep the example simple. The test dataset can be used like a validation dataset and evaluated at the end of each training

epoch. This will result in model evaluation scores on the train and test dataset each epoch that can be plotted later.

```
...
# fit model
history = model.fit(trainX, trainY, epochs=100, batch_size=64, validation_data=(testX,
    testY), verbose=0)
```

Listing 20.9: Example of fitting a defined model.

Once the model is fit, we can evaluate it directly on the test dataset.

```
...
# evaluate model
_, acc = model.evaluate(testX, testY, verbose=0)
```

Listing 20.10: Example of evaluating a fit model.

20.3.5 Present Results

Once the model has been evaluated, we can present the results. There are two key aspects to present: the diagnostics of the learning behavior of the model during training and the estimation of the model performance. First, the diagnostics involve creating a line plot showing model performance on the train and test set during training. These plots are valuable for getting an idea of whether a model is overfitting, underfitting, or has a good fit for the dataset.

We will create a single figure with two subplots, one for loss and one for accuracy. The blue lines will indicate model performance on the training dataset and orange lines will indicate performance on the hold out test dataset. The `summarize_diagnostics()` function below creates and shows this plot given the collected training histories. The plot is saved to file, specifically a file with the same name as the script with a `.png` extension.

```
# plot diagnostic learning curves
def summarize_diagnostics(history):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(history.history['loss'], color='blue', label='train')
    pyplot.plot(history.history['val_loss'], color='orange', label='test')
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Classification Accuracy')
    pyplot.plot(history.history['accuracy'], color='blue', label='train')
    pyplot.plot(history.history['val_accuracy'], color='orange', label='test')
    # save plot to file
    filename = sys.argv[0].split('/')[-1]
    pyplot.savefig(filename + '_plot.png')
    pyplot.close()
```

Listing 20.11: Example of a function for plotting learning curves.

Next, we can report the final model performance on the test dataset. This can be achieved by printing the classification accuracy directly.

```
...
print('> %.3f' % (acc * 100.0))
```

Listing 20.12: Example of summarizing model performance.

20.3.6 Complete Example

We need a function that will drive the test harness. This involves calling all the defined functions. The `run_test_harness()` function below implements this and can be called to kick-off the evaluation of a given model.

```
# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # prepare pixel data
    trainX, testX = prep_pixels(trainX, testX)
    # define model
    model = define_model()
    # fit model
    history = model.fit(trainX, trainY, epochs=100, batch_size=64, validation_data=(testX,
        testY), verbose=0)
    # evaluate model
    _, acc = model.evaluate(testX, testY, verbose=0)
    print('> %.3f' % (acc * 100.0))
    # learning curves
    summarize_diagnostics(history)
```

Listing 20.13: Example of a function for running the test harness.

We now have everything we need for the test harness. The complete code example for the test harness for the CIFAR-10 dataset is listed below.

```
# test harness for evaluating models on the cifar10 dataset
# NOTE: no model is defined, this example will not run
import sys
from matplotlib import pyplot
from keras.datasets import cifar10
from keras.utils import to_categorical
from keras.models import Sequential

# load train and test dataset
def load_dataset():
    # load dataset
    (trainX, trainY), (testX, testY) = cifar10.load_data()
    # one hot encode target values
    trainY = to_categorical(trainY)
    testY = to_categorical(testY)
    return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
    # convert from integers to floats
    train_norm = train.astype('float32')
    test_norm = test.astype('float32')
    # normalize to range 0-1
    train_norm = train_norm / 255.0
    test_norm = test_norm / 255.0
    return train_norm, test_norm
```

```

test_norm = test_norm / 255.0
# return normalized images
return train_norm, test_norm

# define cnn model
def define_model():
    model = Sequential()
    # ...
    return model

# plot diagnostic learning curves
def summarize_diagnostics(history):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(history.history['loss'], color='blue', label='train')
    pyplot.plot(history.history['val_loss'], color='orange', label='test')
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Classification Accuracy')
    pyplot.plot(history.history['accuracy'], color='blue', label='train')
    pyplot.plot(history.history['val_accuracy'], color='orange', label='test')
    # save plot to file
    filename = sys.argv[0].split('/')[-1]
    pyplot.savefig(filename + '_plot.png')
    pyplot.close()

# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # prepare pixel data
    trainX, testX = prep_pixels(trainX, testX)
    # define model
    model = define_model()
    # fit model
    history = model.fit(trainX, trainY, epochs=100, batch_size=64, validation_data=(testX,
        testY), verbose=0)
    # evaluate model
    _, acc = model.evaluate(testX, testY, verbose=0)
    print('> %.3f' % (acc * 100.0))
    # learning curves
    summarize_diagnostics(history)

# entry point, run the test harness
run_test_harness()

```

Listing 20.14: Example of defining a test harness for modeling the CIFAR-10 dataset.

This test harness can evaluate any CNN models we may wish to evaluate on the CIFAR-10 dataset and can run on the CPU or GPU. Note: as is, no model is defined, so this complete example cannot be run. Next, let's look at how we can define and evaluate a baseline model.

20.4 How to Develop a Baseline Model

We can now investigate a baseline model for the CIFAR-10 dataset. A baseline model will establish a minimum model performance to which all of our other models can be compared, as well as a model architecture that we can use as the basis of study and improvement. A good starting point is the general architectural principles of the VGG models. These are a good starting point because they achieved top performance in the ILSVRC 2014 competition and because the modular structure of the architecture is easy to understand and implement. For more details on the VGG model, see the 2015 paper *Very Deep Convolutional Networks for Large-Scale Image Recognition*.

The architecture involves stacking convolutional layers with small 3×3 filters followed by a max pooling layer. Together, these layers form a block, and these blocks can be repeated where the number of filters in each block is increased with the depth of the network such as 32, 64, 128, 256 for the first four blocks of the model. Padding is used on the convolutional layers to ensure the height and width of the output feature maps matches the inputs. We can explore this architecture on the CIFAR-10 problem and compare a model with this architecture with 1, 2, and 3 blocks. Each layer will use the ReLU activation function and the He weight initialization, which are generally best practices. For example, a 3-block VGG-style architecture can be defined in Keras as follows:

```
# example of a 3-block vgg style architecture
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same', input_shape=(32, 32, 3)))
model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(MaxPooling2D((2, 2)))
...
...
```

Listing 20.15: Example of defining a VGG-style feature extraction model.

This defines the feature detector part of the model. This must be coupled with a classifier part of the model that interprets the features and makes a prediction as to which class a given photo belongs. This can be fixed for each model that we investigate. First, the feature maps output from the feature extraction part of the model must be flattened. We can then interpret them with one or more fully connected layers, and then output a prediction. The output layer must have 10 nodes for the 10 classes and use the softmax activation function.

```
...
# example output part of the model
model.add(Flatten())
model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(10, activation='softmax'))
```

Listing 20.16: Example of defining a classifier output model.

The model will be optimized using stochastic gradient descent. We will use a modest learning rate of 0.001 and a large momentum of 0.9, both of which are good general starting points. The model will optimize the categorical cross-entropy loss function required for multiclass classification and will monitor classification accuracy.

```
...
# compile model
opt = SGD(lr=0.001, momentum=0.9)
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
```

Listing 20.17: Example of defining the optimization algorithm.

We now have enough elements to define our VGG-style baseline models. We can define three different model architectures with 1, 2, and 3 VGG modules which requires that we define 3 separate versions of the `define_model()` function, provided below. To test each model, a new script must be created (e.g. `model_baseline1.py`, `model_baseline2.py`, `model_baseline3.py`) using the test harness defined in the previous section, and with the new version of the `define_model()` function defined below. Let's take a look at each `define_model()` function and the evaluation of the resulting test harness in turn.

```
# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same', input_shape=(32, 32, 3)))
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(10, activation='softmax'))
    # compile model
    opt = SGD(lr=0.001, momentum=0.9)
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

Listing 20.18: Example of defining a 1 VGG block model.

Running the model in the test harness first prints the classification accuracy on the test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved a classification accuracy of just less than 70%.

```
> 67.070
```

Listing 20.19: Example output from evaluating a 1 VGG block model on the CIFAR-10 dataset.

A figure is created and saved to file showing the learning curves of the model during training on the train and test dataset, both with regards to the loss and accuracy. In this case, we can see that the model rapidly overfits the test dataset. This is clear if we look at the plot of loss (top plot), we can see that the model's performance on the training dataset (blue) continues to improve whereas the performance on the test dataset (orange) improves, then starts to get worse at around 15 epochs.

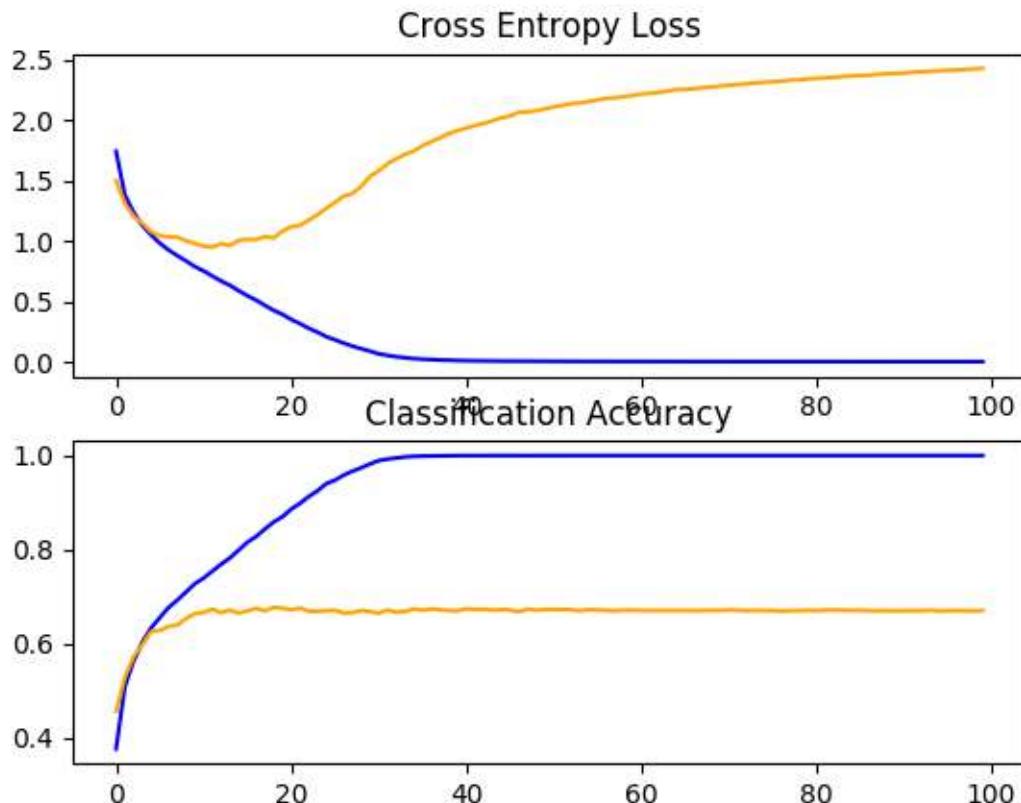


Figure 20.2: Line Plots of Learning Curves for VGG 1 Baseline on the CIFAR-10 Dataset.

20.4.1 Baseline: 2 VGG Blocks

The `define_model()` function for two VGG blocks is listed below.

```
# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same', input_shape=(32, 32, 3)))
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
```

```
model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(10, activation='softmax'))
# compile model
opt = SGD(lr=0.001, momentum=0.9)
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
return model
```

Listing 20.20: Example of defining a 2 VGG blocks model.

Running the model in the test harness first prints the classification accuracy on the test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model with two blocks performs better than the model with a single block: a good sign.

```
> 71.080
```

Listing 20.21: Example output from evaluating a 2 VGG block model on the CIFAR-10 dataset.

A figure showing learning curves is created and saved to file. In this case, we continue to see strong overfitting.

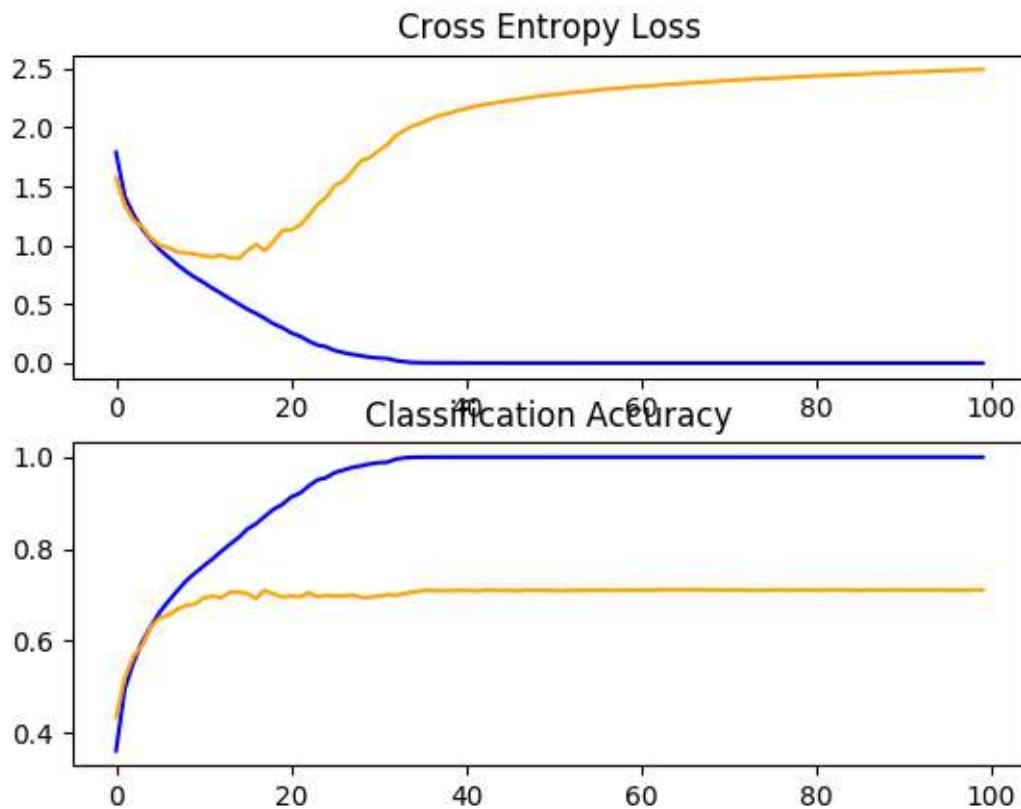


Figure 20.3: Line Plots of Learning Curves for VGG 2 Baseline on the CIFAR-10 Dataset.

20.4.2 Baseline: 3 VGG Blocks

The `define_model()` function for three VGG blocks is listed below.

```
# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same', input_shape=(32, 32, 3)))
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
```

```

model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(10, activation='softmax'))
# compile model
opt = SGD(lr=0.001, momentum=0.9)
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
return model

```

Listing 20.22: Example of defining a 3 VGG blocks model.

Running the model in the test harness first prints the classification accuracy on the test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, yet another modest increase in performance is seen as the depth of the model was increased.

```
> 73.500
```

Listing 20.23: Example output from evaluating a 3 VGG block model on the CIFAR-10 dataset.

Reviewing the figures showing the learning curves, again we see dramatic overfitting within the first 20 training epochs.

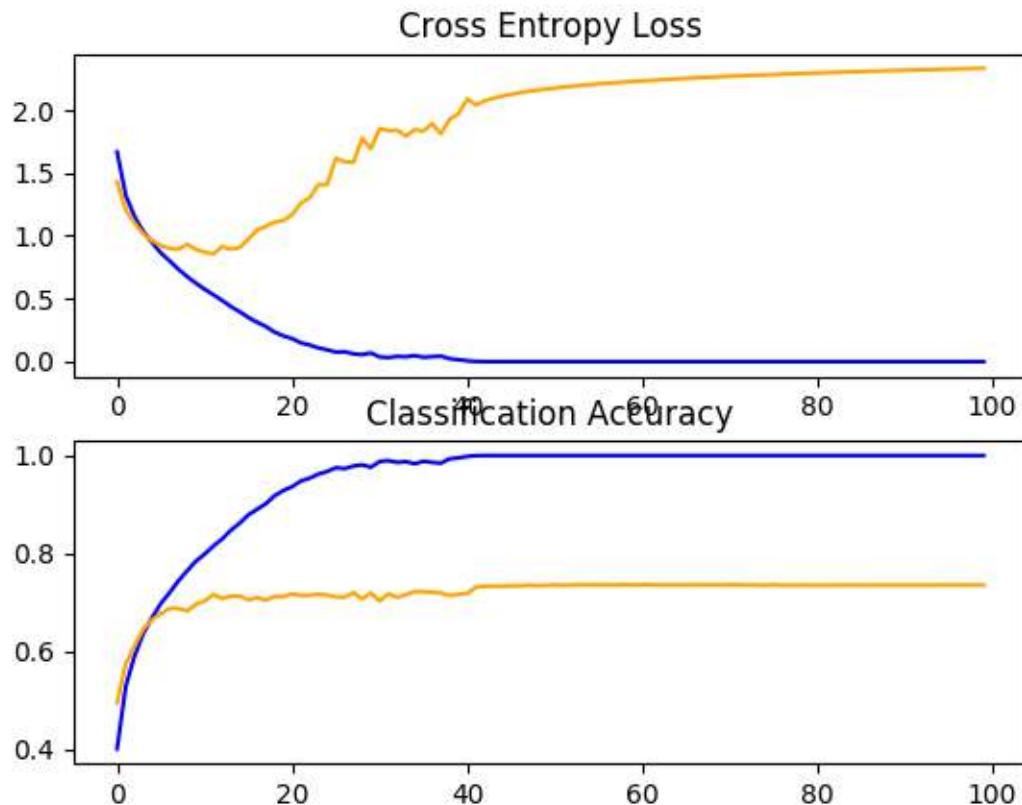


Figure 20.4: Line Plots of Learning Curves for VGG 3 Baseline on the CIFAR-10 Dataset.

20.4.3 Discussion

We have explored three different models with a VGG-based architecture. The results can be summarized below, although we must assume some variance in these results given the stochastic nature of the algorithm:

- **VGG 1:** 67.070%.
- **VGG 2:** 71.080%.
- **VGG 3:** 73.500%.

In all cases, the model was able to learn the training dataset, showing an improvement on the training dataset that at least continued to 40 epochs, and perhaps more. This is a good sign, as it shows that the problem is learnable and that all three models have sufficient capacity to learn the problem. The results of the model on the test dataset showed an improvement in classification accuracy with each increase in the depth of the model. It is possible that this trend would continue if models with four and five layers were evaluated, and this might make an interesting extension. Nevertheless, all three models showed the same pattern of dramatic overfitting at around 15-to-20 epochs.

These results suggest that the model with three VGG blocks is a good starting point or baseline model for our investigation. The results also suggest that the model is in need of regularization to address the rapid overfitting of the test dataset. More generally, the results suggest that it may be useful to investigate techniques that slow down the convergence (rate of learning) of the model. This may include techniques such as data augmentation as well as learning rate schedules, changes to the batch size, and perhaps more. In the next section, we will investigate some of these ideas for improving model performance.

20.5 How to Develop an Improved Model

Now that we have established a baseline model, the VGG architecture with three blocks, we can investigate modifications to the model and the training algorithm that seek to improve performance. We will look at two main areas to address the severe overfitting observed, namely regularization and data augmentation.

20.5.1 Dropout Regularization

Dropout is a simple technique that will randomly drop nodes out of the network. It has a regularizing effect as the remaining nodes must adapt to pick-up the slack of the removed nodes. Dropout can be added to the model by adding new `Dropout` layers, where the amount of nodes removed is specified as a parameter. There are many patterns for adding Dropout to a model, in terms of where in the model to add the layers and how much dropout to use. In this case, we will add `Dropout` layers after each max pooling layer and after the fully connected layer, and use a fixed dropout rate of 20% (e.g. retain 80% of the nodes). The updated VGG 3 baseline model with dropout is listed below.

```
# define cnn model
def define_model():
```

```

model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same', input_shape=(32, 32, 3)))
model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.2))
model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.2))
model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
# compile model
opt = SGD(lr=0.001, momentum=0.9)
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
return model

```

Listing 20.24: Example of defining a 3 VGG block model with dropout.

The full code listing is provided below for completeness.

```

# baseline model with dropout on the cifar10 dataset
import sys
from matplotlib import pyplot
from keras.datasets import cifar10
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.optimizers import SGD

# load train and test dataset
def load_dataset():
    # load dataset
    (trainX, trainY), (testX, testY) = cifar10.load_data()
    # one hot encode target values
    trainY = to_categorical(trainY)
    testY = to_categorical(testY)
    return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
    # convert from integers to floats

```

```
train_norm = train.astype('float32')
test_norm = test.astype('float32')
# normalize to range 0-1
train_norm = train_norm / 255.0
test_norm = test_norm / 255.0
# return normalized images
return train_norm, test_norm

# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same', input_shape=(32, 32, 3)))
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dropout(0.2))
    model.add(Dense(10, activation='softmax'))
    # compile model
    opt = SGD(lr=0.001, momentum=0.9)
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    return model

# plot diagnostic learning curves
def summarize_diagnostics(history):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(history.history['loss'], color='blue', label='train')
    pyplot.plot(history.history['val_loss'], color='orange', label='test')
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Classification Accuracy')
    pyplot.plot(history.history['accuracy'], color='blue', label='train')
    pyplot.plot(history.history['val_accuracy'], color='orange', label='test')
    # save plot to file
    filename = sys.argv[0].split('/')[-1]
    pyplot.savefig(filename + '_plot.png')
    pyplot.close()

# run the test harness for evaluating a model
```

```
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # prepare pixel data
    trainX, testX = prep_pixels(trainX, testX)
    # define model
    model = define_model()
    # fit model
    history = model.fit(trainX, trainY, epochs=100, batch_size=64, validation_data=(testX,
        testY), verbose=0)
    # evaluate model
    _, acc = model.evaluate(testX, testY, verbose=0)
    print('> %.3f' % (acc * 100.0))
    # learning curves
    summarize_diagnostics(history)

# entry point, run the test harness
run_test_harness()
```

Listing 20.25: Example of evaluating a 3-block VGG model with dropout.

Running the model in the test harness prints the classification accuracy on the test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see a jump in classification accuracy by about 10% from about 73% without dropout to about 83% with dropout.

```
> 83.450
```

Listing 20.26: Example output from evaluating a 3-block VGG model with dropout.

Reviewing the learning curve for the model, we can see that overfitting has been reduced. The model converges well for about 40 or 50 epochs, at which point there is no further improvement on the test dataset. This is a great result. We could elaborate upon this model and add early stopping with a patience of about 10 epochs to save a well-performing model on the test set during training at around the point that no further improvements are observed. We could also try exploring a learning rate schedule that drops the learning rate after improvements on the test set stall. Dropout has performed well, and we do not know that the chosen rate of 20% is the best. We could explore other dropout rates, as well as different positioning of the dropout layers in the model architecture.

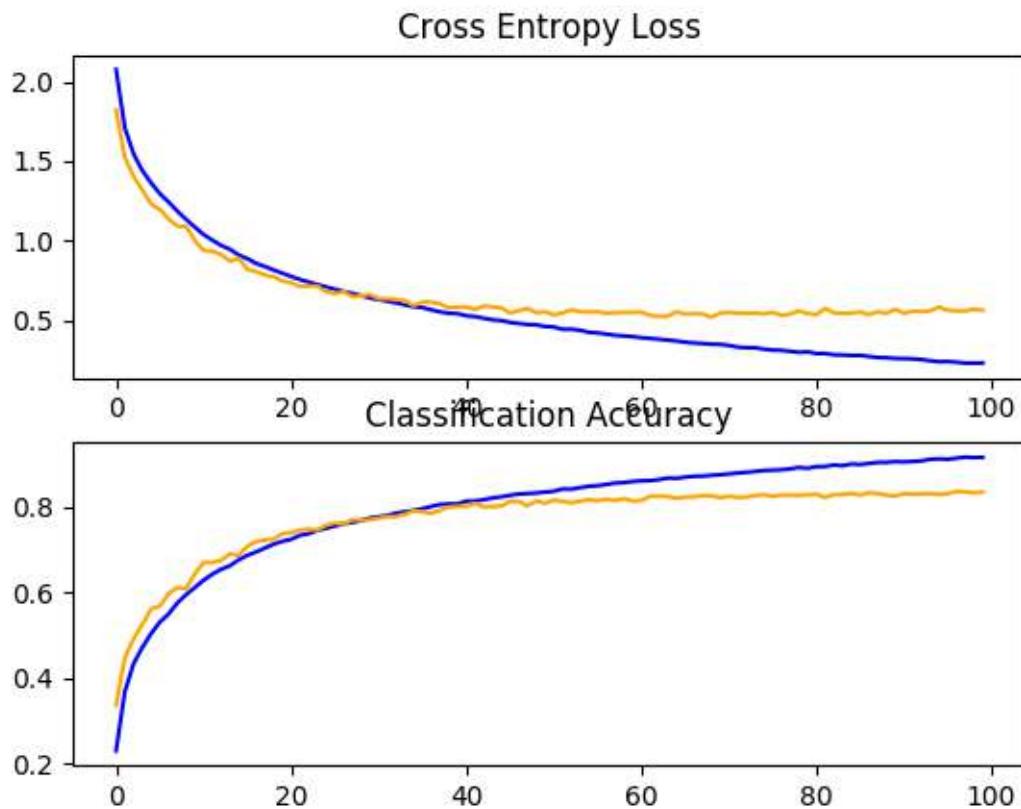


Figure 20.5: Line Plots of Learning Curves for Baseline Model With Dropout on the CIFAR-10 Dataset.

20.5.2 Data Augmentation

Data augmentation involves making copies of the examples in the training dataset with small random modifications. This has a regularizing effect as it both expands the training dataset and allows the model to learn the same general features, although in a more generalized manner (data augmentation was introduced in Chapter 9). There are many types of data augmentation that could be applied. Given that the dataset is comprised of small photos of objects, we do not want to use augmentation that distorts the images too much, so that useful features in the images can be preserved and used.

The types of random augmentations that could be useful include a horizontal flip, minor shifts of the image, and perhaps small zooming or cropping of the image. We will investigate the effect of simple augmentation on the baseline image, specifically horizontal flips and 10% shifts in the height and width of the image. This can be implemented in Keras using the `ImageDataGenerator` class; for example:

```
# create data generator
datagen = ImageDataGenerator(width_shift_range=0.1, height_shift_range=0.1,
    horizontal_flip=True)
# prepare iterator
it_train = datagen.flow(trainX, trainY, batch_size=64)
```

Listing 20.27: Example of preparing the image data generator for augmentation.

This can be used during training by passing the iterator to the `model.fit_generator()` function and defining the number of batches in a single epoch.

```
# fit model
steps = int(trainX.shape[0] / 64)
history = model.fit_generator(it_train, steps_per_epoch=steps, epochs=100,
    validation_data=(testX, testY), verbose=0)
```

Listing 20.28: Example of fitting the model with data augmentation.

No changes to the model are required. The updated version of the `run_test_harness()` function to support data augmentation is listed below.

```
# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # prepare pixel data
    trainX, testX = prep_pixels(trainX, testX)
    # define model
    model = define_model()
    # create data generator
    datagen = ImageDataGenerator(width_shift_range=0.1, height_shift_range=0.1,
        horizontal_flip=True)
    # prepare iterator
    it_train = datagen.flow(trainX, trainY, batch_size=64)
    # fit model
    steps = int(trainX.shape[0] / 64)
    history = model.fit_generator(it_train, steps_per_epoch=steps, epochs=100,
        validation_data=(testX, testY), verbose=0)
    # evaluate model
    _, acc = model.evaluate(testX, testY, verbose=0)
    print('> %.3f' % (acc * 100.0))
    # learning curves
    summarize_diagnostics(history)
```

Listing 20.29: Example of the updated function for running the test harness with data augmentation.

The full code listing is provided below for completeness.

```
# baseline model with data augmentation on the cifar10 dataset
import sys
from matplotlib import pyplot
from keras.datasets import cifar10
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD
from keras.preprocessing.image import ImageDataGenerator
```

```
# load train and test dataset
def load_dataset():
    # load dataset
    (trainX, trainY), (testX, testY) = cifar10.load_data()
    # one hot encode target values
    trainY = to_categorical(trainY)
    testY = to_categorical(testY)
    return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
    # convert from integers to floats
    train_norm = train.astype('float32')
    test_norm = test.astype('float32')
    # normalize to range 0-1
    train_norm = train_norm / 255.0
    test_norm = test_norm / 255.0
    # return normalized images
    return train_norm, test_norm

# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same', input_shape=(32, 32, 3)))
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(10, activation='softmax'))
    # compile model
    opt = SGD(lr=0.001, momentum=0.9)
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    return model

# plot diagnostic learning curves
def summarize_diagnostics(history):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(history.history['loss'], color='blue', label='train')
    pyplot.plot(history.history['val_loss'], color='orange', label='test')
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Classification Accuracy')
```

```

pyplot.plot(history.history['accuracy'], color='blue', label='train')
pyplot.plot(history.history['val_accuracy'], color='orange', label='test')
# save plot to file
filename = sys.argv[0].split('/')[-1]
pyplot.savefig(filename + '_plot.png')
pyplot.close()

# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # prepare pixel data
    trainX, testX = prep_pixels(trainX, testX)
    # define model
    model = define_model()
    # create data generator
    datagen = ImageDataGenerator(width_shift_range=0.1, height_shift_range=0.1,
        horizontal_flip=True)
    # prepare iterator
    it_train = datagen.flow(trainX, trainY, batch_size=64)
    # fit model
    steps = int(trainX.shape[0] / 64)
    history = model.fit_generator(it_train, steps_per_epoch=steps, epochs=100,
        validation_data=(testX, testY), verbose=0)
    # evaluate model
    _, acc = model.evaluate(testX, testY, verbose=0)
    print('> %.3f' % (acc * 100.0))
    # learning curves
    summarize_diagnostics(history)

# entry point, run the test harness
run_test_harness()

```

Listing 20.30: Example of evaluating a 3-block VGG model with data augmentation.

Running the model in the test harness prints the classification accuracy on the test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we see another large improvement in model performance, much like we saw with dropout. We see an improvement of about 11% from about 73% for the baseline model to about 84%.

> 84.470

Listing 20.31: Example output from evaluating a 3-block VGG model with data augmentation.

Reviewing the learning curves, we see a similar improvement in model performances as we do with dropout, although the plot of loss suggests that model performance on the test set may have stalled slightly sooner than it did with dropout. The results suggest that perhaps a configuration that used both dropout and data augmentation might be effective.

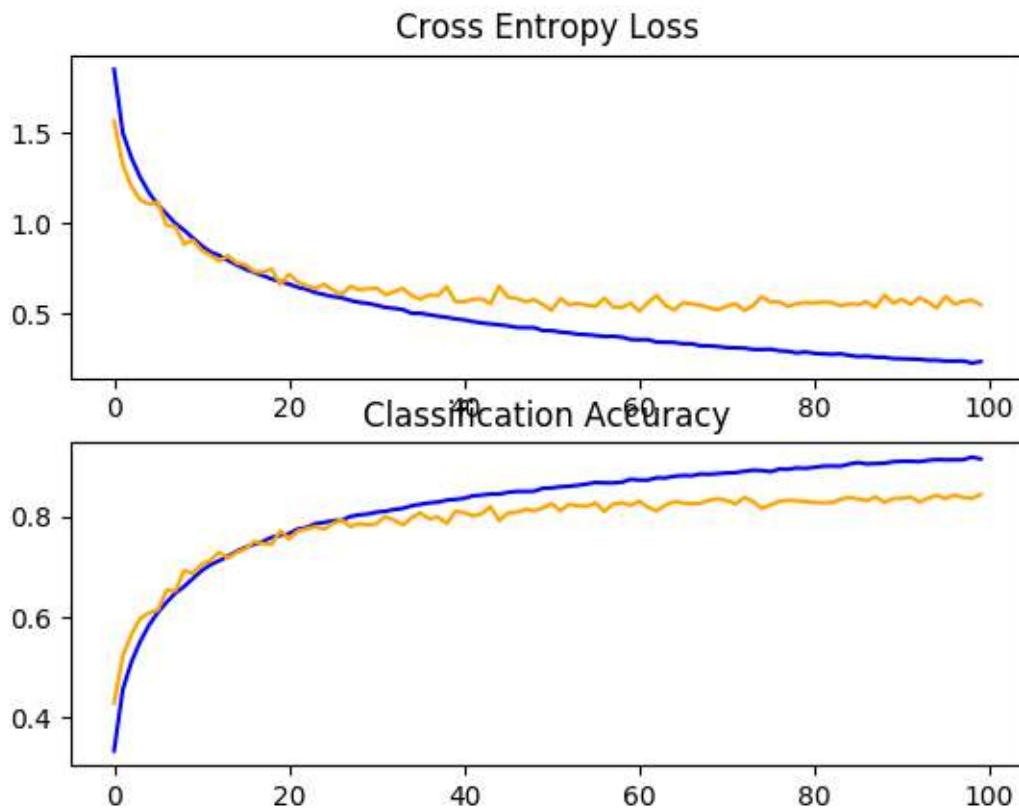


Figure 20.6: Line Plots of Learning Curves for Baseline Model With Data Augmentation on the CIFAR-10 Dataset.

20.5.3 Dropout and Data Augmentation

In the previous two sections, we discovered that both dropout and data augmentation resulted in a significant improvement in model performance. In this section, we can experiment with combining both of these changes to the model to see if a further improvement can be achieved. Specifically, whether using both regularization techniques together results in better performance than either technique used alone. The full code listing of a model with fixed dropout and data augmentation is provided below for completeness.

```
# baseline model with dropout and data augmentation on the cifar10 dataset
import sys
from matplotlib import pyplot
from keras.datasets import cifar10
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD
from keras.preprocessing.image import ImageDataGenerator
```

```
from keras.layers import Dropout

# load train and test dataset
def load_dataset():
    # load dataset
    (trainX, trainY), (testX, testY) = cifar10.load_data()
    # one hot encode target values
    trainY = to_categorical(trainY)
    testY = to_categorical(testY)
    return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
    # convert from integers to floats
    train_norm = train.astype('float32')
    test_norm = test.astype('float32')
    # normalize to range 0-1
    train_norm = train_norm / 255.0
    test_norm = test_norm / 255.0
    # return normalized images
    return train_norm, test_norm

# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same', input_shape=(32, 32, 3)))
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dropout(0.2))
    model.add(Dense(10, activation='softmax'))

    # compile model
    opt = SGD(lr=0.001, momentum=0.9)
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    return model

# plot diagnostic learning curves
def summarize_diagnostics(history):
    # plot loss
    pyplot.subplot(211)
```

```

pyplot.title('Cross Entropy Loss')
pyplot.plot(history.history['loss'], color='blue', label='train')
pyplot.plot(history.history['val_loss'], color='orange', label='test')
# plot accuracy
pyplot.subplot(212)
pyplot.title('Classification Accuracy')
pyplot.plot(history.history['accuracy'], color='blue', label='train')
pyplot.plot(history.history['val_accuracy'], color='orange', label='test')
# save plot to file
filename = sys.argv[0].split('/')[-1]
pyplot.savefig(filename + '_plot.png')
pyplot.close()

# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # prepare pixel data
    trainX, testX = prep_pixels(trainX, testX)
    # define model
    model = define_model()
    # create data generator
    datagen = ImageDataGenerator(width_shift_range=0.1, height_shift_range=0.1,
        horizontal_flip=True)
    # prepare iterator
    it_train = datagen.flow(trainX, trainY, batch_size=64)
    # fit model
    steps = int(trainX.shape[0] / 64)
    history = model.fit_generator(it_train, steps_per_epoch=steps, epochs=200,
        validation_data=(testX, testY), verbose=0)
    # evaluate model
    _, acc = model.evaluate(testX, testY, verbose=0)
    print('> %.3f' % (acc * 100.0))
    # learning curves
    summarize_diagnostics(history)

# entry point, run the test harness
run_test_harness()

```

Listing 20.32: Example of evaluating a 3-block VGG model with dropout and data augmentation.

Running the model in the test harness prints the classification accuracy on the test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that as we would have hoped, using both regularization techniques together has resulted in a further lift in model performance on the test set. In this case, combining fixed dropout with about 83% and data augmentation with about 84% has resulted in an improvement to about 86% classification accuracy.

```
> 85.880
```

Listing 20.33: Example output from evaluating a 3-block VGG model with dropout and data augmentation.

Reviewing the learning curves, we can see that the convergence behavior of the model is also better than either fixed dropout and data augmentation alone. Learning has been slowed without overfitting, allowing continued improvement. The plot also suggests that learning may not have stalled and may have continued to improve if allowed to continue, but perhaps very modestly. Results might be further improved if a pattern of increasing dropout was used instead of a fixed dropout rate throughout the depth of the model.

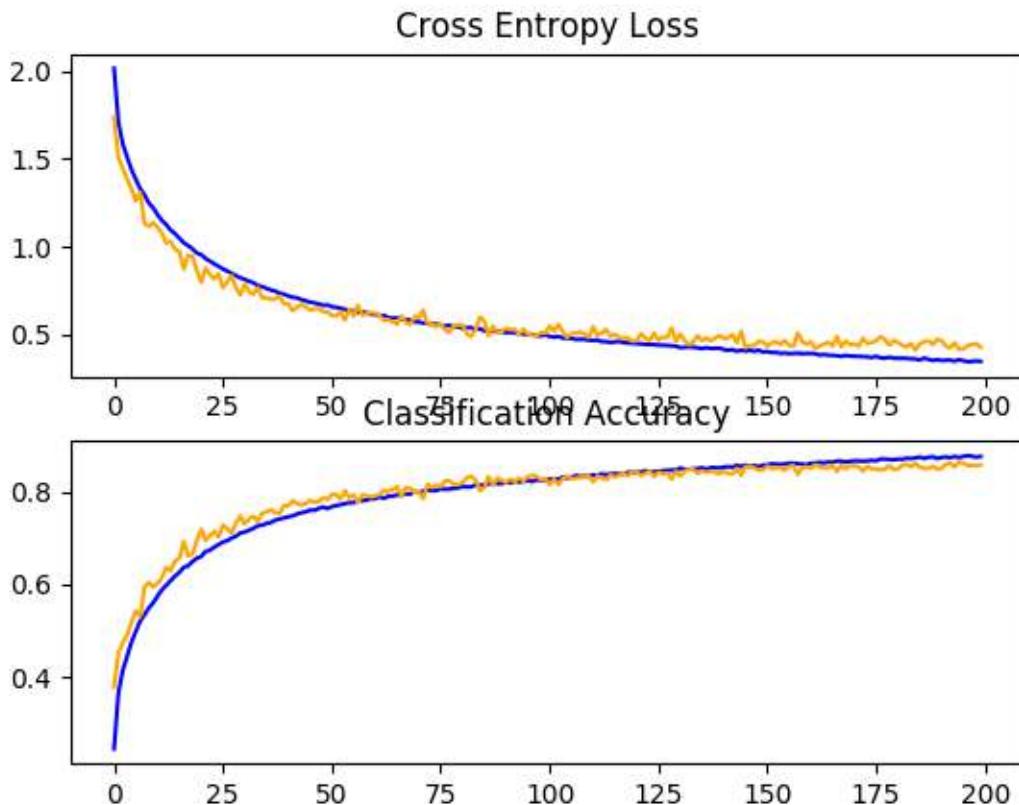


Figure 20.7: Line Plots of Learning Curves for Baseline Model With Dropout and Data Augmentation on the CIFAR-10 Dataset.

20.5.4 Dropout and Data Augmentation and Batch Normalization

We can expand upon the previous example in a few specific ways. First, we can increase the number of training epochs from 200 to 400, to give the model more of an opportunity to improve. Next, we can add batch normalization in an effort to stabilize the learning and perhaps accelerate the learning process. To offset this acceleration, we can increase the regularization by changing the dropout from a fixed pattern to an increasing pattern.

Finally, we can change dropout from a fixed amount at each level, to instead using an increasing trend with less dropout in early layers and more dropout in later layers, increasing 10% each time from 20% after the first block to 50% at the fully connected layer. This type of increasing dropout with the depth of the model is a common pattern. It is effective as it forces

layers deep in the model to regularize more than layers closer to the input. The updated model definition is listed below.

```
# baseline model with dropout and data augmentation on the cifar10 dataset
import sys
from matplotlib import pyplot
from keras.datasets import cifar10
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD
from keras.preprocessing.image import ImageDataGenerator
from keras.layers import Dropout
from keras.layers import BatchNormalization

# load train and test dataset
def load_dataset():
    # load dataset
    (trainX, trainY), (testX, testY) = cifar10.load_data()
    # one hot encode target values
    trainY = to_categorical(trainY)
    testY = to_categorical(testY)
    return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
    # convert from integers to floats
    train_norm = train.astype('float32')
    test_norm = test.astype('float32')
    # normalize to range 0-1
    train_norm = train_norm / 255.0
    test_norm = test_norm / 255.0
    # return normalized images
    return train_norm, test_norm

# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same', input_shape=(32, 32, 3)))
    model.add(BatchNormalization())
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(BatchNormalization())
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D((2, 2)))
```

```
model.add(Dropout(0.3))
model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.4))
model.add(Flatten())
model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
# compile model
opt = SGD(lr=0.001, momentum=0.9)
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
return model

# plot diagnostic learning curves
def summarize_diagnostics(history):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(history.history['loss'], color='blue', label='train')
    pyplot.plot(history.history['val_loss'], color='orange', label='test')
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Classification Accuracy')
    pyplot.plot(history.history['accuracy'], color='blue', label='train')
    pyplot.plot(history.history['val_accuracy'], color='orange', label='test')
    # save plot to file
    filename = sys.argv[0].split('/')[-1]
    pyplot.savefig(filename + '_plot.png')
    pyplot.close()

# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # prepare pixel data
    trainX, testX = prep_pixels(trainX, testX)
    # define model
    model = define_model()
    # create data generator
    datagen = ImageDataGenerator(width_shift_range=0.1, height_shift_range=0.1,
        horizontal_flip=True)
    # prepare iterator
    it_train = datagen.flow(trainX, trainY, batch_size=64)
    # fit model
    steps = int(trainX.shape[0] / 64)
    history = model.fit_generator(it_train, steps_per_epoch=steps, epochs=400,
        validation_data=(testX, testY), verbose=0)
    # evaluate model
    _, acc = model.evaluate(testX, testY, verbose=0)
    print('> %.3f' % (acc * 100.0))
```

```
# learning curves
summarize_diagnostics(history)

# entry point, run the test harness
run_test_harness()
```

Listing 20.34: Example of evaluating a 3-block VGG model with increasing dropout data augmentation and batch norm.

Running the model in the test harness prints the classification accuracy on the test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that we achieved a further lift in model performance to about 89% accuracy, improving upon using the combination of dropout and data augmentation alone at about 85%.

```
> 88.620
```

Listing 20.35: Example output from evaluating a 3-block VGG model with increasing dropout data augmentation and batch norm.

Reviewing the learning curves, we can see the training of the model shows continued improvement for nearly the duration of 400 epochs. We can see perhaps a slight drop-off on the test dataset at around 300 epochs, but the improvement trend does continue. The model may benefit from further training epochs.

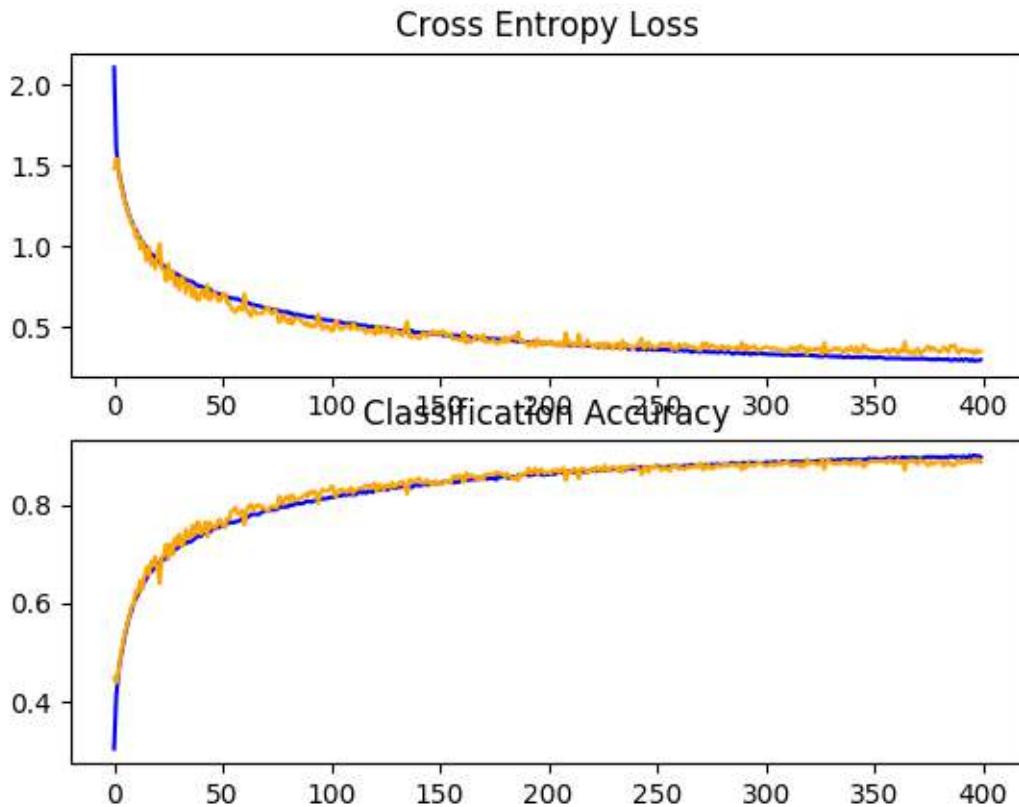


Figure 20.8: Line Plots of Learning Curves for Baseline Model With Increasing Dropout, Data Augmentation, and Batch Normalization on the CIFAR-10 Dataset.

20.5.5 Discussion

In this section, we explored four approaches designed to slow down the convergence of the model. A summary of the results is provided below:

- **Baseline + Dropout:** 83.450%.
- **Baseline + Data Augmentation:** 84.470%.
- **Baseline + Dropout + Data Augmentation:** 85.880%.
- **Baseline + Increasing Dropout + Data Augmentation + Batch Norm:** 88.620%.

The results suggest that both dropout and data augmentation are having the desired effect. The further combinations show that the model is now learning well and we have good control over the rate of learning without overfitting. We might be able to achieve further improvements with additional regularization. This could be achieved with more aggressive dropout in later layers. It is possible that further addition of weight decay may improve the model.

So far, we have not tuned the hyperparameters of the learning algorithm, such as the learning rate, which is perhaps the most important hyperparameter. We may expect further

improvements with adaptive changes to the learning rate, such as use of an adaptive learning rate technique such as Adam. These types of changes may help to refine the model once converged.

20.6 How to Finalize the Model and Make Predictions

The process of model improvement may continue for as long as we have ideas and the time and resources to test them out. At some point, a final model configuration must be chosen and adopted. In this case, we will use the best model after improvement, specifically the VGG-3 baseline with increasing dropout, data augmentation and batch normalization. First, we will finalize our model by fitting a model on the entire training dataset and saving the model to file for later use. We could load the model and evaluate its performance on the hold out test dataset, to get an idea of how well the chosen model actually performs in practice. This is not needed in this case, as we used the test dataset as the validation dataset and we already know how well the model will perform. Finally, we will use the saved model to make a prediction on a single image.

20.6.1 Save Final Model

A final model is typically fit on all available data, such as the combination of all train and test dataset. In this tutorial, we will demonstrate the final model fit only on the just training dataset to keep the example simple. The first step is to fit the final model on the entire training dataset.

```
...
# fit model
model.fit(trainX, trainY, epochs=100, batch_size=64, verbose=0)
```

Listing 20.36: Example of fitting the final model.

Once fit, we can save the final model to an H5 file by calling the `save()` function on the model and pass in the chosen filename.

```
...
# save model
model.save('final_model.h5')
```

Listing 20.37: Example of saving the final model.

Note: saving and loading a Keras model requires that the `h5py` library is installed on your workstation. The complete example of fitting the final model on the training dataset and saving it to file is listed below.

```
# save the final model to file
from keras.datasets import cifar10
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import BatchNormalization
from keras.optimizers import SGD
```

```
from keras.preprocessing.image import ImageDataGenerator

# load train and test dataset
def load_dataset():
    # load dataset
    (trainX, trainY), (testX, testY) = cifar10.load_data()
    # one hot encode target values
    trainY = to_categorical(trainY)
    testY = to_categorical(testY)
    return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
    # convert from integers to floats
    train_norm = train.astype('float32')
    test_norm = test.astype('float32')
    # normalize to range 0-1
    train_norm = train_norm / 255.0
    test_norm = test_norm / 255.0
    # return normalized images
    return train_norm, test_norm

# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same', input_shape=(32, 32, 3)))
    model.add(BatchNormalization())
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(BatchNormalization())
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.3))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(BatchNormalization())
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.4))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(BatchNormalization())
    model.add(Dropout(0.5))
    model.add(Dense(10, activation='softmax'))

    # compile model
    opt = SGD(lr=0.001, momentum=0.9)
```

```

model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
return model

# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # prepare pixel data
    trainX, testX = prep_pixels(trainX, testX)
    # define model
    model = define_model()
    # create data generator
    datagen = ImageDataGenerator(width_shift_range=0.1, height_shift_range=0.1,
        horizontal_flip=True)
    # prepare iterator
    it_train = datagen.flow(trainX, trainY, batch_size=64)
    # fit model
    steps = int(trainX.shape[0] / 64)
    model.fit_generator(it_train, steps_per_epoch=steps, epochs=400, validation_data=(testX,
        testY), verbose=0)
    # save model
    model.save('final_model.h5')

# entry point, run the test harness
run_test_harness()

```

Listing 20.38: Example of fitting and saving the final model.

After running this example you will now have a 4.3-megabyte file with the name `final_model.h5` in your current working directory.

20.6.2 Make Prediction

We can use our saved model to make a prediction on new images. The model assumes that new images are color, they have been segmented so that one image contains one centered object, and the size of the image is square with the size 32×32 pixels. Below is an image extracted from the CIFAR-10 test dataset.



Figure 20.9: Deer.

You can save it in your current working directory with the filename `sample_image.png`.

- Download the Deer Image (`sample_image.png`).¹

¹https://machinelearningmastery.com/wp-content/uploads/2019/02/sample_image-1.png

We will pretend this is an entirely new and unseen image, prepared in the required way, and see how we might use our saved model to predict the integer that the image represents. For this example, we expect class 4 for Deer. First, we can load the image and force it to the size to be 32×32 pixels. The loaded image can then be resized to have a single channel and represent a single sample in a dataset. The `load_image()` function implements this and will return the loaded image ready for classification. Importantly, the pixel values are prepared in the same way as the pixel values were prepared for the training dataset when fitting the final model, in this case, normalized.

```
# load and prepare the image
def load_image(filename):
    # load the image
    img = load_img(filename, target_size=(32, 32))
    # convert to array
    img = img_to_array(img)
    # reshape into a single sample with 3 channels
    img = img.reshape(1, 32, 32, 3)
    # prepare pixel data
    img = img.astype('float32')
    img = img / 255.0
    return img
```

Listing 20.39: Example of a function for loading and preparing an image for making a prediction.

Next, we can load the model as in the previous section and call the `predict_classes()` function to predict the object in the image.

```
...
# predict the class
result = model.predict_classes(img)
```

Listing 20.40: Example of making a prediction with a prepared image.

The complete example is listed below.

```
# make a prediction for a new image.
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.models import load_model

# load and prepare the image
def load_image(filename):
    # load the image
    img = load_img(filename, target_size=(32, 32))
    # convert to array
    img = img_to_array(img)
    # reshape into a single sample with 3 channels
    img = img.reshape(1, 32, 32, 3)
    # prepare pixel data
    img = img.astype('float32')
    img = img / 255.0
    return img

# load an image and predict the class
def run_example():
    # load the image
```

```
img = load_image('sample_image.png')
# load model
model = load_model('final_model.h5')
# predict the class
result = model.predict_classes(img)
print(result[0])

# entry point, run the example
run_example()
```

Listing 20.41: Example of making a prediction with the final model.

Running the example first loads and prepares the image, loads the model, and then correctly predicts that the loaded image represents a `deer` or class 4.

4

Listing 20.42: Example output making a prediction with the final model.

20.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Pixel Scaling.** Explore alternate techniques for scaling the pixels, such as centering and standardization, and compare performance.
- **Learning Rates.** Explore alternate learning rates, adaptive learning rates, and learning rate schedules and compare performance.
- **Transfer Learning.** Explore using transfer learning, such as a pre-trained VGG-16 model on this dataset.

If you explore any of these extensions, I'd love to know.

20.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

20.8.1 API

- Keras Datasets API.
<https://keras.io/datasets/>
- Keras Datasets Code.
<https://github.com/keras-team/keras/tree/master/keras/datasets>

20.8.2 Articles

- Classification datasets results, What is the class of this image?
http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html
- CIFAR-10, Wikipedia.
<https://en.wikipedia.org/wiki/CIFAR-10>
- The CIFAR-10 dataset and CIFAR-100 datasets.
<https://www.cs.toronto.edu/~kriz/cifar.html>
- CIFAR-10 — Object Recognition in Images, Kaggle.
<https://www.kaggle.com/c/cifar-10>
- Simple CNN 90%+, Pasquale Giovenale, Kaggle.
<https://www.kaggle.com/c/cifar-10/discussion/40237>

20.9 Summary

In this tutorial, you discovered how to develop a convolutional neural network model from scratch for object photo classification. Specifically, you learned:

- How to develop a test harness to develop a robust evaluation of a model and establish a baseline of performance for a classification task.
- How to explore extensions to a baseline model to improve learning and model capacity.
- How to develop a finalized model, evaluate the performance of the final model, and use it to make predictions on new images.

20.9.1 Next

In the next section, you will discover how to develop a deep convolutional neural network model for classifying photographs of dogs and cats.

Chapter 21

How to Classify Photographs of Dogs and Cats

The Dogs vs. Cats dataset is a standard computer vision dataset that involves classifying photos as either containing a dog or cat. Although the problem sounds simple, it was only effectively addressed in the last few years using deep learning convolutional neural networks. While the dataset is effectively solved, it can be used as the basis for learning and practicing how to develop, evaluate, and use convolutional deep learning neural networks for image classification from scratch.

This includes how to develop a robust test harness for estimating the performance of the model, how to explore improvements to the model, and how to save the model and later load it to make predictions on new data. In this tutorial, you will discover how to develop a convolutional neural network to classify photos of dogs and cats. After completing this tutorial, you will know:

- How to load and prepare photos of dogs and cats for modeling.
- How to develop a convolutional neural network for photo classification from scratch and improve model performance.
- How to develop a model for photo classification using transfer learning.

Let's get started.

21.1 Tutorial Overview

This tutorial is divided into six parts; they are:

1. Dogs vs. Cats Prediction Problem
2. Dogs vs. Cats Dataset Preparation
3. Develop a Baseline CNN Model
4. Develop Model Improvements
5. Explore Transfer Learning
6. How to Finalize the Model and Make Predictions

21.2 Dogs vs. Cats Prediction Problem

The dogs vs cats dataset refers to a dataset used for a Kaggle machine learning competition held in 2013. The dataset is comprised of photos of dogs and cats provided as a subset of photos from a much larger dataset of 3 million manually annotated photos. The dataset was developed as a partnership between Petfinder.com and Microsoft.

The dataset was originally used as a CAPTCHA (or Completely Automated Public Turing test to tell Computers and Humans Apart), that is, a task that it is believed a human finds trivial, but cannot be solved by a machine, used on websites to distinguish between human users and bots. Specifically, the task was referred to as *Asirra* or Animal Species Image Recognition for Restricting Access, a type of CAPTCHA. The task was described in the 2007 paper titled *Asirra: A CAPTCHA that Exploits Interest-Aligned Manual Image Categorization*.

We present Asirra, a CAPTCHA that asks users to identify cats out of a set of 12 photographs of both cats and dogs. Asirra is easy for users; user studies indicate it can be solved by humans 99.6% of the time in under 30 seconds. Barring a major advance in machine vision, we expect computers will have no better than a 1/54,000 chance of solving it.

— *Asirra: A CAPTCHA that Exploits Interest-Aligned Manual Image Categorization*, 2007.

At the time that the competition was posted, the state-of-the-art result was achieved with an SVM and described in a 2007 paper with the title *Machine Learning Attacks Against the Asirra CAPTCHA* that achieved 80% classification accuracy. It was this paper that demonstrated that the task was no longer a suitable task for a CAPTCHA soon after the task was proposed.

... we describe a classifier which is 82.7% accurate in telling apart the images of cats and dogs used in Asirra. This classifier is a combination of support-vector machine classifiers trained on color and texture features extracted from images. [...] Our results suggest caution against deploying Asirra without safeguards.

— *Machine Learning Attacks Against the Asirra CAPTCHA*, 2007.

The Kaggle competition provided 25,000 labeled photos: 12,500 dogs and the same number of cats. Predictions were then required on a test dataset of 12,500 unlabeled photographs. The competition was won by Pierre Sermanet (currently a research scientist at Google Brain) who achieved a classification accuracy of about 98.914% on a 70% subsample of the test dataset. His method was later described as part of the 2013 paper titled *OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks*. The dataset is straightforward to understand and small enough to fit into memory. As such, it has become a good *hello world* or *getting started* computer vision dataset for beginners when getting started with convolutional neural networks. As such, it is routine to achieve approximately 80% accuracy with a manually designed convolutional neural network and 90%+ accuracy using transfer learning on this task.

21.3 Dogs vs. Cats Dataset Preparation

The dataset can be downloaded for free from the Kaggle website, although I believe you must have a Kaggle account. If you do not have a Kaggle account, sign-up first. Download the dataset by visiting the Dogs vs. Cats Data page and click the *Download All* button.

- Dogs vs Cats Data Download Page.¹

This will download the 850-megabyte file `dogs-vs-cats.zip` to your workstation. Unzip the file and you will see `train.zip`, `train1.zip` and a `.csv` file. Unzip the `train.zip` file, as we will be focusing only on this dataset. You will now have a folder called `train/` that contains 25,000 `.jpg` files of dogs and cats. The photos are labeled by their filename, with the word `dog` or `cat`. The file naming convention is as follows:

```
cat.0.jpg
...
cat.124999.jpg
dog.0.jpg
dog.124999.jpg
```

Listing 21.1: Sample of files in the dataset.

21.3.1 Plot Dog and Cat Photos

Looking at a few random photos in the directory, you can see that the photos are color and have different shapes and sizes. For example, let's load and plot the first nine photos of dogs in a single figure. The complete example is listed below.

```
# plot dog photos from the dogs vs cats dataset
from matplotlib import pyplot
from matplotlib.image import imread
# define location of dataset
folder = 'train/'
# plot first few images
for i in range(9):
    # define subplot
    pyplot.subplot(330 + 1 + i)
    # define filename
    filename = folder + 'dog.' + str(i) + '.jpg'
    # load image pixels
    image = imread(filename)
    # plot raw pixel data
    pyplot.imshow(image)
# show the figure
pyplot.show()
```

Listing 21.2: Example of plotting photographs of dogs.

Running the example creates a figure showing the first nine photos of dogs in the dataset. We can see that some photos are landscape format, some are portrait format, and some are square.

¹<https://www.kaggle.com/c/dogs-vs-cats/data>

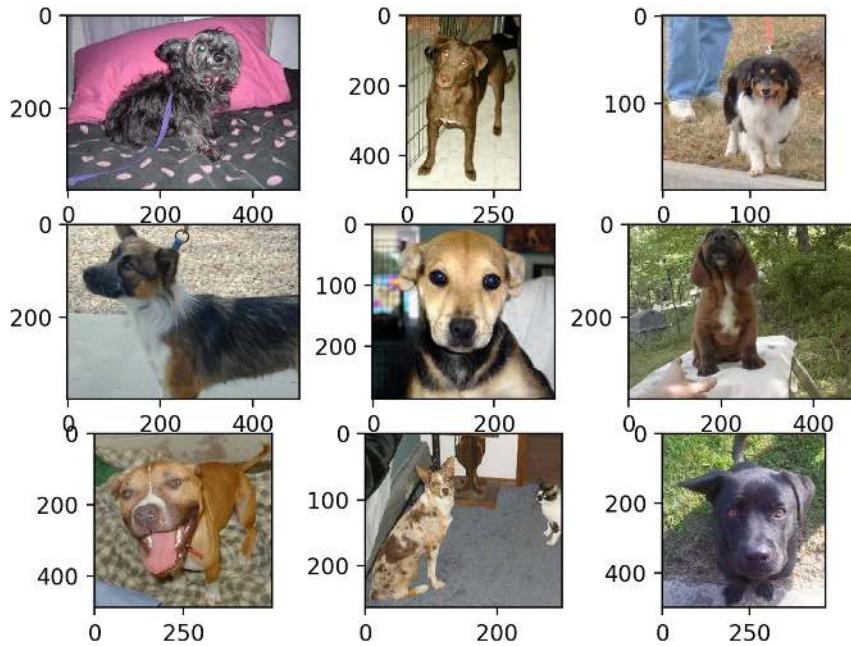


Figure 21.1: Plot of the First Nine Photos of Dogs in the Dogs vs Cats Dataset.

We can update the example and change it to plot cat photos instead; the complete example is listed below.

```
# plot cat photos from the dogs vs cats dataset
from matplotlib import pyplot
from matplotlib.image import imread
# define location of dataset
folder = 'train/'
# plot first few images
for i in range(9):
    # define subplot
    pyplot.subplot(330 + 1 + i)
    # define filename
    filename = folder + 'cat.' + str(i) + '.jpg'
    # load image pixels
    image = imread(filename)
    # plot raw pixel data
    pyplot.imshow(image)
# show the figure
pyplot.show()
```

Listing 21.3: Example of plotting photographs of cats.

Again, we can see that the photos are all different sizes. We can also see a photo where the cat is barely visible (bottom left corner) and another that has two cats (lower right corner). This suggests that any classifier fit on this problem will have to be robust.

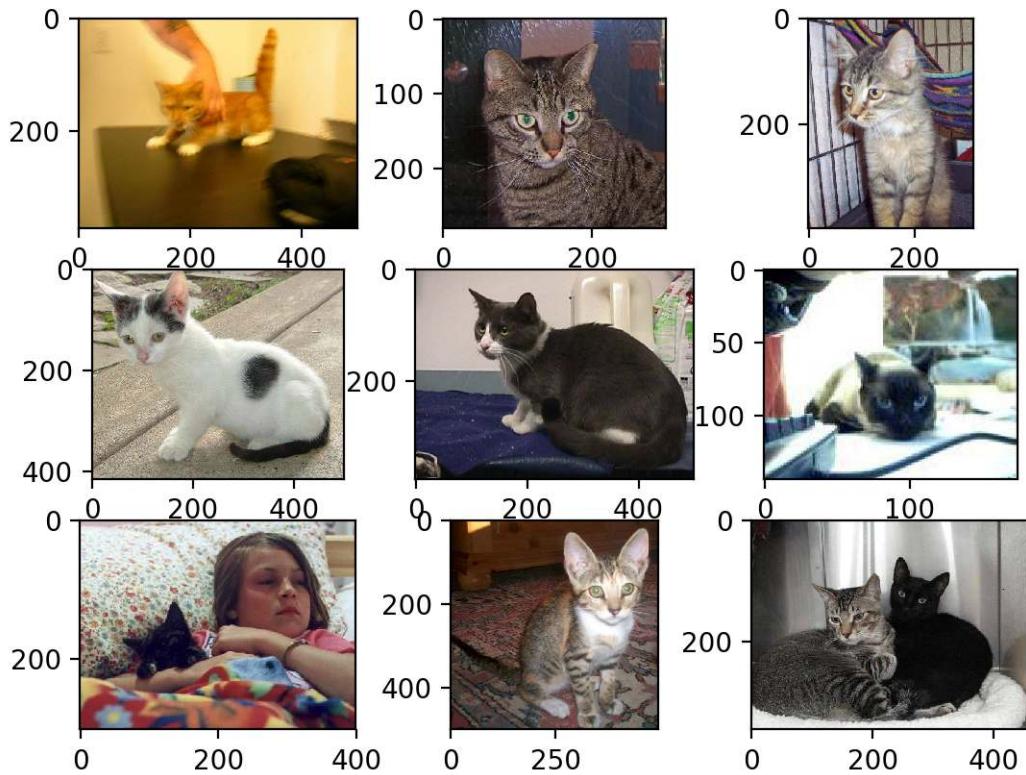


Figure 21.2: Plot of the First Nine Photos of Cats in the Dogs vs Cats Dataset.

21.3.2 Select Standardized Photo Size

The photos will have to be reshaped prior to modeling so that all images have the same shape. This is often a small square image. There are many ways to achieve this, although the most common is a simple resize operation that will stretch and deform the aspect ratio of each image and force it into the new shape. We could load all photos and look at the distribution of the photo widths and heights, then design a new photo size that best reflects what we are most likely to see in practice. Smaller inputs mean a model that is faster to train, and typically this concern dominates the choice of image size. In this case, we will follow this approach and choose a fixed size of 200×200 pixels.

21.3.3 Pre-Process Photo Sizes (Optional)

If we want to load all of the images into memory, we can estimate that it would require about 12 gigabytes of RAM. That is 25,000 images with $200 \times 200 \times 3$ pixels each, or 3,000,000,000 32-bit pixel values. We could load all of the images, reshape them, and store them as a single NumPy array. This could fit into RAM on many modern machines, but not all, especially if you only have 8 gigabytes to work with. We can write custom code to load the images into memory and resize them as part of the loading process, then save them ready for modeling. The

example below uses the Keras image processing API to load all 25,000 photos in the training dataset and reshapes them to 200×200 square photos. The label is also determined for each photo based on the filenames. A tuple of photos and labels is then saved.

```
# load dogs vs cats dataset, reshape and save to a new file
from os import listdir
from numpy import asarray
from numpy import save
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
# define location of dataset
folder = 'train/'
photos, labels = list(), list()
# enumerate files in the directory
for file in listdir(folder):
    # determine class
    output = 0.0
    if file.startswith('cat'):
        output = 1.0
    # load image
    photo = load_img(folder + file, target_size=(200, 200))
    # convert to numpy array
    photo = img_to_array(photo)
    # store
    photos.append(photo)
    labels.append(output)
# convert to numpy arrays
photos = asarray(photos)
labels = asarray(labels)
print(photos.shape, labels.shape)
# save the reshaped photos
save('dogs_vs_cats_photos.npy', photos)
save('dogs_vs_cats_labels.npy', labels)
```

Listing 21.4: Example of pre-processing all photos to the same size.

Running the example may take about one minute to load all of the images into memory and prints the shape of the loaded data to confirm it was loaded correctly. **Note:** running this example assumes you have more than 12 gigabytes of RAM. You can skip this example if you do not have sufficient RAM; it is only provided as a demonstration.

```
(25000, 200, 200, 3) (25000,)
```

Listing 21.5: Example output from pre-processing all photos to the same size.

At the end of the run, two files with the names `dogs_vs_cats_photos.npy` and `dogs_vs_cats_labels.npy` are created that contain all of the resized images and their associated class labels. The files are only about 12 gigabytes in size together and are significantly faster to load than the individual images. The prepared data can be loaded directly; for example:

```
# load and confirm the shape
from numpy import load
photos = load('dogs_vs_cats_photos.npy')
labels = load('dogs_vs_cats_labels.npy')
print(photos.shape, labels.shape)
```

Listing 21.6: Example of loading the pre-processed photos dataset.

21.3.4 Pre-Process Photos into Standard Directories

Alternately, we can load the images progressively using the Keras `ImageDataGenerator` class and `flow_from_directory()` API. This will be slower to execute but will require less RAM. This API prefers data to be divided into separate `train/` and `test/` directories, and under each directory to have a subdirectory for each class, e.g. a `train/dog/` and a `train/cat/` subdirectories and the same for test. Images are then organized under the subdirectories. We can write a script to create a copy of the dataset with this preferred structure. We will randomly select 25% of the images (or 6,250) to be used in a test dataset. First, we need to create the directory structure as follows:

```
dataset_dogs_vs_cats
├── test
│   ├── cats
│   └── dogs
└── train
    ├── cats
    └── dogs
```

Listing 21.7: Summary of the preferred directory structure.

We can create directories in Python using the `makedirs()` function and use a loop to create the `dog/` and `cat/` subdirectories for both the `train/` and `test/` directories.

```
...
# create directories
dataset_home = 'dataset_dogs_vs_cats/'
subdirs = ['train/', 'test/']
for subdir in subdirs:
    # create label subdirectories
    labeldirs = ['dogs/', 'cats/']
    for labldir in labeldirs:
        newdir = dataset_home + subdir + labldir
        makedirs(newdir, exist_ok=True)
```

Listing 21.8: Example of creating the required directories.

Next, we can enumerate all image files in the dataset and copy them into the `dogs/` or `cats/` subdirectory based on their filename. Additionally, we can randomly decide to hold back 25% of the images into the test dataset. This is done consistently by fixing the seed for the pseudorandom number generator so that we get the same split of data each time the code is run.

```
...
# seed random number generator
seed(1)
# define ratio of pictures to use for validation
val_ratio = 0.25
# copy training dataset images into subdirectories
src_directory = 'train/'
for file in.listdir(src_directory):
    src = src_directory + '/' + file
    dst_dir = 'train/'
    if random() < val_ratio:
        dst_dir = 'test/'
    if file.startswith('cat'):
        dst = dataset_home + dst_dir + 'cats/' + file
```

```

    copyfile(src, dst)
elif file.startswith('dog'):
    dst = dataset_home + dst_dir + 'dogs/' + file
    copyfile(src, dst)

```

Listing 21.9: Example of copying photographs into the desired directory structure.

The complete code example is listed below and assumes that you have the images in the downloaded `train.zip` unzipped in the current working directory in `train/`.

```

# organize dataset into a useful structure
from os import makedirs
from os import listdir
from shutil import copyfile
from random import seed
from random import random
# create directories
dataset_home = 'dataset_dogs_vs_cats/'
subdirs = ['train/', 'test/']
for subdir in subdirs:
    # create label subdirectories
    labeldirs = ['dogs/', 'cats/']
    for labldir in labeldirs:
        newdir = dataset_home + subdir + labldir
        makedirs(newdir, exist_ok=True)
# seed random number generator
seed(1)
# define ratio of pictures to use for validation
val_ratio = 0.25
# copy training dataset images into subdirectories
src_directory = 'train/'
for file in listdir(src_directory):
    src = src_directory + '/' + file
    dst_dir = 'train/'
    if random() < val_ratio:
        dst_dir = 'test/'
    if file.startswith('cat'):
        dst = dataset_home + dst_dir + 'cats/' + file
        copyfile(src, dst)
    elif file.startswith('dog'):
        dst = dataset_home + dst_dir + 'dogs/' + file
        copyfile(src, dst)

```

Listing 21.10: Example of restructuring the cats and dogs dataset into directories.

After running the example, you will now have a new `dataset_dogs_vs_cats/` directory with a `train/` and `val/` subfolders and further `dogs/` can `cats/` subdirectories, exactly as designed.

21.4 Develop a Baseline CNN Model

In this section, we can develop a baseline convolutional neural network model for the dogs vs. cats dataset. A baseline model will establish a minimum model performance to which all of our other models can be compared, as well as a model architecture that we can use as the basis of study and improvement. A good starting point is the general architectural principles of the

VGG models. These are a good starting point because they achieved top performance in the ILSVRC 2014 competition and because the modular structure of the architecture is easy to understand and implement. For more details on the VGG model, see the 2015 paper *Very Deep Convolutional Networks for Large-Scale Image Recognition*.

The architecture involves stacking convolutional layers with small 3×3 filters followed by a max pooling layer. Together, these layers form a block, and these blocks can be repeated where the number of filters in each block is increased with the depth of the network such as 32, 64, 128, 256 for the first four blocks of the model. Padding is used on the convolutional layers to ensure the height and width shapes of the output feature maps matches the inputs. We can explore this architecture on the dogs vs cats problem and compare a model with this architecture with 1, 2, and 3 blocks. Each layer will use the ReLU activation function and the He weight initialization, which are generally best practices. For example, a 3-block VGG-style architecture where each block has a single convolutional and pooling layer can be defined in Keras as follows:

```
...
# block 1
model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same', input_shape=(200, 200, 3)))
model.add(MaxPooling2D((2, 2)))
# block 2
model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(MaxPooling2D((2, 2)))
# block 3
model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(MaxPooling2D((2, 2)))
```

Listing 21.11: Example of defining a 3-block VGG-style feature extraction model.

We can create a function named `define_model()` that will define a model and return it ready to be fit on the dataset. This function can then be customized to define different baseline models, e.g. versions of the model with 1, 2, or 3 VGG style blocks. The model will be fit with stochastic gradient descent and we will start with a modest learning rate of 0.001 and a momentum of 0.9. The problem is a binary classification task, requiring the prediction of one value of either 0 or 1. An output layer with 1 node and a sigmoid activation will be used and the model will be optimized using the binary cross-entropy loss function. Below is an example of the `define_model()` function for defining a convolutional neural network model for the dogs vs. cats problem with one VGG-style block.

```
# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same', input_shape=(200, 200, 3)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(1, activation='sigmoid'))
# compile model
opt = SGD(lr=0.001, momentum=0.9)
model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
```

```
    return model
```

Listing 21.12: Example of a function for defining the baseline model.

It can be called to prepare a model as needed, for example:

```
...
# define model
model = define_model()
```

Listing 21.13: Example calling the function to define the model.

Next, we need to prepare the data. This involves first defining an instance of the `ImageDataGenerator` that will scale the pixel values to the range of 0-1.

```
...
# create data generator
datagen = ImageDataGenerator(rescale=1.0/255.0)
```

Listing 21.14: Example preparing the image data generator with pixel normalization.

Next, iterators need to be prepared for both the train and test datasets. We can use the `flow_from_directory()` function on the data generator and create one iterator for each of the `train/` and `test/` directories. We must specify that the problem is a binary classification problem via the `class_mode` argument, and to load the images with the size of 200×200 pixels via the `target_size` argument. We will fix the batch size at 64.

```
...
# prepare iterators
train_it = datagen.flow_from_directory('dataset_dogs_vs_cats/train/',
    class_mode='binary', batch_size=64, target_size=(200, 200))
test_it = datagen.flow_from_directory('dataset_dogs_vs_cats/test/',
    class_mode='binary', batch_size=64, target_size=(200, 200))
```

Listing 21.15: Example preparing the image dataset iterators.

We can then fit the model using the train iterator (`train_it`) and use the test iterator (`test_it`) as a validation dataset during training. The number of steps for the train and test iterators must be specified. This is the number of batches that will comprise one epoch. This can be specified via the length of each iterator, and will be the total number of images in the train and test directories divided by the batch size (64). The model will be fit for 20 epochs, a small number to check if the model can learn the problem.

```
...
# fit model
history = model.fit_generator(train_it, steps_per_epoch=len(train_it),
    validation_data=test_it, validation_steps=len(test_it), epochs=20, verbose=0)
```

Listing 21.16: Example fitting the model with the image dataset iterators.

Once fit, the final model can be evaluated on the test dataset directly and the classification accuracy reported.

```
...
# evaluate model
_, acc = model.evaluate_generator(test_it, steps=len(test_it), verbose=0)
print('> %.3f' % (acc * 100.0))
```

Listing 21.17: Example evaluating the model with the image dataset iterators.

Finally, we can create a plot of the history collected during training stored in the `history` variable returned from the call to `fit_generator()`. The History contains the model accuracy and loss on the test and training dataset at the end of each epoch. Line plots of these measures over training epochs provide learning curves that we can use to get an idea of whether the model is overfitting, underfitting, or has a good fit. The `summarize_diagnostics()` function below takes the history directory and creates a single figure with a line plot of the loss and another for the accuracy. The figure is then saved to file with a filename based on the name of the script. This is helpful if we wish to evaluate many variations of the model in different files and create line plots automatically for each.

```
# plot diagnostic learning curves
def summarize_diagnostics(history):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(history.history['loss'], color='blue', label='train')
    pyplot.plot(history.history['val_loss'], color='orange', label='test')
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Classification Accuracy')
    pyplot.plot(history.history['accuracy'], color='blue', label='train')
    pyplot.plot(history.history['val_accuracy'], color='orange', label='test')
    # save plot to file
    filename = sys.argv[0].split('/')[-1]
    pyplot.savefig(filename + '_plot.png')
    pyplot.close()
```

Listing 21.18: Example of a function for plotting learning curves of model performance.

We can tie all of this together into a simple test harness for testing a model configuration. The complete example of evaluating a one-block baseline model on the dogs and cats dataset is listed below.

```
# 1-vgg block baseline model for the dogs vs cats dataset
import sys
from matplotlib import pyplot
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD
from keras.preprocessing.image import ImageDataGenerator

# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same', input_shape=(200, 200, 3)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(1, activation='sigmoid'))
    # compile model
    opt = SGD(lr=0.001, momentum=0.9)
```

```

model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
return model

# plot diagnostic learning curves
def summarize_diagnostics(history):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(history.history['loss'], color='blue', label='train')
    pyplot.plot(history.history['val_loss'], color='orange', label='test')
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Classification Accuracy')
    pyplot.plot(history.history['accuracy'], color='blue', label='train')
    pyplot.plot(history.history['val_accuracy'], color='orange', label='test')
    # save plot to file
    filename = sys.argv[0].split('/')[-1]
    pyplot.savefig(filename + '_plot.png')
    pyplot.close()

# run the test harness for evaluating a model
def run_test_harness():
    # define model
    model = define_model()
    # create data generator
    datagen = ImageDataGenerator(rescale=1.0/255.0)
    # prepare iterators
    train_it = datagen.flow_from_directory('dataset_dogs_vs_cats/train/',
        class_mode='binary', batch_size=64, target_size=(200, 200))
    test_it = datagen.flow_from_directory('dataset_dogs_vs_cats/test/',
        class_mode='binary', batch_size=64, target_size=(200, 200))
    # fit model
    history = model.fit_generator(train_it, steps_per_epoch=len(train_it),
        validation_data=test_it, validation_steps=len(test_it), epochs=20, verbose=0)
    # evaluate model
    _, acc = model.evaluate_generator(test_it, steps=len(test_it), verbose=0)
    print('> %.3f' % (acc * 100.0))
    # learning curves
    summarize_diagnostics(history)

# entry point, run the test harness
run_test_harness()

```

Listing 21.19: Example of a baseline model for the dogs and cats dataset.

Now that we have a test harness, let's look at the evaluation of three simple baseline models.

21.4.1 One Block VGG Model

The one-block VGG model has a single convolutional layer with 32 filters followed by a max pooling layer. The `define_model()` function for this model was defined in the previous section but is provided again below for completeness.

```

# define cnn model
def define_model():
    model = Sequential()

```

```
model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same', input_shape=(200, 200, 3)))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='sigmoid'))
# compile model
opt = SGD(lr=0.001, momentum=0.9)
model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
return model
```

Listing 21.20: Example of a function for defining a 1-VGG block baseline model.

Running this example first prints the size of the train and test datasets, confirming that the dataset was loaded correctly. The model is then fit and evaluated, which takes approximately 20 minutes on modern GPU hardware.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved an accuracy of about 72% on the test dataset.

```
Found 18697 images belonging to 2 classes.
Found 6303 images belonging to 2 classes.
> 72.331
```

Listing 21.21: Example output from evaluating a 1-VGG block baseline model.

A figure is also created showing a line plot for the loss and another for the accuracy of the model on both the train (blue) and test (orange) datasets. Reviewing this plot, we can see that the model has overfit the training dataset at about 12 epochs.

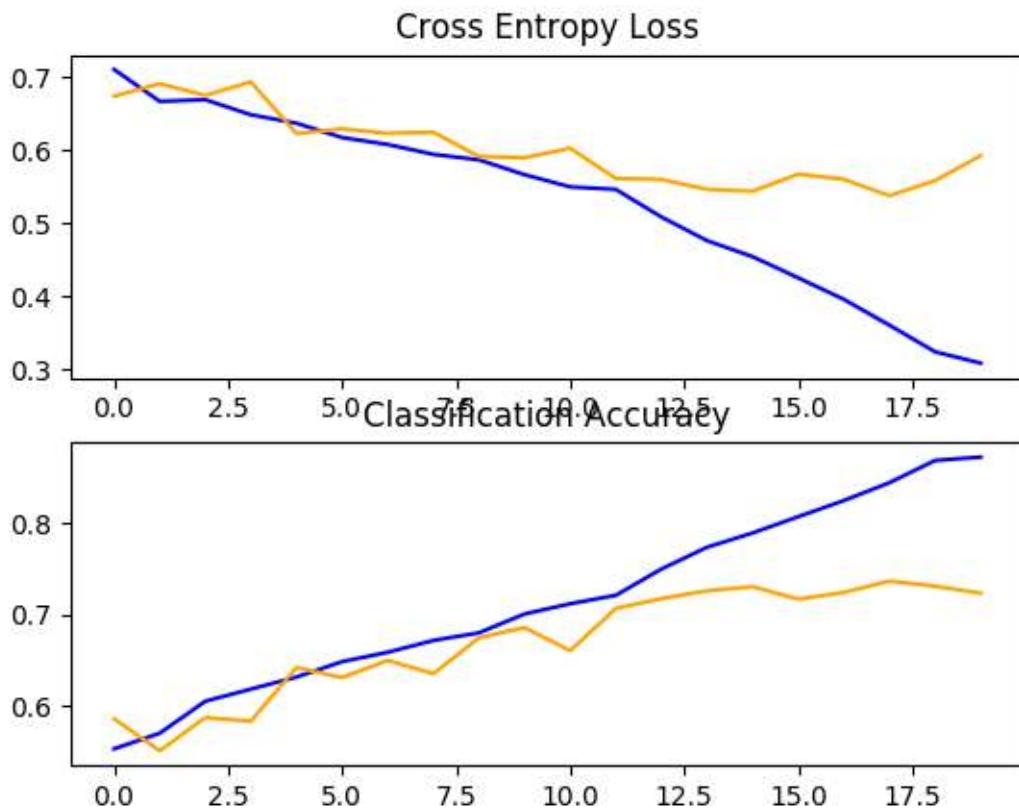


Figure 21.3: Line Plots of Loss and Accuracy Learning Curves for the Baseline Model With One VGG Block on the Dogs and Cats Dataset.

21.4.2 Two Block VGG Model

The two-block VGG model extends the one block model and adds a second block with 64 filters. The `define_model()` function for this model is provided below for completeness.

```
# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same', input_shape=(200, 200, 3)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(1, activation='sigmoid'))
    # compile model
    opt = SGD(lr=0.001, momentum=0.9)
    model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

Listing 21.22: Example of a function for defining a 2-VGG block baseline model.

Running this example again prints the size of the train and test datasets, confirming that the dataset was loaded correctly. The model is fit and evaluated and the performance on the test dataset is reported.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved a small improvement in performance from about 72% with one block to about 76% accuracy with two blocks.

```
Found 18697 images belonging to 2 classes.
Found 6303 images belonging to 2 classes.
> 76.646
```

Listing 21.23: Example output from evaluating a 2-VGG block baseline model.

Reviewing the plot of the learning curves, we can see that again the model appears to have overfit the training dataset, perhaps sooner, in this case at around eight training epochs. This is likely the result of the increased capacity of the model, and we might expect this trend of sooner overfitting to continue with the next model.

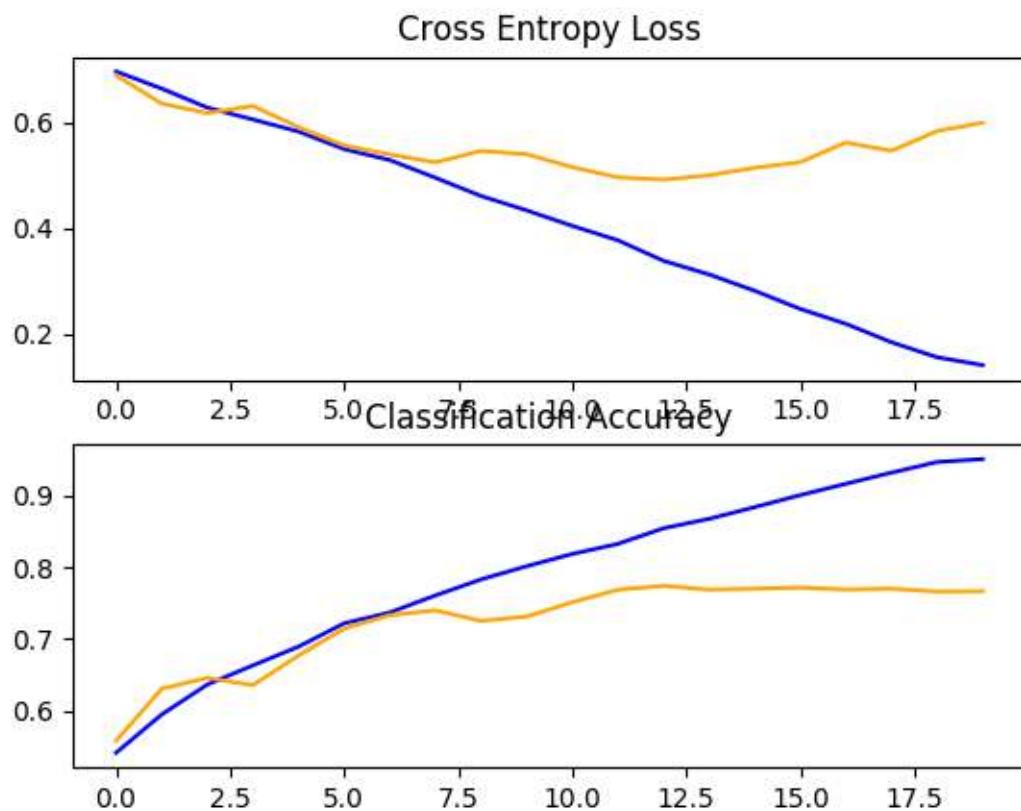


Figure 21.4: Line Plots of Loss and Accuracy Learning Curves for the Baseline Model With Two VGG Blocks on the Dogs and Cats Dataset.

21.4.3 Three Block VGG Model

The three-block VGG model extends the two block model and adds a third block with 128 filters. The `define_model()` function for this model was defined in the previous section but is provided again below for completeness.

```
# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same', input_shape=(200, 200, 3)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(1, activation='sigmoid'))
    # compile model
    opt = SGD(lr=0.001, momentum=0.9)
    model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

Listing 21.24: Example of a function for defining a 3-VGG block baseline model.

Running this example prints the size of the train and test datasets, confirming that the dataset was loaded correctly. The model is fit and evaluated and the performance on the test dataset is reported.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that we achieved a further lift in performance from about 76% with two blocks to about 80% accuracy with three blocks. This result is good, as it is close to the prior state-of-the-art reported in the paper using an SVM at about 82% accuracy.

```
Found 18697 images belonging to 2 classes.
Found 6303 images belonging to 2 classes.
> 80.184
```

Listing 21.25: Example output from evaluating a 3-VGG block baseline model.

Reviewing the plot of the learning curves, we can see a similar trend of overfitting, in this case perhaps pushed back as far as to epoch five or six.

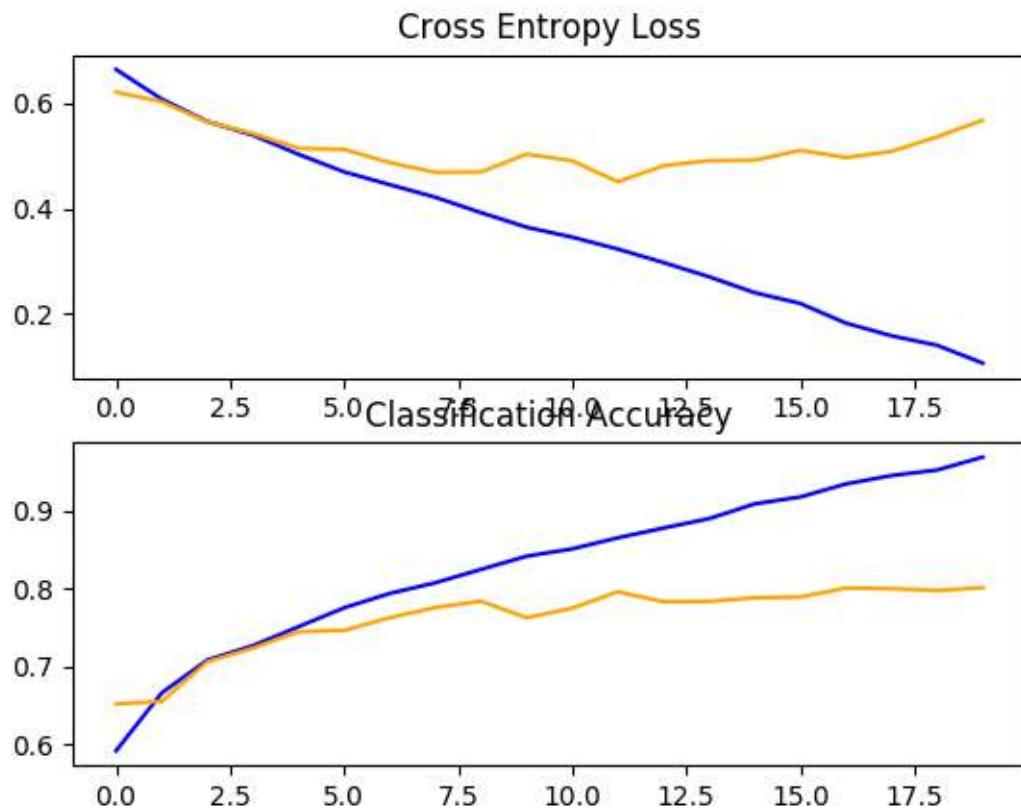


Figure 21.5: Line Plots of Loss and Accuracy Learning Curves for the Baseline Model With Three VGG Block on the Dogs and Cats Dataset.

21.4.4 Discussion

We have explored three different models with a VGG-based architecture. The results can be summarized below, although we must assume some variance in these results given the stochastic nature of the algorithm:

- **Baseline VGG 1:** 72.331%.
- **Baseline VGG 2:** 76.646%.
- **Baseline VGG 3:** 80.184%.

We see a trend of improved performance with the increase in capacity, but also a similar case of overfitting occurring earlier and earlier in the run. The results suggest that the model will likely benefit from regularization techniques. This may include techniques such as dropout, weight decay, and data augmentation. The latter can also boost performance by encouraging the model to learn features that are further invariant to position by expanding the training dataset.

21.5 Develop Model Improvements

In the previous section, we developed a baseline model using VGG-style blocks and discovered a trend of improved performance with increased model capacity. In this section, we will start with the baseline model with three VGG blocks (i.e. VGG 3) and explore some simple improvements to the model. From reviewing the learning curves for the model during training, the model showed strong signs of overfitting. We can explore two approaches to attempt to address this overfitting: dropout regularization and data augmentation. Both of these approaches are expected to slow the rate of improvement during training and hopefully counter the overfitting of the training dataset. As such, we will increase the number of training epochs from 20 to 50 to give the model more space for refinement.

21.5.1 Dropout Regularization

Dropout regularization is a computationally cheap way to regularize a deep neural network. Dropout works by probabilistically removing, or *dropping out*, inputs to a layer, which may be input variables in the data sample or activations from a previous layer. It has the effect of simulating a large number of networks with very different network structures and, in turn, making nodes in the network generally more robust to the inputs.

Typically, a small amount of dropout can be applied after each VGG block, with more dropout applied to the fully connected layers near the output layer of the model. Below is the `define_model()` function for an updated version of the baseline model with the addition of Dropout. In this case, a dropout of 20% is applied after each VGG block, with a larger dropout rate of 50% applied after the fully connected layer in the classifier part of the model.

```
# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same', input_shape=(200, 200, 3)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dropout(0.5))
    model.add(Dense(1, activation='sigmoid'))
    # compile model
    opt = SGD(lr=0.001, momentum=0.9)
    model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

Listing 21.26: Example of a function for defining a 3-VGG block baseline model with dropout.

The full code listing of the baseline model with the addition of dropout on the dogs vs. cats dataset is listed below for completeness.

```
# baseline model with dropout for the dogs vs cats dataset
import sys
from matplotlib import pyplot
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.optimizers import SGD
from keras.preprocessing.image import ImageDataGenerator

# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same', input_shape=(200, 200, 3)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dropout(0.5))
    model.add(Dense(1, activation='sigmoid'))
    # compile model
    opt = SGD(lr=0.001, momentum=0.9)
    model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
    return model

# plot diagnostic learning curves
def summarize_diagnostics(history):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(history.history['loss'], color='blue', label='train')
    pyplot.plot(history.history['val_loss'], color='orange', label='test')
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Classification Accuracy')
    pyplot.plot(history.history['accuracy'], color='blue', label='train')
    pyplot.plot(history.history['val_accuracy'], color='orange', label='test')
    # save plot to file
    filename = sys.argv[0].split('/')[-1]
    pyplot.savefig(filename + '_plot.png')
    pyplot.close()
```

```
# run the test harness for evaluating a model
def run_test_harness():
    # define model
    model = define_model()
    # create data generator
    datagen = ImageDataGenerator(rescale=1.0/255.0)
    # prepare iterator
    train_it = datagen.flow_from_directory('dataset_dogs_vs_cats/train/',
        class_mode='binary', batch_size=64, target_size=(200, 200))
    test_it = datagen.flow_from_directory('dataset_dogs_vs_cats/test/',
        class_mode='binary', batch_size=64, target_size=(200, 200))
    # fit model
    history = model.fit_generator(train_it, steps_per_epoch=len(train_it),
        validation_data=test_it, validation_steps=len(test_it), epochs=50, verbose=0)
    # evaluate model
    _, acc = model.evaluate_generator(test_it, steps=len(test_it), verbose=0)
    print('> %.3f' % (acc * 100.0))
    # learning curves
    summarize_diagnostics(history)

# entry point, run the test harness
run_test_harness()
```

Listing 21.27: Example of evaluating the 3-VGG block model with dropout for the dogs and cats dataset.

Running the example first fits the model, then reports the model performance on the hold out test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see a small lift in model performance from about 80% accuracy for the baseline model to about 81% with the addition of dropout.

```
Found 18697 images belonging to 2 classes.
Found 6303 images belonging to 2 classes.
> 81.279
```

Listing 21.28: Example output from evaluating the 3-VGG block model with dropout for the dogs and cats dataset.

Reviewing the learning curves, we can see that dropout has had an effect on the rate of improvement of the model on both the train and test sets. Overfitting has been reduced or delayed, although performance may begin to stall towards the end of the run. The results suggest that further training epochs may result in further improvement of the model. It may also be interesting to explore perhaps a slightly higher dropout rate after the VGG blocks in addition to the increase in training epochs.

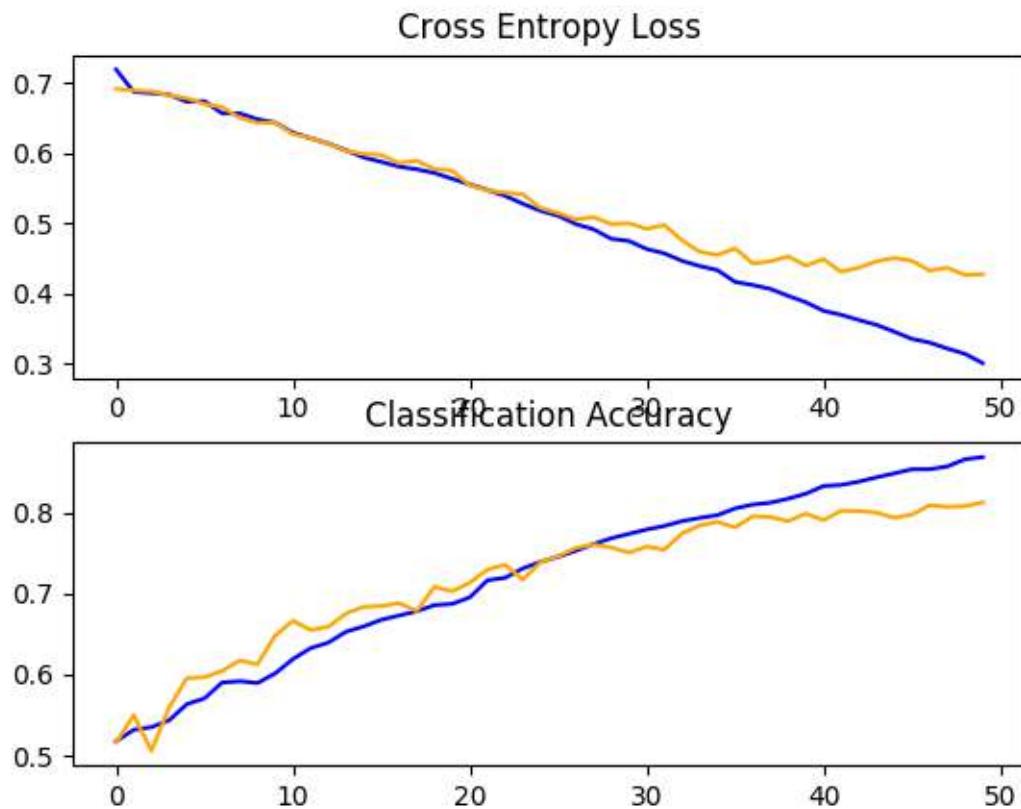


Figure 21.6: Line Plots of Loss and Accuracy Learning Curves for the Baseline Model With Dropout on the Dogs and Cats Dataset.

21.5.2 Image Data Augmentation

Image data augmentation is a technique that can be used to artificially expand the size of a training dataset by creating modified versions of images in the dataset. Training deep learning neural network models on more data can result in more skillful models, and the augmentation techniques can create variations of the images that can improve the ability of the fit models to generalize what they have learned to new images. Data augmentation can also act as a regularization technique, adding noise to the training data, and encouraging the model to learn the same features, invariant to their position in the input (data augmentation was introduced in Chapter 9).

Small changes to the input photos of dogs and cats might be useful for this problem, such as small shifts and horizontal flips. These augmentations can be specified as arguments to the `ImageDataGenerator` used for the training dataset. The augmentations should not be used for the test dataset, as we wish to evaluate the performance of the model on the unmodified photographs. This requires that we have a separate `ImageDataGenerator` instance for the train and test dataset, then iterators for the train and test sets created from the respective data generators. For example:

```
# create data generators
```

```

train_datagen = ImageDataGenerator(rescale=1.0/255.0,
    width_shift_range=0.1, height_shift_range=0.1, horizontal_flip=True)
test_datagen = ImageDataGenerator(rescale=1.0/255.0)
# prepare iterators
train_it = train_datagen.flow_from_directory('dataset_dogs_vs_cats/train/',
    class_mode='binary', batch_size=64, target_size=(200, 200))
test_it = test_datagen.flow_from_directory('dataset_dogs_vs_cats/test/',
    class_mode='binary', batch_size=64, target_size=(200, 200))

```

Listing 21.29: Example defining the image data generators for augmentation.

In this case, photos in the training dataset will be augmented with small (10%) random horizontal and vertical shifts and random horizontal flips that create a mirror image of a photo. Photos in both the train and test steps will have their pixel values scaled in the same way. The full code listing of the baseline model with training data augmentation for the dogs and cats dataset is listed below for completeness.

```

# baseline model with data augmentation for the dogs vs cats dataset
import sys
from matplotlib import pyplot
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD
from keras.preprocessing.image import ImageDataGenerator

# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same', input_shape=(200, 200, 3)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(1, activation='sigmoid'))
    # compile model
    opt = SGD(lr=0.001, momentum=0.9)
    model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
    return model

# plot diagnostic learning curves
def summarize_diagnostics(history):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(history.history['loss'], color='blue', label='train')
    pyplot.plot(history.history['val_loss'], color='orange', label='test')
    # plot accuracy

```

```

pyplot.subplot(212)
pyplot.title('Classification Accuracy')
pyplot.plot(history.history['accuracy'], color='blue', label='train')
pyplot.plot(history.history['val_accuracy'], color='orange', label='test')
# save plot to file
filename = sys.argv[0].split('/')[-1]
pyplot.savefig(filename + '_plot.png')
pyplot.close()

# run the test harness for evaluating a model
def run_test_harness():
    # define model
    model = define_model()
    # create data generators
    train_datagen = ImageDataGenerator(rescale=1.0/255.0,
        width_shift_range=0.1, height_shift_range=0.1, horizontal_flip=True)
    test_datagen = ImageDataGenerator(rescale=1.0/255.0)
    # prepare iterators
    train_it = train_datagen.flow_from_directory('dataset_dogs_vs_cats/train/',
        class_mode='binary', batch_size=64, target_size=(200, 200))
    test_it = test_datagen.flow_from_directory('dataset_dogs_vs_cats/test/',
        class_mode='binary', batch_size=64, target_size=(200, 200))
    # fit model
    history = model.fit_generator(train_it, steps_per_epoch=len(train_it),
        validation_data=test_it, validation_steps=len(test_it), epochs=50, verbose=0)
    # evaluate model
    _, acc = model.evaluate_generator(test_it, steps=len(test_it), verbose=0)
    print('> %.3f' % (acc * 100.0))
    # learning curves
    summarize_diagnostics(history)

# entry point, run the test harness
run_test_harness()

```

Listing 21.30: Example of evaluating the 3-VGG block model with data augmentation for the dogs and cats dataset.

Running the example first fits the model, then reports the model performance on the hold out test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see a lift in performance of about 5% from about 80% for the baseline model to about 86% for the baseline model with simple data augmentation.

```
> 85.816
```

Listing 21.31: Example output from evaluating the 3-VGG block model with data augmentation for the dogs and cats dataset.

Reviewing the learning curves, we can see that it appears the model is capable of further learning with both the loss on the train and test dataset still decreasing even at the end of the run. Repeating the experiment with 100 or more epochs will very likely result in a better performing model. It may be interesting to explore other augmentations that may further

encourage the learning of features invariant to their position in the input, such as minor rotations and zooms.

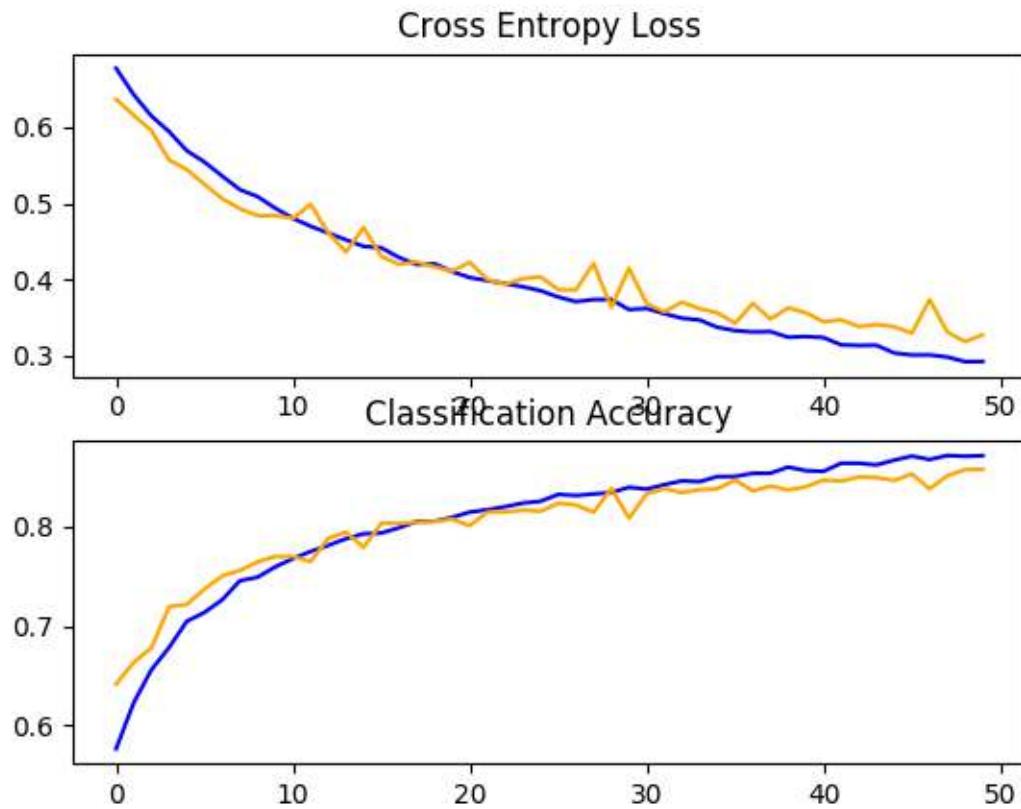


Figure 21.7: Line Plots of Loss and Accuracy Learning Curves for the Baseline Model With Data Augmentation on the Dogs and Cats Dataset.

21.5.3 Discussion

We have explored three different improvements to the baseline model. The results can be summarized below, although we must assume some variance in these results given the stochastic nature of the algorithm:

- **Baseline VGG3 + Dropout:** 81.279%.
- **Baseline VGG3 + Data Augmentation:** 85.816%.

As suspected, the addition of regularization techniques slows the progression of the learning algorithms and reduces overfitting, resulting in improved performance on the holdout dataset. It is likely that the combination of both approaches with further increase in the number of training epochs will result in further improvements. This is just the beginning of the types of improvements that can be explored on this dataset. In addition to tweaks to the regularization methods described, other regularization methods could be explored such as weight decay and early stopping.

It may be worth exploring changes to the learning algorithm such as changes to the learning rate, use of a learning rate schedule, or an adaptive learning rate such as Adam. Alternate model architectures may also be worth exploring. The chosen baseline model is expected to offer more capacity than may be required for this problem and a smaller model may faster to train and in turn could result in better performance.

21.6 Explore Transfer Learning

Transfer learning involves using all or parts of a model trained on a related task. Keras provides a range of pre-trained models that can be loaded and used wholly or partially via the Keras Applications API. A useful model for transfer learning is one of the VGG models, such as VGG-16 with 16 layers that at the time it was developed, achieved top results on the ImageNet photo classification challenge. The model is comprised of two main parts, the feature extractor part of the model that is made up of VGG blocks, and the classifier part of the model that is made up of fully connected layers and the output layer.

We can use the feature extraction part of the model and add a new classifier part of the model that is tailored to the dogs and cats dataset. Specifically, we can hold the weights of all of the convolutional layers fixed during training, and only train new fully connected layers that will learn to interpret the features extracted from the model and make a binary classification. This can be achieved by loading the VGG-16 model, removing the fully connected layers from the output-end of the model, then adding the new fully connected layers to interpret the model output and make a prediction. The classifier part of the model can be removed automatically by setting the `include_top` argument to `False`, which also requires that the shape of the input also be specified for the model, in this case `(224, 224, 3)`. This means that the loaded model ends at the last max pooling layer, after which we can manually add a `Flatten` layer and the new classifier layers. The `define_model()` function below implements this and returns a new model ready for training.

```
# define cnn model
def define_model():
    # load model
    model = VGG16(include_top=False, input_shape=(224, 224, 3))
    # mark loaded layers as not trainable
    for layer in model.layers:
        layer.trainable = False
    # add new classifier layers
    flat1 = Flatten()(model.layers[-1].output)
    class1 = Dense(128, activation='relu', kernel_initializer='he_uniform')(flat1)
    output = Dense(1, activation='sigmoid')(class1)
    # define new model
    model = Model(inputs=model.inputs, outputs=output)
    # compile model
    opt = SGD(lr=0.001, momentum=0.9)
    model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

Listing 21.32: Example of a function for defining a pre-trained model.

Once created, we can train the model as before on the training dataset. Not a lot of training will be required in this case, as only the new fully connected and output layer have trainable

weights. As such, we will fix the number of training epochs at 10. The VGG16 model was trained on a specific ImageNet challenge dataset. As such, it is configured to expected input images to have the shape 224×224 pixels. We will use this as the target size when loading photos from the dogs and cats dataset.

The model also expects images to be centered. That is, to have the mean pixel values from each channel (red, green, and blue) as calculated on the ImageNet training dataset subtracted from the input. Keras provides a function to perform this preparation for individual photos via the `preprocess_input()` function. Nevertheless, we can achieve the same effect with the `ImageDataGenerator` by setting the `featurewise_center` argument to `True` and manually specifying the mean pixel values to use when centering as the mean values from the ImageNet training dataset: [123.68, 116.779, 103.939]. The full code listing of the VGG model for transfer learning on the dogs vs. cats dataset is listed below.

```
# vgg16 model used for transfer learning on the dogs and cats dataset
import sys
from matplotlib import pyplot
from keras.applications.vgg16 import VGG16
from keras.models import Model
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD
from keras.preprocessing.image import ImageDataGenerator

# define cnn model
def define_model():
    # load model
    model = VGG16(include_top=False, input_shape=(224, 224, 3))
    # mark loaded layers as not trainable
    for layer in model.layers:
        layer.trainable = False
    # add new classifier layers
    flat1 = Flatten()(model.layers[-1].output)
    class1 = Dense(128, activation='relu', kernel_initializer='he_uniform')(flat1)
    output = Dense(1, activation='sigmoid')(class1)
    # define new model
    model = Model(inputs=model.inputs, outputs=output)
    # compile model
    opt = SGD(lr=0.001, momentum=0.9)
    model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
    return model

# plot diagnostic learning curves
def summarize_diagnostics(history):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(history.history['loss'], color='blue', label='train')
    pyplot.plot(history.history['val_loss'], color='orange', label='test')
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Classification Accuracy')
    pyplot.plot(history.history['accuracy'], color='blue', label='train')
    pyplot.plot(history.history['val_accuracy'], color='orange', label='test')
    # save plot to file
```

```

filename = sys.argv[0].split('/')[-1]
pyplot.savefig(filename + '_plot.png')
pyplot.close()

# run the test harness for evaluating a model
def run_test_harness():
    # define model
    model = define_model()
    # create data generator
    datagen = ImageDataGenerator(featurewise_center=True)
    # specify imagenet mean values for centering
    datagen.mean = [123.68, 116.779, 103.939]
    # prepare iterator
    train_it = datagen.flow_from_directory('dataset_dogs_vs_cats/train/',
        class_mode='binary', batch_size=64, target_size=(224, 224))
    test_it = datagen.flow_from_directory('dataset_dogs_vs_cats/test/',
        class_mode='binary', batch_size=64, target_size=(224, 224))
    # fit model
    history = model.fit_generator(train_it, steps_per_epoch=len(train_it),
        validation_data=test_it, validation_steps=len(test_it), epochs=10, verbose=1)
    # evaluate model
    _, acc = model.evaluate_generator(test_it, steps=len(test_it), verbose=0)
    print('> %.3f' % (acc * 100.0))
    # learning curves
    summarize_diagnostics(history)

# entry point, run the test harness
run_test_harness()

```

Listing 21.33: Example of evaluating a pre-trained model on the dogs and cats dataset.

Running the example first fits the model, then reports the model performance on the hold out test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved very impressive results with a classification accuracy of about 97% on the holdout test dataset.

```

Found 18697 images belonging to 2 classes.
Found 6303 images belonging to 2 classes.
> 97.636

```

Listing 21.34: Example output from evaluating a pre-trained model on the dogs and cats dataset.

Reviewing the learning curves, we can see that the model fits the dataset quickly. It does not show strong overfitting, although the results suggest that perhaps additional capacity in the classifier and/or the use of regularization might be helpful. There are many improvements that could be made to this approach, including adding dropout regularization to the classifier part of the model and perhaps even fine-tuning the weights of some or all of the layers in the feature detector part of the model.

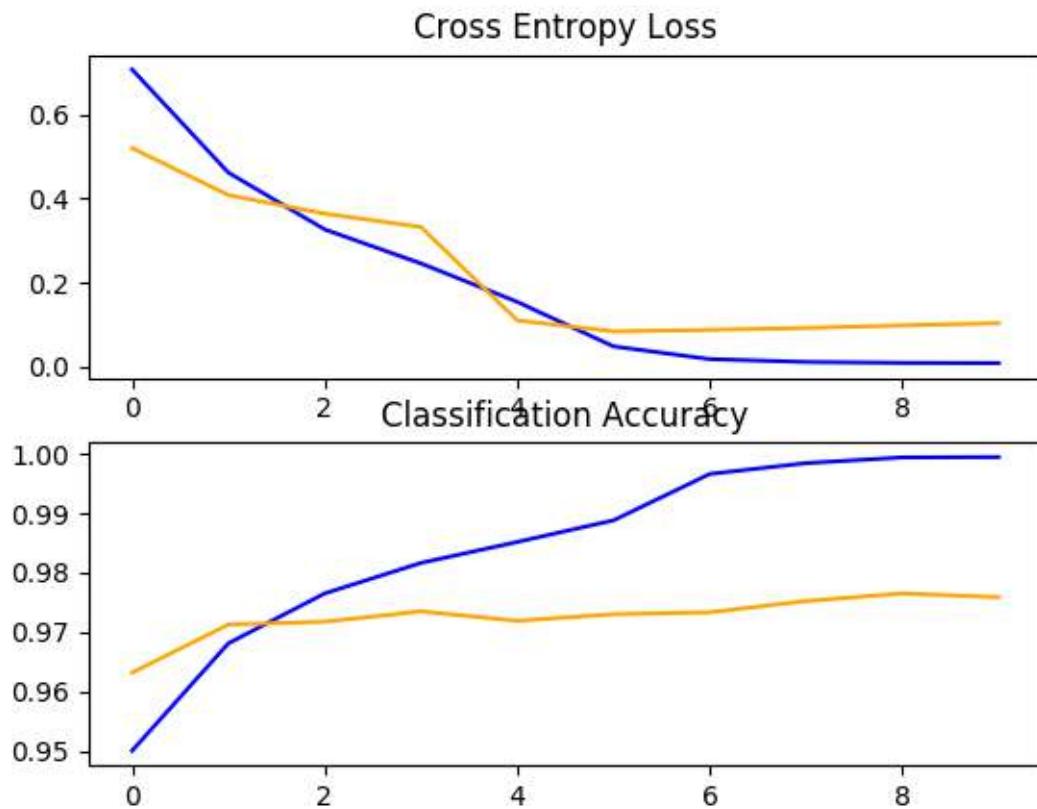


Figure 21.8: Line Plots of Loss and Accuracy Learning Curves for the VGG16 Transfer Learning Model on the Dogs and Cats Dataset.

21.7 How to Finalize the Model and Make Predictions

The process of model improvement may continue for as long as we have ideas and the time and resources to test them out. At some point, a final model configuration must be chosen and adopted. In this case, we will keep things simple and use the VGG-16 transfer learning approach as the final model. First, we will finalize our model by fitting a model on the entire training dataset and saving the model to file for later use. We will then load the saved model and use it to make a prediction on a single image.

21.7.1 Prepare Final Dataset

A final model is typically fit on all available data, such as the combination of all train and test datasets. In this tutorial, we will demonstrate the final model fit only on the training dataset as we only have labels for the training dataset. The first step is to prepare the training dataset so that it can be loaded by the `ImageDataGenerator` class via the `flow_from_directory()` function. Specifically, we need to create a new directory with all training images organized into `dogs/` and `cats/` subdirectories without any separation into `train/` or `test/` directories. This can be achieved by updating the script we developed at the beginning of the tutorial. In this

case, we will create a new `finalize_dogs_vs_cats/` folder with `dogs/` and `cats/` subfolders for the entire training dataset. The structure will look as follows:

```
finalize_dogs_vs_cats
    cats
    dogs
```

Listing 21.35: Example of the desired structure for the final training dataset.

The updated script is listed below for completeness.

```
# organize dataset into a useful structure
from os import makedirs
from os import listdir
from shutil import copyfile
# create directories
dataset_home = 'finalize_dogs_vs_cats/'
# create label subdirectories
labldirs = ['dogs/', 'cats/']
for labldir in labldirs:
    newdir = dataset_home + labldir
    makedirs(newdir, exist_ok=True)
# copy training dataset images into subdirectories
src_directory = 'dogs-vs-cats/train/'
for file in listdir(src_directory):
    src = src_directory + '/' + file
    if file.startswith('cat'):
        dst = dataset_home + 'cats/' + file
        copyfile(src, dst)
    elif file.startswith('dog'):
        dst = dataset_home + 'dogs/' + file
        copyfile(src, dst)
```

Listing 21.36: Example preparing the dataset for training the final model.

21.7.2 Save Final Model

We are now ready to fit a final model on the entire training dataset. The `flow_from_directory()` must be updated to load all of the images from the new `finalize_dogs_vs_cats/` directory.

```
...
# prepare iterator
train_it = datagen.flow_from_directory('finalize_dogs_vs_cats/',
    class_mode='binary', batch_size=64, target_size=(224, 224))
```

Listing 21.37: Example preparing the image dataset iterator for training the final model.

Additionally, the call to `fit_generator()` no longer needs to specify a validation dataset.

```
...
# fit model
model.fit_generator(train_it, steps_per_epoch=len(train_it), epochs=10, verbose=0)
```

Listing 21.38: Example fitting the final model.

Once fit, we can save the final model to an H5 file by calling the `save()` function on the model and pass in the chosen filename.

```
...
# save model
model.save('final_model.h5')
```

Listing 21.39: Example saving the final model.

Note, saving and loading a Keras model requires that the `h5py` library is installed on your workstation. The complete example of fitting the final model on the training dataset and saving it to file is listed below.

```
# save the final model to file
from keras.applications.vgg16 import VGG16
from keras.models import Model
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD
from keras.preprocessing.image import ImageDataGenerator

# define cnn model
def define_model():
    # load model
    model = VGG16(include_top=False, input_shape=(224, 224, 3))
    # mark loaded layers as not trainable
    for layer in model.layers:
        layer.trainable = False
    # add new classifier layers
    flat1 = Flatten()(model.layers[-1].output)
    class1 = Dense(128, activation='relu', kernel_initializer='he_uniform')(flat1)
    output = Dense(1, activation='sigmoid')(class1)
    # define new model
    model = Model(inputs=model.inputs, outputs=output)
    # compile model
    opt = SGD(lr=0.001, momentum=0.9)
    model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
    return model

# run the test harness for evaluating a model
def run_test_harness():
    # define model
    model = define_model()
    # create data generator
    datagen = ImageDataGenerator(featurewise_center=True)
    # specify imagenet mean values for centering
    datagen.mean = [123.68, 116.779, 103.939]
    # prepare iterator
    train_it = datagen.flow_from_directory('finalize_dogs_vs_cats/',
                                           class_mode='binary', batch_size=64, target_size=(224, 224))
    # fit model
    model.fit_generator(train_it, steps_per_epoch=len(train_it), epochs=10, verbose=0)
    # save model
    model.save('final_model.h5')

    # entry point, run the test harness
run_test_harness()
```

Listing 21.40: Example fitting and saving the final model.

After running this example, you will now have a large 81-megabyte file with the name `final_model.h5` in your current working directory.

21.7.3 Make Prediction

We can use our saved model to make a prediction on new images. The model assumes that new images are color and they have been segmented so that one image contains at least one dog or cat. Below is an image extracted from the test dataset for the dogs and cats competition. It has no label, but we can clearly tell it is a photo of a dog.



Figure 21.9: Photograph of a Dog.

You can save it in your current working directory with the filename `sample_image.jpg`.

- Download Dog Photograph (`sample_image.jpg`).²

We will pretend this is an entirely new and unseen image, prepared in the required way, and see how we might use our saved model to predict the integer that the image represents. For this example, we expect class 1 for Dog. Note: the subdirectories of images, one for each class, are loaded by the `flow_from_directory()` function in alphabetical order and assigned an integer for each class. The subdirectory `cat` comes before `dog`, therefore the class labels are assigned the integers: `cat=0`, `dog=1`. This can be changed via the `classes` argument in calling `flow_from_directory()` when training the model. First, we can load the image and force it to the size to be 224×224 pixels. The loaded image can then be resized to have a single sample in a dataset. The pixel values must also be centered to match the way that the data was prepared during the training of the model. The `load_image()` function implements this and will return the loaded image ready for classification.

```
# load and prepare the image
def load_image(filename):
    # load the image
```

²https://machinelearningmastery.com/wp-content/uploads/2019/03/sample_image.jpg

```

img = load_img(filename, target_size=(224, 224))
# convert to array
img = img_to_array(img)
# reshape into a single sample with 3 channels
img = img.reshape(1, 224, 224, 3)
# center pixel data
img = img.astype('float32')
img = img - [123.68, 116.779, 103.939]
return img

```

Listing 21.41: Example of a function for loading and preparing an image for making a prediction.

Next, we can load the model as in the previous section and call the `predict()` function to predict the content in the image as a number between 0 and 1 for cat and dog respectively.

```

...
# predict the class
result = model.predict(img)

```

Listing 21.42: Example of making a prediction with a prepared image.

```

# make a prediction for a new image.
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.models import load_model

# load and prepare the image
def load_image(filename):
    # load the image
    img = load_img(filename, target_size=(224, 224))
    # convert to array
    img = img_to_array(img)
    # reshape into a single sample with 3 channels
    img = img.reshape(1, 224, 224, 3)
    # center pixel data
    img = img.astype('float32')
    img = img - [123.68, 116.779, 103.939]
    return img

# load an image and predict the class
def run_example():
    # load the image
    img = load_image('sample_image.jpg')
    # load model
    model = load_model('final_model.h5')
    # predict the class
    result = model.predict(img)
    print(result[0])

# entry point, run the example
run_example()

```

Listing 21.43: Example making a prediction with the final model.

Running the example first loads and prepares the image, loads the model, and then correctly predicts that the loaded image represents a dog or class 1.

1

Listing 21.44: Example output from making a prediction with the final model.

21.8 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Tune Regularization.** Explore minor changes to the regularization techniques used on the baseline model, such as different dropout rates and different image augmentation.
- **Tune Learning Rate.** Explore changes to the learning algorithm used to train the baseline model, such as alternate learning rate, a learning rate schedule, or an adaptive learning rate algorithm such as Adam.
- **Alternate Pre-Trained Model.** Explore an alternate pre-trained model for transfer learning on the problem, such as Inception or ResNet.

If you explore any of these extensions, I'd love to know.

21.9 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

21.9.1 Papers

- *Asirra: A CAPTCHA that Exploits Interest-Aligned Manual Image Categorization*, 2007.
<https://dl.acm.org/citation.cfm?id=1315291>
- *Machine Learning Attacks Against the Asirra CAPTCHA*, 2007.
<https://dl.acm.org/citation.cfm?id=1455838>
- *OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks*, 2013.
<https://arxiv.org/abs/1312.6229>

21.9.2 API

- Keras Applications API.
<https://keras.io/applications/>
- Keras Image Processing API.
<https://keras.io/preprocessing/image/>
- Keras Sequential Model API.
<https://keras.io/models/sequential/>

21.9.3 Articles

- Dogs vs Cats Kaggle Competition.
<https://www.kaggle.com/c/dogs-vs-cats/>
- Dogs vs. Cats Redux: Kernels Edition.
<https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition>
- Dogs vs Cats Dataset, Kaggle.
<https://www.kaggle.com/c/dogs-vs-cats/data>

21.10 Summary

In this tutorial, you discovered how to develop a convolutional neural network to classify photos of dogs and cats. Specifically, you learned:

- How to load and prepare photos of dogs and cats for modeling.
- How to develop a convolutional neural network for photo classification from scratch and improve model performance.
- How to develop a model for photo classification using transfer learning.

21.10.1 Next

In the next section, you will discover how to develop a deep learning convolutional neural network model for labeling satellite photographs of the Amazon rainforest.

Chapter 22

How to Label Satellite Photographs of the Amazon Rainforest

The Planet dataset has become a standard computer vision benchmark that involves classifying or tagging the contents satellite photos of Amazon tropical rainforest. The dataset was the basis of a data science competition on the Kaggle website and was effectively solved. Nevertheless, it can be used as the basis for learning and practicing how to develop, evaluate, and use convolutional deep learning neural networks for image classification from scratch. This includes how to develop a robust test harness for estimating the performance of the model, how to explore improvements to the model, and how to save the model and later load it to make predictions on new data. In this tutorial, you will discover how to develop a convolutional neural network to classify satellite photos of the Amazon tropical rainforest.

After completing this tutorial, you will know:

- How to load and prepare satellite photos of the Amazon tropical rainforest for modeling.
- How to develop a convolutional neural network for photo classification from scratch and improve model performance.
- How to develop a final model and use it to make predictions on new data.

Let's get started.

22.1 Tutorial Overview

This tutorial is divided into seven parts; they are:

1. Introduction to the Planet Dataset
2. How to Prepare Data for Modeling
3. Model Evaluation Measure
4. How to Evaluate a Baseline Model
5. How to Improve Model Performance

6. How to use Transfer Learning
7. How to Finalize the Model and Make Predictions

22.2 Introduction to the Planet Dataset

The *Planet: Understanding the Amazon from Space* competition was held on Kaggle in 2017. The competition involved classifying small squares of satellite images taken from space of the Amazon rainforest in Brazil in terms of 17 classes, such as *agriculture*, *clear*, and *water*. Given the name of the competition, the dataset is often referred to simply as the *Planet dataset*. The color images were provided in both TIFF and JPEG format with the size 256×256 pixels. A total of 40,779 images were provided in the training dataset and 40,669 images were provided in the test set for which predictions were required.

The problem is an example of a multi-label image classification task, where one or more class labels must be predicted for each label. This is different from multiclass classification, where each image is assigned one from among many classes. The multiple class labels were provided for each image in the training dataset with an accompanying file that mapped the image filename to the string class labels. The competition was run for approximately four months (April to July in 2017) and a total of 938 teams participated, generating much discussion around the use of data preparation, data augmentation, and the use of convolutional neural networks.

The competition was won by a competitor named *bestfitting* with a public leaderboard F-beta score of 0.93398 on 66% of the test dataset and a private leaderboard F-beta score of 0.93317 on 34% of the test dataset. His approach was described in the article *Planet: Understanding the Amazon from Space, 1st Place Winner's Interview*¹ and involved a pipeline and ensemble of a large number of models, mostly convolutional neural networks with transfer learning. It was a challenging competition, although the dataset remains freely available (if you have a Kaggle account), and provides a good benchmark problem for practicing image classification with convolutional neural networks for aerial and satellite datasets. As such, it is routine to achieve an F-beta score of greater than 0.8 with a manually designed convolutional neural network and an F-beta score 0.89+ using transfer learning on this task.

22.3 How to Prepare Data for Modeling

The first step is to download the dataset. In order to download the data files, you must have a Kaggle account. The dataset can be downloaded from the Planet Data page.

- [Planet: Understanding the Amazon from Space Data Download Page](#).²

This page lists all of the files provided for the competition, although we do not need to download all of the files. The specific files required for this tutorial are as follows:

- `train-jpg.tar.7z` (600 megabytes)
- `train_v2.csv.zip` (159 kilobytes)

¹<https://goo.gl/DYMEl3>

²<https://www.kaggle.com/c/planet-understanding-the-amazon-from-space/data>

Note that the JPEG zip files might be listed without the `.7z` extension. If so, download the `.tar` versions. To download a given file, click the small icon of the download button that appears next to the file when you hover over it with the mouse, as seen in the picture below. Note that you must agree to the competition rules before the website will allow you to download the dataset. If you are having trouble, click on the *Late Submission* button.

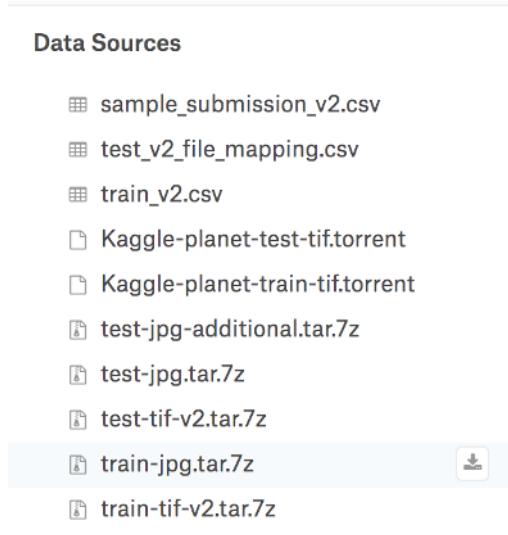


Figure 22.1: Example of Download Button to Download Files for the Planet Dataset.

Once you have downloaded the dataset files, you must unzip them. The `.zip` files for the CSV files can be unzipped using your favorite unzipping program. The `.7z` files that contain the JPEG images can also be unzipped using your favorite unzipping program. If this is a new zip format for you, you may need additional software, such as *The Unarchiver*³ software on MacOS, or *p7zip*⁴ on many platforms. For example, on the command line on most POSIX-based workstations the `.7z` files can be decompressed using the *p7zip* and `tar` files as follows:

```
7z x test-jpg.tar.7z
tar -xvf test-jpg.tar
7z x train-jpg.tar.7z
tar -xvf train-jpg.tar
```

Listing 22.1: Example of commands for unzipping the downloaded dataset.

Once unzipped, you will now have a CSV file and a directory in your current working directory, as follows:

```
train-jpg/
train_v2.csv
```

Listing 22.2: Example of the unzipped dataset.

Inspecting the folder, you will see many JPEG files. Inspecting the `train_v2.csv` file, you will see a mapping of JPEG files in the training dataset (`train-jpg/`) and their mapping to class labels separated by a space for each; for example:

³<https://itunes.apple.com/au/app/the-unarchiver/id425424353?mt=12>

⁴<http://p7zip.sourceforge.net/>

```

image_name,tags
train_0,haze primary
train_1,agriculture clear primary water
train_2,clear primary
train_3,clear primary
train_4,agriculture clear habitation primary road
...

```

Listing 22.3: Sample of the label file.

The dataset must be prepared before modeling. There are at least two approaches we could explore; they are: an in-memory approach and a progressive loading approach. The dataset could be prepared with the intent of loading the entire training dataset into memory when fitting models. This will require a machine with sufficient RAM to hold all of the images (e.g. 32GB or 64GB of RAM), such as an Amazon EC2 instance, although training models will be significantly faster.

Alternately, the dataset could be loaded as-needed during training, batch by batch. This would require developing a data generator. Training models would be significantly slower, but training could be performed on workstations with less RAM (e.g. 8GB or 16GB). In this tutorial, we will use the former approach. As such, I strongly encourage you to run the tutorial on an Amazon EC2 instance with sufficient RAM and access to a GPU, such as the affordable p3.2xlarge instance on the Deep Learning AMI (Amazon Linux) AMI, which costs approximately \$3 USD per hour (for more information, see Appendix C). If using an EC2 instance is not an option for you, then I will give hints below on how to further reduce the size of the training dataset so that it will fit into memory on your workstation so that you can complete this tutorial.

22.3.1 Visualize Dataset

The first step is to inspect some of the images in the training dataset. We can do this by loading some images and plotting multiple images in one figure using Matplotlib. The complete example is listed below.

```

# plot the first 9 images in the planet dataset
from matplotlib import pyplot
from matplotlib.image import imread
# define location of dataset
folder = 'train-jpg/'
# plot first few images
for i in range(9):
    # define subplot
    pyplot.subplot(330 + 1 + i)
    # define filename
    filename = folder + 'train_' + str(i) + '.jpg'
    # load image pixels
    image = imread(filename)
    # plot raw pixel data
    pyplot.imshow(image)
# show the figure
pyplot.show()

```

Listing 22.4: Example of plotting photographs from the dataset.

Running the example creates a figure that plots the first nine images in the training dataset. We can see that the images are indeed satellite photos of the rain forest. Some show significant haze, others show trees, roads, or rivers and other structures. The plots suggests that modeling may benefit from data augmentation as well as simple techniques to make the features in the images more visible.

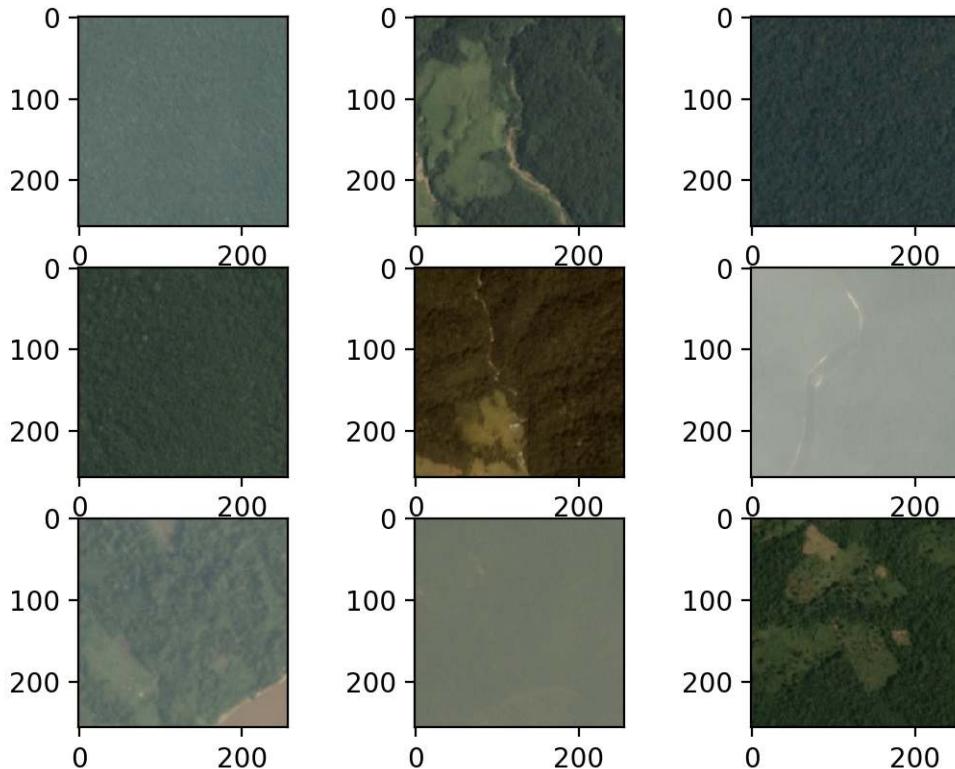


Figure 22.2: Figure Showing the First Nine Images From the Planet Dataset.

22.3.2 Create Mappings

The next step involves understanding the labels that may be assigned to each image. We can load the CSV mapping file for the training dataset (`train_v2.csv`) directly using the `read_csv()` Pandas function. The complete example is listed below.

```
# load and summarize the mapping file for the planet dataset
from pandas import read_csv
# load file as CSV
filename = 'train_v2.csv'
mapping_csv = read_csv(filename)
# summarize properties
print(mapping_csv.shape)
print(mapping_csv[:10])
```

Listing 22.5: Example of loading and summarizing the mappings file.

Running the example first summarizes the shape of the training dataset. We can see that there are indeed 40,479 training images known to the mapping file. Next, the first 10 rows of the file are summarized. We can see that the second column of the file contains a space-separated list of tags to assign to each image.

(40479, 2)		
	image_name	tags
0	train_0	haze primary
1	train_1	agriculture clear primary water
2	train_2	clear primary
3	train_3	clear primary
4	train_4	agriculture clear habitation primary road
5	train_5	haze primary water
6	train_6	agriculture clear cultivation primary water
7	train_7	haze primary
8	train_8	agriculture clear cultivation primary
9	train_9	agriculture clear cultivation primary road

Listing 22.6: Example output from loading and summarizing the mappings file.

We will need the set of all known tags to be assigned to images, as well as a unique and consistent integer to apply to each tag. This is so that we can develop a target vector for each image with a one hot encoding, e.g. a vector with all zeros and a one at the index for each tag applied to the image. This can be achieved by looping through each row in the `tags` column, splitting the tags by space, and storing them in a set. We will then have a set of all known tags. For example:

```
# create a set of labels
labels = set()
for i in range(len(mapping_csv)):
    # convert spaced separated tags into an array of tags
    tags = mapping_csv['tags'][i].split(' ')
    # add tags to the set of known labels
    labels.update(tags)
```

Listing 22.7: Example of building a set of unique tags.

This can then be ordered alphabetically and each tag assigned an integer based on this alphabetic rank. This will mean that the same tag will always be assigned the same integer for consistency.

```
# convert set of labels to a list to list
labels = list(labels)
# order set alphabetically
labels.sort()
```

Listing 22.8: Example of ordering a list of unique tags.

We can create a dictionary that maps tags to integers so that we can encode the training dataset for modeling. We can also create a dictionary with the reverse mapping from integers to string tag values, so later when the model makes a prediction, we can turn it into something readable.

```
# dict that maps labels to integers, and the reverse
labels_map = {labels[i]:i for i in range(len(labels))}
```

```
inv_labels_map = {i:labels[i] for i in range(len(labels))}
```

Listing 22.9: Example of creating a mapping of tags to integers, and the inverse.

We can tie all of this together into a convenience function called `create_tag_mapping()` that will take the loaded DataFrame containing the `train_v2.csv` data and return a mapping and inverse mapping dictionaries.

```
# create a mapping of tags to integers given the loaded mapping file
def create_tag_mapping(mapping_csv):
    # create a set of all known tags
    labels = set()
    for i in range(len(mapping_csv)):
        # convert spaced separated tags into an array of tags
        tags = mapping_csv['tags'][i].split(' ')
        # add tags to the set of known labels
        labels.update(tags)
    # convert set of labels to a list to list
    labels = list(labels)
    # order set alphabetically
    labels.sort()
    # dict that maps labels to integers, and the reverse
    labels_map = {labels[i]:i for i in range(len(labels))}
    inv_labels_map = {i:labels[i] for i in range(len(labels))}
    return labels_map, inv_labels_map
```

Listing 22.10: Example of a function to prepare a mapping of tags to integers and the inverse.

We can test out this function to see how many and what tags we have to work with; the complete example is listed below.

```
# create a mapping of tags to integers
from pandas import read_csv

# create a mapping of tags to integers given the loaded mapping file
def create_tag_mapping(mapping_csv):
    # create a set of all known tags
    labels = set()
    for i in range(len(mapping_csv)):
        # convert spaced separated tags into an array of tags
        tags = mapping_csv['tags'][i].split(' ')
        # add tags to the set of known labels
        labels.update(tags)
    # convert set of labels to a list to list
    labels = list(labels)
    # order set alphabetically
    labels.sort()
    # dict that maps labels to integers, and the reverse
    labels_map = {labels[i]:i for i in range(len(labels))}
    inv_labels_map = {i:labels[i] for i in range(len(labels))}
    return labels_map, inv_labels_map

# load file as CSV
filename = 'train_v2.csv'
mapping_csv = read_csv(filename)
# create a mapping of tags to integers
mapping, inv_mapping = create_tag_mapping(mapping_csv)
```

```
print(len(mapping))
print(mapping)
```

Listing 22.11: Example of creating a mapping of tags to integers.

Running the example, we can see that we have a total of 17 tags in the dataset. We can also see the mapping dictionary where each tag is assigned a consistent and unique integer. The tags appear to be sensible descriptions of the types of features we may see in a given satellite image. It might be interesting as a further extension to explore the distribution of tags across images to see if their assignment or use in the training dataset is balanced or imbalanced. This could give further insight into how difficult the prediction problem may be.

```
17

{'agriculture': 0, 'artisinal_mine': 1, 'bare_ground': 2, 'blooming': 3, 'blow_down': 4,
 'clear': 5, 'cloudy': 6, 'conventional_mine': 7, 'cultivation': 8, 'habitation': 9,
 'haze': 10, 'partly_cloudy': 11, 'primary': 12, 'road': 13, 'selective_logging': 14,
 'slash_burn': 15, 'water': 16}
```

Listing 22.12: Example output from creating a mapping of tags to integers.

We also need a mapping of training set filenames to the tags for the image. This is a simple dictionary with the filename of the image as the key and the list of tags as the value. The `create_file_mapping()` below implements this, also taking the loaded `DataFrame` as an argument and returning the mapping with the tag value for each filename stored as a list.

```
# create a mapping of filename to tags
def create_file_mapping(mapping_csv):
    mapping = dict()
    for i in range(len(mapping_csv)):
        name, tags = mapping_csv['image_name'][i], mapping_csv['tags'][i]
        mapping[name] = tags.split(' ')
    return mapping
```

Listing 22.13: Example of a function create a mapping of image filenames to tags.

We can now prepare the image component of the dataset.

22.3.3 Create In-Memory Dataset

We need to be able to load the JPEG images into memory. This can be achieved by enumerating all files in the `train-jpg/` folder. Keras provides a simple API to load an image from file via the `load_img()` function and to convert it to a NumPy array via the `img_to_array()` function. As part of loading an image, we can force the size to be smaller to save memory and speed up training. In this case, we will halve the size of the image from 256×256 to 128×128 . We will also store the pixel values as an unsigned 8-bit integer (e.g. values between 0 and 255).

```
...
# load image
photo = load_img(filename, target_size=(128,128))
# convert to numpy array
photo = img_to_array(photo, dtype='uint8')
```

Listing 22.14: Example of loading an image with a preferred size.

The photo will represent an input to the model, but we also require an output for the photo. We can then retrieve the tags for the loaded image using the filename without the extension using the prepared filename-to-tags mapping prepared with the `create_file_mapping()` function developed in the previous section.

```
...
# get tags
tags = file_mapping(filename[:-4])
```

Listing 22.15: Example of getting tags for a filename.

We need to one hot encode the tags for the image. This means that we will require a 17-element vector with a 1 value for each tag present. We can get the index of where to place the 1 values from the mapping of tags to integers created via the `create_tag_mapping()` function developed in the previous section. The `one_hot_encode()` function below implements this, given a list of tags for an image and the mapping of tags to integers as arguments, and it will return a 17 element NumPy array that describes a one hot encoding of the tags for one photo.

```
# create a one hot encoding for one list of tags
def one_hot_encode(tags, mapping):
    # create empty vector
    encoding = zeros(len(mapping), dtype='uint8')
    # mark 1 for each tag in the vector
    for tag in tags:
        encoding[mapping[tag]] = 1
    return encoding
```

Listing 22.16: Example of a function for one hot encoding tags.

We can now load the input (photos) and output (one hot encoded vector) elements for the entire training dataset. The `load_dataset()` function below implements this given the path to the JPEG images, the mapping of files to tags, and the mapping of tags to integers as inputs; it will return NumPy arrays for the X and y elements for modeling.

```
# load all images into memory
def load_dataset(path, file_mapping, tag_mapping):
    photos, targets = list(), list()
    # enumerate files in the directory
    for filename in listdir(folder):
        # load image
        photo = load_img(path + filename, target_size=(128,128))
        # convert to numpy array
        photo = img_to_array(photo, dtype='uint8')
        # get tags
        tags = file_mapping[filename[:-4]]
        # one hot encode tags
        target = one_hot_encode(tags, tag_mapping)
        # store
        photos.append(photo)
        targets.append(target)
    X = asarray(photos, dtype='uint8')
    y = asarray(targets, dtype='uint8')
    return X, y
```

Listing 22.17: Example of a function for loading the dataset.

Note: this will load the entire training dataset into memory and may require at least $128 \times 128 \times 3 \times 40,479$ images $\times 8\text{bits}$, or about 2 gigabytes RAM just to hold the loaded photos. If you run out of memory here, or later when modeling (when pixels are 16 or 32 bits), try reducing the size of the loaded photos to 32×32 and/or stop the loop after loading 20,000 photographs. Once loaded, we can save these NumPy arrays to file for later use. We could use the `save()` or `savez()` NumPy functions to save the arrays direction. Instead, we will use the `savez_compressed()` NumPy function to save both arrays in one function call in a compressed format, saving a few more megabytes. Loading the arrays of smaller images will be significantly faster than loading the raw JPEG images each time during modeling.

```
...
# save both arrays to one file in compressed format
savez_compressed('planet_data.npz', X, y)
```

Listing 22.18: Example of saving arrays in a compressed format.

We can tie all of this together and prepare the Planet dataset for in-memory modeling and save it to a new single file for fast loading later. The complete example is listed below.

```
# load and prepare planet dataset and save to file
from os import listdir
from numpy import zeros
from numpy import asarray
from numpy import savez_compressed
from pandas import read_csv
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array

# create a mapping of tags to integers given the loaded mapping file
def create_tag_mapping(mapping_csv):
    # create a set of all known tags
    labels = set()
    for i in range(len(mapping_csv)):
        # convert spaced separated tags into an array of tags
        tags = mapping_csv['tags'][i].split(' ')
        # add tags to the set of known labels
        labels.update(tags)
    # convert set of labels to a list to list
    labels = list(labels)
    # order set alphabetically
    labels.sort()
    # dict that maps labels to integers, and the reverse
    labels_map = {labels[i]:i for i in range(len(labels))}
    inv_labels_map = {i:labels[i] for i in range(len(labels))}
    return labels_map, inv_labels_map

# create a mapping of filename to a list of tags
def create_file_mapping(mapping_csv):
    mapping = dict()
    for i in range(len(mapping_csv)):
        name, tags = mapping_csv['image_name'][i], mapping_csv['tags'][i]
        mapping[name] = tags.split(' ')
    return mapping

# create a one hot encoding for one list of tags
```

```

def one_hot_encode(tags, mapping):
    # create empty vector
    encoding = zeros(len(mapping), dtype='uint8')
    # mark 1 for each tag in the vector
    for tag in tags:
        encoding[mapping[tag]] = 1
    return encoding

# load all images into memory
def load_dataset(path, file_mapping, tag_mapping):
    photos, targets = list(), list()
    # enumerate files in the directory
    for filename in listdir(folder):
        # load image
        photo = load_img(path + filename, target_size=(128,128))
        # convert to numpy array
        photo = img_to_array(photo, dtype='uint8')
        # get tags
        tags = file_mapping[filename[:-4]]
        # one hot encode tags
        target = one_hot_encode(tags, tag_mapping)
        # store
        photos.append(photo)
        targets.append(target)
    X = asarray(photos, dtype='uint8')
    y = asarray(targets, dtype='uint8')
    return X, y

# load the mapping file
filename = 'train_v2.csv'
mapping_csv = read_csv(filename)
# create a mapping of tags to integers
tag_mapping, _ = create_tag_mapping(mapping_csv)
# create a mapping of filenames to tag lists
file_mapping = create_file_mapping(mapping_csv)
# load the jpeg images
folder = 'train-jpg/'
X, y = load_dataset(folder, file_mapping, tag_mapping)
print(X.shape, y.shape)
# save both arrays to one file in compressed format
savez_compressed('planet_data.npz', X, y)

```

Listing 22.19: Example preparing the dataset and saving the results in a compressed format.

Running the example first loads the entire dataset and summarizes the shape. We can confirm that the input samples (X) are 128×128 color images and that the output samples are 17-element vectors. At the end of the run, a single file `planet_data.npz` is saved containing the dataset that is approximately 1.2 gigabytes in size, saving about 700 megabytes due to compression.

(40479, 128, 128, 3) (40479, 17)

Listing 22.20: Example output from preparing the dataset and saving the results in a compressed format.

The dataset can be loaded easily later using the `load()` NumPy function, as follows:

```
# load prepared planet dataset
from numpy import load
data = load('planet_data.npz')
X, y = data['arr_0'], data['arr_1']
print('Loaded: ', X.shape, y.shape)
```

Listing 22.21: Example of loading the prepared dataset.

Running this small example confirms that the dataset is correctly loaded.

```
Loaded: (40479, 128, 128, 3) (40479, 17)
```

Listing 22.22: Example output from loading the prepared dataset.

22.4 Model Evaluation Measure

Before we start modeling, we must select a performance metric. Classification accuracy is often appropriate for binary classification tasks with a balanced number of examples in each class. In this case, we are working neither with a binary or multiclass classification task; instead, it is a multi-label classification task and the number of labels are not balanced, with some used more heavily than others. As such, the Kaggle competition organizes chose the F-beta metric (also called $F\beta$), specifically the F2 score. This is a metric that is related to the F1 score (also called F-measure). The F1 score calculates the average of the recall and the precision. You may remember that the precision and recall are calculated as follows:

$$\text{precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}} \quad (22.1)$$

$$\text{recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}} \quad (22.2)$$

Precision describes how good a model is at predicting the positive class. Recall describes how good the model is at predicting the positive class when the actual outcome is positive. The F1 is the mean of these two scores, specifically the harmonic mean instead of the arithmetic mean because the values are proportions. F1 is preferred over accuracy when evaluating the performance of a model on an imbalanced dataset, with a value between 0 and 1 for worst and best possible scores.

$$F1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \quad (22.3)$$

$$\text{F-beta} = (1 + \text{beta}^2) \times \frac{\text{precision} \times \text{recall}}{\text{beta}^2 \times \text{precision} + \text{recall}} \quad (22.4)$$

A common value of beta is 2, and this was the value used in the competition, where recall valued twice as highly as precision. This is often referred to as the F2 score. The idea of a positive and negative class only makes sense for a binary classification problem. As we are predicting multiple classes, the idea of positive and negative and related terms are calculated for each class in a one vs. rest manner, then averaged across each class. The scikit-learn library provides an implementation of F-beta via the `fbeta_score()` function. We can call this function

to evaluate a set of predictions and specify a beta value of 2 and the `average` argument set to `samples`.

```
...
# calculate F2 score
score = fbeta_score(y_true, y_pred, 2, average='samples')
```

Listing 22.23: Example of calculating the F2 score.

For example, we can test this on our prepared dataset. We can split our loaded dataset into separate train and test datasets that we can use to train and evaluate models on this problem. This can be achieved using the `train_test_split()` and specifying a `random_state` argument so that the same data split is given each time the code is run. We will use 70% for the training set and 30% for the test set.

```
...
# split dataset
trainX, testX, trainY, testY = train_test_split(X, y, test_size=0.3, random_state=1)
```

Listing 22.24: Example of splitting the dataset into train and test sets.

The `load_dataset()` function below implements this by loading the saved dataset, splitting it into train and test components, and returning them ready for use.

```
# load train and test dataset
def load_dataset():
    # load dataset
    data = load('planet_data.npz')
    X, y = data['arr_0'], data['arr_1']
    # separate into train and test datasets
    trainX, testX, trainY, testY = train_test_split(X, y, test_size=0.3, random_state=1)
    print(trainX.shape, trainY.shape, testX.shape, testY.shape)
    return trainX, trainY, testX, testY
```

Listing 22.25: Example of a function for loading and splitting the prepared dataset.

We can then make a prediction of all classes or all tags for all photographs, e.g. one values in the one hot encoded vectors.

```
...
# make all one predictions
train_yhat = asarray([ones(trainY.shape[1]) for _ in range(trainY.shape[0])])
test_yhat = asarray([ones(testY.shape[1]) for _ in range(testY.shape[0])])
```

Listing 22.26: Example of predicting all tags on all photos.

The predictions can then be evaluated using the scikit-learn `fbeta_score()` function with the true values in the train and test dataset.

```
...
train_score = fbeta_score(trainY, train_yhat, 2, average='samples')
test_score = fbeta_score(testY, test_yhat, 2, average='samples')
```

Listing 22.27: Example of calculate F2 scores for naive predictions.

Tying this together, the complete example is listed below.

```

# test f-beta score
from numpy import load
from numpy import ones
from numpy import asarray
from sklearn.model_selection import train_test_split
from sklearn.metrics import fbeta_score

# load train and test dataset
def load_dataset():
    # load dataset
    data = load('planet_data.npz')
    X, y = data['arr_0'], data['arr_1']
    # separate into train and test datasets
    trainX, testX, trainY, testY = train_test_split(X, y, test_size=0.3, random_state=1)
    print(trainX.shape, trainY.shape, testX.shape, testY.shape)
    return trainX, trainY, testX, testY

# load dataset
trainX, trainY, testX, testY = load_dataset()
# make all one predictions
train_yhat = asarray([ones(trainY.shape[1]) for _ in range(trainY.shape[0])])
test_yhat = asarray([ones(testY.shape[1]) for _ in range(testY.shape[0])])
# evaluate predictions
train_score = fbeta_score(trainY, train_yhat, 2, average='samples')
test_score = fbeta_score(testY, test_yhat, 2, average='samples')
print('All Ones: train=% .3f, test=% .3f' % (train_score, test_score))

```

Listing 22.28: Example of evaluating a naive prediction on the train and test datasets.

Running this example first loads the prepared dataset, then splits it into train and test sets and the shape of the prepared datasets is reported. We can see that we have a little more than 28,000 examples in the training dataset and a little more than 12,000 examples in the test set. Next, the all-one predictions are prepared and then evaluated and the scores are reported. We can see that an all ones prediction for both datasets results in a score of about 0.48.

```
(28335, 128, 128, 3) (28335, 17) (12144, 128, 128, 3) (12144, 17)
All Ones: train=0.484, test=0.483
```

Listing 22.29: Example output from evaluating a naive prediction on the train and test datasets.

We will require a version of the F-beta score calculation in Keras to use as a metric. Keras used to support this metric for binary classification problems (2 classes) prior to version 2.0 of the library. This code can be used as the basis for defining a new metric function that can be used with Keras. A version of this function is also proposed in a Kaggle kernel titled *F-beta score for Keras*⁵. This new function is listed below.

```

from keras import backend

# calculate fbeta score for multiclass/label classification
def fbeta(y_true, y_pred, beta=2):
    # clip predictions
    y_pred = backend.clip(y_pred, 0, 1)
    # calculate elements

```

⁵<https://www.kaggle.com/arsenyinfo/f-beta-score-for-keras>

```

tp = backend.sum(backend.round(backend.clip(y_true * y_pred, 0, 1)), axis=1)
fp = backend.sum(backend.round(backend.clip(y_pred - y_true, 0, 1)), axis=1)
fn = backend.sum(backend.round(backend.clip(y_true - y_pred, 0, 1)), axis=1)
# calculate precision
p = tp / (tp + fp + backend.epsilon())
# calculate recall
r = tp / (tp + fn + backend.epsilon())
# calculate fbeta, averaged across each class
bb = beta ** 2
fbeta_score = backend.mean((1 + bb) * (p * r) / (bb * p + r + backend.epsilon()))
return fbeta_score

```

Listing 22.30: Example of calculating F-beta scores as a metric in Keras.

It can be used when compiling a model in Keras, specified via the metrics argument; for example:

```

...
model.compile(... metrics=[fbeta])

```

Listing 22.31: Example of using the F-beta metric in Keras.

We can test this new function and compare results to the scikit-learn function as follows.

```

# compare f-beta score between sklearn and keras
from numpy import load
from numpy import ones
from numpy import asarray
from sklearn.model_selection import train_test_split
from sklearn.metrics import fbeta_score
from keras import backend

# load train and test dataset
def load_dataset():
    # load dataset
    data = load('planet_data.npz')
    X, y = data['arr_0'], data['arr_1']
    # separate into train and test datasets
    trainX, testX, trainY, testY = train_test_split(X, y, test_size=0.3, random_state=1)
    print(trainX.shape, trainY.shape, testX.shape, testY.shape)
    return trainX, trainY, testX, testY

# calculate fbeta score for multi-class/label classification
def fbeta(y_true, y_pred, beta=2):
    # clip predictions
    y_pred = backend.clip(y_pred, 0, 1)
    # calculate elements
    tp = backend.sum(backend.round(backend.clip(y_true * y_pred, 0, 1)), axis=1)
    fp = backend.sum(backend.round(backend.clip(y_pred - y_true, 0, 1)), axis=1)
    fn = backend.sum(backend.round(backend.clip(y_true - y_pred, 0, 1)), axis=1)
    # calculate precision
    p = tp / (tp + fp + backend.epsilon())
    # calculate recall
    r = tp / (tp + fn + backend.epsilon())
    # calculate fbeta, averaged across each class
    bb = beta ** 2
    fbeta_score = backend.mean((1 + bb) * (p * r) / (bb * p + r + backend.epsilon()))
    return fbeta_score

```

```

    return fbeta_score

# load dataset
trainX, trainY, testX, testY = load_dataset()
# make all one predictions
train_yhat = asarray([ones(trainY.shape[1]) for _ in range(trainY.shape[0])])
test_yhat = asarray([ones(testY.shape[1]) for _ in range(testY.shape[0])])
# evaluate predictions with sklearn
train_score = fbeta_score(trainY, train_yhat, 2, average='samples')
test_score = fbeta_score(testY, test_yhat, 2, average='samples')
print('All Ones (sklearn): train=% .3f, test=% .3f' % (train_score, test_score))
# evaluate predictions with keras
train_score = fbeta(backend.variable(trainY), backend.variable(train_yhat))
test_score = fbeta(backend.variable(testY), backend.variable(test_yhat))
print('All Ones (keras): train=% .3f, test=% .3f' % (train_score, test_score))

```

Listing 22.32: Example comparing the Keras metric and scikit-learn implementations of F-beta.

Running the example loads the datasets as before, and in this case, the F-beta is calculated using both scikit-learn and Keras. We can see that both functions achieve the same result.

```
(28335, 128, 128, 3) (28335, 17) (12144, 128, 128, 3) (12144, 17)
All Ones (sklearn): train=0.484, test=0.483
All Ones (keras): train=0.484, test=0.483
```

Listing 22.33: Example output from comparing the Keras metric and scikit-learn implementations of F-beta.

We can use the score of 0.483 on the test set as a naive forecast to which all models in the subsequent sections can be compared to determine if they are skillful or not.

22.5 How to Evaluate a Baseline Model

We are now ready to develop and evaluate a baseline convolutional neural network model for the prepared planet dataset. We will design a baseline model with a VGG-type structure. That is blocks of convolutional layers with small 3×3 filters followed by a max pooling layer, with this pattern repeating with a doubling in the number of filters with each block added. Specifically, each block will have two convolutional layers with 3×3 filters, ReLU activation and He weight initialization with same padding, ensuring the output feature maps have the same width and height. These will be followed by a max pooling layer with a 3×3 kernel. Three of these blocks will be used with 32, 64 and 128 filters respectively.

```

...
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same', input_shape=(128, 128, 3)))
model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(MaxPooling2D((2, 2)))

```

```
model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(MaxPooling2D((2, 2)))
```

Listing 22.34: Example of defining a 3-block VGG-style feature extraction model.

The output of the final pooling layer will be flattened and fed to a fully connected layer for interpretation then finally to an output layer for prediction. The model must produce a 17-element vector with a prediction between 0 and 1 for each output class.

```
...
model.add(Flatten())
model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(17, activation='sigmoid'))
```

Listing 22.35: Example of defining a multi-label classification output model.

If this were a multiclass classification problem, we would use a softmax activation function and the categorical cross-entropy loss function. This would not be appropriate for multi-label classification, as we expect the model to output multiple 1 values, not a single 1 value. In this case, we will use the sigmoid activation function in the output layer and optimize the binary cross-entropy loss function. The model will be optimized with minibatch stochastic gradient descent with a conservative learning rate of 0.01 and a momentum of 0.9, and the model will keep track of the *fbeta* metric during training.

```
...
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(optimizer=opt, loss='binary_crossentropy', metrics=[fbeta])
```

Listing 22.36: Example of defining the optimization algorithm and metric.

The `define_model()` function below ties all of this together and parameterizes the shape of the input and output, in case you want to experiment by changing these values or reuse the code on another dataset. The function will return a model ready to be fit on the planet dataset.

```
# define cnn model
def define_model(in_shape=(128, 128, 3), out_shape=17):
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same', input_shape=in_shape))
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
        padding='same'))
    model.add(MaxPooling2D((2, 2)))
```

```

model.add(Flatten())
model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(out_shape, activation='sigmoid'))
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(optimizer=opt, loss='binary_crossentropy', metrics=[fbeta])
return model

```

Listing 22.37: Example of a function for defining a model.

The choice of this model as the baseline model is somewhat arbitrary. You may want to explore with other baseline models that have fewer layers or different learning rates. We can use the `load_dataset()` function developed in the previous section to load the dataset and split it into train and test sets for fitting and evaluating a defined model. The pixel values will be normalized before fitting the model. We will achieve this by defining an `ImageDataGenerator` instance and specifying the rescale argument as $\frac{1.0}{255.0}$. This will normalize pixel values per batch to 32-bit floating point values, which might be more memory efficient than rescaling all of the pixel values at once in memory.

```

...
# create data generator
datagen = ImageDataGenerator(rescale=1.0/255.0)

```

Listing 22.38: Example of preparing an image data generator with normalization.

We can create iterators from this data generator for both the train and test sets, and in this case, we will use the relatively large batch size of 128 images to accelerate learning.

```

...
# prepare iterators
train_it = datagen.flow(trainX, trainY, batch_size=128)
test_it = datagen.flow(testX, testY, batch_size=128)

```

Listing 22.39: Example of preparing an image dataset iterators.

The defined model can then be fit using the train iterator, and the test iterator can be used to evaluate the test dataset at the end of each epoch. The model will be fit for 50 epochs.

```

...
# fit model
history = model.fit_generator(train_it, steps_per_epoch=len(train_it),
                               validation_data=test_it, validation_steps=len(test_it), epochs=50, verbose=0)

```

Listing 22.40: Example of fitting the model on the dataset iterators.

Once fit, we can calculate the final loss and F-beta scores on the test dataset to estimate the skill of the model.

```

...
# evaluate model
loss, fbeta = model.evaluate_generator(test_it, steps=len(test_it), verbose=0)
print('> loss=%3f, fbeta=%3f' % (loss, fbeta))

```

Listing 22.41: Example of evaluating the model on the dataset iterators.

The `fit_generator()` function called to fit the model returns a dictionary containing the loss and F-beta scores recorded each epoch on the train and test dataset. We can create a

plot of these values that can provide insight into the learning dynamics of the model. The `summarize_diagnostics()` function will create a figure from this recorded history data with one plot showing loss and another the F-beta scores for the model at the end of each training epoch on the train dataset (blue lines) and test dataset (orange lines). The created figure is saved to a PNG file with the same filename as the script with a `_plot.png` extension. This allows the same test harness to be used with multiple different script files for different model configurations, saving the learning curves in separate files along the way.

```
# plot diagnostic learning curves
def summarize_diagnostics(history):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(history.history['loss'], color='blue', label='train')
    pyplot.plot(history.history['val_loss'], color='orange', label='test')
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Fbeta')
    pyplot.plot(history.history['fbeta'], color='blue', label='train')
    pyplot.plot(history.history['val_fbeta'], color='orange', label='test')
    # save plot to file
    filename = sys.argv[0].split('/')[-1]
    pyplot.savefig(filename + '_plot.png')
    pyplot.close()
```

Listing 22.42: Example of a function for plotting learning curves of performance.

We can tie this together and define a function `run_test_harness()` to drive the test harness, including the loading and preparation of the data as well as definition, fit, and evaluation of the model.

```
# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # create data generator
    datagen = ImageDataGenerator(rescale=1.0/255.0)
    # prepare iterators
    train_it = datagen.flow(trainX, trainY, batch_size=128)
    test_it = datagen.flow(testX, testY, batch_size=128)
    # define model
    model = define_model()
    # fit model
    history = model.fit_generator(train_it, steps_per_epoch=len(train_it),
        validation_data=test_it, validation_steps=len(test_it), epochs=50, verbose=0)
    # evaluate model
    loss, fbeta = model.evaluate_generator(test_it, steps=len(test_it), verbose=0)
    print('> loss=%f, fbeta=%f' % (loss, fbeta))
    # learning curves
    summarize_diagnostics(history)
```

Listing 22.43: Example of a function for driving the test harness.

The complete example of evaluating a baseline model on the planet dataset is listed below.

```
# baseline model for the planet dataset
```

```
import sys
from numpy import load
from matplotlib import pyplot
from sklearn.model_selection import train_test_split
from keras import backend
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD

# load train and test dataset
def load_dataset():
    # load dataset
    data = load('planet_data.npz')
    X, y = data['arr_0'], data['arr_1']
    # separate into train and test datasets
    trainX, testX, trainY, testY = train_test_split(X, y, test_size=0.3, random_state=1)
    print(trainX.shape, trainY.shape, testX.shape, testY.shape)
    return trainX, trainY, testX, testY

# calculate fbeta score for multi-class/label classification
def fbeta(y_true, y_pred, beta=2):
    # clip predictions
    y_pred = backend.clip(y_pred, 0, 1)
    # calculate elements
    tp = backend.sum(backend.round(backend.clip(y_true * y_pred, 0, 1)), axis=1)
    fp = backend.sum(backend.round(backend.clip(y_pred - y_true, 0, 1)), axis=1)
    fn = backend.sum(backend.round(backend.clip(y_true - y_pred, 0, 1)), axis=1)
    # calculate precision
    p = tp / (tp + fp + backend.epsilon())
    # calculate recall
    r = tp / (tp + fn + backend.epsilon())
    # calculate fbeta, averaged across each class
    bb = beta ** 2
    fbeta_score = backend.mean((1 + bb) * (p * r) / (bb * p + r + backend.epsilon()))
    return fbeta_score

# define cnn model
def define_model(in_shape=(128, 128, 3), out_shape=17):
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same', input_shape=in_shape))
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
```

```

    padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(out_shape, activation='sigmoid'))
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(optimizer=opt, loss='binary_crossentropy', metrics=[fbeta])
return model

# plot diagnostic learning curves
def summarize_diagnostics(history):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(history.history['loss'], color='blue', label='train')
    pyplot.plot(history.history['val_loss'], color='orange', label='test')
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Fbeta')
    pyplot.plot(history.history['fbeta'], color='blue', label='train')
    pyplot.plot(history.history['val_fbeta'], color='orange', label='test')
    # save plot to file
    filename = sys.argv[0].split('/')[-1]
    pyplot.savefig(filename + '_plot.png')
    pyplot.close()

# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # create data generator
    datagen = ImageDataGenerator(rescale=1.0/255.0)
    # prepare iterators
    train_it = datagen.flow(trainX, trainY, batch_size=128)
    test_it = datagen.flow(testX, testY, batch_size=128)
    # define model
    model = define_model()
    # fit model
    history = model.fit_generator(train_it, steps_per_epoch=len(train_it),
        validation_data=test_it, validation_steps=len(test_it), epochs=50, verbose=0)
    # evaluate model
    loss, fbeta = model.evaluate_generator(test_it, steps=len(test_it), verbose=0)
    print('> loss=%f, fbeta=%f' % (loss, fbeta))
    # learning curves
    summarize_diagnostics(history)

# entry point, run the test harness
run_test_harness()

```

Listing 22.44: Example of evaluating the baseline model on the planet dataset.

Running the example first loads the dataset and splits it into train and test sets. The shape of the input and output elements of each of the train and test datasets is printed, confirming that the same data split was performed as before. The model is fit and evaluated, and an F-beta score for the final model on the test dataset is reported.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the baseline model achieved an F-beta score of about 0.831, which is quite a bit better than the naive score of 0.483 reported in the previous section. This suggests that the baseline model is skillful.

```
(28335, 128, 128, 3) (28335, 17) (12144, 128, 128, 3) (12144, 17)
> loss=0.470, fbeta=0.831
```

Listing 22.45: Example output from evaluating the baseline model on the planet dataset.

A figure is also created and saved to file showing plots of the learning curves for the model on the train and test sets with regard to both loss and F-beta. In this case, the plot of the loss learning curves suggests that the model has overfit the training dataset, perhaps around epoch 20 out of 50, although the overfitting has not seemingly negatively impacted the performance of the model on the test dataset with regard to the F-beta score.

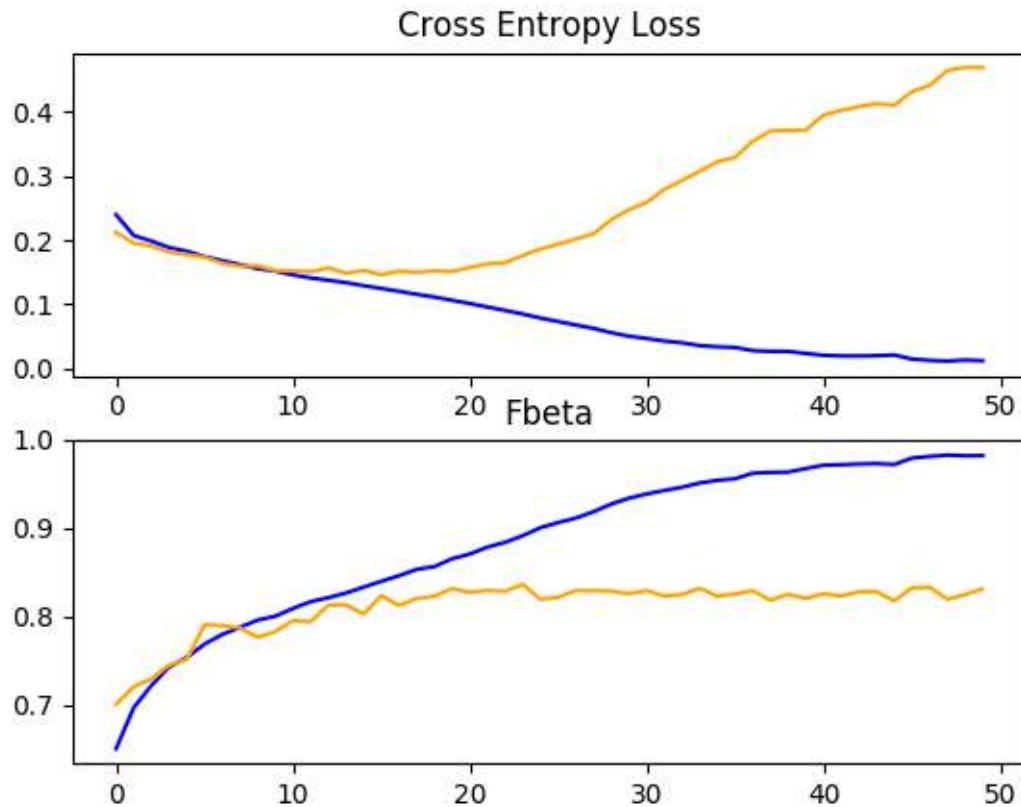


Figure 22.3: Line Plots Showing Loss and F-beta Learning Curves for the Baseline Model on the Train and Test Datasets on the Planet Problem.

Now that we have a baseline model for the dataset, we have a strong basis for experimentation and improvement. We will explore some ideas for improving the performance of the model in the next section.

22.6 How to Improve Model Performance

In the previous section, we defined a baseline model that can be used as the basis for improvement on the planet dataset. The model achieved a reasonable F-beta score, although the learning curves suggested that the model had overfit the training dataset. Two common approaches to explore to address overfitting are dropout regularization and data augmentation. Both have the effect of disrupting and slowing down the learning process, specifically the rate that the model improves over training epochs. We will explore both of these methods in this section. Given that we expect the rate of learning to be slowed, we give the model more time to learn by increasing the number of training epochs from 50 to 200.

22.6.1 Dropout Regularization

Dropout regularization is a computationally cheap way to regularize a deep neural network. Dropout works by probabilistically removing, or *dropping out*, inputs to a layer, which may be input variables in the data sample or activations from a previous layer. It has the effect of simulating a large number of networks with very different network structures and, in turn, making nodes in the network generally more robust to the inputs. Typically, a small amount of dropout can be applied after each VGG block, with more dropout applied to the fully connected layers near the output layer of the model. Below is the `define_model()` function for an updated version of the baseline model with the addition of Dropout. In this case, a dropout of 20% is applied after each VGG block, with a larger dropout rate of 50% applied after the fully connected layer in the classifier part of the model.

```
# define cnn model
def define_model(in_shape=(128, 128, 3), out_shape=17):
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same', input_shape=in_shape))
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dropout(0.5))
    model.add(Dense(out_shape, activation='sigmoid'))
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(optimizer=opt, loss='binary_crossentropy', metrics=[fbeta])
```

```
    return model
```

Listing 22.46: Example of a function for defining the model with dropout.

The full code listing of the baseline model with the addition of dropout on the planet dataset is listed below for completeness.

```
# baseline model with dropout on the planet dataset
import sys
from numpy import load
from matplotlib import pyplot
from sklearn.model_selection import train_test_split
from keras import backend
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.optimizers import SGD

# load train and test dataset
def load_dataset():
    # load dataset
    data = load('planet_data.npz')
    X, y = data['arr_0'], data['arr_1']
    # separate into train and test datasets
    trainX, testX, trainY, testY = train_test_split(X, y, test_size=0.3, random_state=1)
    print(trainX.shape, trainY.shape, testX.shape, testY.shape)
    return trainX, trainY, testX, testY

# calculate fbeta score for multi-class/label classification
def fbeta(y_true, y_pred, beta=2):
    # clip predictions
    y_pred = backend.clip(y_pred, 0, 1)
    # calculate elements
    tp = backend.sum(backend.round(backend.clip(y_true * y_pred, 0, 1)), axis=1)
    fp = backend.sum(backend.round(backend.clip(y_pred - y_true, 0, 1)), axis=1)
    fn = backend.sum(backend.round(backend.clip(y_true - y_pred, 0, 1)), axis=1)
    # calculate precision
    p = tp / (tp + fp + backend.epsilon())
    # calculate recall
    r = tp / (tp + fn + backend.epsilon())
    # calculate fbeta, averaged across each class
    bb = beta ** 2
    fbeta_score = backend.mean((1 + bb) * (p * r) / (bb * p + r + backend.epsilon()))
    return fbeta_score

# define cnn model
def define_model(in_shape=(128, 128, 3), out_shape=17):
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same', input_shape=in_shape))
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(MaxPooling2D((2, 2)))
```

```
model.add(Dropout(0.2))
model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.2))
model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
    padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
model.add(Dropout(0.5))
model.add(Dense(out_shape, activation='sigmoid'))
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(optimizer=opt, loss='binary_crossentropy', metrics=[fbeta])
return model

# plot diagnostic learning curves
def summarize_diagnostics(history):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(history.history['loss'], color='blue', label='train')
    pyplot.plot(history.history['val_loss'], color='orange', label='test')
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Fbeta')
    pyplot.plot(history.history['fbeta'], color='blue', label='train')
    pyplot.plot(history.history['val_fbeta'], color='orange', label='test')
    # save plot to file
    filename = sys.argv[0].split('/')[-1]
    pyplot.savefig(filename + '_plot.png')
    pyplot.close()

# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # create data generator
    datagen = ImageDataGenerator(rescale=1.0/255.0)
    # prepare iterators
    train_it = datagen.flow(trainX, trainY, batch_size=128)
    test_it = datagen.flow(testX, testY, batch_size=128)
    # define model
    model = define_model()
    # fit model
    history = model.fit_generator(train_it, steps_per_epoch=len(train_it),
        validation_data=test_it, validation_steps=len(test_it), epochs=200, verbose=0)
    # evaluate model
    loss, fbeta = model.evaluate_generator(test_it, steps=len(test_it), verbose=0)
    print('> loss=%f, fbeta=%f' % (loss, fbeta))
```

```
# learning curves
summarize_diagnostics(history)

# entry point, run the test harness
run_test_harness()
```

Listing 22.47: Example of evaluating the baseline model with dropout on the planet dataset.

Running the example first fits the model, then reports the model performance on the hold out test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see a small lift in model performance from an F-beta score of about 0.831 for the baseline model to about 0.859 with the addition of dropout.

```
(28335, 128, 128, 3) (28335, 17) (12144, 128, 128, 3) (12144, 17)
> loss=0.190, fbeta=0.859
```

Listing 22.48: Example output from evaluating the baseline model with dropout on the planet dataset.

Reviewing the learning curves, we can see that dropout has had some effect on the rate of improvement of the model on both the train and test sets. Overfitting has been reduced or delayed, although performance may begin to stall towards the middle of the run, around epoch 100. The results suggest that further regularization may be required. This could be achieved by a larger dropout rate and/or perhaps the addition of weight decay. Additionally, the batch size could be decreased and the learning rate decreased, both of which may further slow the rate of improvement by the model, perhaps with a positive effect on reducing the overfitting of the training dataset.

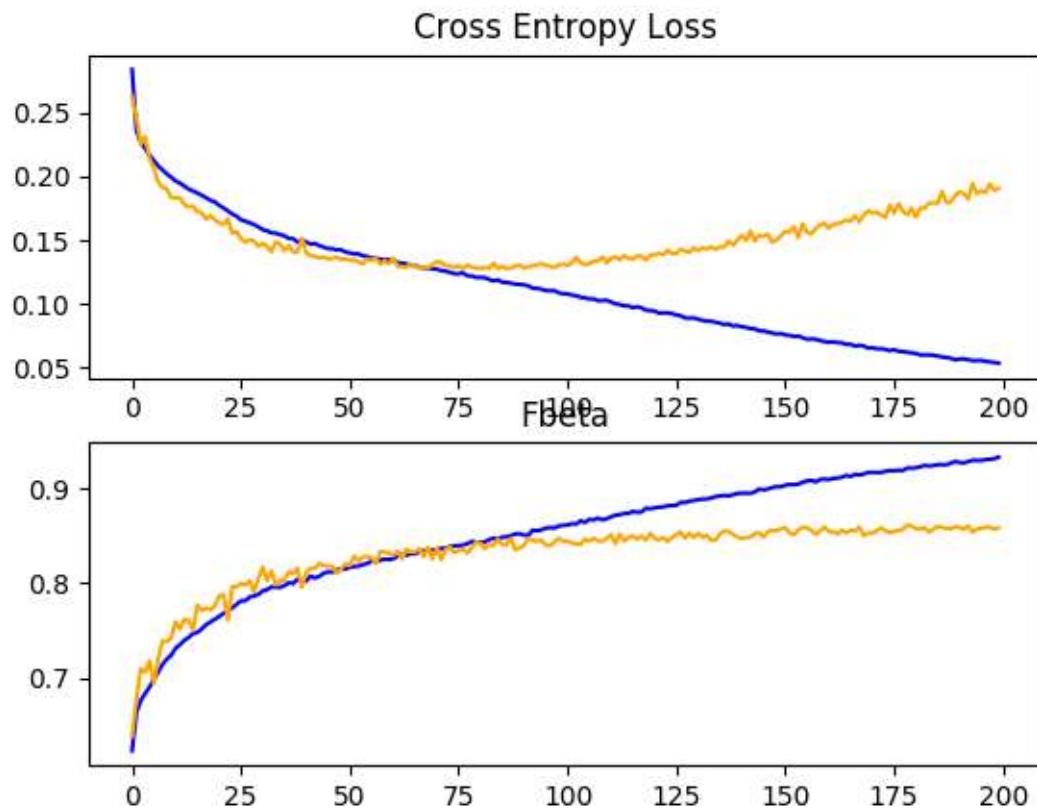


Figure 22.4: Line Plots Showing Loss and F-beta Learning Curves for the Baseline Model With Dropout on the Train and Test Datasets on the Planet Problem.

22.6.2 Image Data Augmentation

Image data augmentation is a technique that can be used to artificially expand the size of a training dataset by creating modified versions of images in the dataset. Training deep learning neural network models on more data can result in more skillful models, and the augmentation techniques can create variations of the images that can improve the ability of the fit models to generalize what they have learned to new images. Data augmentation can also act as a regularization technique, adding noise to the training data and encouraging the model to learn the same features, invariant to their position in the input (data augmentation was introduced in Chapter 9).

Small changes to the input photos of the satellite photos might be useful for this problem, such as horizontal flips, vertical flips, rotations, zooms, and perhaps more. These augmentations can be specified as arguments to the `ImageDataGenerator` instance, used for the training dataset. The augmentations should not be used for the test dataset, as we wish to evaluate the performance of the model on the unmodified photographs. This requires that we have a separate `ImageDataGenerator` instance for the train and test dataset, then iterators for the train and test sets created from the respective data generators. For example:

```
# create data generator
```

```

train_datagen = ImageDataGenerator(rescale=1.0/255.0, horizontal_flip=True,
    vertical_flip=True, rotation_range=90)
test_datagen = ImageDataGenerator(rescale=1.0/255.0)
# prepare iterators
train_it = train_datagen.flow(trainX, trainY, batch_size=128)
test_it = test_datagen.flow(testX, testY, batch_size=128)

```

Listing 22.49: Example of preparing the image data generators for augmentation.

In this case, photos in the training dataset will be augmented with random horizontal and vertical flips as well as random rotations of up to 90 degrees. Photos in both the train and test steps will have their pixel values scaled in the same way as we did for the baseline model. The full code listing of the baseline model with training data augmentation for the planet dataset is listed below for completeness.

```

# baseline model with data augmentation for the planet dataset
import sys
from numpy import load
from matplotlib import pyplot
from sklearn.model_selection import train_test_split
from keras import backend
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD

# load train and test dataset
def load_dataset():
    # load dataset
    data = load('planet_data.npz')
    X, y = data['arr_0'], data['arr_1']
    # separate into train and test datasets
    trainX, testX, trainY, testY = train_test_split(X, y, test_size=0.3, random_state=1)
    print(trainX.shape, trainY.shape, testX.shape, testY.shape)
    return trainX, trainY, testX, testY

# calculate fbeta score for multi-class/label classification
def fbeta(y_true, y_pred, beta=2):
    # clip predictions
    y_pred = backend.clip(y_pred, 0, 1)
    # calculate elements
    tp = backend.sum(backend.round(backend.clip(y_true * y_pred, 0, 1)), axis=1)
    fp = backend.sum(backend.round(backend.clip(y_pred - y_true, 0, 1)), axis=1)
    fn = backend.sum(backend.round(backend.clip(y_true - y_pred, 0, 1)), axis=1)
    # calculate precision
    p = tp / (tp + fp + backend.epsilon())
    # calculate recall
    r = tp / (tp + fn + backend.epsilon())
    # calculate fbeta, averaged across each class
    bb = beta ** 2
    fbeta_score = backend.mean((1 + bb) * (p * r) / (bb * p + r + backend.epsilon()))
    return fbeta_score

```

```
# define cnn model
def define_model(in_shape=(128, 128, 3), out_shape=17):
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same', input_shape=in_shape))
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform',
                    padding='same'))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(out_shape, activation='sigmoid'))
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(optimizer=opt, loss='binary_crossentropy', metrics=[fbeta])
return model

# plot diagnostic learning curves
def summarize_diagnostics(history):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(history.history['loss'], color='blue', label='train')
    pyplot.plot(history.history['val_loss'], color='orange', label='test')
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Fbeta')
    pyplot.plot(history.history['fbeta'], color='blue', label='train')
    pyplot.plot(history.history['val_fbeta'], color='orange', label='test')
    # save plot to file
    filename = sys.argv[0].split('/')[-1]
    pyplot.savefig(filename + '_plot.png')
    pyplot.close()

# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # create data generator
    train_datagen = ImageDataGenerator(rescale=1.0/255.0, horizontal_flip=True,
                                        vertical_flip=True, rotation_range=90)
    test_datagen = ImageDataGenerator(rescale=1.0/255.0)
    # prepare iterators
    train_it = train_datagen.flow(trainX, trainY, batch_size=128)
    test_it = test_datagen.flow(testX, testY, batch_size=128)
    # define model
    model = define_model()
```

```
# fit model
history = model.fit_generator(train_it, steps_per_epoch=len(train_it),
    validation_data=test_it, validation_steps=len(test_it), epochs=200, verbose=0)
# evaluate model
loss, fbeta = model.evaluate_generator(test_it, steps=len(test_it), verbose=0)
print('> loss=% .3f, fbeta=% .3f' % (loss, fbeta))
# learning curves
summarize_diagnostics(history)

# entry point, run the test harness
run_test_harness()
```

Listing 22.50: Example of evaluating the baseline model with data augmentation on the planet dataset.

Running the example first fits the model, then reports the model performance on the hold out test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see a lift in performance of about 0.06 from an F-beta score of about 0.831 for the baseline model to a score of about 0.882 for the baseline model with simple data augmentation. This is a large improvement, larger than we saw with dropout.

```
(28335, 128, 128, 3) (28335, 17) (12144, 128, 128, 3) (12144, 17)
> loss=0.103, fbeta=0.882
```

Listing 22.51: Example output from evaluating the baseline model with data augmentation on the planet dataset.

Reviewing the learning curves, we can see that the overfitting has been dramatically impacted. Learning continues well past 100 epochs, although may show signs of leveling out towards the end of the run. The results suggest that further augmentation or other types of regularization added to this configuration may be helpful. It may be interesting to explore additional image augmentations that may further encourage the learning of features invariant to their position in the input, such as zooms and shifts.

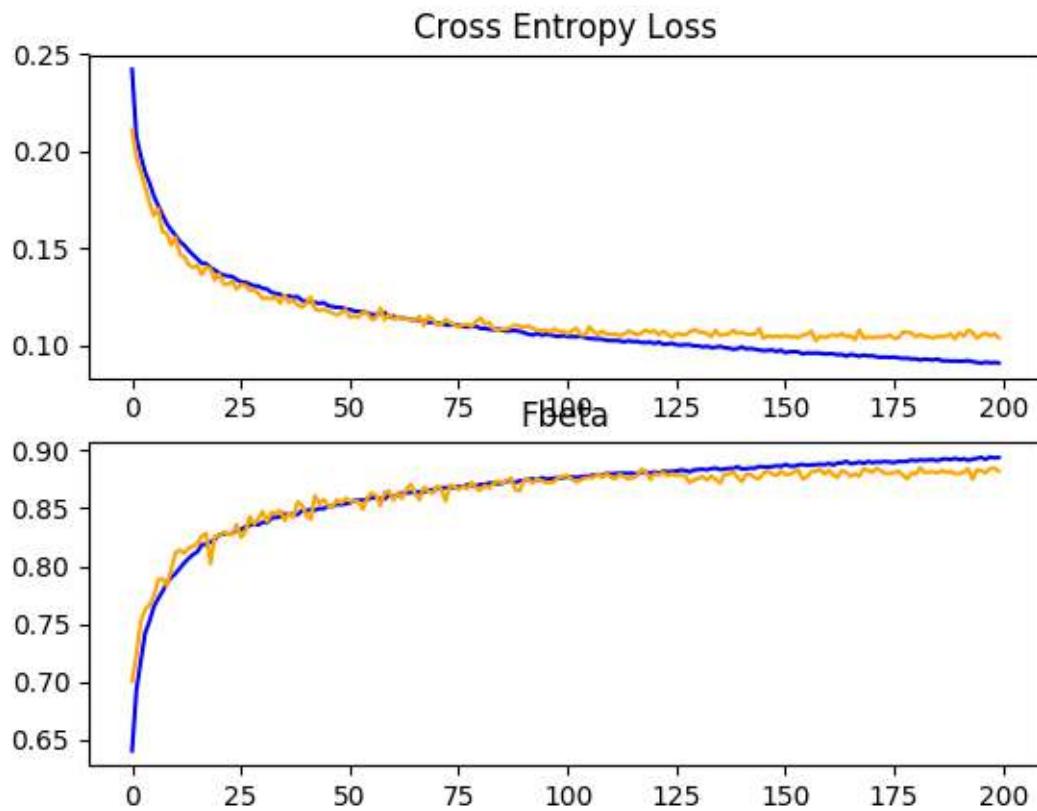


Figure 22.5: Line Plots Showing Loss and F-beta Learning Curves for the Baseline Model With Data Augmentation on the Train and Test Datasets on the Planet Problem.

22.6.3 Discussion

We have explored two different improvements to the baseline model. The results can be summarized below, although we must assume some variance in these results given the stochastic nature of the algorithm:

- **Baseline + Dropout Regularization:** 0.859.
- **Baseline + Data Augmentation:** 0.882.

As suspected, the addition of regularization techniques slows the progression of the learning algorithms and reduces overfitting, resulting in improved performance on the holdout dataset. It is likely that the combination of both approaches with a further increase in the number of training epochs will result in further improvements. That is, the combination of both dropout with data augmentation. This is just the beginning of the types of improvements that can be explored on this dataset. In addition to tweaks to the regularization methods described, other regularization methods could be explored such as weight decay and early stopping. It may be worth exploring changes to the learning algorithm, such as changes to the learning rate, use of a learning rate schedule, or an adaptive learning rate such as Adam. Alternate model

architectures may also be worth exploring. The chosen baseline model offers more capacity than may be required for this problem and a smaller model may faster to train and in turn could result in better performance.

22.7 How to Use Transfer Learning

Transfer learning involves using all or parts of a model trained on a related task. Keras provides a range of pre-trained models that can be loaded and used wholly or partially via the Keras Applications API. A useful model for transfer learning is one of the VGG models, such as VGG-16 with 16 layers that, at the time it was developed, achieved top results on the ImageNet photo classification challenge. The model is comprised of two main parts: the feature extractor part of the model that is made up of VGG blocks, and the classifier part of the model that is made up of fully connected layers and the output layer.

We can use the feature extraction part of the model and add a new classifier part of the model that is tailored to the planets dataset. Specifically, we can hold the weights of all of the convolutional layers fixed during training and only train new fully connected layers that will learn to interpret the features extracted from the model and make a suite of binary classifications. This can be achieved by loading the VGG-16 model, removing the fully connected layers from the output-end of the model, then adding the new fully connected layers to interpret the model output and make a prediction. The classifier part of the model can be removed automatically by setting the `include_top` argument to `False`, which also requires that the shape of the input be specified for the model, in this case `(128, 128, 3)`. This means that the loaded model ends at the last max pooling layer, after which we can manually add a `Flatten` layer and the new classifier fully-connected layers. The `define_model()` function below implements this and returns a new model ready for training.

```
# define cnn model
def define_model(in_shape=(128, 128, 3), out_shape=17):
    # load model
    model = VGG16(include_top=False, input_shape=in_shape)
    # mark loaded layers as not trainable
    for layer in model.layers:
        layer.trainable = False
    # add new classifier layers
    flat1 = Flatten()(model.layers[-1].output)
    class1 = Dense(128, activation='relu', kernel_initializer='he_uniform')(flat1)
    output = Dense(out_shape, activation='sigmoid')(class1)
    # define new model
    model = Model(inputs=model.inputs, outputs=output)
    # compile model
    opt = SGD(lr=0.01, momentum=0.9)
    model.compile(optimizer=opt, loss='binary_crossentropy', metrics=[fbeta])
    return model
```

Listing 22.52: Example of defining a pre-trained model for the planet dataset.

Once created, we can train the model as before on the training dataset. Not a lot of training will be required in this case, as only the new fully connected and output layers have trainable weights. As such, we will fix the number of training epochs at 10. The VGG16 model was trained on a specific ImageNet challenge dataset. As such, the model expects images to be centered.

That is, to have the mean pixel values from each channel (red, green, and blue) as calculated on the ImageNet training dataset subtracted from the input. Keras provides a function to perform this preparation for individual photos via the `preprocess_input()` function. Nevertheless, we can achieve the same effect with the image data generator, by setting the `featurewise_center` argument to `True` and manually specifying the mean pixel values to use when centering as the mean values from the ImageNet training dataset: [123.68, 116.779, 103.939].

```
# create data generator
datagen = ImageDataGenerator(featurewise_center=True)
# specify imagenet mean values for centering
datagen.mean = [123.68, 116.779, 103.939]
```

Listing 22.53: Example of preparing the image data generator for the pre-trained model.

The full code listing of the VGG-16 model for transfer learning on the planet dataset is listed below.

```
# vgg16 transfer learning on the planet dataset
import sys
from numpy import load
from matplotlib import pyplot
from sklearn.model_selection import train_test_split
from keras import backend
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD
from keras.applications.vgg16 import VGG16
from keras.models import Model
from keras.preprocessing.image import ImageDataGenerator

# load train and test dataset
def load_dataset():
    # load dataset
    data = load('planet_data.npz')
    X, y = data['arr_0'], data['arr_1']
    # separate into train and test datasets
    trainX, testX, trainY, testY = train_test_split(X, y, test_size=0.3, random_state=1)
    print(trainX.shape, trainY.shape, testX.shape, testY.shape)
    return trainX, trainY, testX, testY

# calculate fbeta score for multi-class/label classification
def fbeta(y_true, y_pred, beta=2):
    # clip predictions
    y_pred = backend.clip(y_pred, 0, 1)
    # calculate elements
    tp = backend.sum(backend.round(backend.clip(y_true * y_pred, 0, 1)), axis=1)
    fp = backend.sum(backend.round(backend.clip(y_pred - y_true, 0, 1)), axis=1)
    fn = backend.sum(backend.round(backend.clip(y_true - y_pred, 0, 1)), axis=1)
    # calculate precision
    p = tp / (tp + fp + backend.epsilon())
    # calculate recall
    r = tp / (tp + fn + backend.epsilon())
    # calculate fbeta, averaged across each class
    bb = beta ** 2
    fbeta_score = backend.mean((1 + bb) * (p * r) / (bb * p + r + backend.epsilon()))
    return fbeta_score
```

```
# define cnn model
def define_model(in_shape=(128, 128, 3), out_shape=17):
    # load model
    model = VGG16(include_top=False, input_shape=in_shape)
    # mark loaded layers as not trainable
    for layer in model.layers:
        layer.trainable = False
    # add new classifier layers
    flat1 = Flatten()(model.layers[-1].output)
    class1 = Dense(128, activation='relu', kernel_initializer='he_uniform')(flat1)
    output = Dense(out_shape, activation='sigmoid')(class1)
    # define new model
    model = Model(inputs=model.inputs, outputs=output)
    # compile model
    opt = SGD(lr=0.01, momentum=0.9)
    model.compile(optimizer=opt, loss='binary_crossentropy', metrics=[fbeta])
    return model

# plot diagnostic learning curves
def summarize_diagnostics(history):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(history.history['loss'], color='blue', label='train')
    pyplot.plot(history.history['val_loss'], color='orange', label='test')
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Fbeta')
    pyplot.plot(history.history['fbeta'], color='blue', label='train')
    pyplot.plot(history.history['val_fbeta'], color='orange', label='test')
    # save plot to file
    filename = sys.argv[0].split('/')[-1]
    pyplot.savefig(filename + '_plot.png')
    pyplot.close()

# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # create data generator
    datagen = ImageDataGenerator(featurewise_center=True)
    # specify imagenet mean values for centering
    datagen.mean = [123.68, 116.779, 103.939]
    # prepare iterators
    train_it = datagen.flow(trainX, trainY, batch_size=128)
    test_it = datagen.flow(testX, testY, batch_size=128)
    # define model
    model = define_model()
    # fit model
    history = model.fit_generator(train_it, steps_per_epoch=len(train_it),
        validation_data=test_it, validation_steps=len(test_it), epochs=20, verbose=0)
    # evaluate model
    loss, fbeta = model.evaluate_generator(test_it, steps=len(test_it), verbose=0)
    print('> loss=%f, fbeta=%f' % (loss, fbeta))
    # learning curves
```

```
summarize_diagnostics(history)

# entry point, run the test harness
run_test_harness()
```

Listing 22.54: Example of evaluating the pre-trained model on the planet dataset.

Running the example first fits the model, then reports the model performance on the hold out test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved an F-beta score of about 0.860, which is better than the baseline model, but not as good as the baseline model with image data augmentation.

```
(28335, 128, 128, 3) (28335, 17) (12144, 128, 128, 3) (12144, 17)
> loss=0.152, fbeta=0.860
```

Listing 22.55: Example output from evaluating the pre-trained model on the planet dataset.

Reviewing the learning curves, we can see that the model fits the dataset quickly, showing strong overfitting within just a few training epochs. The results suggest that the model could benefit from regularization to address overfitting and perhaps other changes to the model or learning process to slow the rate of improvement.

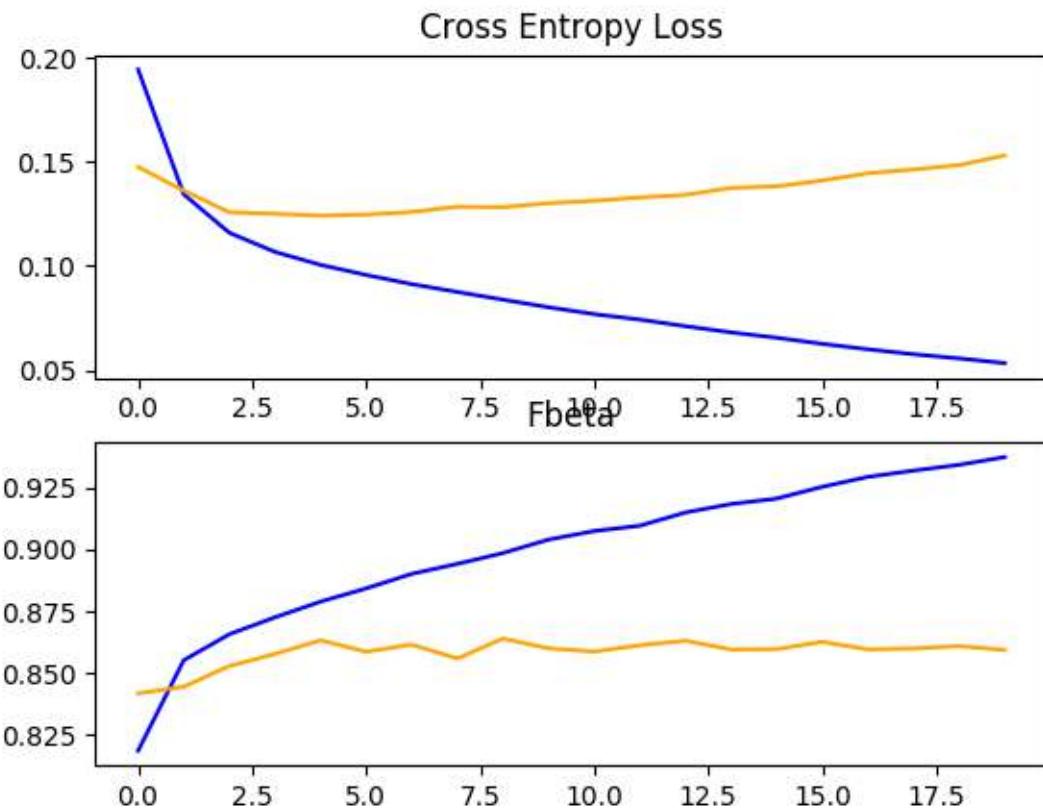


Figure 22.6: Line Plots Showing Loss and F-beta Learning Curves for the VGG-16 Model on the Train and Test Datasets on the Planet Problem.

The VGG-16 model was designed to classify photographs of objects into one of 1,000 categories. As such, it was designed to pick out fine-grained features of objects. We can guess that the features learned by the model by the deeper layers will represent higher order features seen in the ImageNet dataset that may not be directly relevant to the classification of satellite photos of the Amazon rainforest. To address this, we can re-fit the VGG-16 model and allow the training algorithm to fine tune the weights for some of the layers in the model. In this case, we will make the three convolutional layers (and pooling layer for consistency) as trainable. The updated version of the `define_model()` function is listed below.

```
# define cnn model
def define_model(in_shape=(128, 128, 3), out_shape=17):
    # load model
    model = VGG16(include_top=False, input_shape=in_shape)
    # mark loaded layers as not trainable
    for layer in model.layers:
        layer.trainable = False
    # allow last vgg block to be trainable
    model.get_layer('block5_conv1').trainable = True
    model.get_layer('block5_conv2').trainable = True
    model.get_layer('block5_conv3').trainable = True
    model.get_layer('block5_pool').trainable = True
    # add new classifier layers
```

```
flat1 = Flatten()(model.layers[-1].output)
class1 = Dense(128, activation='relu', kernel_initializer='he_uniform')(flat1)
output = Dense(out_shape, activation='sigmoid')(class1)
# define new model
model = Model(inputs=model.inputs, outputs=output)
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(optimizer=opt, loss='binary_crossentropy', metrics=[fbeta])
return model
```

Listing 22.56: Example of defining a pre-trained model with some trainable layers for the planet dataset.

The example of transfer learning with VGG-16 on the planet dataset can then be re-run with this modification.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we see a lift in model performance as compared to the VGG-16 model feature extraction model used as-is improving the F-beta score from about 0.860 to about 0.879. The score is close to the F-beta score seen with the baseline model with the addition of image data augmentation.

```
(28335, 128, 128, 3) (28335, 17) (12144, 128, 128, 3) (12144, 17)
> loss=0.210, fbeta=0.879
```

Listing 22.57: Example output from evaluating the pre-trained model with some trainable layers on the planet dataset.

Reviewing the learning curves, we can see that the model still shows signs of overfitting the training dataset relatively early in the run. The results suggest that perhaps the model could benefit from the use of dropout and/or other regularization methods.

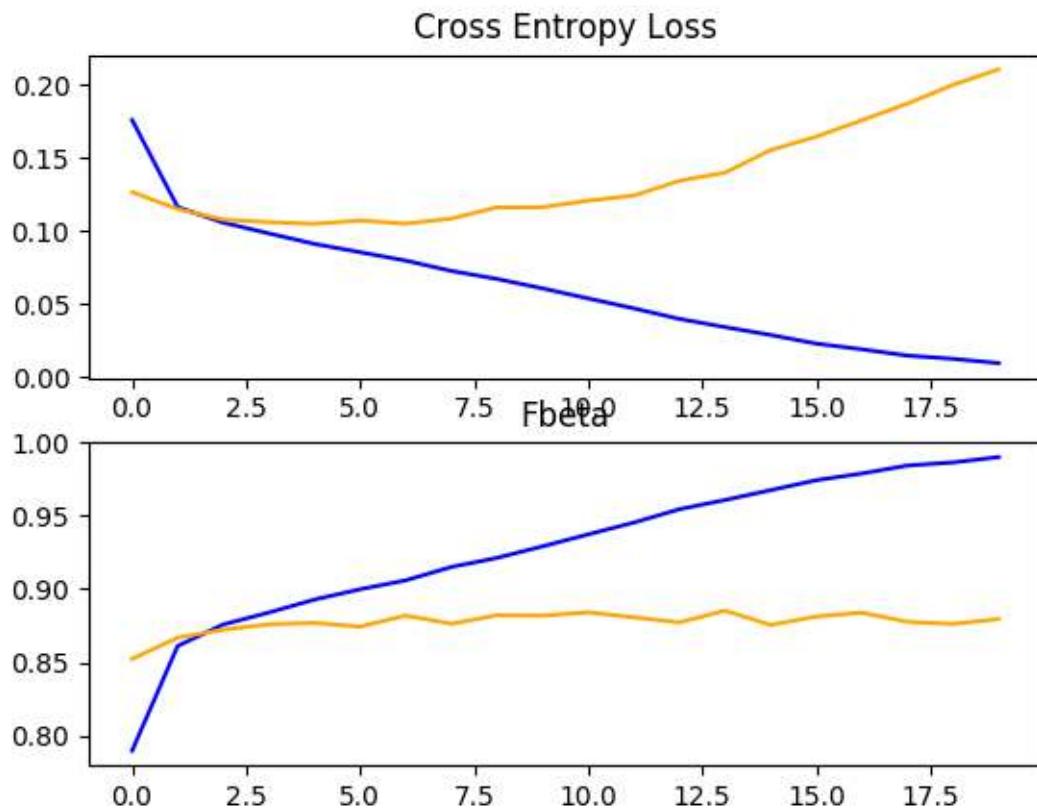


Figure 22.7: Line Plots Showing Loss and F-beta Learning Curves for the VGG-16 Model with Fine Tuning on the Train and Test Datasets on the Planet Problem.

Given that we saw a large improvement with the use of data augmentation on the baseline model, it may be interesting to see if data augmentation can be used to improve the performance of the VGG-16 model with fine-tuning. In this case, the same `define_model()` function can be used, although in this case the `run_test_harness()` can be updated to use image data augmentation as was performed in the previous section. We expect that the addition of data augmentation will slow the rate of improvement. As such we will increase the number of training epochs from 20 to 50 to give the model more time to converge. The complete example of VGG-16 with fine-tuning and data augmentation is listed below.

```
# vgg with fine-tuning and data augmentation for the planet dataset
import sys
from numpy import load
from matplotlib import pyplot
from sklearn.model_selection import train_test_split
from keras import backend
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD
from keras.applications.vgg16 import VGG16
from keras.models import Model
from keras.preprocessing.image import ImageDataGenerator
```

```
# load train and test dataset
def load_dataset():
    # load dataset
    data = load('planet_data.npz')
    X, y = data['arr_0'], data['arr_1']
    # separate into train and test datasets
    trainX, testX, trainY, testY = train_test_split(X, y, test_size=0.3, random_state=1)
    print(trainX.shape, trainY.shape, testX.shape, testY.shape)
    return trainX, trainY, testX, testY

# calculate fbeta score for multi-class/label classification
def fbeta(y_true, y_pred, beta=2):
    # clip predictions
    y_pred = backend.clip(y_pred, 0, 1)
    # calculate elements
    tp = backend.sum(backend.round(backend.clip(y_true * y_pred, 0, 1)), axis=1)
    fp = backend.sum(backend.round(backend.clip(y_pred - y_true, 0, 1)), axis=1)
    fn = backend.sum(backend.round(backend.clip(y_true - y_pred, 0, 1)), axis=1)
    # calculate precision
    p = tp / (tp + fp + backend.epsilon())
    # calculate recall
    r = tp / (tp + fn + backend.epsilon())
    # calculate fbeta, averaged across each class
    bb = beta ** 2
    fbeta_score = backend.mean((1 + bb) * (p * r) / (bb * p + r + backend.epsilon()))
    return fbeta_score

# define cnn model
def define_model(in_shape=(128, 128, 3), out_shape=17):
    # load model
    model = VGG16(include_top=False, input_shape=in_shape)
    # mark loaded layers as not trainable
    for layer in model.layers:
        layer.trainable = False
    # allow last vgg block to be trainable
    model.get_layer('block5_conv1').trainable = True
    model.get_layer('block5_conv2').trainable = True
    model.get_layer('block5_conv3').trainable = True
    model.get_layer('block5_pool').trainable = True
    # add new classifier layers
    flat1 = Flatten()(model.layers[-1].output)
    class1 = Dense(128, activation='relu', kernel_initializer='he_uniform')(flat1)
    output = Dense(out_shape, activation='sigmoid')(class1)
    # define new model
    model = Model(inputs=model.inputs, outputs=output)
    # compile model
    opt = SGD(lr=0.01, momentum=0.9)
    model.compile(optimizer=opt, loss='binary_crossentropy', metrics=[fbeta])
    return model

# plot diagnostic learning curves
def summarize_diagnostics(history):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
```

```

pyplot.plot(history.history['loss'], color='blue', label='train')
pyplot.plot(history.history['val_loss'], color='orange', label='test')
# plot accuracy
pyplot.subplot(212)
pyplot.title('Fbeta')
pyplot.plot(history.history['fbeta'], color='blue', label='train')
pyplot.plot(history.history['val_fbeta'], color='orange', label='test')
# save plot to file
filename = sys.argv[0].split('/')[-1]
pyplot.savefig(filename + '_plot.png')
pyplot.close()

# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # create data generator
    train_datagen = ImageDataGenerator(featurewise_center=True, horizontal_flip=True,
                                       vertical_flip=True, rotation_range=90)
    test_datagen = ImageDataGenerator(featurewise_center=True)
    # specify imagenet mean values for centering
    train_datagen.mean = [123.68, 116.779, 103.939]
    test_datagen.mean = [123.68, 116.779, 103.939]
    # prepare iterators
    train_it = train_datagen.flow(trainX, trainY, batch_size=128)
    test_it = test_datagen.flow(testX, testY, batch_size=128)
    # define model
    model = define_model()
    # fit model
    history = model.fit_generator(train_it, steps_per_epoch=len(train_it),
                                  validation_data=test_it, validation_steps=len(test_it), epochs=50, verbose=0)
    # evaluate model
    loss, fbeta = model.evaluate_generator(test_it, steps=len(test_it), verbose=0)
    print('> loss=%f, fbeta=%f' % (loss, fbeta))
    # learning curves
    summarize_diagnostics(history)

# entry point, run the test harness
run_test_harness()

```

Listing 22.58: Example of evaluating the pre-trained model with fine tuning on the planet dataset.

Running the example first fits the model, then reports the model performance on the hold out test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see yet another further lift in model performance from an F-beta score of about 0.879 to an F-beta score of about 0.891.

```
(28335, 128, 128, 3) (28335, 17) (12144, 128, 128, 3) (12144, 17)
> loss=0.100, fbeta=0.891
```

Listing 22.59: Example output from evaluating the pre-trained model with fine tuning on the planet dataset.

Reviewing the learning curves, we can see that data augmentation again has had a large impact on model overfitting, in this case stabilizing the learning and delaying overfitting perhaps until epoch 20.

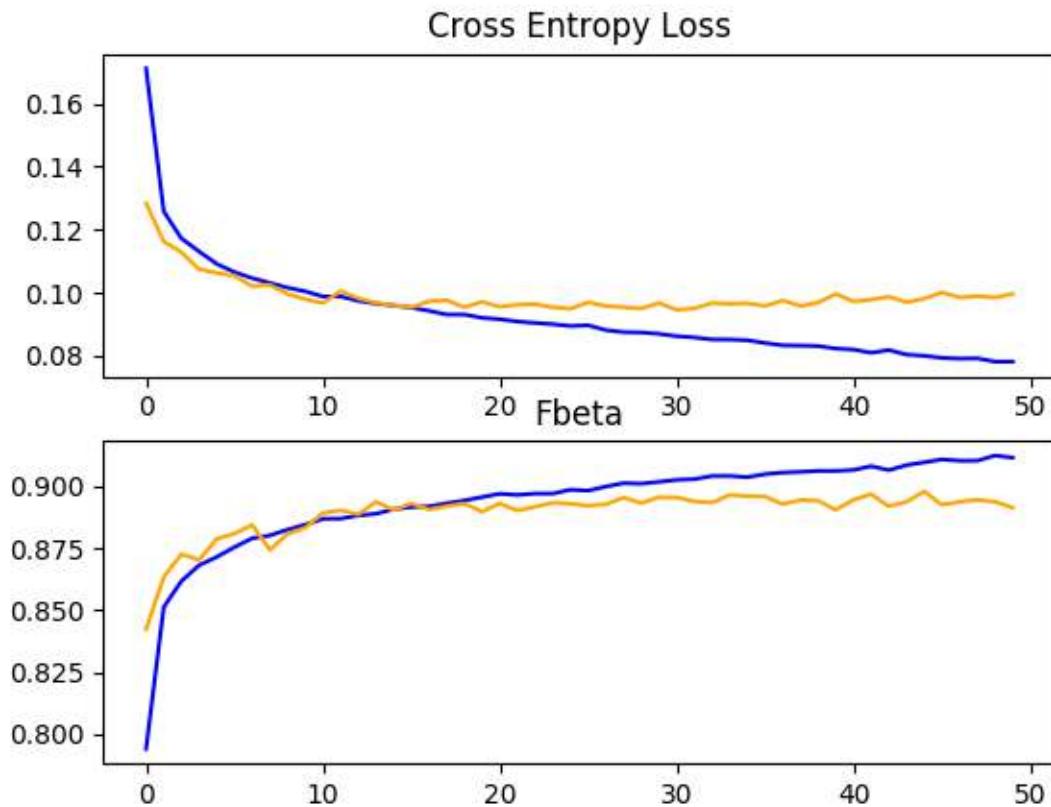


Figure 22.8: Line Plots Showing Loss and F-beta Learning Curves for the VGG-16 Model With fine-tuning and Data Augmentation on the Train and Test Datasets on the Planet Problem.

22.7.1 Discussion

We have explored three different cases of transfer learning in this section. The results can be summarized below, although we must assume some variance in these results given the stochastic nature of the learning algorithm:

- **VGG-16 Model:** 0.860.
- **VGG-16 Model + fine-tuning:** 0.879.
- **VGG-16 Model + fine-tuning + Data Augmentation:** 0.891.

The choice of the VGG-16 model was somewhat arbitrary, given that it is a smaller and well-understood model. Other models could be used as the basis for transfer learning, such as ResNet, that may achieve better performance. Further, more fine-tuning may also result in better performance. This might include tuning the weights of more of the feature extractor

layers, perhaps with a smaller learning rate. This might also include the modification of the model to add regularization, such as dropout.

22.8 How to Finalize the Model and Make Predictions

The process of model improvement may continue for as long as we have ideas and the time and resources to test them out. At some point, a final model configuration must be chosen and adopted. In this case, we will keep things simple and use the VGG-16 transfer learning, fine-tuning, and data augmentation as the final model. First, we will finalize our model by fitting a model on the entire training dataset and saving the model to file for later use. We will then load the saved model and use it to make a prediction on a single image.

22.8.1 Save Final Model

The first step is to fit a final model on the entire training dataset. The `load_dataset()` function can be updated to no longer split the loaded dataset into train and test sets.

```
# load train and test dataset
def load_dataset():
    # load dataset
    data = load('planet_data.npz')
    X, y = data['arr_0'], data['arr_1']
    return X, y
```

Listing 22.60: Example of loading the dataset for training the final model.

The `define_model()` function can be used as was defined in the previous section for the VGG-16 model with fine-tuning and data augmentation.

```
# define cnn model
def define_model(in_shape=(128, 128, 3), out_shape=17):
    # load model
    model = VGG16(include_top=False, input_shape=in_shape)
    # mark loaded layers as not trainable
    for layer in model.layers:
        layer.trainable = False
    # allow last vgg block to be trainable
    model.get_layer('block5_conv1').trainable = True
    model.get_layer('block5_conv2').trainable = True
    model.get_layer('block5_conv3').trainable = True
    model.get_layer('block5_pool').trainable = True
    # add new classifier layers
    flat1 = Flatten()(model.layers[-1].output)
    class1 = Dense(128, activation='relu', kernel_initializer='he_uniform')(flat1)
    output = Dense(out_shape, activation='sigmoid')(class1)
    # define new model
    model = Model(inputs=model.inputs, outputs=output)
    # compile model
    opt = SGD(lr=0.01, momentum=0.9)
    model.compile(optimizer=opt, loss='binary_crossentropy')
    return model
```

Listing 22.61: Example of a function for defining the final model.

Finally, we only require a single data generator and a single iterator for the training dataset.

```
# create data generator
datagen = ImageDataGenerator(featurewise_center=True, horizontal_flip=True,
    vertical_flip=True, rotation_range=90)
# specify imagenet mean values for centering
datagen.mean = [123.68, 116.779, 103.939]
# prepare iterator
train_it = datagen.flow(X, y, batch_size=128)
```

Listing 22.62: Example of preparing the image data generator for training the final model.

The model will be fit for 50 epochs, after which it will be saved to an `h5` file via a call to the `save()` function on the model.

```
# fit model
model.fit_generator(train_it, steps_per_epoch=len(train_it), epochs=50, verbose=0)
# save model
model.save('final_model.h5')
```

Listing 22.63: Example of fitting and saving the final model.

Note: saving and loading a Keras model requires that the `h5py` library is installed on your workstation. The complete example of fitting the final model on the training dataset and saving it to file is listed below.

```
# save the final model to file
from numpy import load
from keras.preprocessing.image import ImageDataGenerator
from keras.applications.vgg16 import VGG16
from keras.models import Model
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD

# load train and test dataset
def load_dataset():
    # load dataset
    data = load('planet_data.npz')
    X, y = data['arr_0'], data['arr_1']
    return X, y

# define cnn model
def define_model(in_shape=(128, 128, 3), out_shape=17):
    # load model
    model = VGG16(include_top=False, input_shape=in_shape)
    # mark loaded layers as not trainable
    for layer in model.layers:
        layer.trainable = False
    # allow last vgg block to be trainable
    model.get_layer('block5_conv1').trainable = True
    model.get_layer('block5_conv2').trainable = True
    model.get_layer('block5_conv3').trainable = True
    model.get_layer('block5_pool').trainable = True
    # add new classifier layers
    flat1 = Flatten()(model.layers[-1].output)
    class1 = Dense(128, activation='relu', kernel_initializer='he_uniform')(flat1)
```

```
output = Dense(out_shape, activation='sigmoid')(class1)
# define new model
model = Model(inputs=model.inputs, outputs=output)
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(optimizer=opt, loss='binary_crossentropy')
return model

# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    X, y = load_dataset()
    # create data generator
    datagen = ImageDataGenerator(featurewise_center=True, horizontal_flip=True,
        vertical_flip=True, rotation_range=90)
    # specify imagenet mean values for centering
    datagen.mean = [123.68, 116.779, 103.939]
    # prepare iterator
    train_it = datagen.flow(X, y, batch_size=128)
    # define model
    model = define_model()
    # fit model
    model.fit_generator(train_it, steps_per_epoch=len(train_it), epochs=50, verbose=0)
    # save model
    model.save('final_model.h5')

# entry point, run the test harness
run_test_harness()
```

Listing 22.64: Example of fitting and saving the final model.

After running this example you will now have a large 91-megabyte file with the name `final_model.h5` in your current working directory.

22.8.2 Make a Prediction

We can use our saved model to make a prediction on new images. The model assumes that new images are color, and that they have been split into squares with the size of 256×256 . Below is an image extracted from the training dataset, specifically the file `train_1.jpg`.



Figure 22.9: Sample Satellite Image of Amazon Rain Forest For Prediction.

Copy it from your training data directory to the current working directory with the name `sample_image.jpg`, for example:

```
cp train-jpg/train_1.jpg ./sample_image.jpg
```

Listing 22.65: Command to copy a training image for prediction.

According to the mapping file for the training dataset, this file has the tags (in no specific order):

- agriculture
- clear
- primary
- water

We will pretend this is an entirely new and unseen image, prepared in the required way, and see how we might use our saved model to predict the tags that the image represents. First, we can load the image and force it to the size to be 128×128 pixels. The loaded image can then be resized to have a single sample in a dataset. The pixel values must also be centered to match the way that the data was prepared during the training of the model. The `load_image()` function implements this and will return the loaded image ready for classification.

```
# load and prepare the image
def load_image(filename):
    # load the image
    img = load_img(filename, target_size=(128, 128))
    # convert to array
    img = img_to_array(img)
    # reshape into a single sample with 3 channels
    img = img.reshape(1, 128, 128, 3)
    # center pixel data
    img = img.astype('float32')
    img = img - [123.68, 116.779, 103.939]
    return img
```

Listing 22.66: Example of a function to load and prepare an image for making a prediction.

Next, we can load the model as in the previous section and call the `predict()` function to predict the content in the image.

```
...
# predict the class
result = model.predict(img)
```

Listing 22.67: Example of making a prediction with a prepared image.

This will return a 17-element vector with floating point values between 0 and 1 that could be interpreted as probabilities of the model's confidence that the photo could be tagged with each known tag. We can round these probabilities to either 0 or 1 and then use our reverse mapping prepared back in the first section in the `create_tag_mapping()` function to convert the vector indexes that have a 1 value to tags for the image. The `prediction_to_tags()` function below implements this, taking the inverse mapping of integers to tags and the vector predicted by the model for the photo and returning a list of predicted tags.

```
# convert a prediction to tags
def prediction_to_tags(inv_mapping, prediction):
    # round probabilities to {0, 1}
    values = prediction.round()
    # collect all predicted tags
    tags = [inv_mapping[i] for i in range(len(values)) if values[i] == 1.0]
    return tags
```

Listing 22.68: Example of a function for interpreting a one hot encoded prediction.

We can tie all of this together and make a prediction for the new photo. The complete example is listed below.

```
# make a prediction for a new image
from pandas import read_csv
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.models import load_model

# create a mapping of tags to integers given the loaded mapping file
def create_tag_mapping(mapping_csv):
    # create a set of all known tags
    labels = set()
    for i in range(len(mapping_csv)):
        # convert spaced separated tags into an array of tags
        tags = mapping_csv['tags'][i].split(' ')
        # add tags to the set of known labels
        labels.update(tags)
    # convert set of labels to a list to list
    labels = list(labels)
    # order set alphabetically
    labels.sort()
    # dict that maps labels to integers, and the reverse
    labels_map = {labels[i]:i for i in range(len(labels))}
    inv_labels_map = {i:labels[i] for i in range(len(labels))}
    return labels_map, inv_labels_map
```

```

# convert a prediction to tags
def prediction_to_tags(inv_mapping, prediction):
    # round probabilities to {0, 1}
    values = prediction.round()
    # collect all predicted tags
    tags = [inv_mapping[i] for i in range(len(values)) if values[i] == 1.0]
    return tags

# load and prepare the image
def load_image(filename):
    # load the image
    img = load_img(filename, target_size=(128, 128))
    # convert to array
    img = img_to_array(img)
    # reshape into a single sample with 3 channels
    img = img.reshape(1, 128, 128, 3)
    # center pixel data
    img = img.astype('float32')
    img = img - [123.68, 116.779, 103.939]
    return img

# load an image and predict the class
def run_example(inv_mapping):
    # load the image
    img = load_image('sample_image.jpg')
    # load model
    model = load_model('final_model.h5')
    # predict the class
    result = model.predict(img)
    print(result[0])
    # map prediction to tags
    tags = prediction_to_tags(inv_mapping, result[0])
    print(tags)

# load the mapping file
filename = 'train_v2.csv'
mapping_csv = read_csv(filename)
# create a mapping of tags to integers
_, inv_mapping = create_tag_mapping(mapping_csv)
# entry point, run the example
run_example(inv_mapping)

```

Listing 22.69: Example of making a prediction with the final model.

Running the example first loads and prepares the image, loads the model, and then makes a prediction. First, the raw 17-element prediction vector is printed. If we wish, we could pretty-print this vector and summarize the predicted confidence that the photo would be assigned each label. Next, the prediction is rounded and the vector indexes that contain a 1 value are reverse-mapped to their tag string values. The predicted tags are then printed. we can see that the model has correctly predicted the known tags for the provided photo. It might be interesting to repeat this test with an entirely new photo, such as a photo from the test dataset, after you have already manually suggested tags.

[9.0940112e-01 3.6541668e-03 1.5959743e-02 6.8241461e-05 8.5694155e-05]

```
9.9828100e-01 7.4096164e-08 5.5998818e-05 3.6668104e-01 1.2538023e-01  
4.6371704e-04 3.7660234e-04 9.9999273e-01 1.9014676e-01 5.6060363e-04  
1.4613305e-03 9.5227945e-01]  
['agriculture', 'clear', 'primary', 'water']
```

Listing 22.70: Example output from making a prediction with the final model.

22.9 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Tune Learning Rate.** Explore changes to the learning algorithm used to train the baseline model, such as alternate learning rate, a learning rate schedule, or an adaptive learning rate algorithm such as Adam.
- **Regularize Transfer Learning Model.** Explore the addition of further regularization techniques to the transfer learning such as early stopping, dropout, weight decay, and more and compare results.
- **Test-Time Automation.** Update the model to use augmentations during prediction, such as flips, rotations, and/or crops to see if prediction performance on the test dataset can be further improved.

If you explore any of these extensions, I'd love to know.

22.10 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

22.10.1 API

- Keras image utils source code.
https://github.com/keras-team/keras-preprocessing/blob/master/keras_preprocessing/image/utils.py
- Keras Applications API.
<https://keras.io/applications/>
- Keras Image Processing API.
<https://keras.io/preprocessing/image/>
- Keras Sequential Model API.
<https://keras.io/models/sequential/>
- `sklearn.metrics.fbeta_score` API.
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.fbeta_score.html

- `sklearn.model_selection.train_test_split` API.
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html
- Old version of Keras metrics source code (with `fbeta_score`).
<https://github.com/keras-team/keras/blob/4fa7e5d454dd4f3f33f1d756a2a8659f2e789141/keras/metrics.py#L134>
- F-beta score for Keras, Kaggle Kernel.
<https://www.kaggle.com/arsenyinfo/f-beta-score-for-keras>
- `numpy.savetxt_compressed` API.
https://docs.scipy.org/doc/numpy/reference/generated/numpy.savetxt_compressed.html

22.10.2 Articles

- Planet: Understanding the Amazon from Space, Kaggle Competition.
<https://www.kaggle.com/c/planet-understanding-the-amazon-from-space>
- Download the Planet Dataset, Kaggle.
<https://www.kaggle.com/c/planet-understanding-the-amazon-from-space/data>
- Planet Competition Model Evaluation, Kaggle.
<https://www.kaggle.com/c/planet-understanding-the-amazon-from-space#evaluation>
- Planet: Understanding the Amazon from Space, 1st Place Winner's Interview, 2017.
<https://goo.gl/DYMEL3>
- F1 score, Wikipedia.
https://en.wikipedia.org/wiki/F1_score
- Precision and recall, Wikipedia.
https://en.wikipedia.org/wiki/Precision_and_recall

22.11 Summary

In this tutorial, you discovered how to develop a convolutional neural network to classify satellite photos of the Amazon tropical rainforest. Specifically, you learned:

- How to load and prepare satellite photos of the Amazon tropical rainforest for modeling.
- How to develop a convolutional neural network for photo classification from scratch and improve model performance.
- How to develop a final model and use it to make predictions on new data.

22.11.1 Next

This was the final tutorial in this part on image classification. In the next part you will discover how to develop models for object recognition.

Part VI

Object Detection

Overview

In this part you will discover the problem of object recognition and how to develop deep learning models for object detection, such as YOLO and the Region-Based CNN. After reading the chapters in this part, you will know:

- The computer vision task of object recognition comprised of object detection and object localization (Chapter [23](#)).
- The YOLO family of models and how to use a pre-trained model for object recognition on new photographs (Chapter [24](#)).
- The R-CNN family of models and how to use a pre-trained model for object recognition on new photographs (Chapter [25](#)).
- How to develop an object detection model for a new dataset using transfer learning (Chapter [26](#)).

Chapter 23

Deep Learning for Object Recognition

It can be challenging for beginners to distinguish between different related computer vision tasks. For example, image classification is straightforward, but the differences between object localization and object detection can be confusing, especially when all three tasks may be just as equally referred to as object recognition. Image classification involves assigning a class label to an image, whereas object localization involves drawing a bounding box around one or more objects in an image. Object detection is more challenging and combines these two tasks and draws a bounding box around each object of interest in the image and assigns them a class label. Together, all of these problems are referred to as object recognition. In this tutorial, you will discover a gentle introduction to the problem of object recognition and state-of-the-art deep learning models designed to address it. After reading this tutorial, you will know:

- Object recognition refers to a collection of related tasks for identifying objects in digital photographs.
- Region-Based Convolutional Neural Networks, or R-CNNs, are a family of techniques for addressing object localization and recognition tasks, designed for model performance.
- You Only Look Once, or YOLO, is a second family of techniques for object recognition designed for speed and real-time use.

Let's get started.

23.1 Overview

This tutorial is divided into three parts; they are:

1. What is Object Recognition?
2. R-CNN Model Family
3. YOLO Model Family

23.2 What is Object Recognition?

Object recognition is a general term to describe a collection of related computer vision tasks that involve identifying objects in digital photographs. Image classification typically involves predicting the class of one object in an image. Object localization refers to identifying the location of one or more objects in an image and drawing a bounding box around their extent. Object detection combines these two tasks and localizes and classifies one or more objects in an image. When a user or practitioner refers to *object recognition*, they often mean *object detection*.

... we will be using the term object recognition broadly to encompass both image classification (a task requiring an algorithm to determine what object classes are present in the image) as well as object detection (a task requiring an algorithm to localize all objects present in the image)

— *ImageNet Large Scale Visual Recognition Challenge*, 2015.

As such, we can distinguish between these three computer vision tasks:

- **Image Classification:** Predict the type or class of an object in an image.
 - **Input:** An image with a single object, such as a photograph.
 - **Output:** A class label (e.g. one or more integers that are mapped to class labels).
- **Object Localization:** Locate the presence of objects in an image and indicate their location with a bounding box.
 - **Input:** An image with one or more objects, such as a photograph.
 - **Output:** One or more bounding boxes (e.g. defined by a point, width, and height).
- **Object Detection:** Locate the presence of objects with a bounding box and types or classes of the located objects in an image.
 - **Input:** An image with one or more objects, such as a photograph.
 - **Output:** One or more bounding boxes (e.g. defined by a point, width, and height), and a class label for each bounding box.

One further extension to this breakdown of computer vision tasks is object segmentation, also called *object instance segmentation* or *semantic segmentation*, where instances of recognized objects are indicated by highlighting the specific pixels of the object instead of a coarse bounding box. From this breakdown, we can see that object recognition refers to a suite of challenging computer vision tasks.

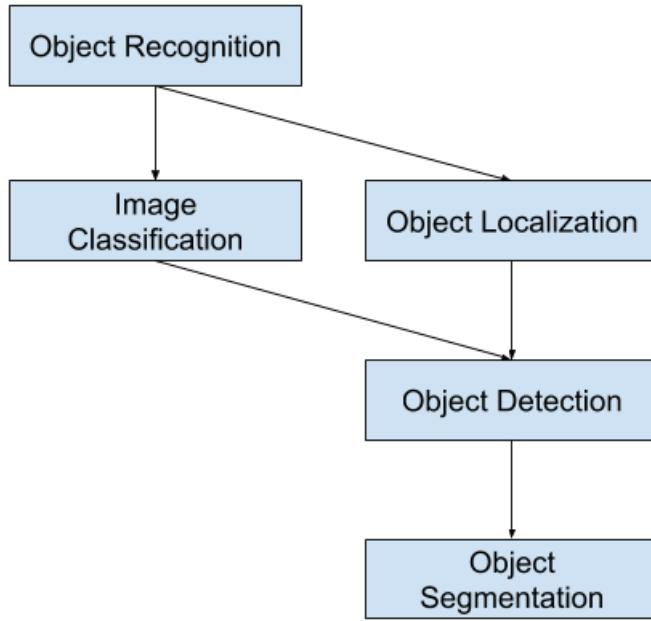


Figure 23.1: Overview of Object Recognition Computer Vision Tasks.

Most of the recent innovations in image recognition problems have come as part of participation in the ILSVRC tasks. This is an annual competition with a separate challenge for each of these three problem types, with the intent of fostering independent and separate improvements at each level that can be leveraged more broadly. For example, see the list of the three corresponding task types below taken from the 2015 ILSVRC review paper:

- **Image classification:** Algorithms produce a list of object categories present in the image.
- **Single-object localization:** Algorithms produce a list of object categories present in the image, along with an axis-aligned bounding box indicating the position and scale of one instance of each object category.
- **Object detection:** Algorithms produce a list of object categories present in the image along with an axis-aligned bounding box indicating the position and scale of every instance of each object category.

We can see that *Single-object localization* is a simpler version of the more broadly defined *Object Localization*, constraining the localization tasks to objects of one type within an image, which we may assume is an easier task. Below is an example comparing single object localization and object detection, taken from the ILSVRC paper. Note the difference in ground truth expectations in each case.

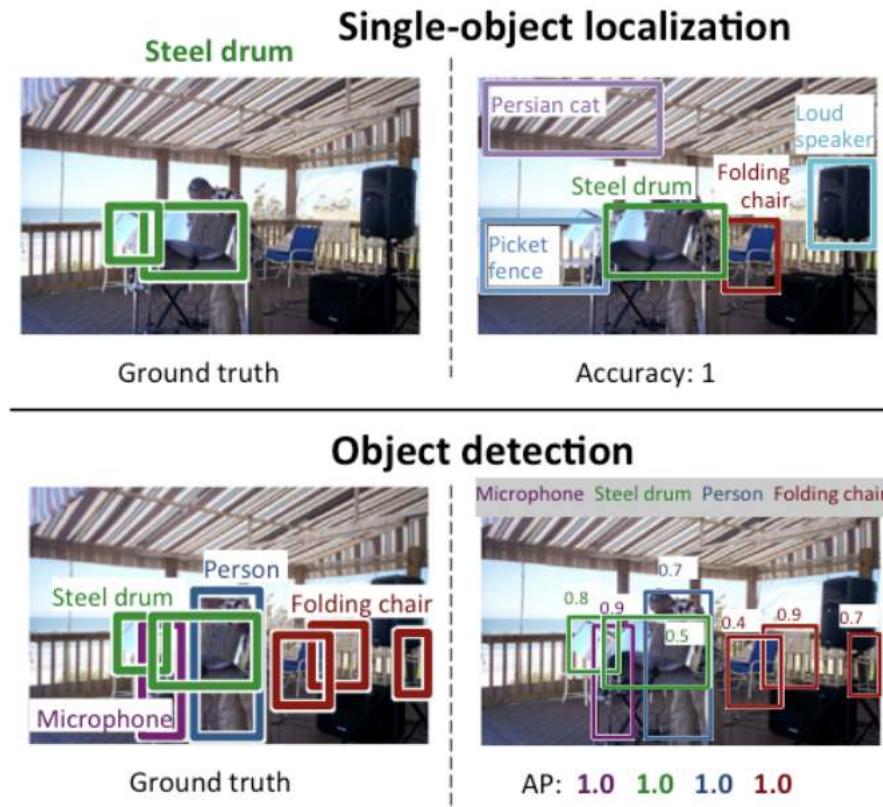


Figure 23.2: Comparison Between Single Object Localization and Object Detection. Taken from the ImageNet paper.

The performance of a model for image classification is evaluated using the mean classification error across the predicted class labels. The performance of a model for single-object localization is evaluated using the distance between the expected and predicted bounding box for the expected class. Whereas the performance of a model for object recognition is evaluated using the precision and recall across each of the best matching bounding boxes for the known objects in the image. Now that we are familiar with the problem of object localization and detection, let's take a look at some recent top-performing deep learning models.

23.3 R-CNN Model Family

The R-CNN family of methods refers to the R-CNN, which may stand for *Regions with CNN Features* or *Region-Based Convolutional Neural Network*, developed by Ross Girshick, et al. This includes the techniques R-CNN, Fast R-CNN, and Faster-RCNN designed and demonstrated for object localization and object recognition. Let's take a closer look at the highlights of each of these techniques in turn.

23.3.1 R-CNN

The R-CNN was described in the 2014 paper by Ross Girshick, et al. from UC Berkeley titled *Rich Feature Hierarchies For Accurate Object Detection And Semantic Segmentation*. It may have been one of the first large and successful application of convolutional neural networks to the

problem of object localization, detection, and segmentation. The approach was demonstrated on benchmark datasets, achieving then state-of-the-art results on the VOC-2012 dataset and the 200-class ILSVRC-2013 object detection dataset. Their proposed R-CNN model is comprised of three modules; they are:

- **Module 1: Region Proposal.** Generate and extract category-independent region proposals, e.g. candidate bounding boxes.
- **Module 2: Feature Extractor.** Extract features from each candidate region, e.g. using a deep convolutional neural network.
- **Module 3: Classifier.** Classify features as one of the known classes, e.g. linear SVM classifier model.

The architecture of the model is summarized in the image below, taken from the paper.

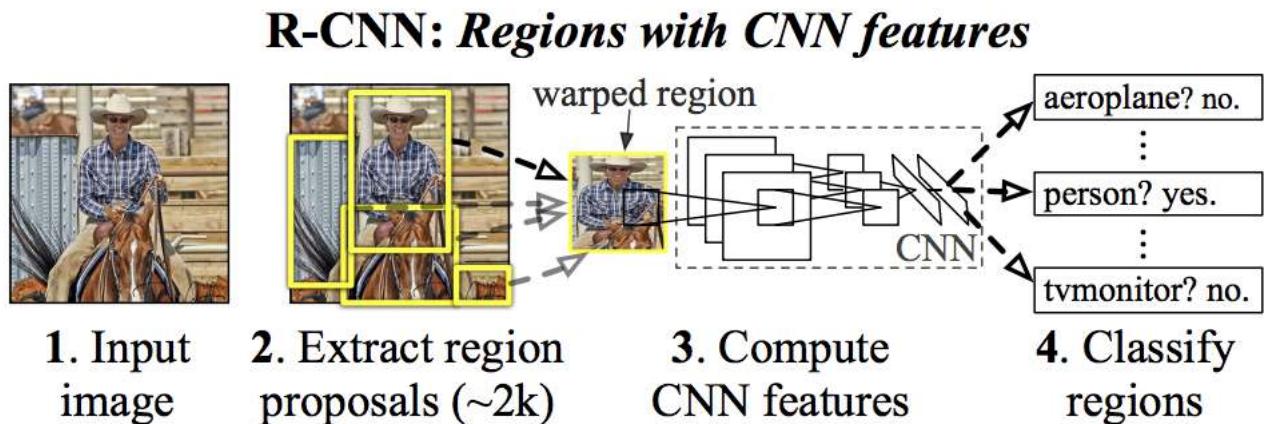


Figure 23.3: Summary of the R-CNN Model Architecture. Taken from the 2014 R-CNN paper.

A computer vision technique is used to propose candidate regions or bounding boxes of potential objects in the image called *selective search*¹, although the flexibility of the design allows other region proposal algorithms to be used. The feature extractor used by the model was the AlexNet deep CNN that won the ILSVRC-2012 image classification competition. The output of the CNN was a 4,096 element vector that describes the contents of the image that is fed to a linear SVM for classification, specifically one SVM is trained for each known class. It is a relatively simple and straightforward application of CNNs to the problem of object localization and recognition. A downside of the approach is that it is slow, requiring a CNN-based feature extraction pass on each of the candidate regions generated by the region proposal algorithm. This is a problem as the paper describes the model operating upon approximately 2,000 proposed regions per image at test-time. Python (Caffe) and MatLab source code for R-CNN as described in the paper was made available in the R-CNN GitHub repository.

¹<https://link.springer.com/article/10.1007/s11263-013-0620-5>

23.3.2 Fast R-CNN

Given the great success of R-CNN, Ross Girshick, then at Microsoft Research, proposed an extension to address the speed issues of R-CNN in a 2015 paper titled *Fast R-CNN*. The paper opens with a review of the limitations of R-CNN, which can be summarized as follows:

- **Training is a multi-stage pipeline.** Involves the preparation and operation of three separate models.
- **Training is expensive in space and time.** Training a deep CNN on so many region proposals per image is very slow.
- **Object detection is slow.** Make predictions using a deep CNN on so many region proposals is very slow.

A prior work was proposed to speed up the technique called spatial pyramid pooling networks, or SPPnets, in the 2014 paper *Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition*. This did speed up the extraction of features, but essentially used a type of forward pass caching algorithm. Fast R-CNN is proposed as a single model instead of a pipeline to learn and output regions and classifications directly. The architecture of the model takes the photograph and a set of region proposals as input that are passed through a deep convolutional neural network. A pre-trained CNN, such as a VGG-16, is used for feature extraction. The end of the deep CNN is a custom layer called a Region of Interest Pooling Layer, or RoI Pooling, that extracts features specific for a given input candidate region. The output of the CNN is then interpreted by a fully connected layer then the model bifurcates into two outputs, one for the class prediction via a softmax layer, and another with a linear output for the bounding box. This process is then repeated multiple times for each region of interest in a given image. The architecture of the model is summarized in the image below, taken from the paper.

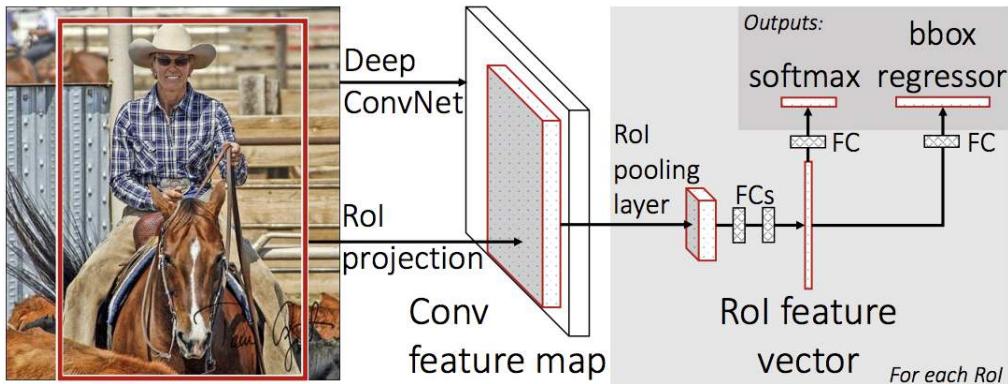


Figure 23.4: Summary of the Fast R-CNN Model Architecture. Taken from the Fast R-CNN paper.

The model is significantly faster to train and to make predictions, yet still requires a set of candidate regions to be proposed along with each input image. Python and C++ (Caffe) source code for Fast R-CNN as described in the paper was made available in a GitHub repository.

23.3.3 Faster R-CNN

The model architecture was further improved for both speed of training and detection by Shaoqing Ren, et al. at Microsoft Research in the 2016 paper titled *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. The architecture was the basis for the first-place results achieved on both the ILSVRC-2015 and MSCOCO-2015 object recognition and detection competition tasks. The architecture was designed to both propose and refine region proposals as part of the training process, referred to as a Region Proposal Network, or RPN. These regions are then used in concert with a Fast R-CNN model in a single model design. These improvements both reduce the number of region proposals and accelerate the test-time operation of the model to near real-time with then state-of-the-art performance.

... our detection system has a frame rate of 5fps (including all steps) on a GPU, while achieving state-of-the-art object detection accuracy on PASCAL VOC 2007, 2012, and MSCOCO datasets with only 300 proposals per image. In ILSVRC and MSCOCO 2015 competitions, Faster R-CNN and RPN are the foundations of the 1st-place winning entries in several tracks

— *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*, 2016.

Although it is a single unified model, the architecture is comprised of two modules:

- **Module 1: Region Proposal Network.** Convolutional neural network for proposing regions and the type of object to consider in the region.
- **Module 2: Fast R-CNN.** Convolutional neural network for extracting features from the proposed regions and outputting the bounding box and class labels.

Both modules operate on the same output of a deep CNN. The region proposal network acts as an attention mechanism for the Fast R-CNN network, informing the second network of where to look or pay attention. The architecture of the model is summarized in the image below, taken from the paper.

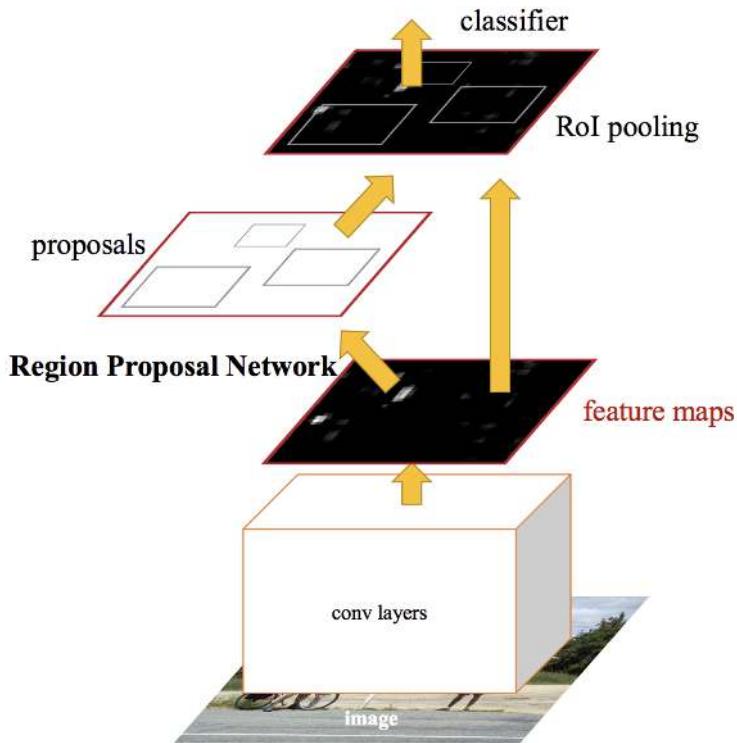


Figure 23.5: Summary of the Faster R-CNN Model Architecture. Taken from the Faster R-CNN paper.

The RPN works by taking the output of a pre-trained deep CNN, such as VGG-16, and passing a small network over the feature map and outputting multiple region proposals and a class prediction for each. Region proposals are bounding boxes, based on so-called anchor boxes or pre-defined shapes designed to accelerate and improve the proposal of regions. The class prediction is binary, indicating the presence of an object, or not, so-called *objectness* of the proposed region. A procedure of alternating training is used where both sub-networks are trained at the same time, although interleaved. This allows the parameters in the feature detector deep CNN to be tailored or fine-tuned for both tasks at the same time.

At the time of writing, this Faster R-CNN architecture is the pinnacle of the R-CNN family of models and continues to achieve near state-of-the-art results on object recognition tasks. A further extension adds support for image segmentation, described in the paper 2017 paper *Mask R-CNN*. Python and C++ (Caffe) source code for Fast R-CNN as described in the paper was made available in a GitHub repository.

23.4 YOLO Model Family

Another popular family of object recognition models is referred to collectively as YOLO or *You Only Look Once*, developed by Joseph Redmon, et al. The R-CNN models may be generally more accurate, yet the YOLO family of models are fast, much faster than R-CNN, achieving object detection in real-time.

23.4.1 YOLO

The YOLO model was first described by Joseph Redmon, et al. in the 2015 paper titled *You Only Look Once: Unified, Real-Time Object Detection*. Note that Ross Girshick, developer of R-CNN, was also an author and contributor to this work, then at Facebook AI Research. The approach involves a single neural network trained end-to-end that takes a photograph as input and predicts bounding boxes and class labels for each bounding box directly. The technique offers lower predictive accuracy (e.g. more localization errors), although operates at 45 frames per second and up to 155 frames per second for a speed-optimized version of the model.

Our unified architecture is extremely fast. Our base YOLO model processes images in real-time at 45 frames per second. A smaller version of the network, Fast YOLO, processes an astounding 155 frames per second ...

— *You Only Look Once: Unified, Real-Time Object Detection*, 2015.

The model works by first splitting the input image into a grid of cells, where each cell is responsible for predicting a bounding box if the center of a bounding box falls within it. Each grid cell predicts a bounding box involving the x , y coordinate and the width and height and the confidence. A class prediction is also based on each cell. For example, an image may be divided into a 7×7 grid and each cell in the grid may predict 2 bounding boxes, resulting in 98 proposed bounding box predictions. The class probabilities map and the bounding boxes with confidences are then combined into a final set of bounding boxes and class labels. The image taken from the paper below summarizes the two outputs of the model.

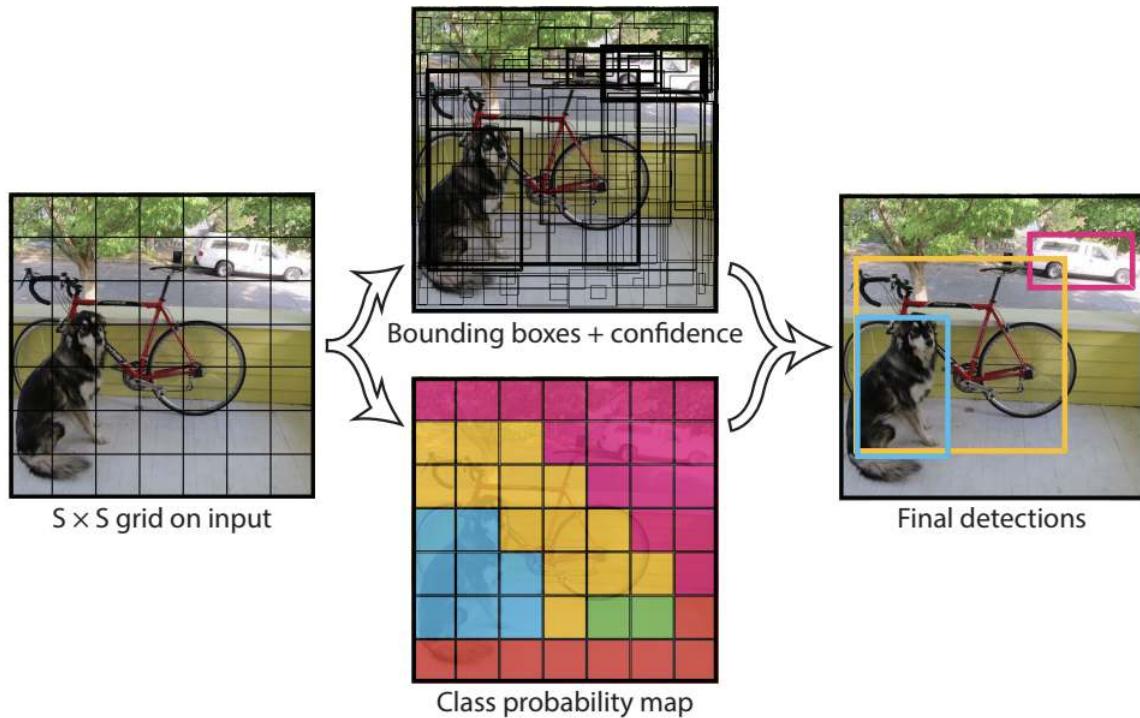


Figure 23.6: Summary of Predictions made by YOLO Model. Taken from the 2015 YOLO paper.

23.4.2 YOLOv2 (YOLO9000) and YOLOv3

The model was updated by Joseph Redmon and Ali Farhadi in an effort to further improve model performance in their 2016 paper titled *YOLO9000: Better, Faster, Stronger*. Although this variation of the model is referred to as YOLO v2, an instance of the model is described that was trained on two object recognition datasets in parallel, capable of predicting 9,000 object classes, hence given the name *YOLO9000*. A number of training and architectural changes were made to the model, such as the use of batch normalization and high-resolution input images.

Like Faster R-CNN, YOLOv2 model makes use of anchor boxes, pre-defined bounding boxes with useful shapes and sizes that are tailored during training. The choice of bounding boxes for the image is pre-processed using a k -means analysis on the training dataset. Importantly, the predicted representation of the bounding boxes is changed to allow small changes to have a less dramatic effect on the predictions, resulting in a more stable model. Rather than predicting position and size directly, offsets are predicted for moving and reshaping the pre-defined anchor boxes relative to a grid cell and damped by a logistic function.

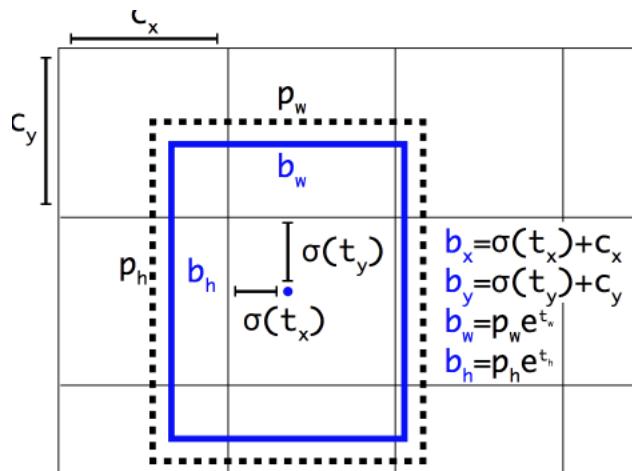


Figure 23.7: Example of the Representation Chosen when Predicting Bounding Box Position and Shape. Taken from the YOLO9000 paper.

Further improvements to the model were proposed by Joseph Redmon and Ali Farhadi in their 2018 paper titled *YOLOv3: An Incremental Improvement*. The improvements were reasonably minor, including a deeper feature detector network and minor representational changes.

23.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

23.5.1 Papers

- *ImageNet Large Scale Visual Recognition Challenge*, 2015.
<https://arxiv.org/abs/1409.0575>

- *Selective Search for Object Recognition.*
<https://link.springer.com/article/10.1007/s11263-013-0620-5>
- *Rich Feature Hierarchies For Accurate Object Detection And Semantic Segmentation*, 2013.
<https://arxiv.org/abs/1311.2524>
- *Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition*, 2014.
<https://arxiv.org/abs/1406.4729>
- *Fast R-CNN*, 2015.
<https://arxiv.org/abs/1504.08083>
- *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*, 2016.
<https://arxiv.org/abs/1506.01497>
- *Mask R-CNN*, 2017.
<https://arxiv.org/abs/1703.06870>
- *You Only Look Once: Unified, Real-Time Object Detection*, 2015.
<https://arxiv.org/abs/1506.02640>
- *YOLO9000: Better, Faster, Stronger*, 2016.
<https://arxiv.org/abs/1612.08242>
- *YOLOv3: An Incremental Improvement*, 2018.
<https://arxiv.org/abs/1804.02767>

23.5.2 Code Projects

- R-CNN: Regions with Convolutional Neural Network Features, GitHub.
<https://github.com/rbgirshick/rcnn>
- Fast R-CNN, GitHub.
<https://github.com/rbgirshick/fast-rcnn>
- Faster R-CNN Python Code, GitHub.
<https://github.com/rbgirshick/py-faster-rcnn>
- YOLO, GitHub.
<https://github.com/pjreddie/darknet/wiki/YOLO:-Real-Time-Object-Detection>

23.5.3 Resources

- Ross Girshick, Homepage.
<http://www.rossgirshick.info/>
- Joseph Redmon, Homepage.
<https://pjreddie.com/>
- YOLO: Real-Time Object Detection, Homepage.
<https://pjreddie.com/darknet/yolo/>

23.6 Summary

In this tutorial, you discovered a gentle introduction to the problem of object recognition and state-of-the-art deep learning models designed to address it. Specifically, you learned:

- Object recognition refers to a collection of related tasks for identifying objects in digital photographs.
- Region-Based Convolutional Neural Networks, or R-CNNs, are a family of techniques for addressing object localization and recognition tasks, designed for model performance.
- You Only Look Once, or YOLO, is a second family of techniques for object recognition designed for speed and real-time use.

23.6.1 Next

In the next section, you will discover how to perform object recognition using the YOLO model.

Chapter 24

How to Perform Object Detection With YOLOv3

Object detection is a task in computer vision that involves identifying the presence, location, and type of one or more objects in a given photograph. It is a challenging problem that involves building upon methods for object recognition (e.g. where are they), object localization (e.g. what are their extent), and object classification (e.g. what are they). In recent years, deep learning techniques are achieving state-of-the-art results for object detection, such as on standard benchmark datasets and in computer vision competitions. Notable is the *You Only Look Once*, or the YOLO, family of Convolutional Neural Networks that achieve near state-of-the-art results with a single end-to-end model that can perform object detection in real-time. In this tutorial, you will discover how to develop a YOLOv3 model for object detection on new photographs.

After completing this tutorial, you will know:

- YOLO-based Convolutional Neural Network family of models for object detection and the most recent variation called YOLOv3.
- The best-of-breed open source library implementation of the YOLOv3 for the Keras deep learning library.
- How to use a pre-trained YOLOv3 to perform object localization and detection on new photographs.

Let's get started.

24.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. YOLO for Object Detection
2. Experiencor YOLO3 Project
3. Object Detection With YOLOv3

24.2 YOLO for Object Detection

Object detection is a computer vision task that involves both localizing one or more objects within an image and classifying each object in the image. It is a challenging computer vision task that requires both successful object localization in order to locate and draw a bounding box around each object in an image, and object classification to predict the correct class of object that was localized. The *You Only Look Once*, or YOLO, family of models are a series of end-to-end deep learning models designed for fast object detection, developed by Joseph Redmon, et al. and first described in the 2015 paper titled *You Only Look Once: Unified, Real-Time Object Detection* (introduced in Chapter 23).

The approach involves a single deep convolutional neural network (originally a version of GoogLeNet, later updated and called DarkNet based on VGG) that splits the input into a grid of cells and each cell directly predicts a bounding box and object classification. The result is a large number of candidate bounding boxes that are consolidated into a final prediction by a post-processing step. There are three main variations of the approach, at the time of writing; they are YOLOv1, YOLOv2, and YOLOv3. The first version proposed the general architecture, whereas the second version refined the design and made use of predefined anchor boxes to improve bounding box proposal, and version three further refined the model architecture and training process. Although the accuracy of the models is close but not as good as Region-Based Convolutional Neural Networks (R-CNNs), they are popular for object detection because of their detection speed, often demonstrated in real-time on video or with camera feed input.

A single neural network predicts bounding boxes and class probabilities directly from full images in one evaluation. Since the whole detection pipeline is a single network, it can be optimized end-to-end directly on detection performance.

— *You Only Look Once: Unified, Real-Time Object Detection*, 2015.

In this tutorial, we will focus on using YOLOv3.

24.3 Experiencor YOLO3 for Keras Project

Source code for each version of YOLO is available, as well as pre-trained models. The official DarkNet GitHub repository contains the source code for the YOLO versions mentioned in the papers, written in C. The repository provides a step-by-step tutorial on how to use the code for object detection. It is a challenging model to implement from scratch, especially for beginners as it requires the development of many customized model elements for training and for prediction. For example, even using a pre-trained model directly requires sophisticated code to distill and interpret the predicted bounding boxes output by the model.

Instead of developing this code from scratch, we can use a third-party implementation. There are many third-party implementations designed for using YOLO with Keras, and none appear to be standardized and designed to be used as a library. The YAD2K project was a de facto standard for YOLOv2 and provided scripts to convert the pre-trained weights into Keras format, use the pre-trained model to make predictions, and provided the code required to distill and interpret the predicted bounding boxes. Many other third-party developers have used this code as a starting point and updated it to support YOLOv3.

Perhaps the most widely used project for using pre-trained YOLO models is called *keras-yolo3: Training and Detecting Objects with YOLO3*¹ by Huynh Ngoc Anh or *experiencor*. The code in the project has been made available under a permissive MIT open source license. Like YAD2K, it provides scripts to both load and use pre-trained YOLO models as well as transfer learning for developing YOLOv3 models on new datasets. He also has a *keras-yolo2* project that provides similar code for YOLOv2 as well as detailed tutorials on how to use the code in the repository. The *keras-yolo3* project appears to be an updated version of that project. Interestingly, *experiencor* has used the model as the basis for some experiments and trained versions of the YOLOv3 on standard object detection problems such as a kangaroo dataset, raccoon dataset, red blood cell detection, and others. He has listed model performance, provided the model weights for download and provided YouTube videos of model behavior. For example:

- Raccoon Detection using YOLO 3.
<https://www.youtube.com/watch?v=lxLyLIL7OsU>

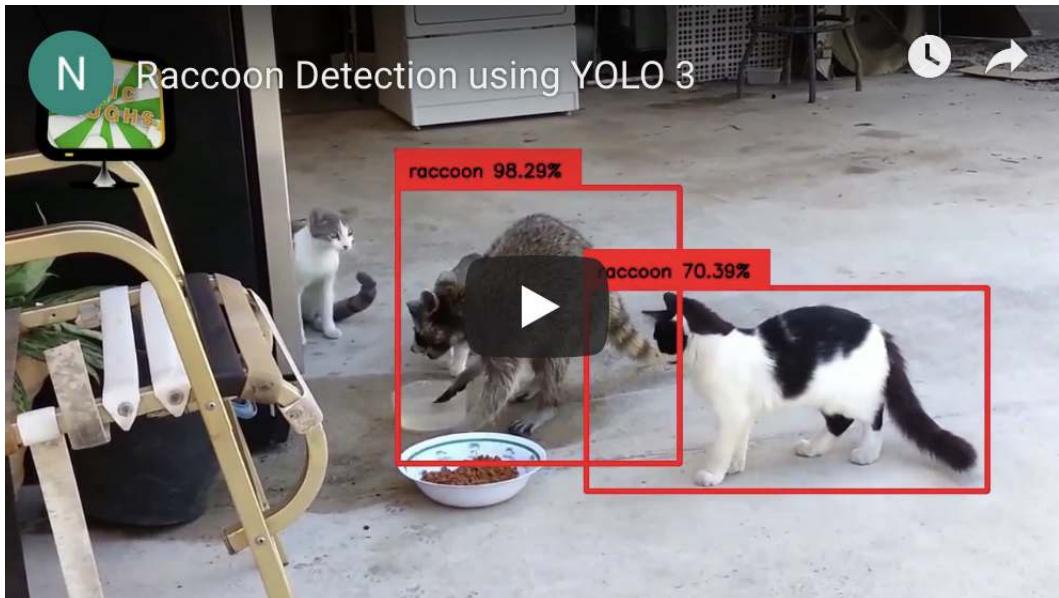


Figure 24.1: Video Showing Raccoon Detection using YOLO 3.

We will use *experiencor*'s *keras-yolo3* project as the basis for performing object detection with a YOLOv3 model in this tutorial².

24.4 Object Detection With YOLOv3

The *keras-yolo3* project provides a lot of capability for using YOLOv3 models, including object detection, transfer learning, and training new models from scratch. In this section, we will use a pre-trained model to perform object detection on an unseen photograph. This capability is available in a single Python file in the repository called `yolo3_one_file_to_detect_them_all.py`

¹<https://github.com/experiencor/keras-yolo3>

²Clone of the project is available here: <https://github.com/jbrownlee/keras-yolo3>

that has about 435 lines. This script is, in fact, a program that will use pre-trained weights to prepare a model and use that model to perform object detection and output a model. It also depends upon OpenCV. Instead of using this program directly, we will reuse elements from this program and develop our own scripts to first prepare and save a Keras YOLOv3 model, and then load the model to make a prediction for a new photograph.

24.4.1 Create and Save Model

The first step is to download the pre-trained model weights. These were trained using the DarkNet code base on the MSCOCO dataset. Download the model weights and place them into your current working directory with the filename `yolov3.weights`. It is a large file (237 megabytes) and may take a moment to download depending on the speed of your internet connection.

- YOLOv3 Pre-trained Model Weights (`yolov3.weights`) (237 MB).³

Next, we need to define a Keras model that has the right number and type of layers to match the downloaded model weights. The model architecture is called *DarkNet* and was originally loosely based on the VGG-16 model. The `yolo3_one_file_to_detect_them_all.py` script provides the `make_yolov3_model()` function to create the model for us, and the helper function `_conv_block()` that is used to create blocks of layers. These two functions can be copied directly from the script. We can now define the Keras model for YOLOv3.

```
...
# define the model
model = make_yolov3_model()
```

Listing 24.1: Example of defining the YOLOv3 model.

Next, we need to load the model weights. The model weights are stored in whatever format that was used by DarkNet. Rather than trying to decode the file manually, we can use the `WeightReader` class provided in the script. To use the `WeightReader`, it is instantiated with the path to our weights file (e.g. `yolov3.weights`). This will parse the file and load the model weights into memory in a format that we can set into our Keras model.

```
...
# load the model weights
weight_reader = WeightReader('yolov3.weights')
```

Listing 24.2: Load the YOLOv3 model weights.

We can then call the `load_weights()` function of the `WeightReader` instance, passing in our defined Keras model to set the weights into the layers.

```
...
# set the model weights into the model
weight_reader.load_weights(model)
```

Listing 24.3: Set weights into defined YOLOv3 model.

That's it; we now have a YOLOv3 model for use. We can save this model to a Keras compatible .h5 model file ready for later use.

³<https://pjreddie.com/media/files/yolov3.weights>

```
...
# save the model to file
model.save('model.h5')
```

Listing 24.4: Save Keras model to file.

We can tie all of this together; the complete code example including functions copied directly from the `yolo3_one_file_to_detect_them_all.py` script is listed below.

```
# create a YOLOv3 Keras model and save it to file
# based on https://github.com/experiencor/keras-yolo3
import struct
import numpy as np
from keras.layers import Conv2D
from keras.layers import Input
from keras.layers import BatchNormalization
from keras.layers import LeakyReLU
from keras.layers import ZeroPadding2D
from keras.layers import UpSampling2D
from keras.layers.merge import add, concatenate
from keras.models import Model

def _conv_block(inp, convs, skip=True):
    x = inp
    count = 0
    for conv in convs:
        if count == (len(conv) - 2) and skip:
            skip_connection = x
        count += 1
        if conv['stride'] > 1: x = ZeroPadding2D(((1,0),(1,0)))(x) # peculiar padding as
            # darknet prefer left and top
        x = Conv2D(conv['filter'],
                   conv['kernel'],
                   strides=conv['stride'],
                   padding='valid' if conv['stride'] > 1 else 'same', # peculiar padding as darknet
                   prefer_left_and_top=True,
                   name='conv_' + str(conv['layer_idx']),
                   use_bias=False if conv['bnorm'] else True)(x)
        if conv['bnorm']: x = BatchNormalization(epsilon=0.001, name='bnorm_' +
            str(conv['layer_idx']))(x)
        if conv['leaky']: x = LeakyReLU(alpha=0.1, name='leaky_' + str(conv['layer_idx']))(x)
    return add([skip_connection, x]) if skip else x

def make_yolov3_model():
    input_image = Input(shape=(None, None, 3))
    # Layer 0 => 4
    x = _conv_block(input_image, [{ 'filter': 32, 'kernel': 3, 'stride': 1, 'bnorm': True,
        'leaky': True, 'layer_idx': 0},
        { 'filter': 64, 'kernel': 3, 'stride': 2, 'bnorm': True, 'leaky': True,
        'layer_idx': 1},
        { 'filter': 32, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,
        'layer_idx': 2},
        { 'filter': 64, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True,
        'layer_idx': 3}])
    # Layer 5 => 8
    x = _conv_block(x, [{ 'filter': 128, 'kernel': 3, 'stride': 2, 'bnorm': True, 'leaky':
```

```

    True, 'layer_idx': 5},
    {'filter': 64, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,
     'layer_idx': 6},
    {'filter': 128, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True,
     'layer_idx': 7}])
# Layer 9 => 11
x = _conv_block(x, [{"filter": 64, "kernel": 1, "stride": 1, "bnorm": True, "leaky": True,
                     "layer_idx": 9},
                     {"filter": 128, "kernel": 3, "stride": 1, "bnorm": True, "leaky": True,
                      "layer_idx": 10}])
# Layer 12 => 15
x = _conv_block(x, [{"filter": 256, "kernel": 3, "stride": 2, "bnorm": True, "leaky": True,
                     "layer_idx": 12},
                     {"filter": 128, "kernel": 1, "stride": 1, "bnorm": True, "leaky": True,
                      "layer_idx": 13},
                     {"filter": 256, "kernel": 3, "stride": 1, "bnorm": True, "leaky": True,
                      "layer_idx": 14}])
# Layer 16 => 36
for i in range(7):
    x = _conv_block(x, [{"filter": 128, "kernel": 1, "stride": 1, "bnorm": True, "leaky": True,
                         "layer_idx": 16+i*3},
                         {"filter": 256, "kernel": 3, "stride": 1, "bnorm": True, "leaky": True,
                          "layer_idx": 17+i*3}])
skip_36 = x
# Layer 37 => 40
x = _conv_block(x, [{"filter": 512, "kernel": 3, "stride": 2, "bnorm": True, "leaky": True,
                     "layer_idx": 37},
                     {"filter": 256, "kernel": 1, "stride": 1, "bnorm": True, "leaky": True,
                      "layer_idx": 38},
                     {"filter": 512, "kernel": 3, "stride": 1, "bnorm": True, "leaky": True,
                      "layer_idx": 39}])
# Layer 41 => 61
for i in range(7):
    x = _conv_block(x, [{"filter": 256, "kernel": 1, "stride": 1, "bnorm": True, "leaky": True,
                         "layer_idx": 41+i*3},
                         {"filter": 512, "kernel": 3, "stride": 1, "bnorm": True, "leaky": True,
                          "layer_idx": 42+i*3}])
skip_61 = x
# Layer 62 => 65
x = _conv_block(x, [{"filter": 1024, "kernel": 3, "stride": 2, "bnorm": True, "leaky": True,
                     "layer_idx": 62},
                     {"filter": 512, "kernel": 1, "stride": 1, "bnorm": True, "leaky": True,
                      "layer_idx": 63},
                     {"filter": 1024, "kernel": 3, "stride": 1, "bnorm": True, "leaky": True,
                      "layer_idx": 64}])
# Layer 66 => 74
for i in range(3):
    x = _conv_block(x, [{"filter": 512, "kernel": 1, "stride": 1, "bnorm": True, "leaky": True,
                         "layer_idx": 66+i*3},
                         {"filter": 1024, "kernel": 3, "stride": 1, "bnorm": True, "leaky": True,
                          "layer_idx": 67+i*3}])
# Layer 75 => 79
x = _conv_block(x, [{"filter": 512, "kernel": 1, "stride": 1, "bnorm": True, "leaky": True,
                     "layer_idx": 75},
                     {"filter": 1024, "kernel": 3, "stride": 1, "bnorm": True, "leaky": True,
                      "layer_idx": 76},

```

```
        {'filter': 512, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,
         'layer_idx': 77},
        {'filter': 1024, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True,
         'layer_idx': 78},
        {'filter': 512, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,
         'layer_idx': 79}], skip=False)
# Layer 80 => 82
yolo_82 = _conv_block(x, [{['filter': 1024, 'kernel': 3, 'stride': 1, 'bnorm': True,
                           'leaky': True, 'layer_idx': 80},
                           {'filter': 255, 'kernel': 1, 'stride': 1, 'bnorm': False, 'leaky': False,
                           'layer_idx': 81}], skip=False)
# Layer 83 => 86
x = _conv_block(x, [{['filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky':
                      True, 'layer_idx': 84}], skip=False)
x = UpSampling2D(2)(x)
x = concatenate([x, skip_61])
# Layer 87 => 91
x = _conv_block(x, [{['filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky':
                      True, 'layer_idx': 87},
                      {'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True,
                       'layer_idx': 88},
                      {'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,
                       'layer_idx': 89},
                      {'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True,
                       'layer_idx': 90},
                      {'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,
                       'layer_idx': 91}], skip=False)
# Layer 92 => 94
yolo_94 = _conv_block(x, [{['filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True,
                           'leaky': True, 'layer_idx': 92},
                           {'filter': 255, 'kernel': 1, 'stride': 1, 'bnorm': False, 'leaky': False,
                           'layer_idx': 93}], skip=False)
# Layer 95 => 98
x = _conv_block(x, [{['filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky':
                      True, 'layer_idx': 96}], skip=False)
x = UpSampling2D(2)(x)
x = concatenate([x, skip_36])
# Layer 99 => 106
yolo_106 = _conv_block(x, [{['filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True,
                            'leaky': True, 'layer_idx': 99},
                            {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True,
                             'layer_idx': 100},
                            {'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,
                             'layer_idx': 101},
                            {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True,
                             'layer_idx': 102},
                            {'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,
                             'layer_idx': 103},
                            {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True,
                             'layer_idx': 104},
                            {'filter': 255, 'kernel': 1, 'stride': 1, 'bnorm': False, 'leaky': False,
                             'layer_idx': 105}], skip=False)
model = Model(input_image, [yolo_82, yolo_94, yolo_106])
return model

class WeightReader:
```

```
def __init__(self, weight_file):
    with open(weight_file, 'rb') as w_f:
        major, = struct.unpack('i', w_f.read(4))
        minor, = struct.unpack('i', w_f.read(4))
        revision, = struct.unpack('i', w_f.read(4))
        if (major*10 + minor) >= 2 and major < 1000 and minor < 1000:
            w_f.read(8)
        else:
            w_f.read(4)
        transpose = (major > 1000) or (minor > 1000)
        binary = w_f.read()
    self.offset = 0
    self.all_weights = np.frombuffer(binary, dtype='float32')

def read_bytes(self, size):
    self.offset = self.offset + size
    return self.all_weights[self.offset-size:self.offset]

def load_weights(self, model):
    for i in range(106):
        try:
            conv_layer = model.get_layer('conv_' + str(i))
            print("loading weights of convolution #" + str(i))
            if i not in [81, 93, 105]:
                norm_layer = model.get_layer('bnorm_' + str(i))
                size = np.prod(norm_layer.get_weights()[0].shape)
                beta = self.read_bytes(size) # bias
                gamma = self.read_bytes(size) # scale
                mean = self.read_bytes(size) # mean
                var = self.read_bytes(size) # variance
                weights = norm_layer.set_weights([gamma, beta, mean, var])
            if len(conv_layer.get_weights()) > 1:
                bias = self.read_bytes(np.prod(conv_layer.get_weights()[1].shape))
                kernel = self.read_bytes(np.prod(conv_layer.get_weights()[0].shape))
                kernel = kernel.reshape(list(reversed(conv_layer.get_weights()[0].shape)))
                kernel = kernel.transpose([2,3,1,0])
                conv_layer.set_weights([kernel, bias])
            else:
                kernel = self.read_bytes(np.prod(conv_layer.get_weights()[0].shape))
                kernel = kernel.reshape(list(reversed(conv_layer.get_weights()[0].shape)))
                kernel = kernel.transpose([2,3,1,0])
                conv_layer.set_weights([kernel])
        except ValueError:
            print("no convolution #" + str(i))

def reset(self):
    self.offset = 0

# define the model
model = make_yolov3_model()
# load the model weights
weight_reader = WeightReader('yolov3.weights')
# set the model weights into the model
weight_reader.load_weights(model)
# save the model to file
model.save('model.h5')
```

Listing 24.5: Example of loading and parsing YOLOv3 model weights and saving a Keras model.

Running the example may take a little less than one minute to execute on modern hardware. As the weight file is loaded, you will see debug information reported about what was loaded, output by the `WeightReader` class.

```
...  
loading weights of convolution #99  
loading weights of convolution #100  
loading weights of convolution #101  
loading weights of convolution #102  
loading weights of convolution #103  
loading weights of convolution #104  
loading weights of convolution #105
```

Listing 24.6: Example output from loading YOLOv3 model weights.

At the end of the run, the `model.h5` file is saved in your current working directory with approximately the same size as the original weight file (237MB), but ready to be loaded and used directly as a Keras model.

24.4.2 Make a Prediction

We need a new photo for object detection, ideally with object classes that we know that the model can recognize from the MSCOCO dataset. We will use a photograph of three zebras taken by Boegh⁴, and released under a permissive license.



Figure 24.2: Photograph of Three Zebras.

Download the photograph and place it in your current working directory with the filename `zebra.jpg`.

- Download Photograph of Three Zebras (`zebra.jpg`).⁵

⁴<https://www.flickr.com/photos/boegh/5676993427/>

⁵<https://machinelearningmastery.com/wp-content/uploads/2019/03/zebra.jpg>

Making a prediction is straightforward, although interpreting the prediction requires some work. The first step is to load the YOLOv3 Keras model. This might be the slowest part of making a prediction.

```
...
# load yolov3 model
model = load_model('model.h5')
```

Listing 24.7: Example of loading the YOLOv3 model.

Next, we need to load our new photograph and prepare it as suitable input to the model. The model expects inputs to be color images with the square shape of 416×416 pixels. We can use the `load_img()` Keras function to load the image and the `target_size` argument to resize the image after loading. We can also use the `img_to_array()` function to convert the loaded PIL image object into a NumPy array, and then rescale the pixel values from 0-255 to 0-1 32-bit floating point values.

```
...
# load the image with the required size
image = load_img('zebra.jpg', target_size=(416, 416))
# convert to numpy array
image = img_to_array(image)
# scale pixel values to [0, 1]
image = image.astype('float32')
image /= 255.0
```

Listing 24.8: Example of loading and preparing an image for making a prediction.

We will want to show the original photo again later, which means we will need to scale the bounding boxes of all detected objects from the square shape back to the original shape. As such, we can load the image and retrieve the original shape.

```
...
# load the image to get its shape
image = load_img('zebra.jpg')
width, height = image.size
```

Listing 24.9: Example of loading an image and getting the dimensions.

We can tie all of this together into a convenience function named `load_image_pixels()` that takes the filename and target size and returns the scaled pixel data ready to provide as input to the Keras model, as well as the original width and height of the image.

```
# load and prepare an image
def load_image_pixels(filename, shape):
    # load the image to get its shape
    image = load_img(filename)
    width, height = image.size
    # load the image with the required size
    image = load_img(filename, target_size=shape)
    # convert to numpy array
    image = img_to_array(image)
    # scale pixel values to [0, 1]
    image = image.astype('float32')
    image /= 255.0
    # add a dimension so that we have one sample
```

```
image = expand_dims(image, 0)
return image, width, height
```

Listing 24.10: Example of a function for loading an image and getting the dimensions.

We can then call this function to load our photo of zebras.

```
...
# define the expected input shape for the model
input_w, input_h = 416, 416
# define our new photo
photo_filename = 'zebra.jpg'
# load and prepare image
image, image_w, image_h = load_image_pixels(photo_filename, (input_w, input_h))
```

Listing 24.11: Example of calling a function to prepare an image for making a prediction.

We can now feed the photo into the Keras model and make a prediction.

```
...
# make prediction
yhat = model.predict(image)
# summarize the shape of the list of arrays
print([a.shape for a in yhat])
```

Listing 24.12: Example making a prediction with a YOLOv3 model.

That's it, at least for making a prediction. The complete example is listed below.

```
# load yolov3 model and perform object detection
# based on https://github.com/experiencor/keras-yolo3
from numpy import expand_dims
from keras.models import load_model
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array

# load and prepare an image
def load_image_pixels(filename, shape):
    # load the image to get its shape
    image = load_img(filename)
    width, height = image.size
    # load the image with the required size
    image = load_img(filename, target_size=shape)
    # convert to numpy array
    image = img_to_array(image)
    # scale pixel values to [0, 1]
    image = image.astype('float32')
    image /= 255.0
    # add a dimension so that we have one sample
    image = expand_dims(image, 0)
    return image, width, height

# load yolov3 model
model = load_model('model.h5')
# define the expected input shape for the model
input_w, input_h = 416, 416
# define our new photo
photo_filename = 'zebra.jpg'
```

```
# load and prepare image
image, image_w, image_h = load_image_pixels(photo_filename, (input_w, input_h))
# make prediction
yhat = model.predict(image)
# summarize the shape of the list of arrays
print([a.shape for a in yhat])
```

Listing 24.13: Example of loading the YOLOv3 model and using it to make a prediction.

Running the example returns a list of three NumPy arrays, the shape of which is displayed as output. These arrays predict both the bounding boxes and class labels but are encoded. They must be interpreted.

```
[(1, 13, 13, 255), (1, 26, 26, 255), (1, 52, 52, 255)]
```

Listing 24.14: Example output from loading the YOLOv3 model and using it to make a prediction.

24.4.3 Make a Prediction and Interpret Result

The output of the model is, in fact, encoded candidate bounding boxes from three different grid sizes, and the boxes are defined in the context of anchor boxes, carefully chosen based on an analysis of the size of objects in the MSCOCO dataset. The script provided by experiencor provides a function called `decode_netout()` that will take each one of the NumPy arrays, one at a time, and decode the candidate bounding boxes and class predictions. Further, any bounding boxes that don't confidently describe an object (e.g. all class probabilities are below a threshold) are ignored. We will use a probability of 60% or 0.6. The function returns a list of `BoundingBox` instances that define the corners of each bounding box in the context of the input image shape and class probabilities.

```
...
# define the anchors
anchors = [[116,90, 156,198, 373,326], [30,61, 62,45, 59,119], [10,13, 16,30, 33,23]]
# define the probability threshold for detected objects
class_threshold = 0.6
boxes = list()
for i in range(len(yhat)):
    # decode the output of the network
    boxes += decode_netout(yhat[i][0], anchors[i], class_threshold, input_h, input_w)
```

Listing 24.15: Example of decoding bound boxes.

Next, the bounding boxes can be stretched back into the shape of the original image. This is helpful as it means that later we can plot the original image and draw the bounding boxes, hopefully detecting real objects. The experiencor script provides the `correct_yolo_boxes()` function to perform this translation of bounding box coordinates, taking the list of bounding boxes, the original shape of our loaded photograph, and the shape of the input to the network as arguments. The coordinates of the bounding boxes are updated directly.

```
...
# correct the sizes of the bounding boxes for the shape of the image
correct_yolo_boxes(boxes, image_h, image_w, input_h, input_w)
```

Listing 24.16: Example of correcting the decoded bounding boxes.

The model has predicted a lot of candidate bounding boxes, and most of the boxes will be referring to the same objects. The list of bounding boxes can be filtered and those boxes that overlap and refer to the same object can be merged. We can define the amount of overlap as a configuration parameter, in this case, 50% or 0.5. This filtering of bounding box regions is generally referred to as non-maximal suppression and is a required post-processing step. The experiencor script provides this via the `do_nms()` function that takes the list of bounding boxes and a threshold parameter. Rather than purging the overlapping boxes, their predicted probability for their overlapping class is cleared. This allows the boxes to remain and be used if they also detect another object type.

```
...
# suppress non-maximal boxes
do_nms(boxes, 0.5)
```

Listing 24.17: Example of non-maximal suppression of bounding boxes.

This will leave us with the same number of boxes, but only very few of interest. We can retrieve just those boxes that strongly predict the presence of an object: that is are more than 60% confident. This can be achieved by enumerating over all boxes and checking the class prediction values. We can then look up the corresponding class label for the box and add it to the list. Each box must be considered for each class label, just in case the same box strongly predicts more than one object. We can develop a `get_boxes()` function that does this and takes the list of boxes, known labels, and our classification threshold as arguments and returns parallel lists of boxes, labels, and scores.

```
# get all of the results above a threshold
def get_boxes(boxes, labels, thresh):
    v_boxes, v_labels, v_scores = list(), list(), list()
    # enumerate all boxes
    for box in boxes:
        # enumerate all possible labels
        for i in range(len(labels)):
            # check if the threshold for this label is high enough
            if box.classes[i] > thresh:
                v_boxes.append(box)
                v_labels.append(labels[i])
                v_scores.append(box.classes[i]*100)
                # don't break, many labels may trigger for one box
    return v_boxes, v_labels, v_scores
```

Listing 24.18: Example of a function for retrieving bounding boxes and class predictions for detected objects.

We can call this function with our list of boxes. We also need a list of strings containing the class labels known to the model in the correct order used during training, specifically those class labels from the MSCOCO dataset. Thankfully, this is provided in the experiencor script.

```
...
# define the labels
labels = ["person", "bicycle", "car", "motorbike", "aeroplane", "bus", "train", "truck",
    "boat", "traffic light", "fire hydrant", "stop sign", "parking meter", "bench",
    "bird", "cat", "dog", "horse", "sheep", "cow", "elephant", "bear", "zebra", "giraffe",
    "backpack", "umbrella", "handbag", "tie", "suitcase", "frisbee", "skis", "snowboard",
    "sports ball", "kite", "baseball bat", "baseball glove", "skateboard", "surfboard",
```

```

"tennis racket", "bottle", "wine glass", "cup", "fork", "knife", "spoon", "bowl",
    "banana",
"apple", "sandwich", "orange", "broccoli", "carrot", "hot dog", "pizza", "donut",
    "cake",
"chair", "sofa", "pottedplant", "bed", "diningtable", "toilet", "tvmonitor", "laptop",
    "mouse",
"remote", "keyboard", "cell phone", "microwave", "oven", "toaster", "sink",
    "refrigerator",
"book", "clock", "vase", "scissors", "teddy bear", "hair drier", "toothbrush"]
# get the details of the detected objects
v_boxes, v_labels, v_scores = get_boxes(boxes, labels, class_threshold)

```

Listing 24.19: Example using MSCOCO class labels.

Now that we have those few boxes of strongly predicted objects, we can summarize them.

```

...
# summarize what we found
for i in range(len(v_boxes)):
    print(v_labels[i], v_scores[i])

```

Listing 24.20: Example of summarizing the detected objects.

We can also plot our original photograph and draw the bounding box around each detected object. This can be achieved by retrieving the coordinates from each bounding box and creating a Rectangle object.

```

...
box = v_boxes[i]
# get coordinates
y1, x1, y2, x2 = box.ymin, box.xmin, box.ymax, box.xmax
# calculate width and height of the box
width, height = x2 - x1, y2 - y1
# create the shape
rect = Rectangle((x1, y1), width, height, fill=False, color='white')
# draw the box
ax.add_patch(rect)

```

Listing 24.21: Example of drawing a bounding box for detected objects.

We can also draw a string with the class label and confidence.

```

...
# draw text and score in top left corner
label = "%s (%.3f)" % (v_labels[i], v_scores[i])
pyplot.text(x1, y1, label, color='white')

```

Listing 24.22: Example of drawing a label for detected objects.

The `draw_boxes()` function below implements this, taking the filename of the original photograph and the parallel lists of bounding boxes, labels and scores, and creates a plot showing all detected objects.

```

# draw all results
def draw_boxes(filename, v_boxes, v_labels, v_scores):
    # load the image
    data = pyplot.imread(filename)
    # plot the image

```

```

pyplot.imshow(data)
# get the context for drawing boxes
ax = pyplot.gca()
# plot each box
for i in range(len(v_boxes)):
    box = v_boxes[i]
    # get coordinates
    y1, x1, y2, x2 = box.ymin, box.xmin, box.ymax, box.xmax
    # calculate width and height of the box
    width, height = x2 - x1, y2 - y1
    # create the shape
    rect = Rectangle((x1, y1), width, height, fill=False, color='white')
    # draw the box
    ax.add_patch(rect)
    # draw text and score in top left corner
    label = "%s (%.3f)" % (v_labels[i], v_scores[i])
    pyplot.text(x1, y1, label, color='white')
# show the plot
pyplot.show()

```

Listing 24.23: Example of a function for drawing detected objects.

We can then call this function to plot our final result.

```

...
# draw what we found
draw_boxes(photo_filename, v_boxes, v_labels, v_scores)

```

Listing 24.24: Example of calling the function to draw the detected objects.

We now have all of the elements required to make a prediction using the YOLOv3 model, interpret the results, and plot them for review. The full code listing, including the original and modified functions taken from the experiencor script, are listed below for completeness.

```

# load yolov3 model and perform object detection
# based on https://github.com/experiencor/keras-yolo3
import numpy as np
from numpy import expand_dims
from keras.models import load_model
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from matplotlib import pyplot
from matplotlib.patches import Rectangle

class BoundBox:
    def __init__(self, xmin, ymin, xmax, ymax, objness = None, classes = None):
        self.xmin = xmin
        self.ymin = ymin
        self.xmax = xmax
        self.ymax = ymax
        self.objness = objness
        self.classes = classes
        self.label = -1
        self.score = -1

    def get_label(self):
        if self.label == -1:

```

```

    self.label = np.argmax(self.classes)

    return self.label

def get_score(self):
    if self.score == -1:
        self.score = self.classes[self.get_label()]

    return self.score

def _sigmoid(x):
    return 1. / (1. + np.exp(-x))

def decode_netout(netout, anchors, obj_thresh, net_h, net_w):
    grid_h, grid_w = netout.shape[:2]
    nb_box = 3
    netout = netout.reshape((grid_h, grid_w, nb_box, -1))
    nb_class = netout.shape[-1] - 5
    boxes = []
    netout[:, :, :, :2] = _sigmoid(netout[:, :, :, :2])
    netout[:, :, :, 4:] = _sigmoid(netout[:, :, :, 4:])
    netout[:, :, :, 5:] = netout[:, :, :, 4:][..., np.newaxis] * netout[:, :, :, 5:]
    netout[:, :, :, 5:] *= netout[:, :, :, 5:] > obj_thresh

    for i in range(grid_h*grid_w):
        row = i / grid_w
        col = i % grid_w
        for b in range(nb_box):
            # 4th element is objectness score
            objectness = netout[int(row)][int(col)][b][4]
            if(objectness.all() <= obj_thresh): continue
            # first 4 elements are x, y, w, and h
            x, y, w, h = netout[int(row)][int(col)][b][:4]
            x = (col + x) / grid_w # center position, unit: image width
            y = (row + y) / grid_h # center position, unit: image height
            w = anchors[2 * b + 0] * np.exp(w) / net_w # unit: image width
            h = anchors[2 * b + 1] * np.exp(h) / net_h # unit: image height
            # last elements are class probabilities
            classes = netout[int(row)][col][b][5:]
            box = BoundBox(x-w/2, y-h/2, x+w/2, y+h/2, objectness, classes)
            boxes.append(box)
    return boxes

def correct_yolo_boxes(boxes, image_h, image_w, net_h, net_w):
    new_w, new_h = net_w, net_h
    for i in range(len(boxes)):
        x_offset, x_scale = (net_w - new_w)/2./net_w, float(new_w)/net_w
        y_offset, y_scale = (net_h - new_h)/2./net_h, float(new_h)/net_h
        boxes[i].xmin = int((boxes[i].xmin - x_offset) / x_scale * image_w)
        boxes[i].xmax = int((boxes[i].xmax - x_offset) / x_scale * image_w)
        boxes[i].ymin = int((boxes[i].ymin - y_offset) / y_scale * image_h)
        boxes[i].ymax = int((boxes[i].ymax - y_offset) / y_scale * image_h)

def _interval_overlap(interval_a, interval_b):
    x1, x2 = interval_a
    x3, x4 = interval_b

```

```
if x3 < x1:
    if x4 < x1:
        return 0
    else:
        return min(x2,x4) - x1
else:
    if x2 < x3:
        return 0
    else:
        return min(x2,x4) - x3

def bbox_iou(box1, box2):
    intersect_w = _interval_overlap([box1.xmin, box1.xmax], [box2.xmin, box2.xmax])
    intersect_h = _interval_overlap([box1.ymin, box1.ymax], [box2.ymin, box2.ymax])
    intersect = intersect_w * intersect_h
    w1, h1 = box1.xmax-box1.xmin, box1.ymax-box1.ymin
    w2, h2 = box2.xmax-box2.xmin, box2.ymax-box2.ymin
    union = w1*h1 + w2*h2 - intersect
    return float(intersect) / union

def do_nms(boxes, nms_thresh):
    if len(boxes) > 0:
        nb_class = len(boxes[0].classes)
    else:
        return
    for c in range(nb_class):
        sorted_indices = np.argsort([-box.classes[c] for box in boxes])
        for i in range(len(sorted_indices)):
            index_i = sorted_indices[i]
            if boxes[index_i].classes[c] == 0: continue
            for j in range(i+1, len(sorted_indices)):
                index_j = sorted_indices[j]
                if bbox_iou(boxes[index_i], boxes[index_j]) >= nms_thresh:
                    boxes[index_j].classes[c] = 0

# load and prepare an image
def load_image_pixels(filename, shape):
    # load the image to get its shape
    image = load_img(filename)
    width, height = image.size
    # load the image with the required size
    image = load_img(filename, target_size=shape)
    # convert to numpy array
    image = img_to_array(image)
    # scale pixel values to [0, 1]
    image = image.astype('float32')
    image /= 255.0
    # add a dimension so that we have one sample
    image = expand_dims(image, 0)
    return image, width, height

# get all of the results above a threshold
def get_boxes(boxes, labels, thresh):
    v_boxes, v_labels, v_scores = list(), list(), list()
    # enumerate all boxes
    for box in boxes:
```

```
# enumerate all possible labels
for i in range(len(labels)):
    # check if the threshold for this label is high enough
    if box.classes[i] > thresh:
        v_boxes.append(box)
        v_labels.append(labels[i])
        v_scores.append(box.classes[i]*100)
    # don't break, many labels may trigger for one box
return v_boxes, v_labels, v_scores

# draw all results
def draw_boxes(filename, v_boxes, v_labels, v_scores):
    # load the image
    data = pyplot.imread(filename)
    # plot the image
    pyplot.imshow(data)
    # get the context for drawing boxes
    ax = pyplot.gca()
    # plot each box
    for i in range(len(v_boxes)):
        box = v_boxes[i]
        # get coordinates
        y1, x1, y2, x2 = box.ymin, box.xmin, box.ymax, box.xmax
        # calculate width and height of the box
        width, height = x2 - x1, y2 - y1
        # create the shape
        rect = Rectangle((x1, y1), width, height, fill=False, color='white')
        # draw the box
        ax.add_patch(rect)
        # draw text and score in top left corner
        label = "%s (%.3f)" % (v_labels[i], v_scores[i])
        pyplot.text(x1, y1, label, color='white')
    # show the plot
    pyplot.show()

# load yolov3 model
model = load_model('model.h5')
# define the expected input shape for the model
input_w, input_h = 416, 416
# define our new photo
photo_filename = 'zebra.jpg'
# load and prepare image
image, image_w, image_h = load_image_pixels(photo_filename, (input_w, input_h))
# make prediction
yhat = model.predict(image)
# summarize the shape of the list of arrays
print([a.shape for a in yhat])
# define the anchors
anchors = [[116,90, 156,198, 373,326], [30,61, 62,45, 59,119], [10,13, 16,30, 33,23]]
# define the probability threshold for detected objects
class_threshold = 0.6
boxes = list()
for i in range(len(yhat)):
    # decode the output of the network
    boxes += decode_netout(yhat[i][0], anchors[i], class_threshold, input_h, input_w)
    # correct the sizes of the bounding boxes for the shape of the image
```

```

correct_yolo_boxes(boxes, image_h, image_w, input_h, input_w)
# suppress non-maximal boxes
do_nms(boxes, 0.5)
# define the labels
labels = ["person", "bicycle", "car", "motorbike", "aeroplane", "bus", "train", "truck",
"boat", "traffic light", "fire hydrant", "stop sign", "parking meter", "bench",
"bird", "cat", "dog", "horse", "sheep", "cow", "elephant", "bear", "zebra", "giraffe",
"backpack", "umbrella", "handbag", "tie", "suitcase", "frisbee", "skis", "snowboard",
"sports ball", "kite", "baseball bat", "baseball glove", "skateboard", "surfboard",
"tennis racket", "bottle", "wine glass", "cup", "fork", "knife", "spoon", "bowl",
"banana",
"apple", "sandwich", "orange", "broccoli", "carrot", "hot dog", "pizza", "donut", "cake",
"chair", "sofa", "pottedplant", "bed", "diningtable", "toilet", "tvmonitor", "laptop",
"mouse",
"remote", "keyboard", "cell phone", "microwave", "oven", "toaster", "sink",
"refrigerator",
"book", "clock", "vase", "scissors", "teddy bear", "hair drier", "toothbrush"]
# get the details of the detected objects
v_boxes, v_labels, v_scores = get_boxes(boxes, labels, class_threshold)
# summarize what we found
for i in range(len(v_boxes)):
    print(v_labels[i], v_scores[i])
# draw what we found
draw_boxes(photo_filename, v_boxes, v_labels, v_scores)

```

Listing 24.25: Example of object detection with YOLOv3 model in Keras.

Running the example again prints the shape of the raw output from the model. This is followed by a summary of the objects detected by the model and their confidence. We can see that the model has detected three zebra, all above 90% likelihood.

```

[(1, 13, 13, 255), (1, 26, 26, 255), (1, 52, 52, 255)]
zebra 94.91060376167297
zebra 99.86329674720764
zebra 96.8708872795105

```

Listing 24.26: Example output from performing object detection with the YOLOv3 model.

A plot of the photograph is created and the three bounding boxes are plotted. We can see that the model has indeed successfully detected the three zebra in the photograph.

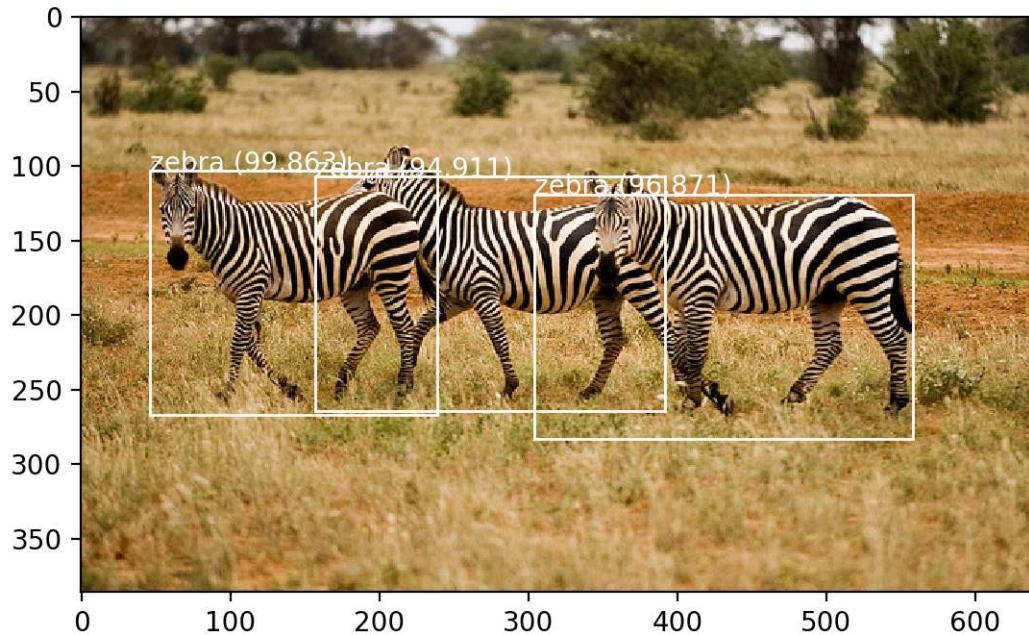


Figure 24.3: Photograph of Three Zebra Each Detected with the YOLOv3 Model and Localized with Bounding Boxes.

24.5 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Your Own Image.** Use the YOLO model to perform object recognition on your own photograph.
- **Video Data.** Use a library such as OpenCV to load a video and perform object detection on each frame of the video.
- **Use DarkNet.** Experiment by downloading and using the official YOLO/DarkNet implementation and compare results to the Keras implementation.

If you explore any of these extensions, I'd love to know.

24.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

24.6.1 Papers

- *You Only Look Once: Unified, Real-Time Object Detection*, 2015.
<https://arxiv.org/abs/1506.02640>
- *YOLO9000: Better, Faster, Stronger*, 2016.
<https://arxiv.org/abs/1612.08242>
- *YOLOv3: An Incremental Improvement*, 2018.
<https://arxiv.org/abs/1804.02767>

24.6.2 API

- `matplotlib.patches.Rectangle` API.
https://matplotlib.org/api/_as_gen/matplotlib.patches.Rectangle.html

24.6.3 Resources

- YOLO: Real-Time Object Detection, Homepage.
<https://pjreddie.com/darknet/yolo/>
- Official DarkNet and YOLO Source Code, GitHub.
<https://github.com/pjreddie/darknet>
- Official YOLO: Real Time Object Detection.
<https://github.com/pjreddie/darknet/wiki/YOLO:-Real-Time-Object-Detection>
- Huynh Ngoc Anh, experiencor, Home Page.
<https://experiencor.github.io/>
- experiencor/keras-yolo3, GitHub.
<https://github.com/experiencor/keras-yolo3>

24.6.4 Other YOLO for Keras Projects

- allanzelener/YAD2K, GitHub.
<https://github.com/allanzelener/YAD2K>
- qqwwweee/keras-yolo3, GitHub.
<https://github.com/qqwwweee/keras-yolo3>
- xiaochus/YOLOv3 GitHub.
<https://github.com/xiaochus/YOL0v3>

24.7 Summary

In this tutorial, you discovered how to develop a YOLOv3 model for object detection on new photographs. Specifically, you learned:

- YOLO-based Convolutional Neural Network family of models for object detection and the most recent variation called YOLOv3.
- The best-of-breed open source library implementation of the YOLOv3 for the Keras deep learning library.
- How to use a pre-trained YOLOv3 to perform object localization and detection on new photographs.

24.7.1 Next

In the next section, you will discover how to perform object localization and object recognition using the Mask R-CNN model.

Chapter 25

How to Perform Object Detection With Mask R-CNN

Object detection is a task in computer vision that involves identifying the presence, location, and type of one or more objects in a given photograph. It is a challenging problem that involves building upon methods for object recognition (e.g. where are they), object localization (e.g. what are their extent), and object classification (e.g. what are they). In recent years, deep learning techniques have achieved state-of-the-art results for object detection, such as on standard benchmark datasets and in computer vision competitions. Most notably is the R-CNN, or Region-Based Convolutional Neural Networks, and the most recent technique called Mask R-CNN that is capable of achieving state-of-the-art results on a range of object detection tasks. In this tutorial, you will discover how to use the Mask R-CNN model to detect objects in new photographs. After completing this tutorial, you will know:

- The region-based Convolutional Neural Network family of models for object detection and the most recent variation called Mask R-CNN.
- The best-of-breed open source library implementation of the Mask R-CNN for the Keras deep learning library.
- How to use a pre-trained Mask R-CNN to perform object localization and detection on new photographs.

Let's get started.

Note: This tutorial requires TensorFlow version 1.15.3 and Keras 2.2.4. It currently does not work with TensorFlow 2.0+ or Keras 2.2.5+ because a third-party library has not been updated at the time of writing.

25.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. R-CNN and Mask R-CNN
2. Matterport Mask R-CNN Project
3. Object Detection with Mask R-CNN

25.2 Mask R-CNN for Object Detection

Object detection is a computer vision task that involves both localizing one or more objects within an image and classifying each object in the image. It is a challenging computer vision task that requires both successful object localization in order to locate and draw a bounding box around each object in an image, and object classification to predict the correct class of object that was localized. An extension of object detection involves marking the specific pixels in the image that belong to each detected object instead of using coarse bounding boxes during object localization. This harder version of the problem is generally referred to as object segmentation or semantic segmentation.

The Region-Based Convolutional Neural Network, or R-CNN, is a family of convolutional neural network models designed for object detection, developed by Ross Girshick, et al (introduced in Chapter 23). There are perhaps four main variations of the approach, resulting in the current pinnacle called Mask R-CNN. The salient aspects of each variation can be summarized as follows:

- **R-CNN:** Bounding boxes are proposed by the *selective search* algorithm, each of which is stretched and features are extracted via a deep convolutional neural network, such as AlexNet, before a final set of object classifications are made with linear SVMs.
- **Fast R-CNN:** Simplified design with a single model, bounding boxes are still specified as input, but a region-of-interest pooling layer is used after the deep CNN to consolidate regions and the model predicts both class labels and regions of interest directly.
- **Faster R-CNN:** Addition of a Region Proposal Network that interprets features extracted from the deep CNN and learns to propose regions-of-interest directly.
- **Mask R-CNN:** Extension of Faster R-CNN that adds an output model for predicting a mask for each detected object.

The Mask R-CNN model introduced in the 2018 paper titled *Mask R-CNN* is the most recent variation of the R-CNN family models and supports both object detection and object segmentation. The paper provides a nice summary of the model lineage to that point:

The Region-based CNN (R-CNN) approach to bounding-box object detection is to attend to a manageable number of candidate object regions and evaluate convolutional networks independently on each RoI. R-CNN was extended to allow attending to RoIs on feature maps using RoIPool, leading to fast speed and better accuracy. Faster R-CNN advanced this stream by learning the attention mechanism with a Region Proposal Network (RPN). Faster R-CNN is flexible and robust to many follow-up improvements, and is the current leading framework in several benchmarks.

— *Mask R-CNN*, 2018.

The family of methods may be among the most effective for object detection, achieving then state-of-the-art results on computer vision benchmark datasets. Although accurate, the models can be slow when making a prediction as compared to alternate models such as YOLO that may be less accurate but are designed for real-time prediction.

25.3 Matterport Mask R-CNN Project

Mask R-CNN is a sophisticated model to implement, especially as compared to a simple or even state-of-the-art deep convolutional neural network model. Source code is available for each version of the R-CNN model, provided in separate GitHub repositories with prototype models based on the Caffe deep learning framework. For example, see the *Further Reading* section at the end of the tutorial. Instead of developing an implementation of the R-CNN or Mask R-CNN model from scratch, we can use a reliable third-party implementation built on top of the Keras deep learning framework.

The best of breed third-party implementations of Mask R-CNN is the Mask R-CNN Project developed by Matterport. The project is open source released under a permissive license (i.e. MIT license) and the code has been widely used on a variety of projects and Kaggle competitions¹. The project is light on API documentation, although it does provide a number of examples in the form of Python Notebooks that you can use to understand how to use the library by example. There are perhaps three main use cases for using the Mask R-CNN model with the Matterport library; they are:

- **Object Detection Application:** Use a pre-trained model for object detection on new images.
- **New Model via Transfer Learning:** Use a pre-trained model as a starting point in developing a model for a new object detection dataset.
- **New Model from Scratch:** Develop a new model from scratch for an object detection dataset.

In order to get familiar with the model and the library, we will look at the first example in the next section.

25.4 Object Detection With Mask R-CNN

In this section, we will use the Matterport Mask R-CNN library to perform object detection on arbitrary photographs. Much like using a pre-trained deep CNN for image classification, e.g. such as VGG-16 trained on an ImageNet dataset, we can use a pre-trained Mask R-CNN model to detect objects in new photographs. In this case, we will use a Mask R-CNN trained on the MSCOCO object detection problem.

25.4.1 Mask R-CNN Installation

The first step is to install the library. At the time of writing, there is no distributed version of the library, so we have to install it manually. The good news is that this is very easy. Installation involves cloning the GitHub repository and running the installation script on your workstation. If you are having trouble, see the installation instructions in the library's readme file.

¹Clone of the project is available here: https://github.com/jbrownlee/Mask_RCNN

Step 1. Clone the Mask R-CNN GitHub Repository

This is as simple as running the following command from your command line:

```
git clone https://github.com/matterport/Mask_RCNN.git
```

Listing 25.1: Example of cloning the Mask_RCNN project.

This will create a new local directory with the name Mask_RCNN that looks as follows:

```
Mask_RCNN
├── assets
└── images
    ├── mrcnn
    └── samples
        ├── balloon
        ├── coco
        ├── nucleus
        └── shapes
```

Listing 25.2: Example of Mask_RCNN project directory structure.

Step 2. Install the Mask R-CNN Library

The library can be installed directly via pip. Change directory into the Mask_RCNN project directory and run the installation script. From the command line, type the following:

```
cd Mask_RCNN
python setup.py install
```

Listing 25.3: Example of installing the Mask_RCNN project.

On Linux or MacOS you may need to install the software with appropriate permissions; for example, you may see an error such as:

```
error: can't create or remove files in install directory
```

Listing 25.4: Example of possible error when installing the Mask_RCNN project.

In that case, install the software with sudo:

```
sudo python setup.py install
```

Listing 25.5: Example of installing the Mask_RCNN project with sudo.

The library will then install directly and you will see a lot of successful installation messages ending with the following:

```
...
Finished processing dependencies for mask-rcnn==2.1
```

Listing 25.6: Example output from installing the Mask_RCNN project.

This confirms that you installed the library successfully and that you have the latest version, which at the time of writing is version 2.1.

Step 3: Confirm the Library Was Installed

It is always a good idea to confirm that the library was installed correctly. You can confirm that the library was installed correctly by querying it via the pip command; for example:

```
pip show mask-rcnn
```

Listing 25.7: Example of checking that the Mask_RCNN library was installed.

You should see output informing you of the version and installation location; for example:

```
Name: mask-rcnn
Version: 2.1
Summary: Mask R-CNN for object detection and instance segmentation
Home-page: https://github.com/matterport/Mask_RCNN
Author: Matterport
Author-email: waleed.abdulla@gmail.com
License: MIT
Location: ...
Requires:
Required-by:
```

Listing 25.8: Example of output from checking that the Mask_RCNN library was installed.

We are now ready to use the library.

25.4.2 Example of Object Localization

We are going to use a pre-trained Mask R-CNN model to detect objects on a new photograph.

Step 1. Download Model Weights

First, download the weights for the pre-trained model, specifically a Mask R-CNN trained on the MS Coco dataset. The weights are available from the GitHub project and the file is about 250 megabytes. Download the model weights to a file with the name `mask_rcnn_coco.h5` in your current working directory.

- Download Model Weights (`mask_rcnn_coco.h5`) (246 megabytes).²

Step 2. Download Sample Photograph

We also need a photograph in which to detect objects. We will use a photograph from Flickr released under a permissive license, specifically a photograph of an elephant taken by Mandy Goldberg³.

²<https://goo.gl/a2p7dS>

³<https://www.flickr.com/photos/viewfrom52/2081198423/>



Figure 25.1: Photograph of an Elephant.

Download the photograph to your current working directory with the filename `elephant.jpg`.

- Download Photograph (`elephant.jpg`).⁴

Step 3. Load Model and Make Prediction

First, the model must be defined via an instance of the `MaskRCNN` class. This class requires a configuration object as a parameter. The configuration object defines how the model might be used during training or inference. In this case, the configuration will only specify the number of images per batch, which will be one, and the number of classes to predict. You can see the full extent of the configuration object and the properties that you can override in the `config.py` file.

```
# define the test configuration
class TestConfig(Config):
    NAME = "test"
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1
    NUM_CLASSES = 1 + 80
```

Listing 25.9: Configuration object for the RCNN model.

We can now define the `MaskRCNN` instance. We will define the model as type *inference* indicating that we are interested in making predictions and not training. We must also specify a directory where any log messages could be written, which in this case will be the current working directory.

⁴<https://machinelearningmastery.com/wp-content/uploads/2019/03/elephant.jpg>

```
...
# define the model
rcnn = MaskRCNN(mode='inference', model_dir='./', config=TestConfig())
```

Listing 25.10: Example of defining the RCNN model.

The next step is to load the weights that we downloaded.

```
...
# load coco model weights
rcnn.load_weights('mask_rcnn_coco.h5', by_name=True)
```

Listing 25.11: Example of loading the pre-trained weights for the RCNN model.

Now we can make a prediction for our image. First, we can load the image and convert it to a NumPy array.

```
...
# load photograph
img = load_img('elephant.jpg')
img = img_to_array(img)
```

Listing 25.12: Example of loading the image for making a prediction.

We can then make a prediction with the model. Instead of calling `predict()` as we would on a normal Keras model, will call the `detect()` function and pass it the single image.

```
...
# make prediction
results = rcnn.detect([img], verbose=0)
```

Listing 25.13: Example of making a prediction with the RCNN model.

The result contains a dictionary for each image that we passed into the `detect()` function, in this case, a list of a single dictionary for the one image. The dictionary has keys for the bounding boxes, masks, and so on, and each key points to a list for multiple possible objects detected in the image. The keys of the dictionary of note are as follows:

- ‘`rois`’: The bound boxes or regions-of-interest (ROI) for detected objects.
- ‘`masks`’: The masks for the detected objects.
- ‘`class_ids`’: The class integers for the detected objects.
- ‘`scores`’: The probability or confidence for each predicted class.

We can draw each box detected in the image by first getting the dictionary for the first image (e.g. `results[0]`), and then retrieving the list of bounding boxes (e.g. `['rois']`).

```
...
# get regions of interest
boxes = results[0]['rois']
```

Listing 25.14: Example of getting the bounding boxes from a prediction.

Each bounding box is defined in terms of the bottom left and top right coordinates of the bounding box in the image

```
...
# interpret bounding box
y1, x1, y2, x2 = boxes[0]
```

Listing 25.15: Example of interpreting a bounding box.

We can use these coordinates to create a `Rectangle` from the Matplotlib API and draw each rectangle over the top of our image.

```
...
# get coordinates
y1, x1, y2, x2 = box
# calculate width and height of the box
width, height = x2 - x1, y2 - y1
# create the shape
rect = Rectangle((x1, y1), width, height, fill=False, color='red')
# draw the box
ax.add_patch(rect)
```

Listing 25.16: Example of drawing a bounding box.

To keep things neat, we can create a function to do this that will take the filename of the photograph and the list of bounding boxes to draw and will show the photo with the boxes.

```
# draw an image with detected objects
def draw_image_with_boxes(filename, boxes_list):
    # load the image
    data = pyplot.imread(filename)
    # plot the image
    pyplot.imshow(data)
    # get the context for drawing boxes
    ax = pyplot.gca()
    # plot each box
    for box in boxes_list:
        # get coordinates
        y1, x1, y2, x2 = box
        # calculate width and height of the box
        width, height = x2 - x1, y2 - y1
        # create the shape
        rect = Rectangle((x1, y1), width, height, fill=False, color='red')
        # draw the box
        ax.add_patch(rect)
    # show the plot
    pyplot.show()
```

Listing 25.17: Example of a function for drawing predicted bounding boxes.

We can now tie all of this together and load the pre-trained model and use it to detect objects in our photograph of an elephant, then draw the photograph with all detected objects. The complete example is listed below.

```
# example of inference with a pre-trained coco model
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from mrcnn.config import Config
from mrcnn.model import MaskRCNN
from matplotlib import pyplot
```

```

from matplotlib.patches import Rectangle

# draw an image with detected objects
def draw_image_with_boxes(filename, boxes_list):
    # load the image
    data = pyplot.imread(filename)
    # plot the image
    pyplot.imshow(data)
    # get the context for drawing boxes
    ax = pyplot.gca()
    # plot each box
    for box in boxes_list:
        # get coordinates
        y1, x1, y2, x2 = box
        # calculate width and height of the box
        width, height = x2 - x1, y2 - y1
        # create the shape
        rect = Rectangle((x1, y1), width, height, fill=False, color='red')
        # draw the box
        ax.add_patch(rect)
    # show the plot
    pyplot.show()

# define the test configuration
class TestConfig(Config):
    NAME = "test"
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1
    NUM_CLASSES = 1 + 80

# define the model
rcnn = MaskRCNN(mode='inference', model_dir='./', config=TestConfig())
# load coco model weights
rcnn.load_weights('mask_rcnn_coco.h5', by_name=True)
# load photograph
img = load_img('elephant.jpg')
img = img_to_array(img)
# make prediction
results = rcnn.detect([img], verbose=0)
# visualize the results
draw_image_with_boxes('elephant.jpg', results[0]['rois'])

```

Listing 25.18: Example of localizing objects with a Mask R-CNN model.

Running the example loads the model and performs object detection. More accurately, we have performed object localization, only drawing bounding boxes around detected objects. In this case, we can see that the model has correctly located the single object in the photo, the elephant, and drawn a red box around it.

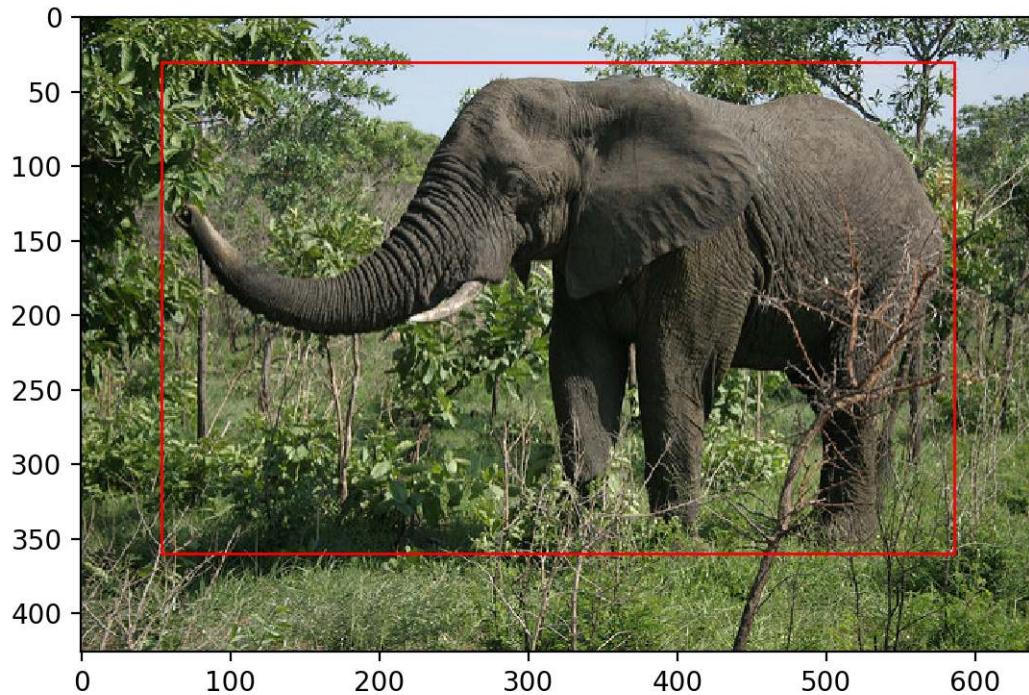


Figure 25.2: Photograph of an Elephant With All Objects Localized With a Bounding Box.

25.4.3 Example of Object Detection

Now that we know how to load the model and use it to make a prediction, let's update the example to perform real object detection. That is, in addition to localizing objects, we want to know what they are. The Mask_RCNN API provides a function called `display_instances()` that will take the array of pixel values for the loaded image and the aspects of the prediction dictionary, such as the bounding boxes, scores, and class labels, and will plot the photo with all of these annotations. One of the arguments is the list of predicted class identifiers available in the '`class_ids`' key of the dictionary. The function also needs a mapping of ids to class labels. The pre-trained model was fit with a dataset that had 80 (81 including background) class labels, helpfully provided as a list in the Mask R-CNN Demo, Notebook Tutorial, listed below.

```
...
# define 81 classes that the coco model knows about
class_names = ['BG', 'person', 'bicycle', 'car', 'motorcycle', 'airplane',
               'bus', 'train', 'truck', 'boat', 'traffic light',
               'fire hydrant', 'stop sign', 'parking meter', 'bench', 'bird',
               'cat', 'dog', 'horse', 'sheep', 'cow', 'elephant', 'bear',
               'zebra', 'giraffe', 'backpack', 'umbrella', 'handbag', 'tie',
               'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
               'kite', 'baseball bat', 'baseball glove', 'skateboard',
               'surfboard', 'tennis racket', 'bottle', 'wine glass', 'cup',
```

```
'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple',
'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza',
'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed',
'dining table', 'toilet', 'tv', 'laptop', 'mouse', 'remote',
'keyboard', 'cell phone', 'microwave', 'oven', 'toaster',
'sink', 'refrigerator', 'book', 'clock', 'vase', 'scissors',
'teddy bear', 'hair drier', 'toothbrush']
```

Listing 25.19: Example of class labels from the MSCOCO dataset.

We can then provide the details of the prediction for the elephant photo to the `display_instances()` function; for example:

```
...
# get dictionary for first prediction
r = results[0]
# show photo with bounding boxes, masks, class labels and scores
display_instances(img, r['rois'], r['masks'], r['class_ids'], class_names, r['scores'])
```

Listing 25.20: Example of displaying the results from detecting objects.

The `display_instances()` function is flexible, allowing you to only draw the mask or only the bounding boxes. You can learn more about this function in the `visualize.py` source file. The complete example with this change using the `display_instances()` function is listed below.

```
# example of inference with a pre-trained coco model
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from mrcnn.visualize import display_instances
from mrcnn.config import Config
from mrcnn.model import MaskRCNN

# define 81 classes that the coco model knows about
class_names = ['BG', 'person', 'bicycle', 'car', 'motorcycle', 'airplane',
               'bus', 'train', 'truck', 'boat', 'traffic light',
               'fire hydrant', 'stop sign', 'parking meter', 'bench', 'bird',
               'cat', 'dog', 'horse', 'sheep', 'cow', 'elephant', 'bear',
               'zebra', 'giraffe', 'backpack', 'umbrella', 'handbag', 'tie',
               'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
               'kite', 'baseball bat', 'baseball glove', 'skateboard',
               'surfboard', 'tennis racket', 'bottle', 'wine glass', 'cup',
               'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple',
               'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza',
               'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed',
               'dining table', 'toilet', 'tv', 'laptop', 'mouse', 'remote',
               'keyboard', 'cell phone', 'microwave', 'oven', 'toaster',
               'sink', 'refrigerator', 'book', 'clock', 'vase', 'scissors',
               'teddy bear', 'hair drier', 'toothbrush']

# define the test configuration
class TestConfig(Config):
    NAME = "test"
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1
    NUM_CLASSES = 1 + 80
```

```
# define the model
rcnn = MaskRCNN(mode='inference', model_dir='./', config=TestConfig())
# load coco model weights
rcnn.load_weights('mask_rcnn_coco.h5', by_name=True)
# load photograph
img = load_img('elephant.jpg')
img = img_to_array(img)
# make prediction
results = rcnn.detect([img], verbose=0)
# get dictionary for first prediction
r = results[0]
# show photo with bounding boxes, masks, class labels and scores
display_instances(img, r['rois'], r['masks'], r['class_ids'], class_names, r['scores'])
```

Listing 25.21: Example of object detection with a Mask R-CNN model.

Running the example shows the photograph of the elephant with the annotations predicted by the Mask R-CNN model, specifically:

- **Bounding Box.** Dotted bounding box around each detected object.
- **Class Label.** Class label assigned each detected object written in the top left corner of the bounding box.
- **Prediction Confidence.** Confidence of class label prediction for each detected object written in the top left corner of the bounding box.
- **Object Mask Outline.** Polygon outline for the mask of each detected object.
- **Object Mask.** Polygon fill for the mask of each detected object.

The result is very impressive and sparks many ideas for how such a powerful pre-trained model could be used in practice.



Figure 25.3: Photograph of an Elephant With All Objects Detected With a Bounding Box and Mask.

25.5 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Your Own Image.** Use the Mask R-CNN model for object detection on your own photograph.
- **Applications.** Brainstorm 3 applications of object detection using the Mask R-CNN model.
- **Apply to Dataset.** Apply the Mask R-CNN model to a standard object recognition dataset and evaluate performance, such as MSCOCO.

If you explore any of these extensions, I'd love to know.

25.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

25.6.1 Papers

- *Rich Feature Hierarchies For Accurate Object Detection And Semantic Segmentation*, 2013.
<https://arxiv.org/abs/1311.2524>
- *Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition*, 2014.
<https://arxiv.org/abs/1406.4729>
- *Fast R-CNN*, 2015.
<https://arxiv.org/abs/1504.08083>
- *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*, 2016.
<https://arxiv.org/abs/1506.01497>
- *Mask R-CNN*, 2017.
<https://arxiv.org/abs/1703.06870>

25.6.2 API

- `matplotlib.patches.Rectangle` API.
https://matplotlib.org/api/_as_gen/matplotlib.patches.Rectangle.html

25.6.3 R-CNN Code Repositories

- R-CNN: Regions with Convolutional Neural Network Features, GitHub.
<https://github.com/rbgirshick/rcnn>
- Fast R-CNN, GitHub.
<https://github.com/rbgirshick/fast-rcnn>
- Faster R-CNN Python Code, GitHub.
<https://github.com/rbgirshick/py-faster-rcnn>
- Detectron, Facebook AI, GitHub.
<https://github.com/facebookresearch/Detectron>
- Mask R-CNN, GitHub.
https://github.com/matterport/Mask_RCNN

25.7 Summary

In this tutorial, you discovered how to use the Mask R-CNN model to detect objects in new photographs. Specifically, you learned:

- The region-based Convolutional Neural Network family of models for object detection and the most recent variation called Mask R-CNN.
- The best-of-breed open source library implementation of the Mask R-CNN for the Keras deep learning library.
- How to use a pre-trained Mask R-CNN to perform object localization and detection on new photographs.

25.7.1 Next

In the next section, you will discover how to develop an object recognition system using Mask R-CNN to identify kangaroos in photographs.

Chapter 26

How to Develop a New Object Detection Model

Object detection is a challenging computer vision task that involves predicting both where the objects are in the image and what type of objects were detected. The Mask Region-based Convolutional Neural Network, or Mask R-CNN, model is one of the state-of-the-art approaches for object recognition tasks. The Matterport Mask R-CNN project provides a library that allows you to develop and train Mask R-CNN Keras models for your own object detection tasks. Using the library can be tricky for beginners and requires the careful preparation of the dataset, although it allows fast training via transfer learning with top performing models trained on challenging object detection tasks, such as MSCOCO. In this tutorial, you will discover how to develop a Mask R-CNN model for kangaroo object detection in photographs. After completing this tutorial, you will know:

- How to prepare an object detection dataset ready for modeling with an R-CNN.
- How to use transfer learning to train an object detection model on a new dataset.
- How to evaluate a fit Mask R-CNN model on a test dataset and make predictions on new photos.

Let's get started.

Note: This tutorial requires TensorFlow version 1.15.3 and Keras 2.2.4. It currently does not work with TensorFlow 2.0+ or Keras 2.2.5+ because a third-party library has not been updated at the time of writing.

26.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. How to Install Mask R-CNN for Keras
2. How to Prepare a Dataset for Object Detection
3. How to Train a Mask R-CNN Model for Kangaroo Detection

4. How to Evaluate a Mask R-CNN Model
5. How to Detect Kangaroos in New Photos

26.2 How to Install Mask R-CNN for Keras

Object detection is a task in computer vision that involves identifying the presence, location, and type of one or more objects in a given image. It is a challenging problem that involves building upon methods for object recognition (e.g. where are they), object localization (e.g. what are their extent), and object classification (e.g. what are they).

The Region-Based Convolutional Neural Network, or R-CNN, is a family of convolutional neural network models designed for object detection, developed by Ross Girshick, et al. (introduced in Chapter 23). There are perhaps four main variations of the approach, resulting in the current pinnacle called Mask R-CNN. The Mask R-CNN introduced in the 2018 paper titled *Mask R-CNN* is the most recent variation of the family of models and supports both object detection and object segmentation. Object segmentation not only involves localizing objects in the image but also specifies a mask for the image, indicating exactly which pixels in the image belong to the object. Mask R-CNN is a sophisticated model to implement, especially as compared to a simple or even state-of-the-art deep convolutional neural network model. Instead of developing an implementation of the R-CNN or Mask R-CNN model from scratch, we can use a reliable third-party implementation built on top of the Keras deep learning framework.

The best-of-breed third-party implementation of Mask R-CNN is the Mask R-CNN Project developed by Matterport. The project is open source released under a permissive license (e.g. MIT license) and the code has been widely used on a variety of projects and Kaggle competitions¹. The first step is to install the library. **Note**, this was covered in Chapter 25 and is summarized here for completeness. If you have already installed the Mask R-CNN library, you can skip this section.

At the time of writing, there is no distributed version of the library, so we have to install it manually. The good news is that this is very easy. Installation involves cloning the GitHub repository and running the installation script on your workstation. If you are having trouble, see the installation instructions in the library's readme file.

26.2.1 Step 1. Clone the Mask R-CNN GitHub Repository

This is as simple as running the following command from your command line:

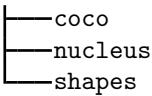
```
git clone https://github.com/matterport/Mask_RCNN.git
```

Listing 26.1: Example of cloning the Mask_RCNN project.

This will create a new local directory with the name Mask_RCNN that looks as follows:

```
Mask_RCNN
├── assets
├── images
└── mrcnn
    └── samples
        └── balloon
```

¹Clone of the project is available here: https://github.com/jbrownlee/Mask_RCNN



Listing 26.2: Example of Mask_RCNN project directory structure.

26.2.2 Step 2. Install the Mask R-CNN Library

The library can be installed directly via pip. Change directory into the Mask_RCNN project directory and run the installation script. From the command line, type the following:

```
cd Mask_RCNN
python setup.py install
```

Listing 26.3: Example of installing the Mask_RCNN project.

On Linux or MacOS you may need to install the software with appropriate permissions; for example, you may see an error such as:

```
error: can't create or remove files in install directory
```

Listing 26.4: Example of possible error when installing the Mask_RCNN project.

In that case, install the software with sudo:

```
sudo python setup.py install
```

Listing 26.5: Example of installing the Mask_RCNN project with sudo.

The library will then install directly and you will see a lot of successful installation messages ending with the following:

```
...
Finished processing dependencies for mask-rcnn==2.1
```

Listing 26.6: Example output from installing the Mask_RCNN project.

This confirms that you installed the library successfully and that you have the latest version, which at the time of writing is version 2.1.

26.2.3 Step 3: Confirm the Library Was Installed

It is always a good idea to confirm that the library was installed correctly. You can confirm that the library was installed correctly by querying it via the pip command; for example:

```
pip show mask-rcnn
```

Listing 26.7: Example of checking that the Mask_RCNN library was installed.

You should see output informing you of the version and installation location; for example:

```
Name: mask-rcnn
Version: 2.1
Summary: Mask R-CNN for object detection and instance segmentation
Home-page: https://github.com/matterport/Mask_RCNN
Author: Matterport
Author-email: waleed.abdulla@gmail.com
```

```
License: MIT
Location: ...
Requires:
Required-by:
```

Listing 26.8: Example of output from checking that the Mask_RCNN library was installed.

We are now ready to use the library.

26.3 How to Prepare a Dataset for Object Detection

Next, we need a dataset to model. In this tutorial, we will use the kangaroo dataset, made available by Huynh Ngoc Anh (*experiencor*). The dataset is comprised of about 160 photographs that contain kangaroos, and XML annotation files that provide bounding boxes for the kangaroos in each photograph. The Mask R-CNN is designed to learn to predict both bounding boxes for objects as well as masks for those detected objects, and the kangaroo dataset does not provide masks. As such, we will use the dataset to learn a kangaroo object detection task, and ignore the masks and not focus on the image segmentation capabilities of the model.

There are a few steps required in order to prepare this dataset for modeling and we will work through each in turn in this section, including downloading the dataset, parsing the annotations file, developing a `KangarooDataset` object that can be used by the Mask_RCNN library, then testing the dataset object to confirm that we are loading images and annotations correctly.

26.3.1 Install Dataset

The first step is to download the dataset into your current working directory. This can be achieved by cloning the GitHub repository directly, as follows:

```
git clone https://github.com/experiencor/kangaroo.git
```

Listing 26.9: Example of cloning the kangaroo dataset project.

This will create a new directory called `kangaroo` with a subdirectory called `images/` that contains all of the JPEG photos of kangaroos and a subdirectory called `annotes/` that contains all of the XML files that describe the locations of kangaroos in each photo.

```
kangaroo
└── annotes
└── images
```

Listing 26.10: Summary of the directories in the kangaroo dataset project.

Looking in each subdirectory, you can see that the photos and annotation files use a consistent naming convention, with filenames using a 5-digit zero-padded numbering system; for example:

```
images/00001.jpg
images/00002.jpg
images/00003.jpg
...
annots/00001.xml
annots/00002.xml
annots/00003.xml
...
```

Listing 26.11: Example of the files in the kangaroo dataset project.

This makes matching photographs and annotation files together very easy. We can also see that the numbering system is not contiguous, that there are some photos missing, e.g. there is no 00007 JPG or XML file. This means that we should focus on loading the list of actual files in the directory rather than using a numbering system.

26.3.2 Parse Annotation File

The next step is to figure out how to load the annotation files. First, open the first annotation file (`annots/00001.xml`) and take a look; you should see:

```
<annotation>
  <folder>Kangaroo</folder>
  <filename>00001.jpg</filename>
  <path>...</path>
  <source>
    <database>Unknown</database>
  </source>
  <size>
    <width>450</width>
    <height>319</height>
    <depth>3</depth>
  </size>
  <segmented>0</segmented>
  <object>
    <name>kangaroo</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>233</xmin>
      <ymin>89</ymin>
      <xmax>386</xmax>
      <ymax>262</ymax>
    </bndbox>
  </object>
  <object>
    <name>kangaroo</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>134</xmin>
      <ymin>105</ymin>
      <xmax>341</xmax>
      <ymax>253</ymax>
    </bndbox>
  </object>
</annotation>
```

Listing 26.12: Example of an annotation file.

We can see that the annotation file contains a `size` element that describes the shape of the photograph, and one or more `object` elements that describe the bounding boxes for the kangaroo objects in the photograph. The size and the bounding boxes are the minimum information that we require from each annotation file. We could write some careful XML parsing code to process these annotation files, and that would be a good idea for a production system. Instead, we will short-cut development and use XPath queries to directly extract the data that we need from each file, e.g. a `//size` query to extract the size element and a `//object` or a `//bndbox` query to extract the bounding box elements. Python provides the `ElementTree` API that can be used to load and parse an XML file and we can use the `find()` and `.findall()` functions to perform the XPath queries on a loaded document. First, the annotation file must be loaded and parsed as an `ElementTree` object.

```
...
# load and parse the file
tree = ElementTree.parse(filename)
```

Listing 26.13: Example of loading and parsing an annotation file.

Once loaded, we can retrieve the root element of the document from which we can perform our XPath queries.

```
...
# get the root of the document
root = tree.getroot()
```

Listing 26.14: Example of getting the root of the parsed XML document.

We can use the `.findall()` function with a query for `./bndbox` to find all `bndbox` elements, then enumerate each to extract the `x` and `y`, `min` and `max` values that define each bounding box. The element text can also be parsed to integer values.

```
...
# extract each bounding box
for box in root.findall('.//bndbox'):
    xmin = int(box.find('xmin').text)
    ymin = int(box.find('ymin').text)
    xmax = int(box.find('xmax').text)
    ymax = int(box.find('ymax').text)
    coors = [xmin, ymin, xmax, ymax]
```

Listing 26.15: Example of extracting the bounding boxes from the XML document.

We can then collect the definition of each bounding box into a list. The dimensions of the image may also be helpful, which can be queried directly.

```
...
# extract image dimensions
width = int(root.find('.//size/width').text)
height = int(root.find('.//size/height').text)
```

Listing 26.16: Example of extracting the image size from the XML document.

We can tie all of this together into a function that will take the annotation filename as an argument, extract the bounding box and image dimension details, and return them for use. The `extract_boxes()` function below implements this behavior.

```
# function to extract bounding boxes from an annotation file
def extract_boxes(filename):
    # load and parse the file
    tree = ElementTree.parse(filename)
    # get the root of the document
    root = tree.getroot()
    # extract each bounding box
    boxes = []
    for box in root.findall('.//bndbox'):
        xmin = int(box.find('xmin').text)
        ymin = int(box.find('ymin').text)
        xmax = int(box.find('xmax').text)
        ymax = int(box.find('ymax').text)
        coors = [xmin, ymin, xmax, ymax]
        boxes.append(coors)
    # extract image dimensions
    width = int(root.find('.//size/width').text)
    height = int(root.find('.//size/height').text)
    return boxes, width, height
```

Listing 26.17: Example of a function for extracting details from an annotation file.

We can test out this function on our annotation files, for example, on the first annotation file in the directory. The complete example is listed below.

```
# example of extracting bounding boxes from an annotation file
from xml.etree import ElementTree

# function to extract bounding boxes from an annotation file
def extract_boxes(filename):
    # load and parse the file
    tree = ElementTree.parse(filename)
    # get the root of the document
    root = tree.getroot()
    # extract each bounding box
    boxes = []
    for box in root.findall('.//bndbox'):
        xmin = int(box.find('xmin').text)
        ymin = int(box.find('ymin').text)
        xmax = int(box.find('xmax').text)
        ymax = int(box.find('ymax').text)
        coors = [xmin, ymin, xmax, ymax]
        boxes.append(coors)
    # extract image dimensions
    width = int(root.find('.//size/width').text)
    height = int(root.find('.//size/height').text)
    return boxes, width, height

# extract details form annotation file
boxes, w, h = extract_boxes('kangaroo/annots/00001.xml')
# summarize extracted details
print(boxes, w, h)
```

Listing 26.18: Example of extracting bounding box details from an annotation file.

Running the example returns a list that contains the details of each bounding box in the annotation file, as well as two integers for the width and height of the photograph.

```
[[233, 89, 386, 262], [134, 105, 341, 253]] 450 319
```

Listing 26.19: Example output from extracting bounding box details from an annotation file.

Now that we know how to load the annotation file, we can look at using this functionality to develop a Dataset object.

26.3.3 Develop KangarooDataset Object

The mask-rcnn library requires that train, validation, and test datasets be managed by a `mrcnn.utils.Dataset` object. This means that a new class must be defined that extends the `mrcnn.utils.Dataset` class and defines a function to load the dataset, with any name you like such as `load_dataset()`, and provide two functions, one for loading a mask called `load_mask()` and one for loading an image reference (path or URL) called `image_reference()`.

```
# class that defines and loads the kangaroo dataset
class KangarooDataset(Dataset):
    # load the dataset definitions
    def load_dataset(self, dataset_dir, is_train=True):
        # ...

    # load the masks for an image
    def load_mask(self, image_id):
        # ...

    # load an image reference
    def image_reference(self, image_id):
        # ...
```

Listing 26.20: Example of a template for the Mask RCNN Dataset object.

To use a Dataset object, it is instantiated, then your custom load function must be called, then finally the built-in `prepare()` function is called. For example, we will create a new class called `KangarooDataset` that will be used as follows:

```
...
# prepare the dataset
train_set = KangarooDataset()
train_set.load_dataset(...)
train_set.prepare()
```

Listing 26.21: Example of using a Mask RCNN Dataset object.

The custom load function, e.g. `load_dataset()` is responsible for both defining the classes and for defining the images in the dataset. Classes are defined by calling the built-in `add_class()` function and specifying the `source` (the name of the dataset), the `class_id` or integer for the class (e.g. 1 for the first class as 0 is reserved for the background class), and the `class_name` (e.g. `kangaroo`).

```
...
# define one class
self.add_class("dataset", 1, "kangaroo")
```

Listing 26.22: Example of defining the class in the dataset.

Objects are defined by a call to the built-in `add_image()` function and specifying the `source` (the name of the dataset), a unique `image_id` (e.g. the filename without the file extension like `00001`), and the path for where the image can be loaded (e.g. `kangaroo/images/00001.jpg`).

This will define an `image_info` dictionary for the image that can be retrieved later via the index or order in which the image was added to the dataset. You can also specify other arguments that will be added to the image info dictionary, such as an `annotation` to define the annotation path.

```
...
# add to dataset
self.add_image('dataset', image_id='00001', path='kangaroo/images/00001.jpg',
annotation='kangaroo/annots/00001.xml')
```

Listing 26.23: Example of adding an image to the dataset.

For example, we can implement a `load_dataset()` function that takes the path to the dataset directory and loads all images in the dataset. Note, testing revealed that there is an issue with image number `00090`, so we will exclude it from the dataset.

```
# load the dataset definitions
def load_dataset(self, dataset_dir):
    # define one class
    self.add_class("dataset", 1, "kangaroo")
    # define data locations
    images_dir = dataset_dir + '/images/'
    annotations_dir = dataset_dir + '/annots/'
    # find all images
    for filename in.listdir(images_dir):
        # extract image id
        image_id = filename[:-4]
        # skip bad images
        if image_id in ['00090']:
            continue
        img_path = images_dir + filename
        ann_path = annotations_dir + image_id + '.xml'
        # add to dataset
        self.add_image('dataset', image_id=image_id, path=img_path, annotation=ann_path)
```

Listing 26.24: Example of defining a function to load the dataset.

We can go one step further and add one more argument to the function to define whether the Dataset instance is for training or test/validation. We have about 160 photos, so we can use about 20%, or the last 32 photos, as a test or validation dataset and the first 131, or 80%, as the training dataset. This division can be made using the integer in the filename, where all photos before photo number 150 will be train and equal or after 150 used for test. The updated `load_dataset()` with support for train and test datasets is provided below.

```
# load the dataset definitions
def load_dataset(self, dataset_dir, is_train=True):
    # define one class
    self.add_class("dataset", 1, "kangaroo")
    # define data locations
```

```

images_dir = dataset_dir + '/images/'
annotations_dir = dataset_dir + '/annots/'
# find all images
for filename in listdir(images_dir):
    # extract image id
    image_id = filename[:-4]
    # skip bad images
    if image_id in ['00090']:
        continue
    # skip all images after 150 if we are building the train set
    if is_train and int(image_id) >= 150:
        continue
    # skip all images before 150 if we are building the test/val set
    if not is_train and int(image_id) < 150:
        continue
    img_path = images_dir + filename
    ann_path = annotations_dir + image_id + '.xml'
    # add to dataset
    self.add_image('dataset', image_id=image_id, path=img_path, annotation=ann_path)

```

Listing 26.25: Example of defining a function to load the dataset as either train or test.

Next, we need to define the `load_mask()` function for loading the mask for a given `image_id`. In this case, the `image_id` is the integer index for an image in the dataset, assigned based on the order that the image was added via a call to `add_image()` when loading the dataset. The function must return an array of one or more masks for the photo associated with the `image_id`, and the classes for each mask. We don't have masks, but we do have bounding boxes. We can load the bounding boxes for a given photo and return them as masks. The library will then infer bounding boxes from our `masks` which will be the same size.

First, we must load the annotation file for the `image_id`. This involves first retrieving the `image info` dict for the `image_id`, then retrieving the annotations path that we stored for the image via our prior call to `add_image()`. We can then use the path in our call to `extract_boxes()` developed in the previous section to get the list of bounding boxes and the dimensions of the image.

```

...
# get details of image
info = self.image_info[image_id]
# define box file location
path = info['annotation']
# load XML
boxes, w, h = self.extract_boxes(path)

```

Listing 26.26: Example of loading an annotation for an image.

We can now define a mask for each bounding box, and an associated class. A mask is a two-dimensional array with the same dimensions as the photograph with all zero values where the object isn't and all one values where the object is in the photograph. We can achieve this by creating a NumPy array with all zero values for the known size of the image and one channel for each bounding box.

```

...
# create one array for all masks, each on a different channel
masks = zeros([h, w, len(boxes)], dtype='uint8')

```

Listing 26.27: Example of creating an empty mask.

Each bounding box is defined as `min` and `max`, `x` and `y` coordinates of the box. These can be used directly to define row and column ranges in the array that can then be marked as 1.

```
...
# create masks
for i in range(len(boxes)):
    box = boxes[i]
    row_s, row_e = box[1], box[3]
    col_s, col_e = box[0], box[2]
    masks[row_s:row_e, col_s:col_e, i] = 1
```

Listing 26.28: Example of marking the bounding boxes in the mask.

All objects have the same class in this dataset. We can retrieve the class index via the `class_names` dictionary, then add it to a list to be returned alongside the masks.

```
...
# get the index for the class
self.class_names.index('kangaroo')
```

Listing 26.29: Example of getting the index for the class.

Tying this together, the complete `load_mask()` function is listed below.

```
# load the masks for an image
def load_mask(self, image_id):
    # get details of image
    info = self.image_info[image_id]
    # define box file location
    path = info['annotation']
    # load XML
    boxes, w, h = self.extract_boxes(path)
    # create one array for all masks, each on a different channel
    masks = zeros([h, w, len(boxes)], dtype='uint8')
    # create masks
    class_ids = list()
    for i in range(len(boxes)):
        box = boxes[i]
        row_s, row_e = box[1], box[3]
        col_s, col_e = box[0], box[2]
        masks[row_s:row_e, col_s:col_e, i] = 1
        class_ids.append(self.class_names.index('kangaroo'))
    return masks, asarray(class_ids, dtype='int32')
```

Listing 26.30: Example of defining a function for loading a mask for an image.

Finally, we must implement the `image_reference()` function. This function is responsible for returning the path or URL for a given `image_id`, which we know is just the `path` property on the `image_info` dict.

```
# load an image reference
def image_reference(self, image_id):
    info = self.image_info[image_id]
    return info['path']
```

Listing 26.31: Example of getting a reference for an image.

And that's it. We have successfully defined a Dataset object for the mask-rcnn library for our Kangaroo dataset. The complete listing of the class and creating a train and test dataset is provided below.

```
# split into train and test set
from os import listdir
from xml.etree import ElementTree
from numpy import zeros
from numpy import asarray
from mrcnn.utils import Dataset

# class that defines and loads the kangaroo dataset
class KangarooDataset(Dataset):
    # load the dataset definitions
    def load_dataset(self, dataset_dir, is_train=True):
        # define one class
        self.add_class("dataset", 1, "kangaroo")
        # define data locations
        images_dir = dataset_dir + '/images/'
        annotations_dir = dataset_dir + '/annots/'
        # find all images
        for filename in listdir(images_dir):
            # extract image id
            image_id = filename[:-4]
            # skip bad images
            if image_id in ['00090']:
                continue
            # skip all images after 150 if we are building the train set
            if is_train and int(image_id) >= 150:
                continue
            # skip all images before 150 if we are building the test/val set
            if not is_train and int(image_id) < 150:
                continue
            img_path = images_dir + filename
            ann_path = annotations_dir + image_id + '.xml'
            # add to dataset
            self.add_image('dataset', image_id=image_id, path=img_path, annotation=ann_path)

    # extract bounding boxes from an annotation file
    def extract_boxes(self, filename):
        # load and parse the file
        tree = ElementTree.parse(filename)
        # get the root of the document
        root = tree.getroot()
        # extract each bounding box
        boxes = list()
        for box in root.findall('.//bndbox'):
            xmin = int(box.find('xmin').text)
            ymin = int(box.find('ymin').text)
            xmax = int(box.find('xmax').text)
            ymax = int(box.find('ymax').text)
            coors = [xmin, ymin, xmax, ymax]
            boxes.append(coors)
        # extract image dimensions
        width = int(root.find('.//size/width').text)
        height = int(root.find('.//size/height').text)
```

```

    return boxes, width, height

# load the masks for an image
def load_mask(self, image_id):
    # get details of image
    info = self.image_info[image_id]
    # define box file location
    path = info['annotation']
    # load XML
    boxes, w, h = self.extract_boxes(path)
    # create one array for all masks, each on a different channel
    masks = zeros([h, w, len(boxes)], dtype='uint8')
    # create masks
    class_ids = list()
    for i in range(len(boxes)):
        box = boxes[i]
        row_s, row_e = box[1], box[3]
        col_s, col_e = box[0], box[2]
        masks[row_s:row_e, col_s:col_e, i] = 1
        class_ids.append(self.class_names.index('kangaroo'))
    return masks, asarray(class_ids, dtype='int32')

# load an image reference
def image_reference(self, image_id):
    info = self.image_info[image_id]
    return info['path']

# train set
train_set = KangarooDataset()
train_set.load_dataset('kangaroo', is_train=True)
train_set.prepare()
print('Train: %d' % len(train_set.image_ids))

# test/val set
test_set = KangarooDataset()
test_set.load_dataset('kangaroo', is_train=False)
test_set.prepare()
print('Test: %d' % len(test_set.image_ids))

```

Listing 26.32: Example of defining and testing a dataset object for the kangaroo dataset.

Running the example successfully loads and prepares the train and test dataset and prints the number of images in each.

```

Train: 131
Test: 32

```

Listing 26.33: Example output from defining and testing a dataset object for the kangaroo dataset.

Now that we have defined the dataset, let's confirm that the images, masks, and bounding boxes are handled correctly.

26.3.4 Test the KangarooDataset Object

The first useful test is to confirm that the images and masks can be loaded correctly. We can test this by creating a dataset and loading an image via a call to the `load_image()` function with an `image_id`, then load the mask for the image via a call to the `load_mask()` function with the same `image_id`.

```
# load an image
image_id = 0
image = train_set.load_image(image_id)
print(image.shape)
# load image mask
mask, class_ids = train_set.load_mask(image_id)
print(mask.shape)
```

Listing 26.34: Example of loading an image and a mask with the kangaroo dataset.

Next, we can plot the photograph using the Matplotlib API, then plot the first mask over the top with an alpha value so that the photograph underneath can still be seen.

```
# plot image
pyplot.imshow(image)
# plot mask
pyplot.imshow(mask[:, :, 0], cmap='gray', alpha=0.5)
pyplot.show()
```

Listing 26.35: Example of plotting a photograph and a mask.

The complete example is listed below.

```
# plot one photograph and mask
from os import listdir
from xml.etree import ElementTree
from numpy import zeros
from numpy import asarray
from mrcnn.utils import Dataset
from matplotlib import pyplot

# class that defines and loads the kangaroo dataset
class KangarooDataset(Dataset):
    # load the dataset definitions
    def load_dataset(self, dataset_dir, is_train=True):
        # define one class
        self.add_class("dataset", 1, "kangaroo")
        # define data locations
        images_dir = dataset_dir + '/images/'
        annotations_dir = dataset_dir + '/annots/'
        # find all images
        for filename in listdir(images_dir):
            # extract image id
            image_id = filename[:-4]
            # skip bad images
            if image_id in ['00090']:
                continue
            # skip all images after 150 if we are building the train set
            if is_train and int(image_id) >= 150:
                continue
```

```
# skip all images before 150 if we are building the test/val set
if not is_train and int(image_id) < 150:
    continue
img_path = images_dir + filename
ann_path = annotations_dir + image_id + '.xml'
# add to dataset
self.add_image('dataset', image_id=image_id, path=img_path, annotation=ann_path)

# extract bounding boxes from an annotation file
def extract_boxes(self, filename):
    # load and parse the file
    tree = ElementTree.parse(filename)
    # get the root of the document
    root = tree.getroot()
    # extract each bounding box
    boxes = list()
    for box in root.findall('.//bndbox'):
        xmin = int(box.find('xmin').text)
        ymin = int(box.find('ymin').text)
        xmax = int(box.find('xmax').text)
        ymax = int(box.find('ymax').text)
        coors = [xmin, ymin, xmax, ymax]
        boxes.append(coors)
    # extract image dimensions
    width = int(root.find('.//size/width').text)
    height = int(root.find('.//size/height').text)
    return boxes, width, height

# load the masks for an image
def load_mask(self, image_id):
    # get details of image
    info = self.image_info[image_id]
    # define box file location
    path = info['annotation']
    # load XML
    boxes, w, h = self.extract_boxes(path)
    # create one array for all masks, each on a different channel
    masks = zeros([h, w, len(boxes)], dtype='uint8')
    # create masks
    class_ids = list()
    for i in range(len(boxes)):
        box = boxes[i]
        row_s, row_e = box[1], box[3]
        col_s, col_e = box[0], box[2]
        masks[row_s:row_e, col_s:col_e, i] = 1
        class_ids.append(self.class_names.index('kangaroo'))
    return masks, asarray(class_ids, dtype='int32')

# load an image reference
def image_reference(self, image_id):
    info = self.image_info[image_id]
    return info['path']

# train set
train_set = KangarooDataset()
train_set.load_dataset('kangaroo', is_train=True)
```

```
train_set.prepare()
# load an image
image_id = 0
image = train_set.load_image(image_id)
print(image.shape)
# load image mask
mask, class_ids = train_set.load_mask(image_id)
print(mask.shape)
# plot image
pyplot.imshow(image)
# plot mask
pyplot.imshow(mask[:, :, 0], cmap='gray', alpha=0.5)
pyplot.show()
```

Listing 26.36: Example of plotting a photograph and the associated mask.

Running the example first prints the shape of the photograph and mask NumPy arrays. We can confirm that both arrays have the same width and height and only differ in terms of the number of channels. We can also see that the first photograph (e.g. `image_id=0`) in this case only has one mask. **Note**, a different photo may be selected, depending on the order in which photographs were loaded from file.

```
(626, 899, 3)
(626, 899, 1)
```

Listing 26.37: Example output from loading a single photo and mask.

A plot of the photograph is also created with the first mask overlaid. In this case, we can see that one kangaroo is present in the photo and that the mask correctly bounds the kangaroo.



Figure 26.1: Photograph of Kangaroo With Object Detection Mask Overlaid.

We could repeat this for the first nine photos in the dataset, plotting each photo in one figure as a subplot and plotting all masks for each photo.

```
...
# plot first few images
for i in range(9):
    # define subplot
    pyplot.subplot(330 + 1 + i)
    # turn off axis labels
    pyplot.axis('off')
    # plot raw pixel data
    image = train_set.load_image(i)
    pyplot.imshow(image)
    # plot all masks
    mask, _ = train_set.load_mask(i)
    for j in range(mask.shape[2]):
        pyplot.imshow(mask[:, :, j], cmap='gray', alpha=0.3)
    # show the figure
    pyplot.show()
```

Listing 26.38: Example of plotting the first few photos and masks in the dataset.

Running the example shows that photos are loaded correctly and that those photos with multiple objects correctly have separate masks defined. **Note**, a different collection of photos

may be selected, depending on the order in which photographs were loaded from file.



Figure 26.2: Plot of First Nine Photos of Kangaroos in the Training Dataset With Object Detection Masks.

Another useful debugging step might be to load all of the *image info* objects in the dataset and print them to the console. This can help to confirm that all of the calls to the `add_image()` function in the `load_dataset()` function worked as expected.

```
...
# enumerate all images in the dataset
for image_id in train_set.image_ids:
    # load image info
    info = train_set.image_info[image_id]
    # display on the console
    print(info)
```

Listing 26.39: Example of displaying information of the photos in the dataset.

Running this code on the loaded training dataset will then show all of the *image info* dictionaries, showing the paths and ids for each image in the dataset.

```
{"id": "00132", "source": "dataset", "path": "kangaroo/images/00132.jpg", "annotation": "kangaroo/annots/00132.xml"}
{"id": "00046", "source": "dataset", "path": "kangaroo/images/00046.jpg", "annotation": "kangaroo/annots/00046.xml"}
{"id": "00052", "source": "dataset", "path": "kangaroo/images/00052.jpg", "annotation": "kangaroo/annots/00052.xml"}
...
```

Listing 26.40: Example output from summarizing the details of the dataset.

Finally, the mask-rcnn library provides utilities for displaying images and masks. We can use some of these built-in functions to confirm that the Dataset is operating correctly. For example, the mask-rcnn library provides the `mrcnn.visualize.display_instances()` function that will show a photograph with bounding boxes, masks, and class labels. This requires that the bounding boxes are extracted from the masks via the `extract_bboxes()` function.

```
...
# define image id
image_id = 1
# load the image
image = train_set.load_image(image_id)
# load the masks and the class ids
mask, class_ids = train_set.load_mask(image_id)
# extract bounding boxes from the masks
bbox = extract_bboxes(mask)
# display image with masks and bounding boxes
display_instances(image, bbox, mask, class_ids, train_set.class_names)
```

Listing 26.41: Example of displaying a photo and mask using the built-in function.

For completeness, the full code listing is provided below.

```
# display image with masks and bounding boxes
from os import listdir
from xml.etree import ElementTree
from numpy import zeros
from numpy import asarray
from mrcnn.utils import Dataset
from mrcnn.visualize import display_instances
from mrcnn.utils import extract_bboxes

# class that defines and loads the kangaroo dataset
class KangarooDataset(Dataset):
    # load the dataset definitions
    def load_dataset(self, dataset_dir, is_train=True):
        # define one class
        self.add_class("dataset", 1, "kangaroo")
        # define data locations
        images_dir = dataset_dir + '/images/'
        annotations_dir = dataset_dir + '/annots/'
        # find all images
        for filename in listdir(images_dir):
            # extract image id
            image_id = filename[:-4]
            # skip bad images
            if image_id in ['00090']:
                continue
            # skip all images after 150 if we are building the train set
            if is_train and int(image_id) >= 150:
                continue
            # skip all images before 150 if we are building the test/val set
            if not is_train and int(image_id) < 150:
                continue
            img_path = images_dir + filename
            ann_path = annotations_dir + image_id + '.xml'
            # add to dataset
```

```
    self.add_image('dataset', image_id=image_id, path=img_path, annotation=ann_path)

# extract bounding boxes from an annotation file
def extract_boxes(self, filename):
    # load and parse the file
    tree = ElementTree.parse(filename)
    # get the root of the document
    root = tree.getroot()
    # extract each bounding box
    boxes = list()
    for box in root.findall('.//bndbox'):
        xmin = int(box.find('xmin').text)
        ymin = int(box.find('ymin').text)
        xmax = int(box.find('xmax').text)
        ymax = int(box.find('ymax').text)
        coors = [xmin, ymin, xmax, ymax]
        boxes.append(coors)
    # extract image dimensions
    width = int(root.find('.//size/width').text)
    height = int(root.find('.//size/height').text)
    return boxes, width, height

# load the masks for an image
def load_mask(self, image_id):
    # get details of image
    info = self.image_info[image_id]
    # define box file location
    path = info['annotation']
    # load XML
    boxes, w, h = self.extract_boxes(path)
    # create one array for all masks, each on a different channel
    masks = zeros([h, w, len(boxes)], dtype='uint8')
    # create masks
    class_ids = list()
    for i in range(len(boxes)):
        box = boxes[i]
        row_s, row_e = box[1], box[3]
        col_s, col_e = box[0], box[2]
        masks[row_s:row_e, col_s:col_e, i] = 1
        class_ids.append(self.class_names.index('kangaroo'))
    return masks, asarray(class_ids, dtype='int32')

# load an image reference
def image_reference(self, image_id):
    info = self.image_info[image_id]
    return info['path']

# train set
train_set = KangarooDataset()
train_set.load_dataset('kangaroo', is_train=True)
train_set.prepare()
# define image id
image_id = 1
# load the image
image = train_set.load_image(image_id)
# load the masks and the class ids
```

```

mask, class_ids = train_set.load_mask(image_id)
# extract bounding boxes from the masks
bbox = extract_bboxes(mask)
# display image with masks and bounding boxes
display_instances(image, bbox, mask, class_ids, train_set.class_names)

```

Listing 26.42: Example of plotting a photograph and the associated mask using the built-in function.

Running the example creates a plot showing the photograph with the mask for each object in a separate color. The bounding boxes match the masks exactly, by design, and are shown with dotted outlines. Finally, each object is marked with the class label, which in this case is kangaroo.



Figure 26.3: Photograph Showing Object Detection Masks, Bounding Boxes, and Class Labels.

Now that we are confident that our dataset is being loaded correctly, we can use it to fit a Mask R-CNN model.

26.4 How to Train Mask R-CNN Model for Kangaroo Detection

A Mask R-CNN model can be fit from scratch, although like other computer vision applications, time can be saved and performance can be improved by using transfer learning. The Mask R-CNN model pre-fit on the MSCOCO object detection dataset can be used as a starting point and then tailored to the specific dataset, in this case, the kangaroo dataset. The first step is to download the model file (architecture and weights) for the pre-fit Mask R-CNN model. The weights are available from the GitHub project and the file is about 250 megabytes. Download the model weights to a file with the name `mask_rcnn_coco.h5` in your current working directory.

- Download Model Weights (`mask_rcnn_coco.h5`) (246 megabytes).²

Next, a configuration object for the model must be defined. This is a new class that extends the `mrcnn.config.Config` class and defines properties of both the prediction problem (such as name and the number of classes) and the algorithm for training the model (such as the learning rate). The configuration must define the name of the configuration via the `NAME` attribute, e.g. `kangaroo_cfg`, that will be used to save details and models to file during the run. The configuration must also define the number of classes in the prediction problem via the `NUM_CLASSES` attribute. In this case, we only have one object type (kangaroo), although there is always an additional class for the background. Finally, we must define the number of samples (photos) used in each training epoch. This will be the number of photos in the training dataset, in this case, 131. Tying this together, our custom `KangarooConfig` class is defined below.

```
# define a configuration for the model
class KangarooConfig(Config):
    # Give the configuration a recognizable name
    NAME = "kangaroo_cfg"
    # Number of classes (background + kangaroo)
    NUM_CLASSES = 1 + 1
    # Number of training steps per epoch
    STEPS_PER_EPOCH = 131

# prepare config
config = KangarooConfig()
```

Listing 26.43: Example of defining a model configuration for training a Mask RCNN model.

Next, we can define our model. This is achieved by creating an instance of the `mrcnn.model.MaskRCNN` class and specifying that the model will be used for training via setting the `mode` argument to ‘`training`’. The `config` argument must also be specified with an instance of our `KangarooConfig` class. Finally, a directory is needed where configuration files can be saved and where checkpoint models can be saved at the end of each epoch. We will use the current working directory.

```
...
# define the model
model = MaskRCNN(mode='training', model_dir='./', config=config)
```

Listing 26.44: Example of defining a Mask RCNN model.

Next, the pre-defined model architecture and weights can be loaded. This can be achieved by calling the `load_weights()` function on the model and specifying the path to the downloaded `mask_rcnn_coco.h5` file. The model will be used as-is, although the class-specific output layers will be removed so that new output layers can be defined and trained. This can be done by specifying the `exclude` argument and listing all of the output layers to exclude or remove from the model after it is loaded. This includes the output layers for the classification label, bounding boxes, and masks.

```
...
# load weights (mscoco)
model.load_weights('mask_rcnn_coco.h5', by_name=True, exclude=["mrcnn_class_logits",
    "mrcnn_bbox_fc", "mrcnn_bbox", "mrcnn_mask"])
```

²<https://goo.gl/a2p7dS>

Listing 26.45: Example of loading pre-trained weights for the Mask RCNN model.

Next, the model can be fit on the training dataset by calling the `train()` function and passing in both the training dataset and the validation dataset. We can also specify the learning rate as the default learning rate in the configuration (0.001). We can also specify what layers to train. In this case, we will only train the heads, that is the output layers of the model.

```
...
# train weights (output layers or 'heads')
model.train(train_set, test_set, learning_rate=config.LEARNING_RATE, epochs=5,
            layers='heads')
```

Listing 26.46: Example of training the Mask RCNN model on the Kangaroo dataset.

We could follow this training with further epochs that fine-tune all of the weights in the model. This could be achieved by using a smaller learning rate and changing the `layer` argument from ‘heads’ to ‘all’. The complete example of training a Mask R-CNN on the kangaroo dataset is listed below. **Note**, this may take some time to execute on the CPU, even with modern hardware. I recommend running the code with a GPU, such as on Amazon EC2, where it will finish in about five minutes on a P3 type hardware (see Appendix C).

```
# fit a mask rcnn on the kangaroo dataset
from os import listdir
from xml.etree import ElementTree
from numpy import zeros
from numpy import asarray
from mrcnn.utils import Dataset
from mrcnn.config import Config
from mrcnn.model import MaskRCNN

# class that defines and loads the kangaroo dataset
class KangarooDataset(Dataset):
    # load the dataset definitions
    def load_dataset(self, dataset_dir, is_train=True):
        # define one class
        self.add_class("dataset", 1, "kangaroo")
        # define data locations
        images_dir = dataset_dir + '/images/'
        annotations_dir = dataset_dir + '/annots/'
        # find all images
        for filename in listdir(images_dir):
            # extract image id
            image_id = filename[:-4]
            # skip bad images
            if image_id in ['00090']:
                continue
            # skip all images after 150 if we are building the train set
            if is_train and int(image_id) >= 150:
                continue
            # skip all images before 150 if we are building the test/val set
            if not is_train and int(image_id) < 150:
                continue
            img_path = images_dir + filename
            ann_path = annotations_dir + image_id + '.xml'
```

```
# add to dataset
self.add_image('dataset', image_id=image_id, path=img_path, annotation=ann_path)

# extract bounding boxes from an annotation file
def extract_boxes(self, filename):
    # load and parse the file
    tree = ElementTree.parse(filename)
    # get the root of the document
    root = tree.getroot()
    # extract each bounding box
    boxes = list()
    for box in root.findall('.//bndbox'):
        xmin = int(box.find('xmin').text)
        ymin = int(box.find('ymin').text)
        xmax = int(box.find('xmax').text)
        ymax = int(box.find('ymax').text)
        coors = [xmin, ymin, xmax, ymax]
        boxes.append(coors)
    # extract image dimensions
    width = int(root.find('.//size/width').text)
    height = int(root.find('.//size/height').text)
    return boxes, width, height

# load the masks for an image
def load_mask(self, image_id):
    # get details of image
    info = self.image_info[image_id]
    # define box file location
    path = info['annotation']
    # load XML
    boxes, w, h = self.extract_boxes(path)
    # create one array for all masks, each on a different channel
    masks = zeros([h, w, len(boxes)], dtype='uint8')
    # create masks
    class_ids = list()
    for i in range(len(boxes)):
        box = boxes[i]
        row_s, row_e = box[1], box[3]
        col_s, col_e = box[0], box[2]
        masks[row_s:row_e, col_s:col_e, i] = 1
        class_ids.append(self.class_names.index('kangaroo'))
    return masks, asarray(class_ids, dtype='int32')

# load an image reference
def image_reference(self, image_id):
    info = self.image_info[image_id]
    return info['path']

# define a configuration for the model
class KangarooConfig(Config):
    # define the name of the configuration
    NAME = "kangaroo_cfg"
    # number of classes (background + kangaroo)
    NUM_CLASSES = 1 + 1
    # number of training steps per epoch
    STEPS_PER_EPOCH = 131
```

```

# prepare train set
train_set = KangarooDataset()
train_set.load_dataset('kangaroo', is_train=True)
train_set.prepare()
print('Train: %d' % len(train_set.image_ids))
# prepare test/val set
test_set = KangarooDataset()
test_set.load_dataset('kangaroo', is_train=False)
test_set.prepare()
print('Test: %d' % len(test_set.image_ids))
# prepare config
config = KangarooConfig()
config.display()
# define the model
model = MaskRCNN(mode='training', model_dir='./', config=config)
# load weights (mscoco) and exclude the output layers
model.load_weights('mask_rcnn_coco.h5', by_name=True, exclude=["mrcnn_class_logits",
    "mrcnn_bbox_fc", "mrcnn_bbox", "mrcnn_mask"])
# train weights (output layers or 'heads')
model.train(train_set, test_set, learning_rate=config.LEARNING_RATE, epochs=5,
    layers='heads')

```

Listing 26.47: Example of training an Mask RCNN model on the Kangaroo dataset.

Running the example will report progress using the standard Keras progress bars. We can see that there are many different train and test loss scores reported for each of the output heads of the network. It can be quite confusing as to which loss to pay attention to. In this example where we are interested in object detection instead of object segmentation, I recommend paying attention to the loss for the classification output on the train and validation datasets (e.g. `mrcnn_class_loss` and `val_mrcnn_class_loss`), as well as the loss for the bounding box output for the train and validation datasets (`mrcnn_bbox_loss` and `val_mrcnn_bbox_loss`).

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```

Epoch 1/5
131/131 [=====] - 106s 811ms/step - loss: 0.8491 - rpn_class_loss:
0.0044 - rpn_bbox_loss: 0.1452 - mrcnn_class_loss: 0.0420 - mrcnn_bbox_loss: 0.2874 -
mrcnn_mask_loss: 0.3701 - val_loss: 1.3402 - val_rpn_class_loss: 0.0160 -
val_rpn_bbox_loss: 0.7913 - val_mrcnn_class_loss: 0.0092 - val_mrcnn_bbox_loss: 0.2263
- val_mrcnn_mask_loss: 0.2975
Epoch 2/5
131/131 [=====] - 69s 526ms/step - loss: 0.4774 - rpn_class_loss:
0.0025 - rpn_bbox_loss: 0.1159 - mrcnn_class_loss: 0.0170 - mrcnn_bbox_loss: 0.1134 -
mrcnn_mask_loss: 0.2285 - val_loss: 0.6261 - val_rpn_class_loss: 8.9502e-04 -
val_rpn_bbox_loss: 0.1624 - val_mrcnn_class_loss: 0.0197 - val_mrcnn_bbox_loss: 0.2148
- val_mrcnn_mask_loss: 0.2282
Epoch 3/5
131/131 [=====] - 67s 515ms/step - loss: 0.4471 - rpn_class_loss:
0.0029 - rpn_bbox_loss: 0.1153 - mrcnn_class_loss: 0.0234 - mrcnn_bbox_loss: 0.0958 -
mrcnn_mask_loss: 0.2097 - val_loss: 1.2998 - val_rpn_class_loss: 0.0144 -
val_rpn_bbox_loss: 0.6712 - val_mrcnn_class_loss: 0.0372 - val_mrcnn_bbox_loss: 0.2645
- val_mrcnn_mask_loss: 0.3125

```

```

Epoch 4/5
131/131 [=====] - 66s 502ms/step - loss: 0.3934 - rpn_class_loss: 0.0026 - rpn_bbox_loss: 0.1003 - mrcnn_class_loss: 0.0171 - mrcnn_bbox_loss: 0.0806 - mrcnn_mask_loss: 0.1928 - val_loss: 0.6709 - val_rpn_class_loss: 0.0016 - val_rpn_bbox_loss: 0.2012 - val_mrcnn_class_loss: 0.0244 - val_mrcnn_bbox_loss: 0.1942 - val_mrcnn_mask_loss: 0.2495
Epoch 5/5
131/131 [=====] - 65s 493ms/step - loss: 0.3357 - rpn_class_loss: 0.0024 - rpn_bbox_loss: 0.0804 - mrcnn_class_loss: 0.0193 - mrcnn_bbox_loss: 0.0616 - mrcnn_mask_loss: 0.1721 - val_loss: 0.8878 - val_rpn_class_loss: 0.0030 - val_rpn_bbox_loss: 0.4409 - val_mrcnn_class_loss: 0.0174 - val_mrcnn_bbox_loss: 0.1752 - val_mrcnn_mask_loss: 0.2513

```

Listing 26.48: Example output from training an Mask RCNN model on the Kangaroo dataset.

A model file is created and saved at the end of each epoch in a subdirectory that starts with `kangaroo.cfg` followed by random characters. A model must be selected for use; in this case, the loss continues to decrease for the bounding boxes on each epoch, so we will use the final model at the end of the run (`mask_rcnn_kangaroo_cfg_0005.h5`). Copy the model file from the config directory into your current working directory. We will use it in the following sections to evaluate the model and make predictions. The results suggest that perhaps more training epochs could be useful, perhaps fine-tuning all of the layers in the model; this might make an interesting extension to the tutorial. Next, let's look at evaluating the performance of this model.

26.5 How to Evaluate a Mask R-CNN Model

The performance of a model for an object recognition task is often evaluated using the mean absolute precision, or mAP. We are predicting bounding boxes so we can determine whether a bounding box prediction is good or not based on how well the predicted and actual bounding boxes overlap. This can be calculated by dividing the area of the overlap by the total area of both bounding boxes, or the intersection divided by the union, referred to as *intersection over union*, or IoU. A perfect bounding box prediction will have an IoU of 1. It is standard to assume a positive prediction of a bounding box if the IoU is greater than 0.5, e.g. they overlap by 50% or more. Precision refers to the percentage of the correctly predicted bounding boxes ($\text{IoU} > 0.5$) out of all bounding boxes predicted. Recall is the percentage of the correctly predicted bounding boxes ($\text{IoU} > 0.5$) out of all objects in the photo.

As we make more predictions, the recall percentage will increase, but precision will drop or become erratic as we start making false positive predictions. The recall (x) can be plotted against the precision (y) for each number of predictions to create a curve or line. We can maximize the value of each point on this line and calculate the average value of the precision or AP for each value of recall. Note: there are variations on how AP is calculated, e.g. the way it is calculated for the widely used PASCAL VOC dataset and the MSCOCO dataset differ.

The average or mean of the average precision (AP) across all of the images in a dataset is called the mean average precision, or mAP. The mask-rcnn library provides a `mrcnn.utils.compute_ap` to calculate the AP and other metrics for a given images. These AP scores can be collected across a dataset and the mean calculated to give an idea at how good the model is at detecting objects in a dataset. First, we must define a new `Config` object to use for making predictions, instead of training. We can extend our previously defined `KangarooConfig` to reuse the parameters.

Instead, we will define a new object with the same values to keep the code compact. The config must change some of the defaults around using the GPU for inference that are different from how they are set for training a model (regardless of whether you are running on the GPU or CPU).

```
# define the prediction configuration
class PredictionConfig(Config):
    # define the name of the configuration
    NAME = "kangaroo_cfg"
    # number of classes (background + kangaroo)
    NUM_CLASSES = 1 + 1
    # simplify GPU config
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1
```

Listing 26.49: Example of defining a new model configuration for evaluating the Mask RCNN model.

Next, we can define the model with the config and set the `mode` argument to ‘`inference`’ instead of ‘`training`’.

```
...
# create config
cfg = PredictionConfig()
# define the model
model = MaskRCNN(mode='inference', model_dir='./', config=cfg)
```

Listing 26.50: Example of defining a new model for making predictions.

Next, we can load the weights from our saved model. We can do that by specifying the path to the model file. In this case, the model file is `mask_rcnn_kangaroo_cfg_0005.h5` in the current working directory.

```
...
# load model weights
model.load_weights('mask_rcnn_kangaroo_cfg_0005.h5', by_name=True)
```

Listing 26.51: Example of loading pre-trained model weights.

Next, we can evaluate the model. This involves enumerating the images in a dataset, making a prediction, and calculating the AP for the prediction before predicting a mean AP across all images. First, the image and ground truth mask can be loaded from the dataset for a given `image_id`. This can be achieved using the `load_image_gt()` convenience function.

```
...
# load image, bounding boxes and masks for the image id
image, image_meta, gt_class_id, gt_bbox, gt_mask = load_image_gt(dataset, cfg, image_id,
    use_mini_mask=False)
```

Listing 26.52: Example of loading ground truth examples for an image.

Next, the pixel values of the loaded image must be scaled in the same way as was performed on the training data, e.g. centered. This can be achieved using the `mold_image()` convenience function.

```
...
# convert pixel values (e.g. center)
```

```
scaled_image = mold_image(image, cfg)
```

Listing 26.53: Example of preparing the image for making a prediction.

The dimensions of the image then need to be expanded one sample in a dataset and used as input to make a prediction with the model.

```
...
sample = expand_dims(scaled_image, 0)
# make prediction
yhat = model.detect(sample, verbose=0)
# extract results for first sample
r = yhat[0]
```

Listing 26.54: Example of using the model to detect objects in the prepared image.

Next, the prediction can be compared to the ground truth and metrics calculated using the `compute_ap()` function.

```
...
# calculate statistics, including AP
AP, _, _, _ = compute_ap(gt_bbox, gt_class_id, gt_mask, r["rois"], r["class_ids"],
r["scores"], r['masks'])
```

Listing 26.55: Example of evaluating the detected objects in a single prediction against the ground truth.

The AP values can be added to a list, then the mean value calculated. Tying this together, the `evaluate_model()` function below implements this and calculates the mAP given a dataset, model and configuration.

```
# calculate the mAP for a model on a given dataset
def evaluate_model(dataset, model, cfg):
    APs = list()
    for image_id in dataset.image_ids:
        # load image, bounding boxes and masks for the image id
        image, _, gt_class_id, gt_bbox, gt_mask = load_image_gt(dataset, cfg, image_id,
        use_mini_mask=False)
        # convert pixel values (e.g. center)
        scaled_image = mold_image(image, cfg)
        # convert image into one sample
        sample = expand_dims(scaled_image, 0)
        # make prediction
        yhat = model.detect(sample, verbose=0)
        # extract results for first sample
        r = yhat[0]
        # calculate statistics, including AP
        AP, _, _, _ = compute_ap(gt_bbox, gt_class_id, gt_mask, r["rois"], r["class_ids"],
        r["scores"], r['masks'])
        # store
        APs.append(AP)
    # calculate the mean AP across all images
    mAP = mean(APs)
    return mAP
```

Listing 26.56: Example of a function for evaluating the model on a dataset.

We can now calculate the mAP for the model on the train and test datasets.

```

...
# evaluate model on training dataset
train_mAP = evaluate_model(train_set, model, cfg)
print("Train mAP: %.3f" % train_mAP)
# evaluate model on test dataset
test_mAP = evaluate_model(test_set, model, cfg)
print("Test mAP: %.3f" % test_mAP)

```

Listing 26.57: Example of evaluating the model on the train and test datasets.

The full code listing is provided below for completeness.

```

# evaluate the mask rcnn model on the kangaroo dataset
from os import listdir
from xml.etree import ElementTree
from numpy import zeros
from numpy import asarray
from numpy import expand_dims
from numpy import mean
from mrcnn.config import Config
from mrcnn.model import MaskRCNN
from mrcnn.utils import Dataset
from mrcnn.utils import compute_ap
from mrcnn.model import load_image_gt
from mrcnn.model import mold_image

# class that defines and loads the kangaroo dataset
class KangarooDataset(Dataset):
    # load the dataset definitions
    def load_dataset(self, dataset_dir, is_train=True):
        # define one class
        self.add_class("dataset", 1, "kangaroo")
        # define data locations
        images_dir = dataset_dir + '/images/'
        annotations_dir = dataset_dir + '/annots/'
        # find all images
        for filename in listdir(images_dir):
            # extract image id
            image_id = filename[:-4]
            # skip bad images
            if image_id in ['00090']:
                continue
            # skip all images after 150 if we are building the train set
            if is_train and int(image_id) >= 150:
                continue
            # skip all images before 150 if we are building the test/val set
            if not is_train and int(image_id) < 150:
                continue
            img_path = images_dir + filename
            ann_path = annotations_dir + image_id + '.xml'
            # add to dataset
            self.add_image('dataset', image_id=image_id, path=img_path, annotation=ann_path)

        # extract bounding boxes from an annotation file
        def extract_boxes(self, filename):
            # load and parse the file

```

```
tree = ElementTree.parse(filename)
# get the root of the document
root = tree.getroot()
# extract each bounding box
boxes = list()
for box in root.findall('.//bndbox'):
    xmin = int(box.find('xmin').text)
    ymin = int(box.find('ymin').text)
    xmax = int(box.find('xmax').text)
    ymax = int(box.find('ymax').text)
    coors = [xmin, ymin, xmax, ymax]
    boxes.append(coors)
# extract image dimensions
width = int(root.find('.//size/width').text)
height = int(root.find('.//size/height').text)
return boxes, width, height

# load the masks for an image
def load_mask(self, image_id):
    # get details of image
    info = self.image_info[image_id]
    # define box file location
    path = info['annotation']
    # load XML
    boxes, w, h = self.extract_boxes(path)
    # create one array for all masks, each on a different channel
    masks = zeros([h, w, len(boxes)], dtype='uint8')
    # create masks
    class_ids = list()
    for i in range(len(boxes)):
        box = boxes[i]
        row_s, row_e = box[1], box[3]
        col_s, col_e = box[0], box[2]
        masks[row_s:row_e, col_s:col_e, i] = 1
        class_ids.append(self.class_names.index('kangaroo'))
    return masks, asarray(class_ids, dtype='int32')

# load an image reference
def image_reference(self, image_id):
    info = self.image_info[image_id]
    return info['path']

# define the prediction configuration
class PredictionConfig(Config):
    # define the name of the configuration
    NAME = "kangaroo_cfg"
    # number of classes (background + kangaroo)
    NUM_CLASSES = 1 + 1
    # simplify GPU config
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1

    # calculate the mAP for a model on a given dataset
def evaluate_model(dataset, model, cfg):
    APs = list()
    for image_id in dataset.image_ids:
```

```

# load image, bounding boxes and masks for the image id
image, _, gt_class_id, gt_bbox, gt_mask = load_image_gt(dataset, cfg, image_id,
    use_mini_mask=False)
# convert pixel values (e.g. center)
scaled_image = mold_image(image, cfg)
# convert image into one sample
sample = expand_dims(scaled_image, 0)
# make prediction
yhat = model.detect(sample, verbose=0)
# extract results for first sample
r = yhat[0]
# calculate statistics, including AP
AP, _, _, _ = compute_ap(gt_bbox, gt_class_id, gt_mask, r["rois"], r["class_ids"],
    r["scores"], r['masks'])
# store
APs.append(AP)
# calculate the mean AP across all images
mAP = mean(APs)
return mAP

# load the train dataset
train_set = KangarooDataset()
train_set.load_dataset('kangaroo', is_train=True)
train_set.prepare()
print('Train: %d' % len(train_set.image_ids))
# load the test dataset
test_set = KangarooDataset()
test_set.load_dataset('kangaroo', is_train=False)
test_set.prepare()
print('Test: %d' % len(test_set.image_ids))
# create config
cfg = PredictionConfig()
# define the model
model = MaskRCNN(mode='inference', model_dir='./', config=cfg)
# load model weights
model.load_weights('mask_rcnn_kangaroo_cfg_0005.h5', by_name=True)
# evaluate model on training dataset
train_mAP = evaluate_model(train_set, model, cfg)
print("Train mAP: %.3f" % train_mAP)
# evaluate model on test dataset
test_mAP = evaluate_model(test_set, model, cfg)
print("Test mAP: %.3f" % test_mAP)

```

Listing 26.58: Example of evaluating the Mask RCNN model on the Kangaroo dataset.

Running the example will make a prediction for each image in the train and test datasets and calculate the mAP for each.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

A mAP above 90% or 95% is a good score. We can see that the mAP score is good on both datasets, and perhaps slightly better on the test dataset, instead of the train dataset. This may be because the dataset is very small, and/or because the model could benefit from further training.

```
Train mAP: 0.929
Test mAP: 0.958
```

Listing 26.59: Example output from evaluating the Mask RCNN model on the Kangaroo dataset.

Now that we have some confidence that the model is sensible, we can use it to make some predictions.

26.6 How to Detect Kangaroos in New Photos

We can use the trained model to detect kangaroos in new photographs, specifically, in photos that we expect to have kangaroos. First, we need a new photo of a kangaroo. We could go to Flickr and find a random photo of a kangaroo. Alternately, we can use any of the photos in the test dataset that were not used to train the model. We have already seen in the previous section how to make a prediction with an image. Specifically, scaling the pixel values and calling `model.detect()`. For example:

```
...
# example of making a prediction
...
# load image
image = ...
# convert pixel values (e.g. center)
scaled_image = mold_image(image, cfg)
# convert image into one sample
sample = expand_dims(scaled_image, 0)
# make prediction
yhat = model.detect(sample, verbose=0)
...
```

Listing 26.60: Example of a template for making a prediction on a new photograph.

Let's take it one step further and make predictions for a number of images in a dataset, then plot the photo with bounding boxes side-by-side with the photo and the predicted bounding boxes. This will provide a visual guide to how good the model is at making predictions. The first step is to load the image and mask from the dataset.

```
...
# load the image and mask
image = dataset.load_image(image_id)
mask, _ = dataset.load_mask(image_id)
```

Listing 26.61: Example of a loading an image and mask from the dataset.

Next, we can make a prediction for the image.

```
...
# convert pixel values (e.g. center)
scaled_image = mold_image(image, cfg)
# convert image into one sample
sample = expand_dims(scaled_image, 0)
# make prediction
yhat = model.detect(sample, verbose=0)[0]
```

Listing 26.62: Example of making a prediction for a single photograph.

Next, we can create a subplot for the ground truth and plot the image with the known bounding boxes.

```
...
# define subplot
pyplot.subplot(n_images, 2, i*2+1)
# turn off axis labels
pyplot.axis('off')
# plot raw pixel data
pyplot.imshow(image)
if i == 0:
    pyplot.title('Actual')
# plot masks
for j in range(mask.shape[2]):
    pyplot.imshow(mask[:, :, j], cmap='gray', alpha=0.3)
```

Listing 26.63: Example of plotting the ground truth masks on the photograph.

We can then create a second subplot beside the first and plot the first, plot the photo again, and this time draw the predicted bounding boxes in red.

```
...
# get the context for drawing boxes
pyplot.subplot(n_images, 2, i*2+2)
# turn off axis labels
pyplot.axis('off')
# plot raw pixel data
pyplot.imshow(image)
if i == 0:
    pyplot.title('Predicted')
ax = pyplot.gca()
# plot each box
for box in yhat['rois']:
    # get coordinates
    y1, x1, y2, x2 = box
    # calculate width and height of the box
    width, height = x2 - x1, y2 - y1
    # create the shape
    rect = Rectangle((x1, y1), width, height, fill=False, color='red')
    # draw the box
    ax.add_patch(rect)
```

Listing 26.64: Example of plotting the predicted masks on the photograph.

We can tie all of this together into a function that takes a dataset, model, and config and creates a plot of the first five photos in the dataset with ground truth and predicted bound boxes.

```
# plot a number of photos with ground truth and predictions
def plot_actual_vs_predicted(dataset, model, cfg, n_images=5):
    # load image and mask
    for i in range(n_images):
        # load the image and mask
        image = dataset.load_image(i)
        mask, _ = dataset.load_mask(i)
        # convert pixel values (e.g. center)
        scaled_image = mold_image(image, cfg)
```

```

# convert image into one sample
sample = expand_dims(scaled_image, 0)
# make prediction
yhat = model.detect(sample, verbose=0)[0]
# define subplot
pyplot.subplot(n_images, 2, i*2+1)
# turn off axis labels
pyplot.axis('off')
# plot raw pixel data
pyplot.imshow(image)
if i == 0:
    pyplot.title('Actual')
# plot masks
for j in range(mask.shape[2]):
    pyplot.imshow(mask[:, :, j], cmap='gray', alpha=0.3)
# get the context for drawing boxes
pyplot.subplot(n_images, 2, i*2+2)
# turn off axis labels
pyplot.axis('off')
# plot raw pixel data
pyplot.imshow(image)
if i == 0:
    pyplot.title('Predicted')
ax = pyplot.gca()
# plot each box
for box in yhat['rois']:
    # get coordinates
    y1, x1, y2, x2 = box
    # calculate width and height of the box
    width, height = x2 - x1, y2 - y1
    # create the shape
    rect = Rectangle((x1, y1), width, height, fill=False, color='red')
    # draw the box
    ax.add_patch(rect)
# show the figure
pyplot.show()

```

Listing 26.65: Example of a function for plotting actual and predicted masks on a number of photographs.

The complete example of loading the trained model and making a prediction for the first few images in the train and test datasets is listed below.

```

# detect kangaroos in photos with mask rcnn model
from os import listdir
from xml.etree import ElementTree
from numpy import zeros
from numpy import asarray
from numpy import expand_dims
from matplotlib import pyplot
from matplotlib.patches import Rectangle
from mrcnn.config import Config
from mrcnn.model import MaskRCNN
from mrcnn.model import mold_image
from mrcnn.utils import Dataset

```

```
# class that defines and loads the kangaroo dataset
class KangarooDataset(Dataset):
    # load the dataset definitions
    def load_dataset(self, dataset_dir, is_train=True):
        # define one class
        self.add_class("dataset", 1, "kangaroo")
        # define data locations
        images_dir = dataset_dir + '/images/'
        annotations_dir = dataset_dir + '/annots/'
        # find all images
        for filename in listdir(images_dir):
            # extract image id
            image_id = filename[:-4]
            # skip bad images
            if image_id in ['00090']:
                continue
            # skip all images after 150 if we are building the train set
            if is_train and int(image_id) >= 150:
                continue
            # skip all images before 150 if we are building the test/val set
            if not is_train and int(image_id) < 150:
                continue
            img_path = images_dir + filename
            ann_path = annotations_dir + image_id + '.xml'
            # add to dataset
            self.add_image('dataset', image_id=image_id, path=img_path, annotation=ann_path)

    # load all bounding boxes for an image
    def extract_boxes(self, filename):
        # load and parse the file
        root = ElementTree.parse(filename)
        boxes = list()
        # extract each bounding box
        for box in root.findall('.//bndbox'):
            xmin = int(box.find('xmin').text)
            ymin = int(box.find('ymin').text)
            xmax = int(box.find('xmax').text)
            ymax = int(box.find('ymax').text)
            coors = [xmin, ymin, xmax, ymax]
            boxes.append(coors)
        # extract image dimensions
        width = int(root.find('.//size/width').text)
        height = int(root.find('.//size/height').text)
        return boxes, width, height

    # load the masks for an image
    def load_mask(self, image_id):
        # get details of image
        info = self.image_info[image_id]
        # define box file location
        path = info['annotation']
        # load XML
        boxes, w, h = self.extract_boxes(path)
        # create one array for all masks, each on a different channel
        masks = zeros([h, w, len(boxes)], dtype='uint8')
        # create masks
```

```
class_ids = list()
for i in range(len(boxes)):
    box = boxes[i]
    row_s, row_e = box[1], box[3]
    col_s, col_e = box[0], box[2]
    masks[row_s:row_e, col_s:col_e, i] = 1
    class_ids.append(self.class_names.index('kangaroo'))
return masks, asarray(class_ids, dtype='int32')

# load an image reference
def image_reference(self, image_id):
    info = self.image_info[image_id]
    return info['path']

# define the prediction configuration
class PredictionConfig(Config):
    # define the name of the configuration
    NAME = "kangaroo_cfg"
    # number of classes (background + kangaroo)
    NUM_CLASSES = 1 + 1
    # simplify GPU config
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1

    # plot a number of photos with ground truth and predictions
    def plot_actual_vs_predicted(dataset, model, cfg, n_images=5):
        # load image and mask
        for i in range(n_images):
            # load the image and mask
            image = dataset.load_image(i)
            mask, _ = dataset.load_mask(i)
            # convert pixel values (e.g. center)
            scaled_image = mold_image(image, cfg)
            # convert image into one sample
            sample = expand_dims(scaled_image, 0)
            # make prediction
            yhat = model.detect(sample, verbose=0)[0]
            # define subplot
            pyplot.subplot(n_images, 2, i*2+1)
            # turn off axis labels
            pyplot.axis('off')
            # plot raw pixel data
            pyplot.imshow(image)
            if i == 0:
                pyplot.title('Actual')
            # plot masks
            for j in range(mask.shape[2]):
                pyplot.imshow(mask[:, :, j], cmap='gray', alpha=0.3)
            # get the context for drawing boxes
            pyplot.subplot(n_images, 2, i*2+2)
            # turn off axis labels
            pyplot.axis('off')
            # plot raw pixel data
            pyplot.imshow(image)
            if i == 0:
                pyplot.title('Predicted')
```

```

ax = pyplot.gca()
# plot each box
for box in yhat['rois']:
    # get coordinates
    y1, x1, y2, x2 = box
    # calculate width and height of the box
    width, height = x2 - x1, y2 - y1
    # create the shape
    rect = Rectangle((x1, y1), width, height, fill=False, color='red')
    # draw the box
    ax.add_patch(rect)
# show the figure
pyplot.show()

# load the train dataset
train_set = KangarooDataset()
train_set.load_dataset('kangaroo', is_train=True)
train_set.prepare()
print('Train: %d' % len(train_set.image_ids))
# load the test dataset
test_set = KangarooDataset()
test_set.load_dataset('kangaroo', is_train=False)
test_set.prepare()
print('Test: %d' % len(test_set.image_ids))
# create config
cfg = PredictionConfig()
# define the model
model = MaskRCNN(mode='inference', model_dir='./', config=cfg)
# load model weights
model_path = 'mask_rcnn_kangaroo_cfg_0005.h5'
model.load_weights(model_path, by_name=True)
# plot predictions for train dataset
plot_actual_vs_predicted(train_set, model, cfg)
# plot predictions for test dataset
plot_actual_vs_predicted(test_set, model, cfg)

```

Listing 26.66: Example of making predictions with the Mask RCNN model.

Running the example first creates a figure showing five photos from the training dataset with the ground truth bounding boxes, with the same photo and the predicted bounding boxes alongside.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model has done well on these examples, finding all of the kangaroos, even in the case where there are two or three in one photo. The second photo down (in the right column) does show a slip-up where the model has predicted a bounding box around the same kangaroo twice.

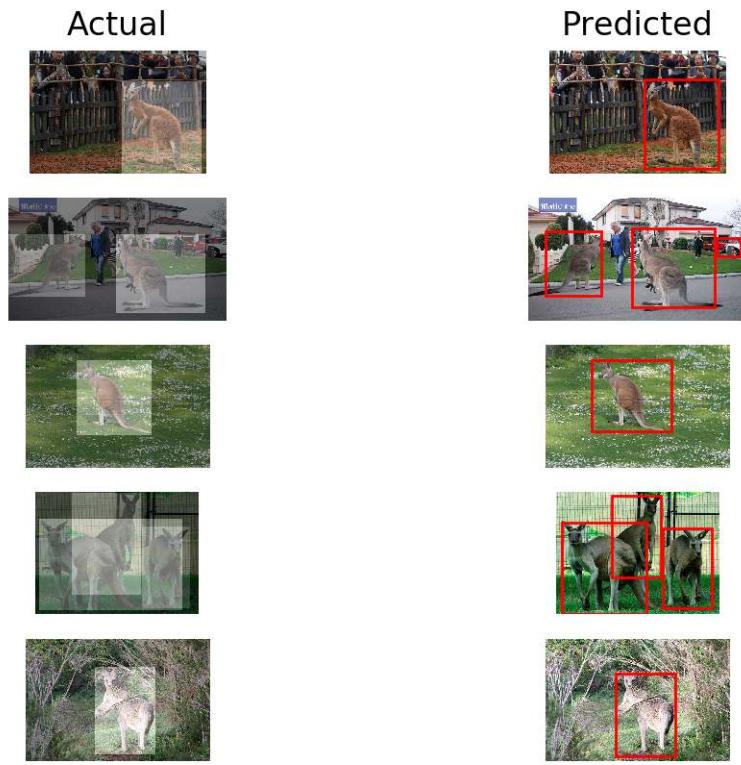


Figure 26.4: Plot of Photos of Kangaroos From the Training Dataset With Ground Truth and Predicted Bounding Boxes.

A second figure is created showing five photos from the test dataset with ground truth bounding boxes and predicted bounding boxes. These are images not seen during training, and again, in each photo, the model has detected the kangaroo. In this case, we can see a few minor object detection errors. Specifically, in the first photo a rock is detected as a kangaroo, and in the last two photos, the same kangaroo was detected twice. No doubt these differences can be ironed out with more training, perhaps with a larger dataset and/or data augmentation, to encourage the model to detect people as background and to detect a given kangaroo once only.

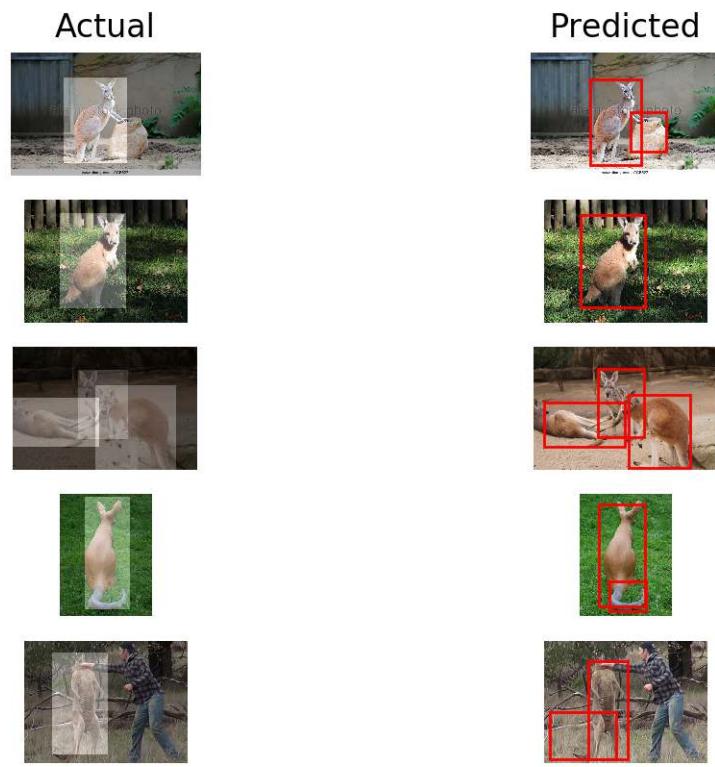


Figure 26.5: Plot of Photos of Kangaroos From the Test Dataset With Ground Truth and Predicted Bounding Boxes.

26.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Tune Model.** Experiment by tuning the model for better performance, such as using a different learning rate.
- **Apply Model.** Apply the model to detect kangaroos in new photographs, such as photos on Flickr.
- **New Dataset.** Find or develop another small dataset for object detection and develop a Mask R-CNN model for it.

If you explore any of these extensions, I'd love to know.

26.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

26.8.1 Papers

- *Mask R-CNN*, 2017.
<https://arxiv.org/abs/1703.06870>

26.8.2 Projects

- Kangaroo Dataset, GitHub.
<https://github.com/experiencor/kangaroo>
- Mask RCNN Project, GitHub.
https://github.com/matterport/Mask_RCNN

26.8.3 APIs

- `xml.etree.ElementTree` API.
<https://docs.python.org/3/library/xml.etree.elementtree.html>
- `matplotlib.patches.Rectangle` API.
https://matplotlib.org/api/_as_gen/matplotlib.patches.Rectangle.html
- `matplotlib.pyplot.subplot` API.
https://matplotlib.org/api/_as_gen/matplotlib.pyplot.subplot.html
- `matplotlib.pyplot.imshow` API.
https://matplotlib.org/api/_as_gen/matplotlib.pyplot.imshow.html

26.8.4 Articles

- Splash of Color: Instance Segmentation with Mask R-CNN and TensorFlow, 2018.
<https://engineering.matterport.com/splash-of-color-instance-segmentation-with-mask-r-cnn-and-tensorflow-7c761e238b46>
- Mask R-CNN — Inspect Balloon Trained Model, Notebook.
https://github.com/matterport/Mask_RCNN/blob/master/samples/balloon/inspect_balloon_model.ipynb
- Mask R-CNN — Train on Shapes Dataset, Notebook.
https://github.com/matterport/Mask_RCNN/blob/master/samples/shapes/train_shapes.ipynb
- mAP (mean Average Precision) for Object Detection, 2018.
https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173

26.9 Summary

In this tutorial, you discovered how to develop a Mask R-CNN model for kangaroo object detection in photographs. Specifically, you learned:

- How to prepare an object detection dataset ready for modeling with an R-CNN.
- How to use transfer learning to train an object detection model on a new dataset.

- How to evaluate a fit Mask R-CNN model on a test dataset and make predictions on new photos.

26.9.1 Next

This was the final tutorial in this part on object recognition. In the next part, you will discover how to develop deep learning models for face detection and face recognition.

Part VII

Face Recognition

Overview

In this part you will discover the computer vision problem of face recognition and how to use top-performing models like MTCNN for face detection and VGGFace2 and FaceNet for face identification and face verification. After reading the chapters in this part, you will know:

- Face recognition is the broad problem that involves first detecting faces in a photograph then either identifying who the person is or verifying their identity (Chapter [27](#)).
- How to perform face detection using classical computer vision methods and the state-of-the-art MTCNN deep learning neural network (Chapter [28](#)).
- About the VGGFace family of models and how to perform face recognition using the VGGFace2 model. (Chapter [29](#)).
- About the FaceNet model from Google and how to perform face recognition using this model (Chapter [30](#)).

Chapter 27

Deep Learning for Face Recognition

Face recognition is the problem of identifying and verifying people in a photograph by their face. It is a task that is trivially performed by humans, even under varying light and when faces are changed by age or obstructed with accessories and facial hair. Nevertheless, it has remained a challenging computer vision problem for decades until recently. Deep learning methods are able to leverage very large datasets of faces and learn rich and compact representations of faces, allowing modern models to first perform as-well and later to outperform the face recognition capabilities of humans. In this tutorial, you will discover the problem of face recognition and how deep learning methods can achieve superhuman performance. After reading this tutorial, you will know:

- Face recognition is the broad problem of identifying or verifying people in photographs and videos.
- Face recognition is a process comprised of detection, alignment, feature extraction, and a recognition task.
- Deep learning models first approached then exceeded human performance for face recognition tasks.

Let's get started.

27.1 Overview

This tutorial is divided into five parts; they are:

1. Faces in Photographs
2. Process of Automatic Face Recognition
3. Face Detection Task
4. Face Recognition Tasks
5. Deep Learning for Face Recognition

27.2 Faces in Photographs

There is often a need to automatically recognize the people in a photograph. There are many reasons why we might want to automatically recognize a person in a photograph. For example:

- We may want to restrict access to a resource to one person, called face authentication.
- We may want to confirm that the person matches their ID, called face verification.
- We may want to assign a name to a face, called face identification.

Generally, we refer to this as the problem of automatic *face recognition* and it may apply to both still photographs or faces in streams of video. Humans can perform this task very easily. We can find the faces in an image and comment as to who the people are, if they are known. We can do this very well, such as when the people have aged, are wearing sunglasses, have different colored hair, are looking in different directions, and so on. We can do this so well that we find faces where there aren't any, such as in clouds. Nevertheless, this remains a hard problem to perform automatically with software, even after 60 or more years of research. Until perhaps very recently.

For example, recognition of face images acquired in an outdoor environment with changes in illumination and/or pose remains a largely unsolved problem. In other words, current systems are still far away from the capability of the human perception system.

— *Face Recognition: A Literature Survey*, 2003.

27.3 Process of Automatic Face Recognition

Face recognition is the problem of identifying or verifying faces in a photograph.

A general statement of the problem of machine recognition of faces can be formulated as follows: given still or video images of a scene, identify or verify one or more persons in the scene using a stored database of faces

— *Face Recognition: A Literature Survey*, 2003.

Face recognition is often described as a process that first involves four steps; they are: face detection, face alignment, feature extraction, and finally face recognition.

1. **Face Detection.** Locate one or more faces in the image and mark with a bounding box.
2. **Face Alignment.** Normalize the face to be consistent with the database, such as geometry and photometrics.
3. **Feature Extraction.** Extract features from the face that can be used for the recognition task.

4. **Face Recognition.** Perform matching of the face against one or more known faces in a prepared database.

A given system may have a separate module or program for each step, which was traditionally the case, or may combine some or all of the steps into a single process. A helpful overview of this process is provided in the book *Handbook of Face Recognition*, provided below:

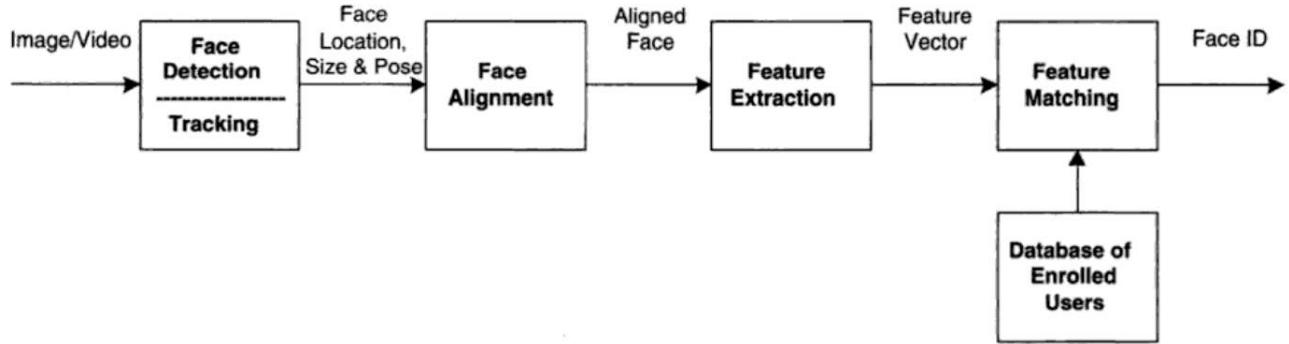


Fig. 1.2. Face recognition processing flow.

Figure 27.1: Overview of the Steps in a Face Recognition Process. Taken from *Handbook of Face Recognition*.

27.4 Face Detection Task

Face detection is the non-trivial first step in face recognition. It is a problem of object recognition that requires that both the location of each face in a photograph is identified (e.g. the position) and the extent of the face is localized (e.g. with a bounding box). Object recognition itself is a challenging problem, although in this case, it is simpler as there is only one type of object, e.g. faces, to be localized, although faces can vary wildly.

The human face is a dynamic object and has a high degree of variability in its appearance, which makes face detection a difficult problem in computer vision.

— *Face Detection: A Survey*, 2001.

Further, because it is the first step in a broader face recognition system, face detection must be robust. For example, a face cannot be recognized if it cannot first be detected. That means faces must be detected with all manner of orientations, angles, light levels, hairstyles, hats, glasses, facial hair, makeup, ages, and so on.

As a visual front-end processor, a face detection system should also be able to achieve the task regardless of illumination, orientation, and camera distance

— *Face Detection: A Survey*, 2001.

The 2001 paper titled *Face Detection: A Survey* provides a taxonomy of face detection methods that can be broadly divided into two main groups:

- Feature-Based.
- Image-Based.

The feature-based face detection uses hand-crafted filters that search for and locate faces in photographs based on a deep knowledge of the domain. They can be very fast and very effective when the filters match, although they can fail dramatically when they don't, e.g. making them somewhat fragile.

... make explicit use of face knowledge and follow the classical detection methodology in which low level features are derived prior to knowledge-based analysis. The apparent properties of the face such as skin color and face geometry are exploited at different system levels.

— *Face Detection: A Survey*, 2001.

Alternately, image-based face detection is holistic and learns how to automatically locate and extract faces from the entire image. Neural networks fit into this class of methods.

... address face detection as a general recognition problem. Image-based representations of faces, for example in 2D intensity arrays, are directly classified into a face group using training algorithms without feature derivation and analysis. [...] these relatively new techniques incorporate face knowledge implicitly into the system through mapping and training schemes.

— *Face Detection: A Survey*, 2001.

Perhaps the dominant method for face detection used for many years (and was used in many cameras) was described in the 2004 paper titled *Robust Real-time Object Detection*, called the detector cascade or simply *cascade*.

Their detector, called detector cascade, consists of a sequence of simple-to-complex face classifiers and has attracted extensive research efforts. Moreover, detector cascade has been deployed in many commercial products such as smartphones and digital cameras. While cascade detectors can accurately find visible upright faces, they often fail to detect faces from different angles, e.g. side view or partially occluded faces.

— *Multi-view Face Detection Using Deep Convolutional Neural Networks*, 2015.

27.5 Face Recognition Tasks

The task of face recognition is broad and can be tailored to the specific needs of a prediction problem. For example, in the 1995 paper titled *Human And Machine Recognition Of Faces: A Survey*, the authors describe three face recognition tasks:

- **Face Matching:** Find the best match for a given face.
- **Face Similarity:** Find faces that are most similar to a given face.
- **Face Transformation:** Generate new faces that are similar to a given face.

They summarize these three separate tasks as follows:

Matching requires that the candidate matching face image be in some set of face images selected by the system. Similarity detection requires in addition to matching that images of faces be found which are similar to a recalled face this requires that the similarity measure used by the recognition system closely match the similarity measures used by humans Transformation applications require that new images created by the system be similar to human recollections of a face.

— *Human And Machine Recognition Of Faces: A Survey*, 1995.

The 2011 book on face recognition titled *Handbook of Face Recognition* describes two main modes for face recognition, as:

- **Face Verification.** A one-to-one mapping of a given face against a known identity (e.g. is this the person?).
- **Face Identification.** A one-to-many mapping for a given face against a database of known faces (e.g. who is this person?).

A face recognition system is expected to identify faces present in images and videos automatically. It can operate in either or both of two modes: (1) face verification (or authentication), and (2) face identification (or recognition).

— Page 1, *Handbook of Face Recognition*. 2011.

We can describe the problem of face recognition as a supervised predictive modeling task trained on samples with inputs and outputs. In all tasks, the input is a photo that contains at least one face, most likely a detected face that may also have been aligned. The output varies based on the type of prediction required for the task; for example:

- It may then be a binary class label or binary class probability in the case of a face verification task.
- It may be a categorical class label or set of probabilities for a face identification task.
- It may be a similarity metric in the case of a similarity type task.

27.6 Deep Learning for Face Recognition

Face recognition has remained an active area of research in computer vision. Perhaps one of the more widely known and adopted *machine learning* methods for face recognition was described in the 1991 paper titled *Face Recognition Using Eigenfaces*. Their method, called simply *Eigenfaces*, was a milestone as it achieved impressive results and demonstrated the capability of simple holistic approaches.

Face images are projected onto a feature space (“face space”) that best encodes the variation among known face images. The face space is defined by the “eigenfaces”, which are the eigenvectors of the set of faces; they do not necessarily correspond to isolated features such as eyes, ears, and noses

— *Face Recognition Using Eigenfaces*, 1991.

The 2018 paper titled *Deep Face Recognition: A Survey*, provides a helpful summary of the state of face recognition research over the last nearly 30 years, highlighting the broad trend from holistic learning methods (such as Eigenfaces), to local handcrafted feature detection, to shallow learning methods, to finally deep learning methods that are currently state-of-the-art.

The holistic approaches dominated the face recognition community in the 1990s. In the early 2000s, handcrafted local descriptors became popular, and the local feature learning approach were introduced in the late 2000s. [...] [shallow learning method] performance steadily improves from around 60% to above 90%, while deep learning boosts the performance to 99.80% in just three years.

— *Deep Face Recognition: A Survey*, 2018.

Given the breakthrough of AlexNet in 2012 for the simpler problem of image classification, there was a flurry of research and publications in 2014 and 2015 on deep learning methods for face recognition. Capabilities quickly achieved near-human-level performance, then exceeded human-level performance on a standard face recognition dataset within a three year period, which is an astounding rate of improvement given the prior decades of effort. There are perhaps four milestone systems on deep learning for face recognition that drove these innovations; they are: DeepFace, the DeepID series of systems, VGGFace, and FaceNet. Let’s briefly touch on each.

DeepFace is a system based on deep convolutional neural networks described by Yaniv Taigman, et al. from Facebook AI Research and Tel Aviv. It was described in the 2014 paper titled *DeepFace: Closing the Gap to Human-Level Performance in Face Verification*. It was perhaps the first major leap forward using deep learning for face recognition, achieving near human-level performance on a standard benchmark dataset.

Our method reaches an accuracy of 97.35% on the Labeled Faces in the Wild (LFW) dataset, reducing the error of the current state-of-the-art by more than 27%, closely approaching human-level performance.

— *DeepFace: Closing the Gap to Human-Level Performance in Face Verification*, 2014.

The DeepID, or *Deep hidden IDentity features*, is a series of systems (e.g. DeepID, DeepID2, etc.), first described by Yi Sun, et al. in their 2014 paper titled *Deep Learning Face Representation from Predicting 10,000 Classes*. Their system was first described much like DeepFace, although was expanded in subsequent publications to support both identification and verification tasks by training via contrastive loss.

The key challenge of face recognition is to develop effective feature representations for reducing intra-personal variations while enlarging inter-personal differences. [...] The face identification task increases the inter-personal variations by drawing DeepID2 features extracted from different identities apart, while the face verification task reduces the intra-personal variations by pulling DeepID2 features extracted from the same identity together, both of which are essential to face recognition.

— *Deep Learning Face Representation by Joint Identification-Verification*, 2014.

The DeepID systems were among the first deep learning models to achieve better-than-human performance on the task, e.g. DeepID2 achieved 99.15% on the Labeled Faces in the Wild (LFW) dataset, which is better-than-human performance of 97.53%. Subsequent systems such as FaceNet and VGGFace improved upon these results.

FaceNet was described by Florian Schroff, et al. at Google in their 2015 paper titled *FaceNet: A Unified Embedding for Face Recognition and Clustering*. Their system achieved then state-of-the-art results and presented an innovation called *triplet loss* that allowed images to be encoded efficiently as feature vectors that allowed rapid similarity calculation and matching via distance calculations.

FaceNet, that directly learns a mapping from face images to a compact Euclidean space where distances directly correspond to a measure of face similarity. [...] Our method uses a deep convolutional network trained to directly optimize the embedding itself, rather than an intermediate bottleneck layer as in previous deep learning approaches. To train, we use triplets of roughly aligned matching / non-matching face patches generated using a novel online triplet mining method

— *FaceNet: A Unified Embedding for Face Recognition and Clustering*, 2015.

The VGGFace (for lack of a better name) was developed by Omkar Parkhi, et al. from the Visual Geometry Group (VGG) at Oxford and was described in their 2015 paper titled *Deep Face Recognition*. In addition to a better-tuned model, the focus of their work was on how to collect a very large training dataset and use this to train a very deep CNN model for face recognition that allowed them to achieve then state-of-the-art results on standard datasets.

... we show how a very large scale dataset (2.6M images, over 2.6K people) can be assembled by a combination of automation and human in the loop

— *Deep Face Recognition*, 2015.

Although these may be the key early milestones in the field of deep learning for computer vision, progress has continued, with much innovation focused on loss functions to effectively train the models.

27.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

27.7.1 Books

- *Handbook of Face Recognition*, Second Edition, 2011.
<https://amzn.to/2EuR80o>

27.7.2 Face Recognition Papers

- *Face Recognition: A Literature Survey*, 2003.
<https://dl.acm.org/citation.cfm?id=954342>
- *Face Detection: A Survey*, 2001.
<https://www.sciencedirect.com/science/article/pii/S107731420190921X>
- *Human And Machine Recognition Of Faces: A Survey*, 1995.
<https://ieeexplore.ieee.org/abstract/document/381842>
- *Robust Real-time Object Detection*, 2004.
<https://link.springer.com/article/10.1023/B:VISI.0000013087.49260.fb>
- *Face Recognition Using Eigenfaces*, 1991.
<https://www.computer.org/csdl/proceedings/cvpr/1991/2148/00/00139758.pdf>

27.7.3 Deep Learning Face Recognition Papers

- *Deep Face Recognition: A Survey*, 2018.
<https://arxiv.org/abs/1804.06655>
- *Deep Face Recognition*, 2015.
http://cis.csuohio.edu/~sschung/CIS660/DeepFaceRecognition_parkhi15.pdf
- *FaceNet: A Unified Embedding for Face Recognition and Clustering*, 2015.
<https://arxiv.org/abs/1503.03832>
- *DeepFace: Closing the Gap to Human-Level Performance in Face Verification*, 2014.
<https://ieeexplore.ieee.org/document/6909616>
- *Deep Learning Face Representation by Joint Identification-Verification*, 2014.
<https://arxiv.org/abs/1406.4773>
- *Deep Learning Face Representation from Predicting 10,000 Classes*, 2014.
<https://dl.acm.org/citation.cfm?id=2679769>
- *Multi-view Face Detection Using Deep Convolutional Neural Networks*, 2015.
<https://arxiv.org/abs/1502.02766>
- *From Facial Parts Responses to Face Detection: A Deep Learning Approach*, 2015.
<https://arxiv.org/abs/1509.06451>

- *Surpassing Human-Level Face Verification Performance on LFW with GaussianFace*, 2014.
<https://arxiv.org/abs/1404.3840>

27.7.4 Articles

- Facial recognition system, Wikipedia.
https://en.wikipedia.org/wiki/Facial_recognition_system
- Facial recognition, Wikipedia.
https://en.wikipedia.org/wiki/Facial_recognition
- Face detection, Wikipedia.
https://en.wikipedia.org/wiki/Face_detection
- Labeled Faces in the Wild Dataset.
<http://vis-www.cs.umass.edu/lfw/>

27.8 Summary

In this tutorial, you discovered the problem of face recognition and how deep learning methods can achieve superhuman performance. Specifically, you learned:

- Face recognition is the broad problem of identifying or verifying people in photographs and videos.
- Face recognition is a process comprised of detection, alignment, feature extraction, and a recognition task.
- Deep learning models first approached then exceeded human performance for face recognition tasks.

27.8.1 Next

In the next section, you will discover how to perform face detection using classical methods as well as a deep learning convolutional neural network.

Chapter 28

How to Detect Faces in Photographs

Face detection is a computer vision problem that involves finding faces in photos. It is a trivial problem for humans to solve and has been solved reasonably well by classical feature-based techniques, such as the cascade classifier. More recently deep learning methods have achieved state-of-the-art results on standard benchmark face detection datasets. One example is the Multi-task Cascade Convolutional Neural Network, or MTCNN for short. In this tutorial, you will discover how to perform face detection in Python using classical and deep learning models. After completing this tutorial, you will know:

- Face detection is a non-trivial computer vision problem for identifying and localizing faces in images.
- Face detection can be performed using the classical feature-based cascade classifier using the OpenCV library.
- State-of-the-art face detection can be achieved using a Multi-task Cascade CNN via the MTCNN library.

Let's get started.

28.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Face Detection
2. Test Photographs
3. Face Detection With OpenCV
4. Face Detection With Deep Learning

28.2 Face Detection

Face detection is a problem in computer vision of locating and localizing one or more faces in a photograph (introduced in Chapter 27). Locating a face in a photograph refers to finding the coordinates of the face in the image, whereas localization refers to demarcating the extent of the face, often via a bounding box around the face.

A general statement of the problem can be defined as follows: Given a still or video image, detect and localize an unknown number (if any) of faces

— *Face Detection: A Survey*, 2001.

Detecting faces in a photograph is easily solved by humans, although this has historically been challenging for computers given the dynamic nature of faces. For example, faces must be detected regardless of orientation or angle they are facing, light levels, clothing, accessories, hair color, facial hair, makeup, age, and so on.

The human face is a dynamic object and has a high degree of variability in its appearance, which makes face detection a difficult problem in computer vision.

— *Face Detection: A Survey*, 2001.

Given a photograph, a face detection system will output zero or more bounding boxes that contain faces. Detected faces can then be provided as input to a subsequent system, such as a face recognition system.

Face detection is a necessary first-step in face recognition systems, with the purpose of localizing and extracting the face region from the background.

— *Face Detection: A Survey*, 2001.

There are perhaps two main approaches to face recognition: feature-based methods that use hand-crafted filters to search for and detect faces, and image-based methods that learn holistically how to extract faces from the entire image.

28.3 Test Photographs

We need test images for face detection in this tutorial. To keep things simple, we will use two test images: one with two faces, and one with many faces. We're not trying to push the limits of face detection, just demonstrate how to perform face detection with normal front-on photographs of people. The first image is a photo of two college students taken by CollegeDegrees360¹ and made available under a permissive license.

¹<https://www.flickr.com/photos/83633410@N07/7658261288/>



Figure 28.1: College Students (`test1.jpg`).

Download the image and place it in your current working directory with the filename `test1.jpg`.

- Download Photo of College Students (`test1.jpg`).²

The second image is a photograph of a number of people on a swim team taken by Bob n Renee³ and released under a permissive license.



Figure 28.2: Swim Team (`test2.jpg`).

²<https://machinelearningmastery.com/wp-content/uploads/2019/03/test1.jpg>

³<https://www.flickr.com/photos/bobnrenee/2370369784/>

Download the image and place it in your current working directory with the filename `test2.jpg`.

- Download Photo of Swim Team (`test2.jpg`).⁴

28.4 Face Detection With OpenCV

Feature-based face detection algorithms are fast and effective and have been used successfully for decades. Perhaps the most successful example is a technique called cascade classifiers first described by Paul Viola and Michael Jones and their 2001 paper titled *Rapid Object Detection using a Boosted Cascade of Simple Features*. In the paper, effective features are learned using the AdaBoost algorithm, although importantly, multiple models are organized into a hierarchy or *cascade*. In the paper, the AdaBoost model is used to learn a range of very simple or weak features in each face, that together provide a robust classifier.

... feature selection is achieved through a simple modification of the AdaBoost procedure: the weak learner is constrained so that each weak classifier returned can depend on only a single feature . As a result each stage of the boosting process, which selects a new weak classifier, can be viewed as a feature selection process.

— *Rapid Object Detection using a Boosted Cascade of Simple Features*, 2001.

The models are then organized into a hierarchy of increasing complexity, called a *cascade*. Simpler classifiers operate on candidate face regions directly, acting like a coarse filter, whereas complex classifiers operate only on those candidate regions that show the most promise as faces.

... a method for combining successively more complex classifiers in a cascade structure which dramatically increases the speed of the detector by focusing attention on promising regions of the image.

— *Rapid Object Detection using a Boosted Cascade of Simple Features*, 2001.

The result is a very fast and effective face detection algorithm that has been the basis for face detection in consumer products, such as cameras.

Their detector, called detector cascade, consists of a sequence of simple-to-complex face classifiers and has attracted extensive research efforts. Moreover, detector cascade has been deployed in many commercial products such as smartphones and digital cameras.

— *Multi-view Face Detection Using Deep Convolutional Neural Networks*, 2015.

It is a modestly complex classifier that has also been tweaked and refined over the last nearly 20 years. A modern implementation of the Classifier Cascade face detection algorithm is provided in the OpenCV library. This is a C++ computer vision library that provides a Python interface. The benefit of this implementation is that it provides pre-trained face detection models, and provides an interface to train a model on your own dataset. OpenCV can be installed by the package manager system on your platform, or via pip; for example:

⁴<https://machinelearningmastery.com/wp-content/uploads/2019/03/test2.jpg>

```
sudo pip install opencv-python
```

Listing 28.1: Example of installing OpenCV via pip.

Once the installation process is complete, it is important to confirm that the library was installed correctly. This can be achieved by importing the library and checking the version number; for example:

```
# check opencv version
import cv2
# print version number
print(cv2.__version__)
```

Listing 28.2: Example of confirming the version of OpenCV.

Running the example will import the library and print the version. In this case, we are using version 4 of the library.

4.1.1

Listing 28.3: Example output from confirming the version of OpenCV.

OpenCV provides the `CascadeClassifier` class that can be used to create a cascade classifier for face detection. The constructor can take a filename as an argument that specifies the XML file for a pre-trained model. OpenCV provides a number of pre-trained models as part of the installation. These are available on your system and are also available on the OpenCV GitHub project. Download a pre-trained model for frontal face detection from the OpenCV GitHub project and place it in your current working directory with the filename `haarcascade_frontalface_default.xml`.

- Download Open Frontal Face Detection Model (`haarcascade_frontalface_default.xml`).⁵

Once downloaded, we can load the model as follows:

```
...
# load the pre-trained model
classifier = CascadeClassifier('haarcascade_frontalface_default.xml')
```

Listing 28.4: Example of loading the pre-trained face detector model.

Once loaded, the model can be used to perform face detection on a photograph by calling the `detectMultiScale()` function. This function will return a list of bounding boxes for all faces detected in the photograph.

```
...
# perform face detection
bboxes = classifier.detectMultiScale(pixels)
# print bounding box for each detected face
for box in bboxes:
    print(box)
```

Listing 28.5: Example of performing face detection with the prepared model.

We can demonstrate this with an example with the college students photograph (`test.jpg`). The photo can be loaded using OpenCV via the `imread()` function.

⁵<https://goo.gl/msZUPm>

```
...
# load the photograph
pixels = imread('test1.jpg')
```

Listing 28.6: Example of loading image pixels from file.

The complete example of performing face detection on the college students photograph with a pre-trained cascade classifier in OpenCV is listed below.

```
# example of face detection with opencv cascade classifier
from cv2 import imread
from cv2 import CascadeClassifier
# load the photograph
pixels = imread('test1.jpg')
# load the pre-trained model
classifier = CascadeClassifier('haarcascade_frontalface_default.xml')
# perform face detection
bboxes = classifier.detectMultiScale(pixels)
# print bounding box for each detected face
for box in bboxes:
    print(box)
```

Listing 28.7: Example of face detection with OpenCV.

Running the example first loads the photograph, then loads and configures the cascade classifier; faces are detected and each bounding box is printed. Each box lists the `x` and `y` coordinates for the bottom-left-hand-corner of the bounding box, as well as the `width` and the `height`. The results suggest that two bounding boxes were detected.

```
[174 75 107 107]
[360 102 101 101]
```

Listing 28.8: Example output from face detection with OpenCV.

We can update the example to plot the photograph and draw each bounding box. This can be achieved by drawing a rectangle for each box directly over the pixels of the loaded image using the `rectangle()` function that takes two points.

```
...
# extract
x, y, width, height = box
x2, y2 = x + width, y + height
# draw a rectangle over the pixels
rectangle(pixels, (x, y), (x2, y2), (0,0,255), 1)
```

Listing 28.9: Example drawing a bounding box as a rectangle.

We can then plot the photograph and keep the window open until we press a key to close it.

```
...
# show the image
imshow('face detection', pixels)
# keep the window open until we press a key
waitKey(0)
# close the window
destroyAllWindows()
```

Listing 28.10: Example showing an image with OpenCV.

The complete example is listed below.

```
# plot photo with detected faces using opencv cascade classifier
from cv2 import imread
from cv2 import imshow
from cv2 import waitKey
from cv2 import destroyAllWindows
from cv2 import CascadeClassifier
from cv2 import rectangle
# load the photograph
pixels = imread('test1.jpg')
# load the pre-trained model
classifier = CascadeClassifier('haarcascade_frontalface_default.xml')
# perform face detection
bboxes = classifier.detectMultiScale(pixels)
# print bounding box for each detected face
for box in bboxes:
    # extract
    x, y, width, height = box
    x2, y2 = x + width, y + height
    # draw a rectangle over the pixels
    rectangle(pixels, (x, y), (x2, y2), (0,0,255), 1)
# show the image
imshow('face detection', pixels)
# keep the window open until we press a key
waitKey(0)
# close the window
destroyAllWindows()
```

Listing 28.11: Example of face detection with OpenCV for first test image and plot the results.

Running the example, we can see that the photograph was plotted correctly and that each face was correctly detected.



Figure 28.3: College Students Photograph With Faces Detected using OpenCV Cascade Classifier.

We can try the same code on the second photograph of the swim team, specifically `test2.jpg`.

```
...
# load the photograph
pixels = imread('test2.jpg')
```

Listing 28.12: Example of face detection with OpenCV for the second test image.

Running the example, we can see that many of the faces were detected correctly, but the result is not perfect. We can see that a face on the first or bottom row of people was detected twice, that a face on the middle row of people was not detected, and that the background on the third or top row was detected as a face.



Figure 28.4: Swim Team Photograph With Faces Detected using OpenCV Cascade Classifier.

The `detectMultiScale()` function provides some arguments to help tune the usage of the classifier. Two parameters of note are `scaleFactor` and `minNeighbors`; for example:

```
...
# perform face detection
bboxes = classifier.detectMultiScale(pixels, 1.1, 3)
```

Listing 28.13: Example of arguments with default values for the `detectMultiScale()` function.

The `scaleFactor` controls how the input image is scaled prior to detection, e.g. is it scaled up or down, which can help to better find the faces in the image. The default value is 1.1 (10% increase), although this can be lowered to values such as 1.05 (5% increase) or raised to values such as 1.4 (40% increase). The `minNeighbors` determines how robust each detection must be in order to be reported, e.g. the number of candidate rectangles that found the face. The

default is 3, but this can be lowered to 1 to detect a lot more faces and will likely increase the false positives, or increase to 6 or more to require a lot more confidence before a face is detected.

The `scaleFactor` and `minNeighbors` often require tuning for a given image or dataset in order to best detect the faces. It may be helpful to perform a sensitivity analysis across a grid of values and see what works well or best in general on one or multiple photographs. A fast strategy may be to lower (or increase for small photos) the `scaleFactor` until all faces are detected, then increase the `minNeighbors` until all false positives disappear, or close to it.

With some tuning, I found that a `scaleFactor` of 1.05 successfully detected all of the faces, but the background detected as a face did not disappear until a `minNeighbors` of 8, after which three faces on the middle row were no longer detected.

```
...
# perform face detection
bboxes = classifier.detectMultiScale(pixels, 1.05, 8)
```

Listing 28.14: Example of tuned arguments for the `detectMultiScale()` function.

The results are not perfect, and perhaps better results can be achieved with further tuning, and perhaps post-processing of the bounding boxes.



Figure 28.5: Swim Team Photograph With Faces Detected Using OpenCV Cascade Classifier After Some Tuning.

28.5 Face Detection With Deep Learning

A number of deep learning methods have been developed and demonstrated for face detection. Perhaps one of the more popular approaches is called the *Multi-Task Cascaded Convolutional Neural Network*, or MTCNN for short, described by Kaipeng Zhang, et al. in the 2016 paper titled *Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks*. The MTCNN is popular because it achieved then state-of-the-art results on a range of benchmark

datasets, and because it is capable of also recognizing other facial features such as eyes and mouth, called landmark detection.

The network uses a cascade structure with three networks; first the image is rescaled to a range of different sizes (called an image pyramid), then the first model (Proposal Network or P-Net) proposes candidate facial regions, the second model (Refine Network or R-Net) filters the bounding boxes, and the third model (Output Network or O-Net) proposes facial landmarks.

The proposed CNNs consist of three stages. In the first stage, it produces candidate windows quickly through a shallow CNN. Then, it refines the windows to reject a large number of non-faces windows through a more complex CNN. Finally, it uses a more powerful CNN to refine the result and output facial landmarks positions.

— *Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks*, 2016.

The image below taken from the paper provides a helpful summary of the three stages from top-to-bottom and the output of each stage left-to-right.

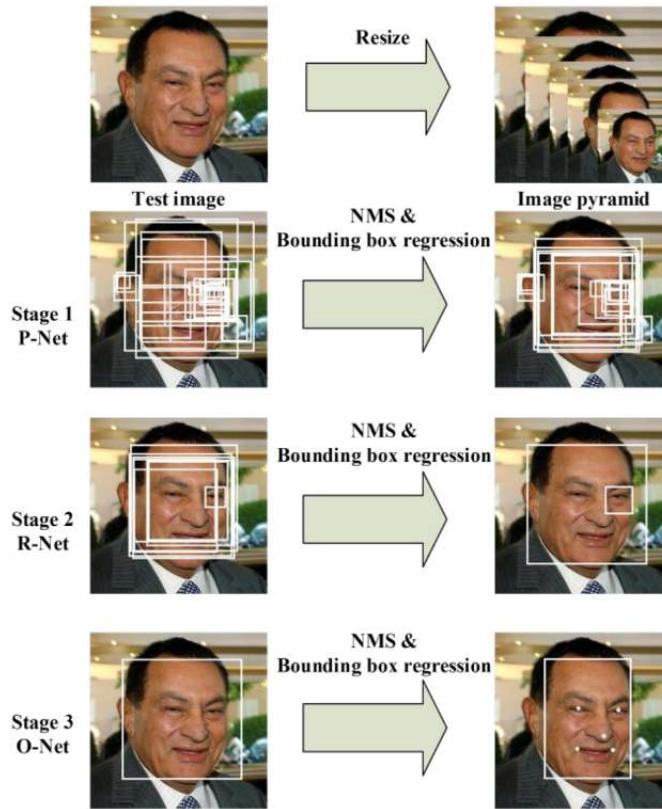


Figure 28.6: Pipeline for the Multi-Task Cascaded Convolutional Neural Network Taken from: Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks.

The model is called a multi-task network because each of the three models in the cascade (P-Net, R-Net and O-Net) are trained on three tasks, e.g. make three types of predictions; they are: face classification, bounding box regression, and facial landmark localization. The three models are not connected directly; instead, outputs of the previous stage are fed as input to the next stage. This allows additional processing to be performed between stages; for example,

non-maximum suppression (NMS) is used to filter the candidate bounding boxes proposed by the first-stage P-Net prior to providing them to the second stage R-Net model.

The MTCNN architecture is reasonably complex to implement. Thankfully, there are open source implementations of the architecture that can be trained on new datasets, as well as pre-trained models that can be used directly for face detection. Of note is the official release with the code and models used in the paper, with the implementation provided in the Caffe deep learning framework. Perhaps the best-of-breed third-party Python-based MTCNN project is called *MTCNN* by Ivan de Paz Centeno, or *ipazc*, made available under a permissive MIT open source license⁶.

The MTCNN project, which we will refer to as *ipazc/MTCNN* to differentiate it from the name of the network, provides an implementation of the MTCNN architecture using TensorFlow and OpenCV. There are two main benefits to this project; first, it provides a top-performing pre-trained model and the second is that it can be installed as a library ready for use in your own code. The library can be installed via `pip`; for example:

```
sudo pip install mtcnn
```

Listing 28.15: Example of installing MTCNN via `pip`.

After successful installation, you should see a message like:

```
Successfully installed mtcnn-0.1.0
```

Listing 28.16: Example of successful installation message.

You can then confirm that the library was installed correctly via `pip`; for example:

```
sudo pip show mtcnn
```

Listing 28.17: Example of querying the installed library with `pip`.

You should see output like that listed below. In this case, you can see that we are using version 0.1.0 of the library.

```
Name: mtcnn
Version: 0.1.0
Summary: Multi-task Cascaded Convolutional Neural Networks for Face Detection, based on
TensorFlow
Home-page: http://github.com/ipazc/mtcnn
Author: Ivn de Paz Centeno
Author-email: ipazc@unileon.es
License: MIT
Location: ...
Requires: opencv-python, keras
Required-by:
```

Listing 28.18: Example output of querying the installed library with `pip`.

You can also confirm that the library was installed correctly via Python, as follows:

```
# confirm mtcnn was installed correctly
import mtcnn
# print version
print(mtcnn.__version__)
```

Listing 28.19: Example of checking the version of the MTCNN library.

⁶Clone of the project is available here: <https://github.com/jbrownlee/mtcnn>

Running the example will load the library, confirming it was installed correctly; and print the version.

```
0.1.0
```

Listing 28.20: Example output from checking the version of the MTCNN library.

Now that we are confident that the library was installed correctly, we can use it for face detection. An instance of the network can be created by calling the `MTCNN()` constructor. By default, the library will use the pre-trained model, although you can specify your own model via the `weights_file` argument and specify a path or URL, for example:

```
...
model = MTCNN(weights_file='filename.npy')
```

Listing 28.21: Example of creating an MTCNN model with a specific set of weights

The minimum box size for detecting a face can be specified via the `min_face_size` argument, which defaults to 20 pixels. The constructor also provides a `scale_factor` argument to specify the scale factor for the input image, which defaults to 0.709. Once the model is configured and loaded, it can be used directly to detect faces in photographs by calling the `detect_faces()` function. This returns a list of dict objects, each providing a number of keys for the details of each face detected, including:

- ‘`box`’: Providing the `x`, `y` of the bottom left of the bounding box, as well as the `width` and `height` of the box.
- ‘`confidence`’: The probability confidence of the prediction.
- ‘`keypoints`’: Providing a dict with dots for the ‘`left_eye`’, ‘`right_eye`’, ‘`nose`’, ‘`mouth_left`’, and ‘`mouth_right`’.

For example, we can perform face detection on the college students photograph as follows:

```
# face detection with mtcnn on a photograph
from matplotlib import pyplot
from mtcnn.mtcnn import MTCNN
# load image from file
filename = 'test1.jpg'
pixels = pyplot.imread(filename)
# create the detector, using default weights
detector = MTCNN()
# detect faces in the image
faces = detector.detect_faces(pixels)
for face in faces:
    print(face)
```

Listing 28.22: Example of basic face detection with the MTCNN library.

Running the example loads the photograph, loads the model, performs face detection, and prints a list of each face detected.

```
{'box': [186, 71, 87, 115], 'confidence': 0.9994562268257141, 'keypoints': {'left_eye': (207, 110), 'right_eye': (252, 119), 'nose': (220, 143), 'mouth_left': (200, 148), 'mouth_right': (244, 159)}}
```

```
{'box': [368, 75, 108, 138], 'confidence': 0.998593270778656, 'keypoints': {'left_eye': (392, 133), 'right_eye': (441, 140), 'nose': (407, 170), 'mouth_left': (388, 180), 'mouth_right': (438, 185)}}}
```

Listing 28.23: Example output from basic face detection with the MTCNN library.

We can draw the boxes on the image by first plotting the image with Matplotlib, then creating a `Rectangle` object using the `x`, `y` and `width` and `height` of a given bounding box; for example:

```
...
# get coordinates
x, y, width, height = result['box']
# create the shape
rect = Rectangle((x, y), width, height, fill=False, color='red')
```

Listing 28.24: Example of drawing a bounding box for a detected face.

Below is a function named `draw_image_with_boxes()` that shows the photograph and then draws a box for each bounding box detected.

```
# draw an image with detected objects
def draw_image_with_boxes(filename, result_list):
    # load the image
    data = pyplot.imread(filename)
    # plot the image
    pyplot.imshow(data)
    # get the context for drawing boxes
    ax = pyplot.gca()
    # plot each box
    for result in result_list:
        # get coordinates
        x, y, width, height = result['box']
        # create the shape
        rect = Rectangle((x, y), width, height, fill=False, color='red')
        # draw the box
        ax.add_patch(rect)
    # show the plot
    pyplot.show()
```

Listing 28.25: Example of a function for drawing a bounding boxes for detected faces.

The complete example making use of this function is listed below.

```
# face detection with mtcnn on a photograph
from matplotlib import pyplot
from matplotlib.patches import Rectangle
from mtcnn.mtcnn import MTCNN

# draw an image with detected objects
def draw_image_with_boxes(filename, result_list):
    # load the image
    data = pyplot.imread(filename)
    # plot the image
    pyplot.imshow(data)
    # get the context for drawing boxes
    ax = pyplot.gca()
    # plot each box
```

```
for result in result_list:
    # get coordinates
    x, y, width, height = result['box']
    # create the shape
    rect = Rectangle((x, y), width, height, fill=False, color='red')
    # draw the box
    ax.add_patch(rect)
# show the plot
pyplot.show()

filename = 'test1.jpg'
# load image from file
pixels = pyplot.imread(filename)
# create the detector, using default weights
detector = MTCNN()
# detect faces in the image
faces = detector.detect_faces(pixels)
# display faces on the original image
draw_image_with_boxes(filename, faces)
```

Listing 28.26: Example of face detection with the MTCNN library and a plot of the results.

Running the example plots the photograph then draws a bounding box for each of the detected faces. We can see that both faces were detected correctly.



Figure 28.7: College Students Photograph With Bounding Boxes Drawn for Each Detected Face Using MTCNN.

We can draw a circle via the `Circle` class for the eyes, nose, and mouth; for example

```
...
# draw the dots
for key, value in result['keypoints'].items():
    # create and draw dot
    dot = Circle(value, radius=2, color='red')
    ax.add_patch(dot)
```

Listing 28.27: Example of drawing the keypoint or landmark items for a detected face.

The complete example with this addition to the `draw_image_with_boxes()` function is listed below.

```
# face detection with mtcnn on a photograph
from matplotlib import pyplot
from matplotlib.patches import Rectangle
from matplotlib.patches import Circle
from mtcnn.mtcnn import MTCNN

# draw an image with detected objects
def draw_image_with_boxes(filename, result_list):
    # load the image
```

```
data = pyplot.imread(filename)
# plot the image
pyplot.imshow(data)
# get the context for drawing boxes
ax = pyplot.gca()
# plot each box
for result in result_list:
    # get coordinates
    x, y, width, height = result['box']
    # create the shape
    rect = Rectangle((x, y), width, height, fill=False, color='red')
    # draw the box
    ax.add_patch(rect)
    # draw the dots
    for _, value in result['keypoints'].items():
        # create and draw dot
        dot = Circle(value, radius=2, color='red')
        ax.add_patch(dot)
# show the plot
pyplot.show()

filename = 'test1.jpg'
# load image from file
pixels = pyplot.imread(filename)
# create the detector, using default weights
detector = MTCNN()
# detect faces in the image
faces = detector.detect_faces(pixels)
# display faces on the original image
draw_image_with_boxes(filename, faces)
```

Listing 28.28: Example of face detection with the MTCNN library and a plot of the results with face landmarks.

The example plots the photograph again with bounding boxes and facial key points. We can see that eyes, nose, and mouth are detected well on each face, although the mouth on the right face could be better detected, with the points looking a little lower than the corners of the mouth.

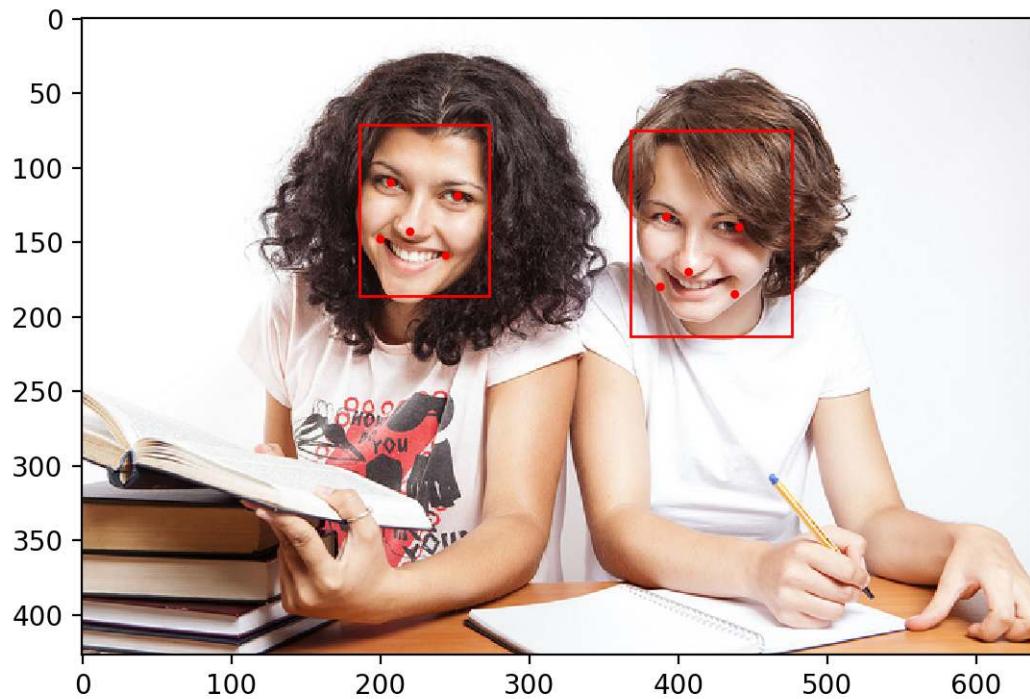


Figure 28.8: College Students Photograph With Bounding Boxes and Facial Keypoints Drawn for Each Detected Face Using MTCNN.

We can now try face detection on the swim team photograph, e.g. the image `test2.jpg`. Running the example, we can see that all thirteen faces were correctly detected and that it looks roughly like all of the facial keypoints are also correct.



Figure 28.9: Swim Team Photograph With Bounding Boxes and Facial Keypoints Drawn for Each Detected Face Using MTCNN.

We may want to extract the detected faces and pass them as input to another system. This can be achieved by extracting the pixel data directly out of the photograph; for example:

```
...
# get coordinates
x1, y1, width, height = result['box']
x2, y2 = x1 + width, y1 + height
# extract face
face = data[y1:y2, x1:x2]
```

Listing 28.29: Example of extracting pixel values for a detected face.

We can demonstrate this by extracting each face and plotting them as separate subplots. You could just as easily save them to file. The `draw_faces()` below extracts and plots each detected face in a photograph.

```
# draw each face separately
def draw_faces(filename, result_list):
    # load the image
    data = pyplot.imread(filename)
    # plot each face as a subplot
    for i in range(len(result_list)):
        # get coordinates
```

```

x1, y1, width, height = result_list[i]['box']
x2, y2 = x1 + width, y1 + height
# define subplot
pyplot.subplot(1, len(result_list), i+1)
pyplot.axis('off')
# plot face
pyplot.imshow(data[y1:y2, x1:x2])
# show the plot
pyplot.show()

```

Listing 28.30: Example of a function to extract and plot pixel values for a detected faces.

The complete example demonstrating this function for the swim team photo is listed below.

```

# extract and plot each detected face in a photograph
from matplotlib import pyplot
from mtcnn.mtcnn import MTCNN

# draw each face separately
def draw_faces(filename, result_list):
    # load the image
    data = pyplot.imread(filename)
    # plot each face as a subplot
    for i in range(len(result_list)):
        # get coordinates
        x1, y1, width, height = result_list[i]['box']
        x2, y2 = x1 + width, y1 + height
        # define subplot
        pyplot.subplot(1, len(result_list), i+1)
        pyplot.axis('off')
        # plot face
        pyplot.imshow(data[y1:y2, x1:x2])
    # show the plot
    pyplot.show()

filename = 'test2.jpg'
# load image from file
pixels = pyplot.imread(filename)
# create the detector, using default weights
detector = MTCNN()
# detect faces in the image
faces = detector.detect_faces(pixels)
# display faces on the original image
draw_faces(filename, faces)

```

Listing 28.31: Example of extracting and plotting faces with the MTCNN library.

Running the example creates a plot that shows each separate face detected in the photograph of the swim team.



Figure 28.10: Plot of Each Separate Face Detected in a Photograph of a Swim Team.

28.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Your Own Image.** Perform face detection using your own photographs.
- **Application Ideas.** Brainstorm 3 application ideas for using face detection models such as MTCNN.
- **Other Classical Models.** Experiment with other pre-trained face detection (or similar) models provided in OpenCV.

If you explore any of these extensions, I'd love to know.

28.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

28.7.1 Books

- Chapter 11 Face Detection, *Handbook of Face Recognition*, Second Edition, 2011.
<https://amzn.to/2EuR80o>

28.7.2 Papers

- *Face Detection: A Survey*, 2001.
<https://www.sciencedirect.com/science/article/pii/S107731420190921X>
- *Rapid Object Detection using a Boosted Cascade of Simple Features*, 2001.
<https://ieeexplore.ieee.org/document/990517>
- *Multi-view Face Detection Using Deep Convolutional Neural Networks*, 2015.
<https://arxiv.org/abs/1502.02766>
- *Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks*, 2016.
<https://arxiv.org/abs/1604.02878>

28.7.3 API

- OpenCV Homepage.
<https://opencv.org/>
- OpenCV GitHub Project.
<https://github.com/opencv/opencv>
- Face Detection using Haar Cascades, OpenCV.
https://docs.opencv.org/3.4.3/d7/d8b/tutorial_py_face_detection.html
- Cascade Classifier Training, OpenCV.
https://docs.opencv.org/3.4.3/dc/d88/tutorial_traincascade.html
- Cascade Classifier, OpenCV.
https://docs.opencv.org/3.4.3/db/d28/tutorial_cascade_classifier.html
- Official MTCNN Project.
https://github.com/kpzheng93/MTCNN_face_detection_alignment
- Python MTCNN Project.
<https://github.com/ipazc/mtcnn>
- `matplotlib.patches.Rectangle` API.
https://matplotlib.org/api/_as_gen/matplotlib.patches.Rectangle.html
- `matplotlib.patches.Circle` API.
https://matplotlib.org/api/_as_gen/matplotlib.patches.Circle.html

28.7.4 Articles

- Face detection, Wikipedia.
https://en.wikipedia.org/wiki/Face_detection
- Cascading classifiers, Wikipedia.
https://en.wikipedia.org/wiki/Cascading_classifiers

28.8 Summary

In this tutorial, you discovered how to perform face detection in Python using classical and deep learning models. Specifically, you learned:

- Face detection is a computer vision problem for identifying and localizing faces in images.
- Face detection can be performed using the classical feature-based cascade classifier using the OpenCV library.
- State-of-the-art face detection can be achieved using a Multi-task Cascade CNN via the MTCNN library.

28.8.1 Next

In the next section, you will discover how to perform face identification and face verification using the VGGFace2 deep learning model.

Chapter 29

How to Perform Face Identification and Verification with VGGFace2

Face recognition is the computer vision task of identifying and verifying a person based on a photograph of their face. Recently, deep learning convolutional neural networks have surpassed classical methods and are achieving state-of-the-art results on standard face recognition datasets. One example of a state-of-the-art model is the VGGFace and VGGFace2 model developed by researchers at the Visual Geometry Group at Oxford.

Although the model can be challenging to implement and resource intensive to train, it can be easily used in standard deep learning libraries such as Keras through the use of freely available pre-trained models and third-party open source libraries. In this tutorial, you will discover how to develop face recognition systems for face identification and verification using the VGGFace2 deep learning model. After completing this tutorial, you will know:

- About the VGGFace and VGGFace2 models for face recognition and how to install the Keras VGGFace library to make use of these models in Python with Keras.
- How to develop a face identification system to predict the name of celebrities in given photographs.
- How to develop a face verification system to confirm the identity of a person given a photograph of their face.

Let's get started.

29.1 Tutorial Overview

This tutorial is divided into six parts; they are:

1. Face Recognition
2. VGGFace and VGGFace2 Models
3. How to Install the keras-vggface Library
4. How to Detect Faces for Face Recognition

5. How to Perform Face Identification With VGGFace2
6. How to Perform Face Verification With VGGFace2

29.2 Face Recognition

Face recognition is the general task of identifying and verifying people from photographs of their face (introduced in Chapter 27). The 2011 book on face recognition titled *Handbook of Face Recognition* describes two main modes for face recognition, as:

- **Face Verification.** A one-to-one mapping of a given face against a known identity (e.g. is this the person?).
- **Face Identification.** A one-to-many mapping for a given face against a database of known faces (e.g. who is this person?).

A face recognition system is expected to identify faces present in images and videos automatically. It can operate in either or both of two modes: (1) face verification (or authentication), and (2) face identification (or recognition).

— Page 1, *Handbook of Face Recognition*. 2011.

We will explore both of these face recognition tasks in this tutorial.

29.3 VGGFace and VGGFace2 Models

The VGGFace refers to a series of models developed for face recognition and demonstrated on benchmark computer vision datasets by members of the Visual Geometry Group (VGG) at the University of Oxford. There are two main VGG models for face recognition at the time of writing; they are VGGFace and VGGFace2. Let's take a closer look at each in turn.

29.3.1 VGGFace Model

The VGGFace model, was described by Omkar Parkhi in the 2015 paper titled *Deep Face Recognition*. A contribution of the paper was a description of how to develop a very large training dataset, required to train modern-convolutional-neural-network-based face recognition systems, to compete with the large datasets used to train models at Facebook and Google.

... [we] propose a procedure to create a reasonably large face dataset whilst requiring only a limited amount of person-power for annotation. To this end we propose a method for collecting face data using knowledge sources available on the web (Section 3). We employ this procedure to build a dataset with over two million faces, and will make this freely available to the research community.

— *Deep Face Recognition*, 2015.

This dataset is then used as the basis for developing deep CNNs for face recognition tasks such as face identification and verification. Specifically, models are trained on the very large dataset, then evaluated on benchmark face recognition datasets, demonstrating that the model is effective at generating generalized features from faces.

They describe the process of training a face classifier first that uses a softmax activation function in the output layer to classify faces as people. This layer is then removed so that the output of the network is a vector feature representation of the face, called a face embedding. The model is then further trained, via fine-tuning, in order that the Euclidean distance between vectors generated for the same identity are made smaller and the vectors generated for different identities is made larger. This is achieved using a triplet loss function.

Triplet-loss training aims at learning score vectors that perform well in the final application, i.e. identity verification by comparing face descriptors in Euclidean space. [...] A triplet (a, p, n) contains an anchor face image as well as a positive $p \neq a$ and negative n examples of the anchor's identity. The projection W' is learned on target datasets

— *Deep Face Recognition*, 2015.

A deep convolutional neural network architecture is used in the VGG style, with blocks of convolutional layers with small kernels and ReLU activations followed by max pooling layers, and the use of fully connected layers in the classifier end of the network.

29.3.2 VGGFace2 Model

Qiong Cao, et al. from the VGG describe a follow-up work in their 2017 paper titled *VGGFace2: A Dataset For Recognizing Faces Across Pose And Age*. They describe VGGFace2 as a much larger dataset that they have collected for the intent of training and evaluating more effective face recognition models.

In this paper, we introduce a new large-scale face dataset named VGGFace2. The dataset contains 3.31 million images of 9131 subjects, with an average of 362.6 images for each subject. Images are downloaded from Google Image Search and have large variations in pose, age, illumination, ethnicity and profession (e.g. actors, athletes, politicians).

— *VGGFace2: A Dataset For Recognizing Faces Across Pose And Age*, 2017.

The paper focuses on how this dataset was collected, curated, and how images were prepared prior to modeling. Nevertheless, VGGFace2 has become the name to refer to the pre-trained models that have provided for face recognition, trained on this dataset. Models are trained on the dataset, specifically a ResNet-50 and a SqueezeNet-ResNet-50 model (called SE-ResNet-50 or SENet), and it is variations of these models that have been made available by the authors, along with the associated code. The models are evaluated on standard face recognition datasets, demonstrating then state-of-the-art performance.

... we demonstrate that deep models (ResNet-50 and SENet) trained on VGGFace2, achieve state-of-the-art performance on [...] benchmarks.

— *VGGFace2: A Dataset For Recognizing Faces Across Pose And Age*, 2017.

A face embedding is predicted by a given model as a 2,048 length vector. The length of the vector is then normalized, e.g. to a length of 1 or unit norm using the L^2 vector norm (Euclidean distance from the origin). This is referred to as the *face descriptor*. The distance between face descriptors (or groups of face descriptors called a *subject template*) is calculated using the Cosine similarity.

The face descriptor is extracted from the layer adjacent to the classifier layer. This leads to a 2048 dimensional descriptor, which is then L^2 normalized

— *VGGFace2: A Dataset For Recognizing Faces Across Pose And Age*, 2017.

29.4 How to Install the keras-vggface Library

The authors of VGGFace2 provide the source code for their models, as well as pre-trained models that can be downloaded with standard deep learning frameworks such as Caffe and PyTorch, although there are no examples for TensorFlow or Keras. We could convert the provided models to TensorFlow or Keras format and develop a model definition in order to load and use these pre-trained models. Thankfully, this work has already been done and can be used directly by third-party projects and libraries. Perhaps the best-of-breed third-party library for using the VGGFace2 (and VGGFace) models in Keras is the `keras-vggface` project and library by Refik Can Malli¹. This library can be installed via pip; for example:

```
sudo pip install git+https://github.com/rcmalli/keras-vggface.git
```

Listing 29.1: Example of installing keras-vggface via pip.

After successful installation, you should then see a message like the following:

```
Successfully installed keras-vggface-0.6
```

Listing 29.2: Example output from installing the keras-vggface library.

You can confirm that the library was installed correctly by querying the installed package:

```
pip show keras-vggface
```

Listing 29.3: Example output confirming the library was installed with pip.

This will summarize the details of the package; for example:

```
Name: keras-vggface
Version: 0.6
Summary: VGGFace implementation with Keras framework
Home-page: https://github.com/rcmalli/keras-vggface
Author: Refik Can MALLI
Author-email: mallir@itu.edu.tr
License: MIT
Location: ...
Requires: numpy, scipy, h5py, pillow, keras, six, pyyaml
Required-by:
```

Listing 29.4: Example output from confirming the library was installed with pip.

¹Clone of the project is available here: <https://github.com/jbrownlee/keras-vggface>

You can also confirm that the library loads correctly by loading it in a script and printing the current version; for example:

```
# check version of keras_vggface
import keras_vggface
# print version
print(keras_vggface.__version__)
```

Listing 29.5: Example of confirming the version of keras-vggface.

Running the example will load the library and print the current version.

```
0.6
```

Listing 29.6: Example output confirming the version of keras-vggface.

29.5 How to Detect Faces for Face Recognition

Before we can perform face recognition, we need to detect faces. Face detection is the process of automatically locating faces in a photograph and localizing them by drawing a bounding box around their extent. In this tutorial, we will also use the Multi-Task Cascaded Convolutional Neural Network, or MTCNN, for face detection, e.g. finding and extracting faces from photos. This is a state-of-the-art deep learning model for face detection, described in the 2016 paper titled *Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks*. **Note**, the installation of the face detection library was covered in Chapter 28 and is summarized again here for completeness.

We will use the implementation provided by Ivan de Paz Centeno in the ipazc/mtcnn project. This can also be installed via pip as follows:

```
sudo pip install mtcnn
```

Listing 29.7: Example of installing mtcnn via pip.

We can confirm that the library was installed correctly by importing the library and printing the version; for example.

```
# confirm mtcnn was installed correctly
import mtcnn
# print version
print(mtcnn.__version__)
```

Listing 29.8: Example of confirming the version of mtcnn.

Running the example prints the current version of the library.

```
0.1.0
```

Listing 29.9: Example output confirming the version of mtcnn.

We can use the mtcnn library to create a face detector and extract faces for our use with the VGGFace face detector models in subsequent sections.

The first step is to load an image as a NumPy array, which we can achieve using the Matplotlib `imread()` function.

```
...
# load image from file
pixels = pyplot.imread(filename)
```

Listing 29.10: Example of loading image pixels from file.

Next, we can create an MTCNN face detector class and use it to detect all faces in the loaded photograph.

```
...
# create the detector, using default weights
detector = MTCNN()
# detect faces in the image
results = detector.detect_faces(pixels)
```

Listing 29.11: Example of creating an MTCNN model and performing face detection

The result is a list of bounding boxes, where each bounding box defines a lower-left-corner of the bounding box, as well as the width and height. If we assume there is only one face in the photo for our experiments, we can determine the pixel coordinates of the bounding box as follows.

```
...
# extract the bounding box from the first face
x1, y1, width, height = results[0]['box']
x2, y2 = x1 + width, y1 + height
```

Listing 29.12: Example of interpreting the bounding box of a detected face.

We can use these coordinates to extract the face.

```
...
# extract the face
face = pixels[y1:y2, x1:x2]
```

Listing 29.13: Example of extracting the pixels of the detected face.

We can then use the PIL library to resize this small image of the face to the required size; specifically, the model expects square input faces with the shape 224×224 .

```
...
# resize pixels to the model size
image = Image.fromarray(face)
image = image.resize((224, 224))
face_array = asarray(image)
```

Listing 29.14: Example of resizing the extracted face.

Tying all of this together, the function `extract_face()` will load a photograph from the loaded filename and return the extracted face. It assumes that the photo contains one face and will return the first face detected.

```
# extract a single face from a given photograph
def extract_face(filename, required_size=(224, 224)):
    # load image from file
    pixels = pyplot.imread(filename)
    # create the detector, using default weights
    detector = MTCNN()
```

```

# detect faces in the image
results = detector.detect_faces(pixels)
# extract the bounding box from the first face
x1, y1, width, height = results[0]['box']
x2, y2 = x1 + width, y1 + height
# extract the face
face = pixels[y1:y2, x1:x2]
# resize pixels to the model size
image = Image.fromarray(face)
image = image.resize(required_size)
face_array = asarray(image)
return face_array

```

Listing 29.15: Example of a function for extracting a face from a photograph.

We can test this function with a photograph. Download a photograph of Sharon Stone taken in 2013 from Wikipedia released under a permissive license².



Figure 29.1: Photograph of Sharon, From Wikipedia (`sharon_stone1.jpg`).

Download the photograph and place it in your current working directory with the filename `sharon_stone1.jpg`.

- Download Photograph of Sharon Stone (`sharon_stone1.jpg`).³

The complete example of loading the photograph of Sharon Stone, extracting the face, and plotting the result is listed below.

```

# example of face detection with mtcnn
from matplotlib import pyplot
from PIL import Image

```

²https://en.wikipedia.org/wiki/File:Sharon_Stone_Cannes_2013_2.jpg

³https://machinelearningmastery.com/wp-content/uploads/2019/03/sharon_stone1.jpg

```

from numpy import asarray
from mtcnn.mtcnn import MTCNN

# extract a single face from a given photograph
def extract_face(filename, required_size=(224, 224)):
    # load image from file
    pixels = pyplot.imread(filename)
    # create the detector, using default weights
    detector = MTCNN()
    # detect faces in the image
    results = detector.detect_faces(pixels)
    # extract the bounding box from the first face
    x1, y1, width, height = results[0]['box']
    x2, y2 = x1 + width, y1 + height
    # extract the face
    face = pixels[y1:y2, x1:x2]
    # resize pixels to the model size
    image = Image.fromarray(face)
    image = image.resize(required_size)
    face_array = asarray(image)
    return face_array

# load the photo and extract the face
pixels = extract_face('sharon_stone1.jpg')
# plot the extracted face
pyplot.imshow(pixels)
# show the plot
pyplot.show()

```

Listing 29.16: Example of face detection with a photograph of Sharon Stone.

Running the example loads the photograph, extracts the face, and plots the result. We can see that the face was correctly detected and extracted. The results suggest that we can use the developed `extract_face()` function as the basis for examples with the VGGFace face recognition model in subsequent sections.

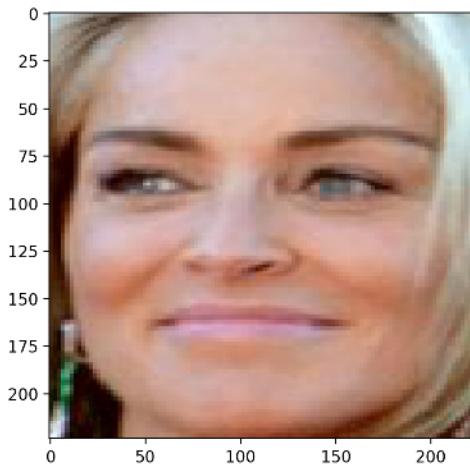


Figure 29.2: Face Detected From a Photograph of Sharon Stone Using an MTCNN Model.

29.6 How to Perform Face Identification With VGGFace2

In this section, we will use the VGGFace2 model to perform face recognition with photographs of celebrities from Wikipedia. A VGGFace model can be created using the `VGGFace()` constructor and specifying the type of model to create via the `model` argument.

```
...
model = VGGFace(model='...')
```

Listing 29.17: Example of creating a VGGFace model.

The keras-vggface library provides three pre-trained VGGModels, a VGGFace1 model via `model='vgg16'` (the default), and two VGGFace2 models '`resnet50`' and '`senet50`'. The example below creates a '`resnet50`' VGGFace2 model and summarizes the shape of the inputs and outputs.

```
# example of creating a face embedding
from keras_vggface.vggface import VGGFace
# create a vggface2 model
model = VGGFace(model='resnet50')
# summarize input and output shape
print('Inputs: %s' % model.inputs)
print('Outputs: %s' % model.outputs)
```

Listing 29.18: Example of defining and loading a VGGFace model.

The first time that a model is created, the library will download the model weights and save them in the `./keras/models/vggface/` directory in your home directory. The size of the weights for the `resnet50` model is about 158 megabytes, so the download may take a few minutes depending on the speed of your internet connection. Running the example prints the shape of the input and output tensors of the model. We can see that the model expects input color images of faces with the shape of 244×244 and the output will be a class prediction of 8,631 people. This makes sense given that the pre-trained models were trained on 8,631 identities in the MS-Celeb-1M dataset (listed in this CSV file⁴).

```
Inputs: [<tf.Tensor 'input_1:0' shape=(?, 224, 224, 3) dtype=float32>]
Outputs: [<tf.Tensor 'classifier/Softmax:0' shape=(?, 8631) dtype=float32>]
```

Listing 29.19: Example output from defining and loading a VGGFace model.

This Keras model can be used directly to predict the probability of a given face belonging to one or more of more than eight thousand known celebrities; for example:

```
...
# perform prediction
yhat = model.predict(samples)
```

Listing 29.20: Example of making a prediction with a VGGFace model.

Once a prediction is made, the class integers can be mapped to the names of the celebrities, and the top five names with the highest probability can be retrieved. This behavior is provided by the `decode_predictions()` function in the keras-vggface library.

⁴http://www.robots.ox.ac.uk/~vgg/data/vgg_face2/meta/identity_meta.csv

```

...
# convert prediction into names
results = decode_predictions(yhat)
# display most likely results
for result in results[0]:
    print('%s: %.3f%%' % (result[0], result[1]*100))

```

Listing 29.21: Example of decoding prediction made by a VGGFace model.

Before we can make a prediction with a face, the pixel values must be scaled in the same way that data was prepared when the VGGFace model was fit. Specifically, the pixel values must be centered on each channel using the mean from the training dataset. This can be achieved using the `preprocess_input()` function provided in the keras-vggface library and specifying the `version=2` so that the images are scaled using the mean values used to train the VGGFace2 models instead of the VGGFace1 models (the default).

```

...
# convert one face into samples
pixels = pixels.astype('float32')
samples = expand_dims(pixels, axis=0)
# prepare the face for the model, e.g. center pixels
samples = preprocess_input(samples, version=2)

```

Listing 29.22: Example of preparing pixels for making a prediction with a VGGFace model.

We can tie all of this together and predict the identity of our Shannon Stone photograph downloaded in the previous section, specifically `sharon_stone1.jpg`. The complete example is listed below.

```

# Example of face detection with a vggface2 model (Sharon Stone)
from numpy import expand_dims
from matplotlib import pyplot
from PIL import Image
from numpy import asarray
from mtcnn.mtcnn import MTCNN
from keras_vggface.vggface import VGGFace
from keras_vggface.utils import preprocess_input
from keras_vggface.utils import decode_predictions

# extract a single face from a given photograph
def extract_face(filename, required_size=(224, 224)):
    # load image from file
    pixels = pyplot.imread(filename)
    # create the detector, using default weights
    detector = MTCNN()
    # detect faces in the image
    results = detector.detect_faces(pixels)
    # extract the bounding box from the first face
    x1, y1, width, height = results[0]['box']
    x2, y2 = x1 + width, y1 + height
    # extract the face
    face = pixels[y1:y2, x1:x2]
    # resize pixels to the model size
    image = Image.fromarray(face)
    image = image.resize(required_size)

```

```
face_array = asarray(image)
return face_array

# load the photo and extract the face
pixels = extract_face('sharon_stone1.jpg')
# convert one face into samples
pixels = pixels.astype('float32')
samples = expand_dims(pixels, axis=0)
# prepare the face for the model, e.g. center pixels
samples = preprocess_input(samples, version=2)
# create a vggface model
model = VGGFace(model='resnet50')
# perform prediction
yhat = model.predict(samples)
# convert prediction into names
results = decode_predictions(yhat)
# display most likely results
for result in results[0]:
    print('%s: %.3f%%' % (result[0][3:-1], result[1]*100))
```

Listing 29.23: Example of face identification of Sharon Stone with a VGGFace model.

Running the example loads the photograph, extracts the single face that we know was present, and then predicts the identity for the face. The top five highest probability names are then displayed. We can see that the model correctly identifies the face as belonging to Sharon Stone with a likelihood of 99.642%.

```
Sharon_Stone: 99.642%
Noelle_Reno: 0.085%
Elisabeth_R\xc3\xb6hm: 0.033%
Anita_Lipnicka: 0.026%
Tina_Maze: 0.019%
```

Listing 29.24: Example output from face identification of Sharon Stone with a VGGFace model.

We can test the model with another celebrity, in this case, a male, Channing Tatum. A photograph of Channing Tatum taken in 2017 is available on Wikipedia under a permissive license⁵.

⁵https://en.wikipedia.org/wiki/File:Channing_Tatum_by_Gage_Skidmore_3.jpg



Figure 29.3: Photograph of Channing Tatum, From Wikipedia (`channing_tatum.jpg`).

Download the photograph and save it in your current working directory with the filename `channing_tatum.jpg`.

- Download Photograph of Channing Tatum (`channing_tatum.jpg`).⁶

Change the code to load the photograph of Channing Tatum instead; for example:

```
...
pixels = extract_face('channing_tatum.jpg')
```

Listing 29.25: Example of extracting the face of Channing Tatum.

Running the example with the new photograph, we can see that the model correctly identifies the face as belonging to Channing Tatum with a likelihood of 94.432%.

```
Channing_Tatum: 93.560%
Les_Miles: 0.163%
Eoghan_Quigg: 0.161%
Nico_Rosberg: 0.159%
Ibrahim_Afellay: 0.068%
```

Listing 29.26: Example output from face identification of Channing Tatum with a VGGFace model.

You might like to try this example with other photographs of celebrities taken from Wikipedia. Try a diverse set of genders, races, and ages. You will discover that the model is not perfect, but for those celebrities that it does know well, it can be effective. You might like to try other versions of the model, such as ‘vgg16’ and ‘senet50’, then compare results. For example, I found that with a photograph of Oscar Isaac⁷, that the ‘vgg16’ is effective, but the VGGFace2 models are not. The model could be used to identify new faces. One approach would be to re-train the model, perhaps just the classifier part of the model, with a new face dataset.

⁶https://machinelearningmastery.com/wp-content/uploads/2019/03/channing_tatum.jpg

⁷https://en.wikipedia.org/wiki/Oscar_Isaac

29.7 How to Perform Face Verification With VGGFace2

A VGGFace2 model can be used for face verification. This involves calculating a face embedding for a new given face and comparing the embedding to the embedding for the single example of the face known to the system. A face embedding is a vector that represents the features extracted from the face. This can then be compared with the vectors generated for other faces. For example, another vector that is close (by some measure) may be the same person, whereas another vector that is far (by some measure) may be a different person.

Typical measures such as Euclidean distance and Cosine distance are calculated between two embeddings and faces are said to match or verify if the distance is below a predefined threshold, often tuned for a specific dataset or application. First, we can load the VGGFace model without the classifier by setting the `include_top` argument to `False`, specifying the shape of the output via the `input_shape` and setting `pooling` to `avg` so that the filter maps at the output end of the model are reduced to a vector using global average pooling.

```
...
# create a vggface model
model = VGGFace(model='resnet50', include_top=False, input_shape=(224, 224, 3),
                 pooling='avg')
```

Listing 29.27: Example of defining and loading the VGGFace model to predict a face embedding.

This model can then be used to make a prediction, which will return a face embedding for one or more faces provided as input.

```
...
# perform prediction
yhat = model.predict(samples)
```

Listing 29.28: Example of predicting a face embedding with a VGGFace model.

We can define a new function that, given a list of filenames for photos containing a face, will extract one face from each photo via the `extract_face()` function developed in a prior section, pre-processing is required for inputs to the VGGFace2 model and can be achieved by calling `preprocess_input()`, then predict a face embedding for each. The `get_embeddings()` function below implements this, returning an array containing an embedding for one face for each provided photograph filename.

```
# extract faces and calculate face embeddings for a list of photo files
def get_embeddings(filenames):
    # extract faces
    faces = [extract_face(f) for f in filenames]
    # convert into an array of samples
    samples = asarray(faces, 'float32')
    # prepare the face for the model, e.g. center pixels
    samples = preprocess_input(samples, version=2)
    # create a vggface model
    model = VGGFace(model='resnet50', include_top=False, input_shape=(224, 224, 3),
                    pooling='avg')
    # perform prediction
    yhat = model.predict(samples)
    return yhat
```

Listing 29.29: Example of a function for predicting face embeddings for a list of photographs.

We can take our photograph of Sharon Stone used previously (e.g. `sharon_stone1.jpg`) as our definition of the identity of Sharon Stone by calculating and storing the face embedding for the face in that photograph. We can then calculate embeddings for faces in other photographs of Sharon Stone and test whether we can effectively verify her identity. We can also use faces from photographs of other people to confirm that they are not verified as Sharon Stone.

Verification can be performed by calculating the Cosine distance between the embedding for the known identity and the embeddings of candidate faces. This can be achieved using the `cosine()` SciPy function. The maximum distance between two embeddings is a score of 1.0, whereas the minimum distance is 0.0. A common cut-off value used for face identity is between 0.4 and 0.6, such as 0.5, although this should be tuned for an application. The `is_match()` function below implements this, calculating the distance between two embeddings and interpreting the result.

```
# determine if a candidate face is a match for a known face
def is_match(known_embedding, candidate_embedding, thresh=0.5):
    # calculate distance between embeddings
    score = cosine(known_embedding, candidate_embedding)
    if score <= thresh:
        print('>face is a Match (%.3f <= %.3f)' % (score, thresh))
    else:
        print('>face is NOT a Match (%.3f > %.3f)' % (score, thresh))
```

Listing 29.30: Example of a function for comparing face embeddings.

We can test out some positive examples by downloading more photos of Sharon Stone from Wikipedia. Specifically, a photograph taken in 2002⁸ (download and save as `sharon_stone2.jpg`), and a photograph taken in 2017⁹ (download and save as `sharon_stone3.jpg`).

- Download Sharon Stone Photograph 2 (`sharon_stone2.jpg`).¹⁰
- Download Sharon Stone Photograph 3 (`sharon_stone3.jpg`).¹¹

We will test these two positive cases and the Channing Tatum photo from the previous section as a negative example. The complete code example of face verification is listed below.

```
# face verification with the VGGFace2 model
from matplotlib import pyplot
from PIL import Image
from numpy import asarray
from scipy.spatial.distance import cosine
from mtcnn.mtcnn import MTCNN
from keras_vggface.vggface import VGGFace
from keras_vggface.utils import preprocess_input

# extract a single face from a given photograph
def extract_face(filename, required_size=(224, 224)):
    # load image from file
    pixels = pyplot.imread(filename)
    # create the detector, using default weights
```

⁸https://en.wikipedia.org/wiki/File:Sharon_Stone..jpg

⁹https://en.wikipedia.org/wiki/File:Sharon_Stone_by_Gage_Skidmore_3.jpg

¹⁰https://machinelearningmastery.com/wp-content/uploads/2019/03/sharon_stone3.jpg

¹¹https://machinelearningmastery.com/wp-content/uploads/2019/03/sharon_stone3.jpg

```

detector = MTCNN()
# detect faces in the image
results = detector.detect_faces(pixels)
# extract the bounding box from the first face
x1, y1, width, height = results[0]['box']
x2, y2 = x1 + width, y1 + height
# extract the face
face = pixels[y1:y2, x1:x2]
# resize pixels to the model size
image = Image.fromarray(face)
image = image.resize(required_size)
face_array = asarray(image)
return face_array

# extract faces and calculate face embeddings for a list of photo files
def get_embeddings(filenames):
    # extract faces
    faces = [extract_face(f) for f in filenames]
    # convert into an array of samples
    samples = asarray(faces, 'float32')
    # prepare the face for the model, e.g. center pixels
    samples = preprocess_input(samples, version=2)
    # create a vggface model
    model = VGGFace(model='resnet50', include_top=False, input_shape=(224, 224, 3),
                    pooling='avg')
    # perform prediction
    yhat = model.predict(samples)
    return yhat

# determine if a candidate face is a match for a known face
def is_match(known_embedding, candidate_embedding, thresh=0.5):
    # calculate distance between embeddings
    score = cosine(known_embedding, candidate_embedding)
    if score <= thresh:
        print('>face is a Match (%.3f <= %.3f)' % (score, thresh))
    else:
        print('>face is NOT a Match (%.3f > %.3f)' % (score, thresh))

# define filenames
filenames = ['sharon_stone1.jpg', 'sharon_stone2.jpg', 'sharon_stone3.jpg',
             'channing_tatum.jpg']
# get embeddings file filenames
embeddings = get_embeddings(filenames)
# define sharon stone
sharon_id = embeddings[0]
# verify known photos of sharon
print('Positive Tests')
is_match(embeddings[0], embeddings[1])
is_match(embeddings[0], embeddings[2])
# verify known photos of other people
print('Negative Tests')
is_match(embeddings[0], embeddings[3])

```

Listing 29.31: Example of face verification with a VGGFace model.

The first photo is taken as the template for Sharon Stone and the remaining photos in the

list are positive and negative photos to test for verification. Running the example, we can see that the system correctly verified the two positive cases given photos of Sharon Stone both earlier and later in time. We can also see that the photo of Channing Tatum is correctly not verified as Sharon Stone. It would be an interesting extension to explore the verification of other negative photos, such as photos of other female celebrities.

```
Positive Tests
>face is a Match (0.418 <= 0.500)
>face is a Match (0.295 <= 0.500)
Negative Tests
>face is NOT a Match (0.709 > 0.500)
```

Listing 29.32: Example output from face verification with a VGGFace model.

29.8 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Other Faces.** Update the example to use other celebrity faces used in the training dataset for the model.
- **Failure Cases.** Find some photos of celebrities known to the model that fail to be identified correctly and see if you can improve the identification with image pre-processing.
- **Embeddings.** Generate embeddings for new faces not used to train the model and determine if they can be used for a basic face validation system.

If you explore any of these extensions, I'd love to know.

29.9 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

29.9.1 Books

- Handbook of Face Recognition, Second Edition, 2011.
<https://amzn.to/2EuR80o>

29.9.2 Papers

- *Deep Face Recognition*, 2015.
<http://www.robots.ox.ac.uk/~vgg/publications/2015/Parkhi15/parkhi15.pdf>
- *VGGFace2: A Dataset For Recognizing Faces Across Pose And Age*, 2017.
<https://arxiv.org/abs/1710.08092>
- *Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks*, 2016.
<https://arxiv.org/abs/1604.02878>

29.9.3 API

- Visual Geometry Group (VGG) Homepage.
<http://www.robots.ox.ac.uk/~vgg/>
- VGGFace Homepage.
http://www.robots.ox.ac.uk/~vgg/software/vgg_face/
- VGGFace2 Homepage.
https://www.robots.ox.ac.uk/~vgg/data/vgg_face2/
- Official VGGFace2 Project, GitHub.
https://github.com/ox-vgg/vgg_face2
- keras-vggface Project, GitHub.
<https://github.com/rcmalli/keras-vggface>
- MS-Celeb-1M Dataset Homepage.
<https://goo.gl/6aveja>
- `scipy.spatial.distance.cosine` API.
<https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.cosine.html>

29.10 Summary

In this tutorial, you discovered how to develop face recognition systems for face identification and verification using the VGGFace2 deep learning model. Specifically, you learned:

- About the VGGFace and VGGFace2 models for face recognition and how to install the Keras VGGFace library to make use of these models in Python with Keras.
- How to develop a face identification system to predict the name of celebrities in given photographs.
- How to develop a face verification system to confirm the identity of a person given a photograph of their face.

29.10.1 Next

In the next section, you will discover how to perform face classification using the FaceNet deep learning model.

Chapter 30

How to Perform Face Classification with FaceNet

Face recognition is a computer vision task of identifying and verifying a person based on a photograph of their face. FaceNet is a face recognition system developed in 2015 by researchers at Google that achieved then state-of-the-art results on a range of face recognition benchmark datasets. The FaceNet system can be used broadly thanks to multiple third-party open source implementations of the model and the availability of pre-trained models.

The FaceNet system can be used to extract high-quality features from faces, called face embeddings, that can then be used to train a face identification system. In this tutorial, you will discover how to develop a face detection system using FaceNet and an SVM classifier to identify people from photographs. After completing this tutorial, you will know:

- About the FaceNet face recognition system developed by Google and open source implementations and pre-trained models.
- How to prepare a face detection dataset including first extracting faces via a face detection system and then extracting face features via face embeddings.
- How to fit, evaluate, and demonstrate an SVM model to predict identities from faces embeddings.

Let's get started.

30.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Face Recognition
2. FaceNet Model
3. How to Load a FaceNet Model in Keras
4. How to Detect Faces for Face Recognition
5. How to Develop a Face Classification System

30.2 Face Recognition

Face recognition is the general task of identifying and verifying people from photographs of their face (introduced in Chapter 27). The 2011 book on face recognition titled *Handbook of Face Recognition* describes two main modes for face recognition, as:

- **Face Verification.** A one-to-one mapping of a given face against a known identity (e.g. is this the person?).
- **Face Identification.** A one-to-many mapping for a given face against a database of known faces (e.g. who is this person?).

A face recognition system is expected to identify faces present in images and videos automatically. It can operate in either or both of two modes: (1) face verification (or authentication), and (2) face identification (or recognition).

— Page 1, *Handbook of Face Recognition*. 2011.

We will focus on the face identification task in this tutorial.

30.3 FaceNet Model

FaceNet is a face recognition system that was described by Florian Schroff, et al. at Google in their 2015 paper titled *FaceNet: A Unified Embedding for Face Recognition and Clustering*. It is a system that, given a picture of a face, will extract high-quality features from the face and predict a 128 element vector representation these features, called a face embedding.

FaceNet, that directly learns a mapping from face images to a compact Euclidean space where distances directly correspond to a measure of face similarity.

— *FaceNet: A Unified Embedding for Face Recognition and Clustering*, 2015.

The model is a deep convolutional neural network trained via a triplet loss function that encourages vectors for the same identity to become more similar (smaller distance), whereas vectors for different identities are expected to become less similar (larger distance). The focus on training a model to create embeddings directly (rather than extracting them from an intermediate layer of a model) was an important innovation in this work.

Our method uses a deep convolutional network trained to directly optimize the embedding itself, rather than an intermediate bottleneck layer as in previous deep learning approaches.

— *FaceNet: A Unified Embedding for Face Recognition and Clustering*, 2015.

These face embeddings were then used as the basis for training classifier systems on standard face recognition benchmark datasets, achieving then-state-of-the-art results.

Our system cuts the error rate in comparison to the best published result by 30% ...

— *FaceNet: A Unified Embedding for Face Recognition and Clustering*, 2015.

The paper also explores other uses of the embeddings, such as clustering to group like-faces based on their extracted features. It is a robust and effective face recognition system, and the general nature of the extracted face embeddings lends the approach to a range of applications.

30.4 How to Load a FaceNet Model in Keras

There are a number of projects that provide tools to train FaceNet-based models and make use of pre-trained models. Perhaps the most prominent is called OpenFace¹ that provides FaceNet models built and trained using the PyTorch deep learning framework. There is a port of OpenFace to Keras, called Keras OpenFace², but at the time of writing, the models appear to require Python 2, which is quite limiting.

Another prominent project is called FaceNet by David Sandberg³ that provides FaceNet models built and trained using TensorFlow. The project looks mature, although at the time of writing does not provide a library-based installation nor clean API. Usefully, David's project provides a number of high-performing pre-trained FaceNet models and there are a number of projects that port or convert these models for use in Keras. A notable example is Keras FaceNet by Hiroki Taniai⁴. His project provides a script for converting the Inception ResNet v1 model from TensorFlow to Keras. He also provides a pre-trained Keras model ready for use. We will use the pre-trained Keras FaceNet model provided by Hiroki Taniai in this tutorial. It was trained on MS-Celeb-1M dataset and expects input images to be color, to have their pixel values whitened (standardized across all three channels), and to have a square shape of 160×160 pixels. The model can be downloaded from here:

- [Download Keras FaceNet Model \(88 megabytes\)](#)⁵

Download the model file and place it in your current working directory with the filename `facenet_keras.h5`. We can load the model directly in Keras using the `load_model()` function; for example:

```
# example of loading the keras facenet model
from keras.models import load_model
# load the model
model = load_model('facenet_keras.h5')
# summarize input and output shape
print(model.inputs)
print(model.outputs)
```

Listing 30.1: Example of loading and summarizing the input and output shape for the FaceNet model.

Running the example loads the model and prints the shape of the input and output tensors. We can see that the model indeed expects square color images as input with the shape 160×160 , and will output a face embedding as a 128 element vector.

```
# [<tf.Tensor 'input_1:0' shape=(?, 160, 160, 3) dtype=float32>]
# [<tf.Tensor 'Bottleneck_BatchNorm/cond/Merge:0' shape=(?, 128) dtype=float32>]
```

Listing 30.2: Example output from loading and summarizing the input and output shape for the FaceNet model.

Now that we have a FaceNet model, we can explore using it.

¹<http://cmusatyalab.github.io/openface/>

²<https://github.com/iwantooxxoox/Keras-OpenFace>

³<https://github.com/davidsandberg/facenet>

⁴<https://github.com/nyoki-mtl/keras-facenet>

⁵<https://drive.google.com/open?id=1pwQ3H4aJ8a6yyJHZkTwtjcL4wYWQb7bn>

30.5 How to Detect Faces for Face Recognition

Before we can perform face recognition, we need to detect faces. Face detection is the process of automatically locating faces in a photograph and localizing them by drawing a bounding box around their extent. In this tutorial, we will also use the Multi-Task Cascaded Convolutional Neural Network, or MTCNN, for face detection, e.g. finding and extracting faces from photos. This is a state-of-the-art deep learning model for face detection, described in the 2016 paper titled *Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks*. Note, the installation of the face detection library was covered in Chapter 28 and is summarized again here for completeness. We will use the implementation provided by Ivan de Paz Centeno in the ipazc/mtcnn project. This can also be installed via pip as follows:

```
sudo pip install mtcnn
```

Listing 30.3: Example of installing mtcnn via pip.

We can confirm that the library was installed correctly by importing the library and printing the version; for example:

```
# confirm mtcnn was installed correctly
import mtcnn
# print version
print(mtcnn.__version__)
```

Listing 30.4: Example of confirming the version of mtcnn.

Running the example prints the current version of the library.

```
0.1.0
```

Listing 30.5: Example output confirming the version of mtcnn.

We can use the mtcnn library to create a face detector and extract faces for our use with the FaceNet face detector models in subsequent sections. The first step is to load an image as a NumPy array, which we can achieve using the PIL library and the *open()* function. We will also convert the image to RGB, just in case the image has an alpha channel or is black and white.

```
...
# load image from file
image = Image.open(filename)
# convert to RGB, if needed
image = image.convert('RGB')
# convert to array
pixels = asarray(image)
```

Listing 30.6: Example of loading image pixels from file.

Next, we can create an MTCNN face detector class and use it to detect all faces in the loaded photograph.

```
...
# create the detector, using default weights
detector = MTCNN()
# detect faces in the image
results = detector.detect_faces(pixels)
```

Listing 30.7: Example of creating an MTCNN model and performing face detection

The result is a list of bounding boxes, where each bounding box defines a lower-left-corner of the bounding box, as well as the width and height. If we assume there is only one face in the photo for our experiments, we can determine the pixel coordinates of the bounding box as follows. Sometimes the library will return a negative pixel index, and I think this is a bug. We can fix this by taking the absolute value of the coordinates.

```
...
# extract the bounding box from the first face
x1, y1, width, height = results[0]['box']
# bug fix
x1, y1 = abs(x1), abs(y1)
x2, y2 = x1 + width, y1 + height
```

Listing 30.8: Example of interpreting the bounding box of a detected face.

We can use these coordinates to extract the face.

```
...
# extract the face
face = pixels[y1:y2, x1:x2]
```

Listing 30.9: Example of extracting the pixels of the detected face.

We can then use the PIL library to resize this small image of the face to the required size; specifically, the model expects square input faces with the shape 160×160 .

```
...
# resize pixels to the model size
image = Image.fromarray(face)
image = image.resize((160, 160))
face_array = asarray(image)
```

Listing 30.10: Example of resizing the extracted face.

Tying all of this together, the function `extract_face()` will load a photograph from the loaded filename and return the extracted face. It assumes that the photo contains one face and will return the first face detected.

```
# function for face detection with mtcnn
from PIL import Image
from numpy import asarray
from mtcnn.mtcnn import MTCNN

# extract a single face from a given photograph
def extract_face(filename, required_size=(160, 160)):
    # load image from file
    image = Image.open(filename)
    # convert to RGB, if needed
    image = image.convert('RGB')
    # convert to array
    pixels = asarray(image)
    # create the detector, using default weights
    detector = MTCNN()
    # detect faces in the image
    results = detector.detect_faces(pixels)
    # extract the bounding box from the first face
    x1, y1, width, height = results[0]['box']
```

```

# bug fix
x1, y1 = abs(x1), abs(y1)
x2, y2 = x1 + width, y1 + height
# extract the face
face = pixels[y1:y2, x1:x2]
# resize pixels to the model size
image = Image.fromarray(face)
image = image.resize(required_size)
face_array = asarray(image)
return face_array

# load the photo and extract the face
pixels = extract_face('...')


```

Listing 30.11: Example of a function for extracting a face from a photograph.

We can use this function to extract faces as needed in the next section that can be provided as input to the FaceNet model.

30.6 How to Develop a Face Classification System

In this section, we will develop a face detection system to predict the identity of a given face. The model will be trained and tested using the *5 Celebrity Faces Dataset* that contains many photographs of five different celebrities. We will use an MTCNN model for face detection, the FaceNet model will be used to create a face embedding for each detected face, then we will develop a Linear Support Vector Machine (SVM) classifier model to predict the identity of a given face.

30.6.1 5 Celebrity Faces Dataset

The 5 Celebrity Faces Dataset is a small dataset that contains photographs of celebrities. It includes photos of: Ben Affleck, Elton John, Jerry Seinfeld, Madonna, and Mindy Kaling. The dataset was prepared and made available by Dan Becker and provided for free download on Kaggle. Note, a Kaggle account is required to download the dataset.

- 5 Celebrity Faces Dataset, Kaggle.⁶

Download the dataset (this may require a Kaggle login), `data.zip` (2.5 megabytes), and unzip it in your local directory with the folder name `5-celebrity-faces-dataset`. You should now have a directory with the following structure (note, there are spelling mistakes in some directory names, and they were left as-is in this example):

```

5-celebrity-faces-dataset
├── train
│   ├── ben_afflek
│   ├── elton_john
│   ├── jerry_seinfeld
│   ├── madonna
│   └── mindy_kaling
└── val

```

⁶<https://www.kaggle.com/dansbecker/5-celebrity-faces-dataset>

```

└── ben_afflek
└── elton_john
└── jerry_seinfeld
└── madonna
└── mindy_kaling

```

Listing 30.12: Example of the directory structure of the dataset.

We can see that there is a training dataset and a validation or test dataset. Looking at some of the photos in the directories, we can see that the photos provide faces with a range of orientations, lighting, and in various sizes. Importantly, each photo contains one face of the person. We will use this dataset as the basis for our classifier, trained on the `train` dataset only and classify faces in the `val` dataset. You can use this same structure to develop a classifier with your own photographs.

30.6.2 Detect Faces

The first step is to detect the face in each photograph and reduce the dataset to a series of faces only. Let's test out our face detector function defined in the previous section, specifically `extract_face()`. Looking in the `5-celebrity-faces-dataset/train/ben_afflek/` directory, we can see that there are 14 photographs of Ben Affleck in the training dataset. We can detect the face in each photograph, and create a plot with 14 faces, with two rows of seven images each. The complete example is listed below.

```

# demonstrate face detection on 5 Celebrity Faces Dataset
from os import listdir
from PIL import Image
from numpy import asarray
from matplotlib import pyplot
from mtcnn.mtcnn import MTCNN

# extract a single face from a given photograph
def extract_face(filename, required_size=(160, 160)):
    # load image from file
    image = Image.open(filename)
    # convert to RGB, if needed
    image = image.convert('RGB')
    # convert to array
    pixels = asarray(image)
    # create the detector, using default weights
    detector = MTCNN()
    # detect faces in the image
    results = detector.detect_faces(pixels)
    # extract the bounding box from the first face
    x1, y1, width, height = results[0]['box']
    # bug fix
    x1, y1 = abs(x1), abs(y1)
    x2, y2 = x1 + width, y1 + height
    # extract the face
    face = pixels[y1:y2, x1:x2]
    # resize pixels to the model size
    image = Image.fromarray(face)
    image = image.resize(required_size)
    face_array = asarray(image)

```

```
return face_array

# specify folder to plot
folder = '5-celebrity-faces-dataset/train/ben_afflek/'
i = 1
# enumerate files
for filename in listdir(folder):
    # path
    path = folder + filename
    # get face
    face = extract_face(path)
    print(i, face.shape)
    # plot
    pyplot.subplot(2, 7, i)
    pyplot.axis('off')
    pyplot.imshow(face)
    i += 1
pyplot.show()
```

Listing 30.13: Example of extracting faces of Ben Affleck.

Running the example takes a moment and reports the progress of each loaded photograph along the way and the shape of the NumPy array containing the face pixel data.

```
1 (160, 160, 3)
2 (160, 160, 3)
3 (160, 160, 3)
4 (160, 160, 3)
5 (160, 160, 3)
6 (160, 160, 3)
7 (160, 160, 3)
8 (160, 160, 3)
9 (160, 160, 3)
10 (160, 160, 3)
11 (160, 160, 3)
12 (160, 160, 3)
13 (160, 160, 3)
14 (160, 160, 3)
```

Listing 30.14: Example output from extracting faces of Ben Affleck.

A figure is created containing the faces detected in the Ben Affleck directory. We can see that each face was correctly detected and that we have a range of lighting, skin tones, and orientations in the detected faces.

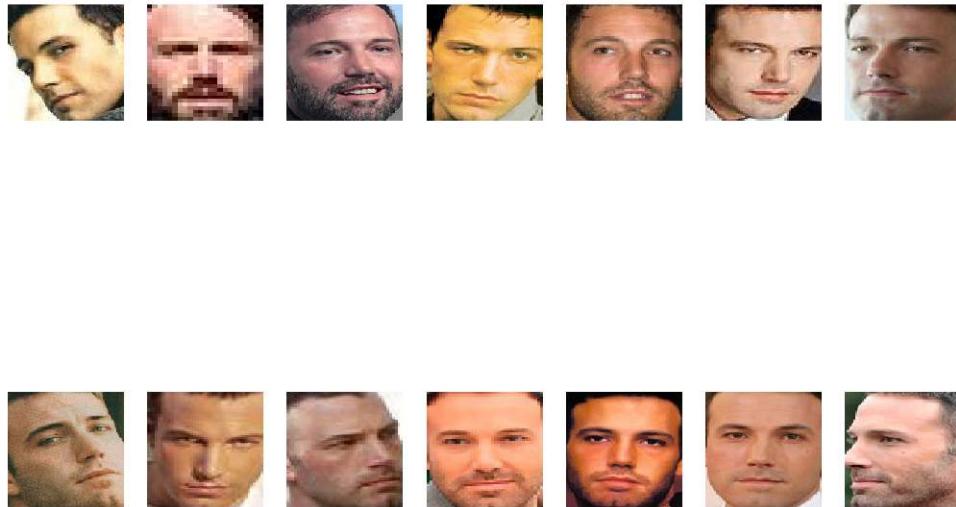


Figure 30.1: Plot of 14 Faces of Ben Affleck Detected From the Training Dataset of the 5 Celebrity Faces Dataset.

So far, so good. Next, we can extend this example to step over each subdirectory for a given dataset (e.g. `train` or `val`), extract the faces, and prepare a dataset with the name as the output label for each detected face. The `load_faces()` function below will load all of the faces into a list for a given directory, e.g. `5-celebrity-faces-dataset/train/ben_afflek/`.

```
# load images and extract faces for all images in a directory
def load_faces(directory):
    faces = list()
    # enumerate files
    for filename in listdir(directory):
        # path
        path = directory + filename
        # get face
        face = extract_face(path)
        # store
        faces.append(face)
    return faces
```

Listing 30.15: Example of a function for loading faces from a directory.

We can call the `load_faces()` function for each subdirectory in the `train` or `val` folders. Each face has one label, the name of the celebrity, which we can take from the directory name. The

`load_dataset()` function below takes a directory name such as `5-celebrity-faces-dataset/train/` and detects faces for each subdirectory (`celebrity`), assigning labels to each detected face. It returns the `X` and `y` elements of the dataset as NumPy arrays.

```
# load a dataset that contains one subdir for each class that in turn contains images
def load_dataset(directory):
    X, y = list(), list()
    # enumerate folders, on per class
    for subdir in listdir(directory):
        # path
        path = directory + subdir + '/'
        # skip any files that might be in the dir
        if not isdir(path):
            continue
        # load all faces in the subdirectory
        faces = load_faces(path)
        # create labels
        labels = [subdir for _ in range(len(faces))]
        # summarize progress
        print('>loaded %d examples for class: %s' % (len(faces), subdir))
        # store
        X.extend(faces)
        y.extend(labels)
    return asarray(X), asarray(y)
```

Listing 30.16: Example of a function for loading all of the faces in a directory.

We can then call this function for the `train` and `val` folders to load all of the data, then save the results in a single compressed NumPy array file via the `savez_compressed()` function.

```
...
# load train dataset
trainX, trainy = load_dataset('5-celebrity-faces-dataset/train/')
print(trainX.shape, trainy.shape)
# load test dataset
testX, testy = load_dataset('5-celebrity-faces-dataset/val/')
print(testX.shape, testy.shape)
# save arrays to one file in compressed format
savez_compressed('5-celebrity-faces-dataset.npz', trainX, trainy, testX, testy)
```

Listing 30.17: Example of saving extracted faces to file.

Tying all of this together, the complete example of detecting all of the faces in the 5 Celebrity Faces Dataset is listed below.

```
# face detection for the 5 Celebrity Faces Dataset
from os import listdir
from os.path import isdir
from PIL import Image
from numpy import savez_compressed
from numpy import asarray
from mtcnn.mtcnn import MTCNN

# extract a single face from a given photograph
def extract_face(filename, required_size=(160, 160)):
    # load image from file
    image = Image.open(filename)
```

```
# convert to RGB, if needed
image = image.convert('RGB')
# convert to array
pixels = asarray(image)
# create the detector, using default weights
detector = MTCNN()
# detect faces in the image
results = detector.detect_faces(pixels)
# extract the bounding box from the first face
x1, y1, width, height = results[0]['box']
# bug fix
x1, y1 = abs(x1), abs(y1)
x2, y2 = x1 + width, y1 + height
# extract the face
face = pixels[y1:y2, x1:x2]
# resize pixels to the model size
image = Image.fromarray(face)
image = image.resize(required_size)
face_array = asarray(image)
return face_array

# load images and extract faces for all images in a directory
def load_faces(directory):
    faces = list()
    # enumerate files
    for filename in listdir(directory):
        # path
        path = directory + filename
        # get face
        face = extract_face(path)
        # store
        faces.append(face)
    return faces

# load a dataset that contains one subdir for each class that in turn contains images
def load_dataset(directory):
    X, y = list(), list()
    # enumerate folders, on per class
    for subdir in listdir(directory):
        # path
        path = directory + subdir + '/'
        # skip any files that might be in the dir
        if not isdir(path):
            continue
        # load all faces in the subdirectory
        faces = load_faces(path)
        # create labels
        labels = [subdir for _ in range(len(faces))]
        # summarize progress
        print('>loaded %d examples for class: %s' % (len(faces), subdir))
        # store
        X.extend(faces)
        y.extend(labels)
    return asarray(X), asarray(y)

# load train dataset
```

```

trainX, trainy = load_dataset('5-celebrity-faces-dataset/train/')
print(trainX.shape, trainy.shape)
# load test dataset
testX, testy = load_dataset('5-celebrity-faces-dataset/val/')
# save arrays to one file in compressed format
savez_compressed('5-celebrity-faces-dataset.npz', trainX, trainy, testX, testy)

```

Listing 30.18: Example of extracting all faces in the dataset and saving the new dataset to file.

Running the example may take a moment. First, all of the photos in the `train` dataset are loaded, then faces are extracted, resulting in 93 samples with square face input and a class label string as output. Then the `val` dataset is loaded, providing 25 samples that can be used as a test dataset. Both datasets are then saved to a compressed NumPy array file called `5-celebrity-faces-dataset.npz` that is about three megabytes and is stored in the current working directory.

```

>loaded 14 examples for class: ben_afflek
>loaded 19 examples for class: madonna
>loaded 17 examples for class: elton_john
>loaded 22 examples for class: mindy_kaling
>loaded 21 examples for class: jerry_seinfeld
(93, 160, 160, 3) (93,)
>loaded 5 examples for class: ben_afflek
>loaded 5 examples for class: madonna
>loaded 5 examples for class: elton_john
>loaded 5 examples for class: mindy_kaling
>loaded 5 examples for class: jerry_seinfeld
(25, 160, 160, 3) (25,)

```

Listing 30.19: Example output from extracting all faces in the dataset and saving the new dataset to file.

This dataset is ready to be provided to a face detection model.

30.6.3 Create Face Embeddings

The next step is to create a face embedding. A face embedding is a vector that represents the features extracted from the face. This can then be compared with the vectors generated for other faces. For example, another vector that is close (by some measure) may be the same person, whereas another vector that is far (by some measure) may be a different person. The classifier model that we want to develop will take a face embedding as input and predict the identity of the face. The FaceNet model will generate this embedding for a given image of a face.

The FaceNet model can be used as part of the classifier itself, or we can use the FaceNet model to pre-process a face to create a face embedding that can be stored and used as input to our classifier model. This latter approach is preferred as the FaceNet model is both large and slow to create a face embedding. We can, therefore, pre-compute the face embeddings for all faces in the train and test (formally `val`) sets in our 5 Celebrity Faces Dataset. First, we can load our detected faces dataset using the `load()` NumPy function.

```

...
# load the face dataset
data = load('5-celebrity-faces-dataset.npz')

```

```
trainX, trainy, testX, testy = data['arr_0'], data['arr_1'], data['arr_2'], data['arr_3']
print('Loaded: ', trainX.shape, trainy.shape, testX.shape, testy.shape)
```

Listing 30.20: Example of loading the extracted faces.

Next, we can load our FaceNet model ready for converting faces into face embeddings.

```
...
# load the facenet model
model = load_model('facenet_keras.h5')
print('Loaded Model')
```

Listing 30.21: Example of loading the FaceNet model.

We can then enumerate each face in the train and test datasets to predict an embedding. To predict an embedding, first the pixel values of the image need to be prepared to meet the expectations of the FaceNet model. This specific implementation of the FaceNet model expects that the pixel values are standardized.

```
...
# scale pixel values
face_pixels = face_pixels.astype('float32')
# standardize pixel values across channels (global)
mean, std = face_pixels.mean(), face_pixels.std()
face_pixels = (face_pixels - mean) / std
```

Listing 30.22: Example of preparing an extracted face for prediction with the FaceNet model.

In order to make a prediction for one example in Keras, we must expand the dimensions so that the face array is one sample.

```
...
# transform face into one sample
samples = expand_dims(face_pixels, axis=0)
```

Listing 30.23: Example of transforming an extracted face into a sample.

We can then use the model to make a prediction and extract the embedding vector.

```
...
# make prediction to get embedding
yhat = model.predict(samples)
# get embedding
embedding = yhat[0]
```

Listing 30.24: Example of predicting a face embedding for an extracted face.

The `get_embedding()` function defined below implements these behaviors and will return a face embedding given a single image of a face and the loaded FaceNet model.

```
# get the face embedding for one face
def get_embedding(model, face_pixels):
    # scale pixel values
    face_pixels = face_pixels.astype('float32')
    # standardize pixel values across channels (global)
    mean, std = face_pixels.mean(), face_pixels.std()
    face_pixels = (face_pixels - mean) / std
    # transform face into one sample
    samples = expand_dims(face_pixels, axis=0)
```

```
# make prediction to get embedding
yhat = model.predict(samples)
return yhat[0]
```

Listing 30.25: Example of a function for predicting a face embedding for an extracted face.

Tying all of this together, the complete example of converting each face into a face embedding in the train and test datasets is listed below.

```
# calculate a face embedding for each face in the dataset using facenet
from numpy import load
from numpy import expand_dims
from numpy import asarray
from numpy import savez_compressed
from keras.models import load_model

# get the face embedding for one face
def get_embedding(model, face_pixels):
    # scale pixel values
    face_pixels = face_pixels.astype('float32')
    # standardize pixel values across channels (global)
    mean, std = face_pixels.mean(), face_pixels.std()
    face_pixels = (face_pixels - mean) / std
    # transform face into one sample
    samples = expand_dims(face_pixels, axis=0)
    # make prediction to get embedding
    yhat = model.predict(samples)
    return yhat[0]

# load the face dataset
data = load('5-celebrity-faces-dataset.npz')
trainX, trainy, testX, testy = data['arr_0'], data['arr_1'], data['arr_2'], data['arr_3']
print('Loaded: ', trainX.shape, trainy.shape, testX.shape, testy.shape)
# load the facenet model
model = load_model('facenet_keras.h5')
print('Loaded Model')
# convert each face in the train set to an embedding
newTrainX = list()
for face_pixels in trainX:
    embedding = get_embedding(model, face_pixels)
    newTrainX.append(embedding)
newTrainX = asarray(newTrainX)
print(newTrainX.shape)
# convert each face in the test set to an embedding
newTestX = list()
for face_pixels in testX:
    embedding = get_embedding(model, face_pixels)
    newTestX.append(embedding)
newTestX = asarray(newTestX)
print(newTestX.shape)
# save arrays to one file in compressed format
savez_compressed('5-celebrity-faces-embeddings.npz', newTrainX, trainy, newTestX, testy)
```

Listing 30.26: Example of predicting face embeddings for all faces in the dataset.

Running the example reports progress along the way. We can see that the face dataset was loaded correctly and so was the model. The train dataset was then transformed into 93 face em-

beddings, each comprised of a 128 element vector. The 25 examples in the test dataset were also suitably converted to face embeddings. The resulting datasets were then saved to a compressed NumPy array that is about 50 kilobytes with the name `5-celebrity-faces-embeddings.npz` in the current working directory.

```
Loaded: (93, 160, 160, 3) (93,) (25, 160, 160, 3) (25,)
Loaded Model
(93, 128)
(25, 128)
```

Listing 30.27: Example output from predicting face embeddings for all faces in the dataset.

We are now ready to develop our face classifier system.

30.6.4 Perform Face Classification

In this section, we will develop a model to classify face embeddings as one of the known celebrities in the 5 Celebrity Faces Dataset. First, we must load the face embeddings dataset.

```
...
# load dataset
data = load('5-celebrity-faces-embeddings.npz')
trainX, trainy, testX, testy = data['arr_0'], data['arr_1'], data['arr_2'], data['arr_3']
print('Dataset: train=%d, test=%d' % (trainX.shape[0], testX.shape[0]))
```

Listing 30.28: Example of loading the face embedding dataset.

Next, the data requires some minor preparation prior to modeling. First, it is a good practice to normalize the face embedding vectors. It is a good practice because the vectors are often compared to each other using a distance metric. In this context, vector normalization means scaling the values until the length or magnitude of the vectors is 1 or unit length. This can be achieved using the `Normalizer` class in scikit-learn. It might even be more convenient to perform this step when the face embeddings are created in the previous step.

```
...
# normalize input vectors
in_encoder = Normalizer(norm='l2')
trainX = in_encoder.transform(trainX)
testX = in_encoder.transform(testX)
```

Listing 30.29: Example of normalizing face embedding vectors.

Next, the string target variables for each celebrity name need to be converted to integers. This can be achieved via the `LabelEncoder` class in scikit-learn.

```
...
# label encode targets
out_encoder = LabelEncoder()
out_encoder.fit(trainy)
trainy = out_encoder.transform(trainy)
testy = out_encoder.transform(testy)
```

Listing 30.30: Example of label encoding target values (celebrity names).

Next, we can fit a model. It is common to use a Linear Support Vector Machine (SVM) when working with normalized face embedding inputs. This is because the method is very

effective at separating the face embedding vectors. We can fit a linear SVM to the training data using the `SVC` class in scikit-learn and setting the `kernel` argument to ‘`linear`’. We may also want probabilities later when making predictions, which can be configured by setting the `probability` argument to `True`.

```
...
# fit model
model = SVC(kernel='linear')
model.fit(trainX, trainy)
```

Listing 30.31: Example of fitting an SVM model to predict identity from face embedding.

Next, we can evaluate the model. This can be achieved by using the `fit` model to make a prediction for each example in the train and test datasets and then calculating the classification accuracy.

```
...
# predict
yhat_train = model.predict(trainX)
yhat_test = model.predict(testX)
# score
score_train = accuracy_score(trainy, yhat_train)
score_test = accuracy_score(testy, yhat_test)
# summarize
print('Accuracy: train=%f, test=%f' % (score_train*100, score_test*100))
```

Listing 30.32: Example of predicting identities and evaluating predictions.

Tying all of this together, the complete example of fitting a Linear SVM on the face embeddings for the 5 Celebrity Faces Dataset is listed below.

```
# develop a classifier for the 5 Celebrity Faces Dataset
from numpy import load
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import Normalizer
from sklearn.svm import SVC
# load dataset
data = load('5-celebrity-faces-embeddings.npz')
trainX, trainy, testX, testy = data['arr_0'], data['arr_1'], data['arr_2'], data['arr_3']
print('Dataset: train=%d, test=%d' % (trainX.shape[0], testX.shape[0]))
# normalize input vectors
in_encoder = Normalizer(norm='l2')
trainX = in_encoder.transform(trainX)
testX = in_encoder.transform(testX)
# label encode targets
out_encoder = LabelEncoder()
out_encoder.fit(trainy)
trainy = out_encoder.transform(trainy)
testy = out_encoder.transform(testy)
# fit model
model = SVC(kernel='linear', probability=True)
model.fit(trainX, trainy)
# predict
yhat_train = model.predict(trainX)
yhat_test = model.predict(testX)
# score
```

```

score_train = accuracy_score(trainy, yhat_train)
score_test = accuracy_score(testy, yhat_test)
# summarize
print('Accuracy: train=% .3f, test=% .3f' % (score_train*100, score_test*100))

```

Listing 30.33: Example of face classification on the 5 celebrity dataset.

Running the example first confirms that the number of samples in the train and test datasets is as we expect. Next, the model is evaluated on the train and test dataset, showing perfect classification accuracy. This is not surprising given the size of the dataset and the power of the face detection and face recognition models used.

```

Dataset: train=93, test=25
Accuracy: train=100.000, test=100.000

```

Listing 30.34: Example output from face classification on the 5 celebrity dataset.

We can make it more interesting by plotting the original face and the prediction. First, we need to load the face dataset, specifically the faces in the test dataset. We could also load the original photos to make it even more interesting.

```

...
# load faces
data = load('5-celebrity-faces-dataset.npz')
testX_faces = data['arr_2']

```

Listing 30.35: Example of loading the extracted faces dataset.

The rest of the example is the same up until we fit the model. First, we need to select a random example from the test set, then get the embedding, face pixels, expected class prediction, and the corresponding name for the class.

```

...
# test model on a random example from the test dataset
selection = choice([i for i in range(testX.shape[0])])
random_face_pixels = testX_faces[selection]
random_face_emb = testX[selection]
random_face_class = testy[selection]
random_face_name = out_encoder.inverse_transform([random_face_class])

```

Listing 30.36: Example of randomly selecting a face for identification.

Next, we can use the face embedding as an input to make a single prediction with the fit model. We can predict both the class integer and the probability of the prediction.

```

...
# prediction for the face
samples = expand_dims(random_face_emb, axis=0)
yhat_class = model.predict(samples)
yhat_prob = model.predict_proba(samples)

```

Listing 30.37: Example of making a prediction for a randomly selected face.

We can then get the name for the predicted class integer, and the probability for this prediction.

```

...
# get name

```

```

class_index = yhat_class[0]
class_probability = yhat_prob[0,class_index] * 100
predict_names = out_encoder.inverse_transform(yhat_class)

```

Listing 30.38: Example of interpreting the prediction of an identity.

We can then print this information.

```

...
print('Predicted: %s (%.3f)' % (predict_names[0], class_probability))
print('Expected: %s' % random_face_name[0])

```

Listing 30.39: Example of summarizing the prediction of an identity.

We can also plot the face pixels along with the predicted name and probability.

```

...
# plot for fun
pyplot.imshow(random_face_pixels)
title = '%s (%.3f)' % (predict_names[0], class_probability)
pyplot.title(title)
pyplot.show()

```

Listing 30.40: Example of plotting the face and identity prediction.

Tying all of this together, the complete example for predicting the identity for a given unseen photo in the test dataset is listed below.

```

# develop a classifier for the 5 Celebrity Faces Dataset
from random import choice
from numpy import load
from numpy import expand_dims
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import Normalizer
from sklearn.svm import SVC
from matplotlib import pyplot
# load faces
data = load('5-celebrity-faces-dataset.npz')
testX_faces = data['arr_2']
# load face embeddings
data = load('5-celebrity-faces-embeddings.npz')
trainX, trainy, testX, testy = data['arr_0'], data['arr_1'], data['arr_2'], data['arr_3']
# normalize input vectors
in_encoder = Normalizer(norm='l2')
trainX = in_encoder.transform(trainX)
testX = in_encoder.transform(testX)
# label encode targets
out_encoder = LabelEncoder()
out_encoder.fit(trainy)
trainy = out_encoder.transform(trainy)
testy = out_encoder.transform(testy)
# fit model
model = SVC(kernel='linear', probability=True)
model.fit(trainX, trainy)
# test model on a random example from the test dataset
selection = choice([i for i in range(testX.shape[0])])
random_face_pixels = testX_faces[selection]
random_face_emb = testX[selection]

```

```

random_face_class = testy[selection]
random_face_name = out_encoder.inverse_transform([random_face_class])
# prediction for the face
samples = expand_dims(random_face_emb, axis=0)
yhat_class = model.predict(samples)
yhat_prob = model.predict_proba(samples)
# get name
class_index = yhat_class[0]
class_probability = yhat_prob[0,class_index] * 100
predict_names = out_encoder.inverse_transform(yhat_class)
print('Predicted: %s (%.3f)' % (predict_names[0], class_probability))
print('Expected: %s' % random_face_name[0])
# plot for fun
pyplot.imshow(random_face_pixels)
title = '%s (%.3f)' % (predict_names[0], class_probability)
pyplot.title(title)
pyplot.show()

```

Listing 30.41: Example of classifying the identity of a randomly selected face and plotting the result.

A different random example from the test dataset will be selected each time the code is run. Try running it a few times. In this case, a photo of Jerry Seinfeld is selected and correctly predicted.

```

Predicted: jerry_seinfeld (88.476)
Expected: jerry_seinfeld

```

Listing 30.42: Example output from classifying the identity of a randomly selected face.

A plot of the chosen face is also created, showing the predicted name and probability in the image title.

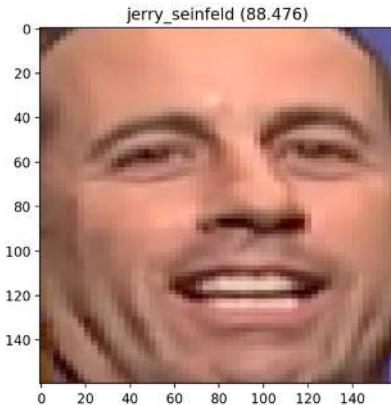


Figure 30.2: Detected Face of Jerry Seinfeld, Correctly Identified by the SVM Classifier.

30.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **New Photographs.** Find new photographs of the celebrities in the dataset and see if they can be correctly classified by the model.
- **New Celebrity.** Collect photographs of an additional celebrity to add to the training dataset, then re-fit the model.
- **Visualize.** Create a plot of the projected face embeddings using PCA or t-SNE, colored by class label and note any natural separation between classes.

If you explore any of these extensions, I'd love to know.

30.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

30.8.1 Books

- *Handbook of Face Recognition*, 2011.
<https://amzn.to/2EuR8Oo>

30.8.2 Papers

- *FaceNet: A Unified Embedding for Face Recognition and Clustering*, 2015.
<https://arxiv.org/abs/1503.03832>

30.8.3 Projects

- OpenFace PyTorch Project.
<http://cmusatyalab.github.io/openface/>
- OpenFace Keras Project, GitHub.
<https://github.com/iwantooxxoox/Keras-OpenFace>
- TensorFlow FaceNet Project, GitHub.
<https://github.com/davidsandberg/facenet>
- Keras FaceNet Project, GitHub.
<https://github.com/nyoki-mtl/keras-facenet>
- MS-Celeb 1M Dataset.
<https://goo.gl/6aveja>

30.8.4 APIs

- `sklearn.preprocessing.Normalizer` API.
<https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.Normalizer.html>
- `sklearn.preprocessing.LabelEncoder` API.
<https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html>

- `sklearn.svm.SVC` API.
<https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html>

30.9 Summary

In this tutorial, you discovered how to develop a face detection system using FaceNet and an SVM classifier to identify people from photographs. Specifically, you learned:

- About the FaceNet face recognition system developed by Google and open source implementations and pre-trained models.
- How to prepare a face detection dataset including first extracting faces via a face detection system and then extracting face features via face embeddings.
- How to fit, evaluate, and demonstrate an SVM model to predict identities from faces embeddings.

30.9.1 Next

This was the final tutorial in this part on face recognition. Next you will discover the Appendix and resources that you can use to dive deeper into the topic of deep learning for computer vision.

Part VIII

Appendix

Appendix A

Getting Help

This is just the beginning of your journey with deep learning for computer vision with Python. As you start to work on methods or expand your existing knowledge of algorithms you may need help. This chapter points out some of the best sources of help.

A.1 Computer Vision Textbooks

A textbook can be a useful resource for practitioners looking for more insight into the theory or background for a specific problem type, image processing method, or statistical technique. The list below provides a selection of some of the top computer vision textbooks:

- *Computer Vision: Algorithms and Applications*, 2010.
<https://amzn.to/2LcIt4J>
- *Computer Vision: Models, Learning, and Inference*, 2012.
<https://amzn.to/2rxrd0F>
- *Computer Vision: A Modern Approach*, 2002.
<https://amzn.to/2rv7AqJ>
- *Introductory Techniques for 3-D Computer Vision*, 1998.
<https://amzn.to/2L9C2zF>
- *Multiple View Geometry in Computer Vision*, 2004.
<https://amzn.to/2LfHLE8>

A.2 Programming Computer Vision Books

Programming books can also be useful when getting started in computer vision. Specifically, for providing more insight into using top computer vision libraries such as PIL and OpenCV. The list below provides a selection of some of the top computer vision programming books:

- *Learning OpenCV 3*, 2017.
<https://amzn.to/2EqF0Pv>

- *Programming Computer Vision with Python*, 2012.
<https://amzn.to/2QKTAAL>
- *Practical Computer Vision with SimpleCV*, 2012.
<https://amzn.to/2QnFqMY>

A.3 Official Keras Destinations

This section lists the official Keras sites that you may find helpful.

- Keras Official Blog.
<https://blog.keras.io/>
- Keras API Documentation.
<https://keras.io/>
- Keras Source Code Project.
<https://github.com/keras-team/keras>

A.4 Where to Get Help with Keras

This section lists the 9 best places I know where you can get help with Keras.

- Keras Users Google Group.
<https://groups.google.com/forum/#!forum/keras-users>
- Keras Slack Channel (you must request to join).
<https://keras-slack-autojoin.herokuapp.com/>
- Keras on Gitter.
<https://gitter.im/Keras-io/Lobby#>
- Keras tag on StackOverflow.
<https://stackoverflow.com/questions/tagged/keras>
- Keras tag on CrossValidated.
<https://stats.stackexchange.com/questions/tagged/keras>
- Keras tag on DataScience.
<https://datascience.stackexchange.com/questions/tagged/keras>
- Keras Topic on Quora.
<https://www.quora.com/topic/Keras>
- Keras GitHub Issues.
<https://github.com/keras-team/keras/issues>
- Keras on Twitter.
<https://twitter.com/hashtag/keras>

A.5 How to Ask Questions

Knowing where to get help is the first step, but you need to know how to get the most out of these resources. Below are some tips that you can use:

- Boil your question down to the simplest form. E.g. not something broad like *my model does not work* or *how does x work*.
- Search for answers before asking questions.
- Provide complete code and error messages.
- Boil your code down to the smallest possible working example that demonstrates the issue.

These are excellent resources both for posting unique questions, but also for searching through the answers to questions on related topics.

A.6 Contact the Author

You are not alone. If you ever have any questions about computer vision or this book, please contact me directly. I will do my best to help.

Jason Brownlee

Jason@MachineLearningMastery.com

Appendix B

How to Setup Python on Your Workstation

It can be difficult to install a Python machine learning environment on some platforms. Python itself must be installed first and then there are many packages to install, and it can be confusing for beginners. In this tutorial, you will discover how to setup a Python machine learning development environment using Anaconda.

After completing this tutorial, you will have a working Python environment to begin learning, practicing, and developing machine learning and deep learning software. These instructions are suitable for Windows, macOS, and Linux platforms. I will demonstrate them on macOS, so you may see some mac dialogs and file extensions.

B.1 Overview

In this tutorial, we will cover the following steps:

1. Download Anaconda
2. Install Anaconda
3. Start and Update Anaconda
4. Install Deep Learning Libraries

Note: The specific versions may differ as the software and libraries are updated frequently.

B.2 Download Anaconda

In this step, we will download the Anaconda Python package for your platform. Anaconda is a free and easy-to-use environment for scientific Python.

- 1. Visit the Anaconda homepage.
<https://www.continuum.io/>
- 2. Click Anaconda from the menu and click Download to go to the download page.
<https://www.continuum.io/downloads>

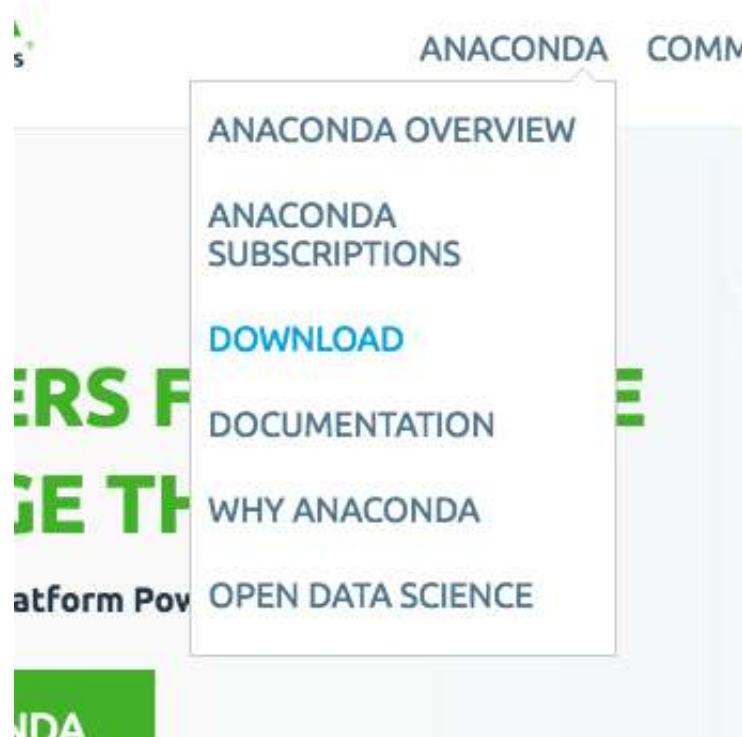


Figure B.1: Click Anaconda and Download.

- 3. Choose the download suitable for your platform (Windows, OSX, or Linux):
 - Choose Python 3.6
 - Choose the Graphical Installer

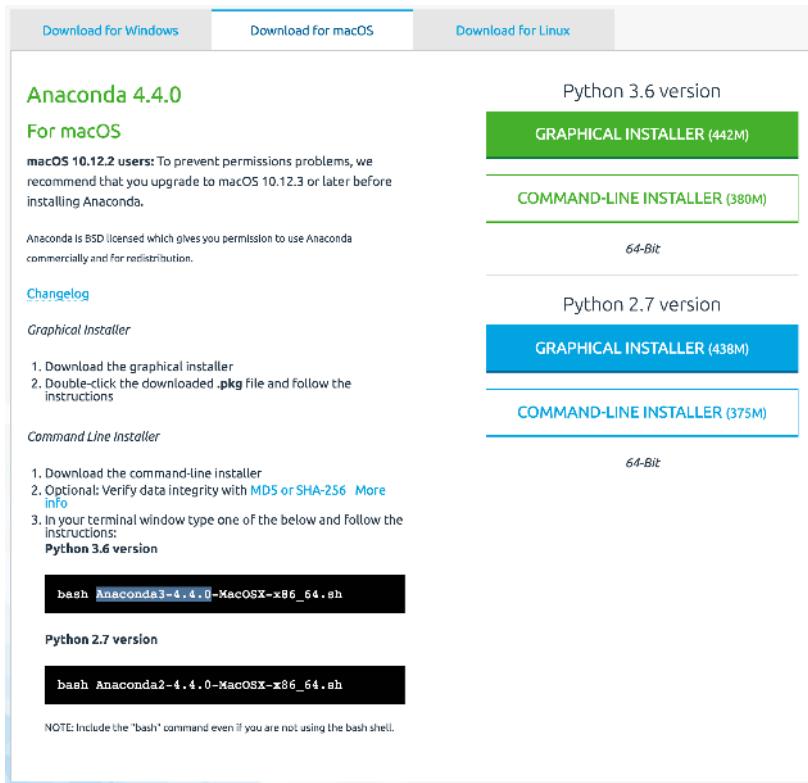


Figure B.2: Choose Anaconda Download for Your Platform.

This will download the Anaconda Python package to your workstation. I'm on macOS, so I chose the macOS version. The file is about 426 MB. You should have a file with a name like:

Anaconda3-4.4.0-MacOSX-x86_64.pkg

Listing B.1: Example filename on macOS.

B.3 Install Anaconda

In this step, we will install the Anaconda Python software on your system. This step assumes you have sufficient administrative privileges to install software on your system.

- 1. Double click the downloaded file.
- 2. Follow the installation wizard.

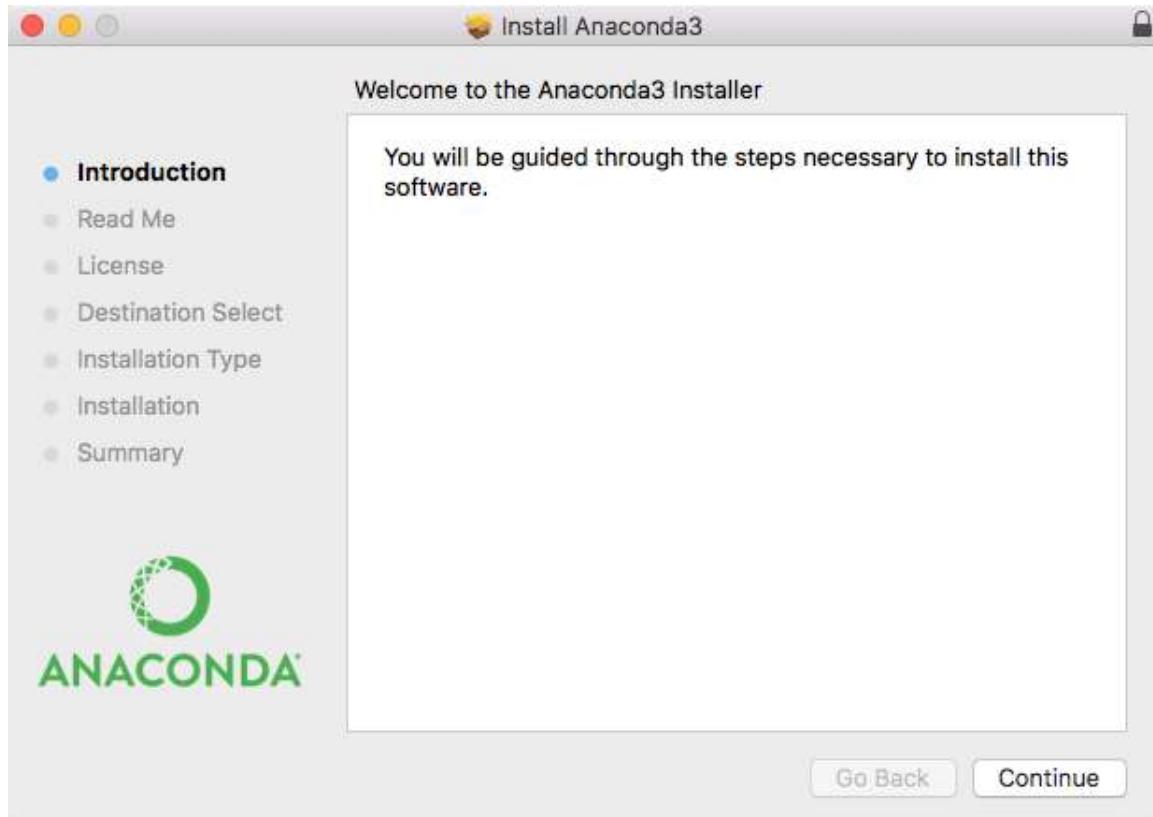


Figure B.3: Anaconda Python Installation Wizard.

Installation is quick and painless. There should be no tricky questions or sticking points.

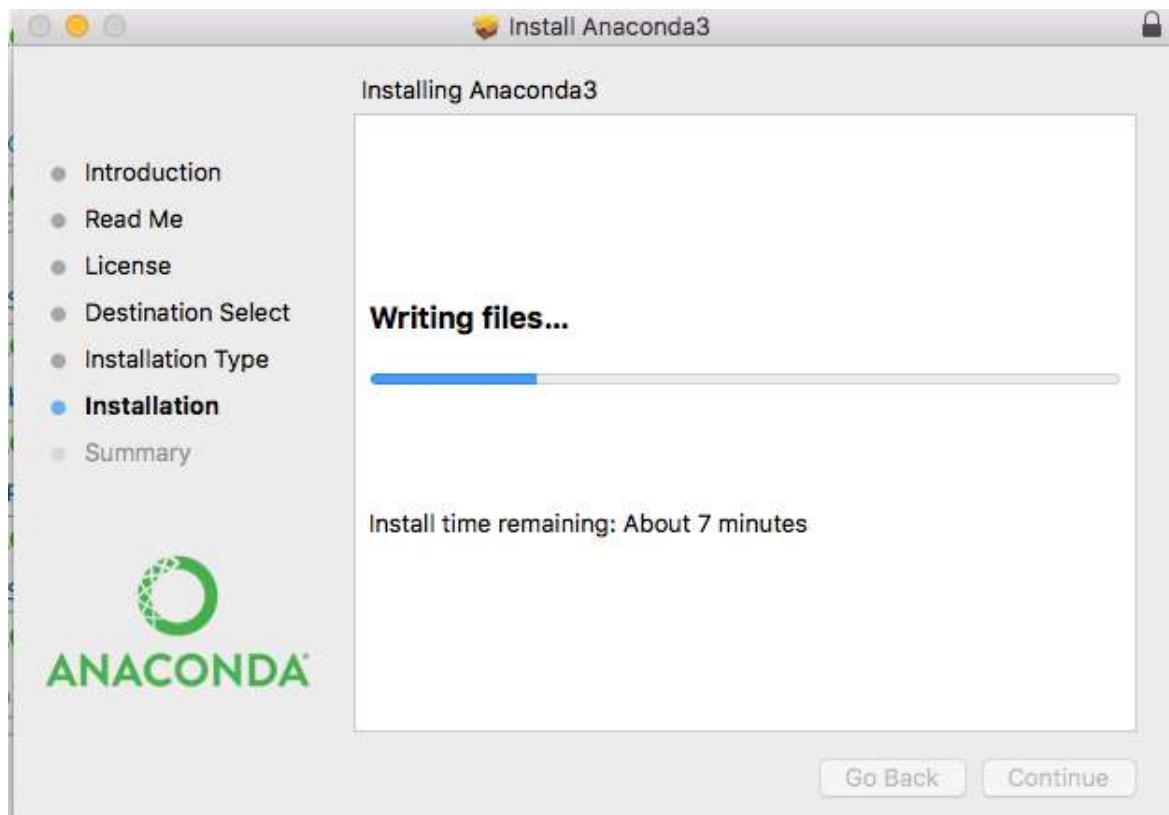


Figure B.4: Anaconda Python Installation Wizard Writing Files.

The installation should take less than 10 minutes and take up a little more than 1 GB of space on your hard drive.

B.4 Start and Update Anaconda

In this step, we will confirm that your Anaconda Python environment is up to date. Anaconda comes with a suite of graphical tools called Anaconda Navigator. You can start Anaconda Navigator by opening it from your application launcher.

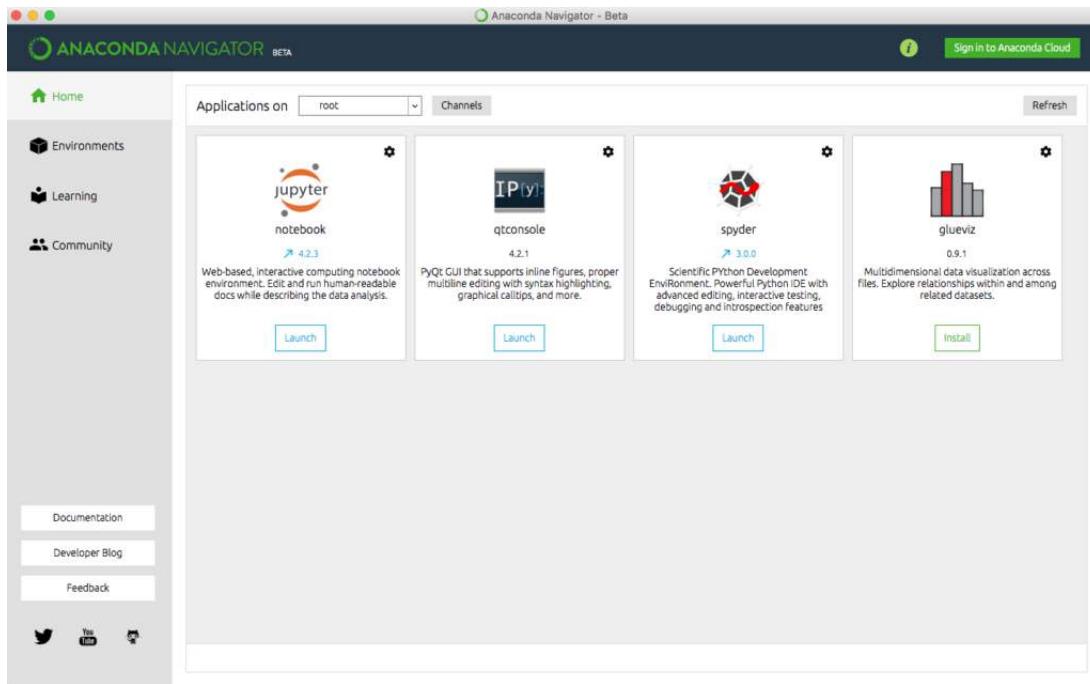


Figure B.5: Anaconda Navigator GUI.

You can use the Anaconda Navigator and graphical development environments later; for now, I recommend starting with the Anaconda command line environment called conda. Conda is fast, simple, it's hard for error messages to hide, and you can quickly confirm your environment is installed and working correctly.

- 1. Open a terminal (command line window).
- 2. Confirm conda is installed correctly, by typing:

```
conda -v
```

Listing B.2: Check the conda version.

You should see the following (or something similar):

```
conda 4.3.21
```

Listing B.3: Example conda version.

- 3. Confirm Python is installed correctly by typing:

```
python -V
```

Listing B.4: Check the Python version.

You should see the following (or something similar):

```
Python 3.6.1 :: Anaconda 4.4.0 (x86_64)
```

Listing B.5: Example Python version.

If the commands do not work or have an error, please check the documentation for help for your platform. See some of the resources in the *Further Reading* section.

- 4. Confirm your conda environment is up-to-date, type:

```
conda update conda
conda update anaconda
```

Listing B.6: Update conda and anaconda.

You may need to install some packages and confirm the updates.

- 5. Confirm your SciPy environment.

The script below will print the version number of the key SciPy libraries you require for machine learning development, specifically: SciPy, NumPy, Matplotlib, Pandas, Statsmodels, and Scikit-learn. You can type `python` and type the commands in directly. Alternatively, I recommend opening a text editor and copy-pasting the script into your editor.

```
# check library version numbers
# scipy
import scipy
print('scipy: %s' % scipy.__version__)
# numpy
import numpy
print('numpy: %s' % numpy.__version__)
# matplotlib
import matplotlib
print('matplotlib: %s' % matplotlib.__version__)
# pandas
import pandas
print('pandas: %s' % pandas.__version__)
# statsmodels
import statsmodels
print('statsmodels: %s' % statsmodels.__version__)
# scikit-learn
import sklearn
print('sklearn: %s' % sklearn.__version__)
```

Listing B.7: Code to check that key Python libraries are installed.

Save the script as a file with the name: `versions.py`. On the command line, change your directory to where you saved the script and type:

```
python versions.py
```

Listing B.8: Run the script from the command line.

You should see output like the following:

```
scipy: 1.5.2
numpy: 1.19.1
matplotlib: 3.3.0
pandas: 1.1.0
statsmodels: 0.11.1
sklearn: 0.23.2
```

Listing B.9: Sample output of versions script.

B.5 Install Deep Learning Libraries

In this step, we will install Python libraries used for deep learning, specifically: Theano, TensorFlow, and Keras. Note: I recommend using Keras for deep learning and Keras only requires one of Theano or TensorFlow to be installed. You do not need both. There may be problems installing TensorFlow on some Windows machines.

- 1. Install the Theano deep learning library by typing:

```
conda install theano
```

Listing B.10: Install Theano with conda.

- 2. Install the TensorFlow deep learning library by typing:

```
conda install -c conda-forge tensorflow
```

Listing B.11: Install TensorFlow with conda.

Alternatively, you may choose to install using pip and a specific version of TensorFlow for your platform.

- 3. Install Keras by typing:

```
pip install keras
```

Listing B.12: Install Keras with pip.

- 4. Confirm your deep learning environment is installed and working correctly.

Create a script that prints the version numbers of each library, as we did before for the SciPy environment.

```
# check deep learning version numbers
# theano
import theano
print('theano: %s' % theano.__version__)
# tensorflow
import tensorflow
print('tensorflow: %s' % tensorflow.__version__)
# keras
import keras
print('keras: %s' % keras.__version__)
```

Listing B.13: Code to check that key deep learning libraries are installed.

Save the script to a file `deep_versions.py`. Run the script by typing:

```
python deep_versions.py
```

Listing B.14: Run script from the command line.

You should see output like:

```
theano: 1.0.5
tensorflow: 2.3.0
keras: 2.4.3
```

Listing B.15: Sample output of the deep learning versions script.

B.6 Further Reading

This section provides resources if you want to know more about Anaconda.

- Anaconda homepage.
<https://www.continuum.io/>
- Anaconda Navigator.
<https://docs.continuum.io/anaconda/navigator.html>
- The conda command line tool.
<http://conda.pydata.org/docs/index.html>
- Instructions for installing TensorFlow in Anaconda.
https://www.tensorflow.org/get_started/os_setup#anaconda_installation

B.7 Summary

Congratulations, you now have a working Python development environment for machine learning and deep learning. You can now learn and practice machine learning and deep learning on your workstation.

Appendix C

How to Setup Amazon EC2 for Deep Learning on GPUs

Large deep learning models require a lot of compute time to run. You can run them on your CPU but it can take hours or days to get a result. If you have access to a GPU on your desktop, you can drastically speed up the training time of your deep learning models. In this project you will discover how you can get access to GPUs to speed up the training of your deep learning models by using the Amazon Web Service (AWS) infrastructure. For less than a dollar per hour and often a lot cheaper you can use this service from your workstation or laptop. After working through this project you will know:

- How to create an account and log-in to Amazon Web Service.
- How to launch a server instance for deep learning.
- How to configure a server instance for faster deep learning on the GPU.

Let's get started.

C.1 Overview

The process is quite simple because most of the work has already been done for us. Below is an overview of the process.

- Setup Your AWS Account.
- Launch Your Server Instance.
- Login and Run Your Code.
- Close Your Server Instance.

Note, it costs money to use a virtual server instance on Amazon. The cost is low for model development (e.g. less than one US dollar per hour), which is why this is so attractive, but it is not free. The server instance runs Linux. It is desirable although not required that you know how to navigate Linux or a Unix-like environment. We're just running our Python scripts, so no advanced skills are needed.

Note: The specific versions may differ as the software and libraries are updated frequently.

C.2 Setup Your AWS Account

You need an account on Amazon Web Services¹.

- 1. You can create account by the Amazon Web Services portal and click *Sign in to the Console*. From there you can sign in using an existing Amazon account or create a new account.

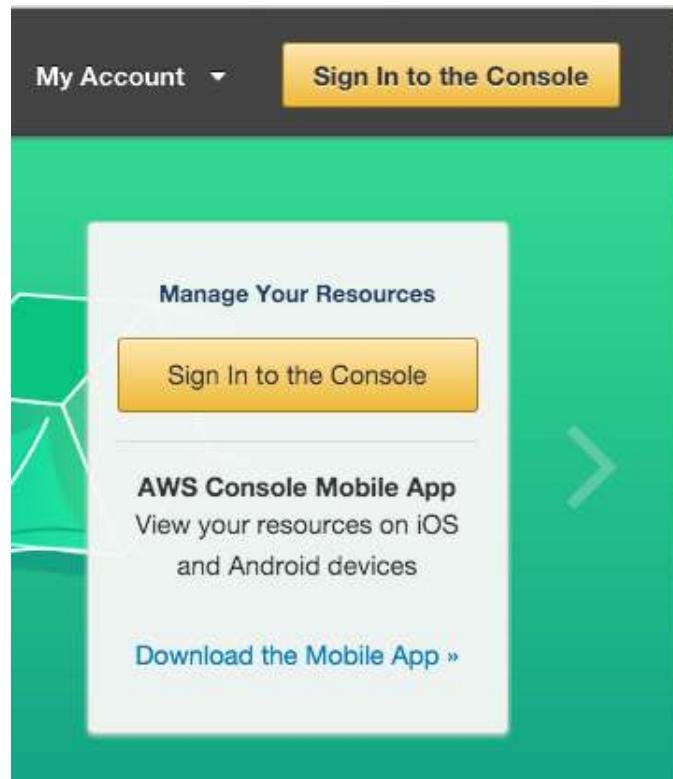


Figure C.1: AWS Sign-in Button

- 2. You will need to provide your details as well as a valid credit card that Amazon can charge. The process is a lot quicker if you are already an Amazon customer and have your credit card on file.

¹<https://aws.amazon.com>



The image shows the 'Sign In or Create an AWS Account' page. At the top is the Amazon Web Services logo. Below it, the heading 'Sign In or Create an AWS Account' is displayed in orange. A question 'What is your email (phone for mobile accounts)?' is followed by a text input field. Below the input field is a label 'E-mail or mobile number:' and another text input field. Underneath these fields are two radio button options: 'I am a new user.' (unselected) and 'I am a returning user and my password is:' (selected). To the right of the selected radio button is a text input field. Below this is a yellow 'Sign in using our secure server' button with a circular arrow icon. At the bottom left is a link 'Forgot your password?'

Figure C.2: AWS Sign-In Form

Once you have an account you can log into the Amazon Web Services console. You will see a range of different services that you can access.

C.3 Launch Your Server Instance

Now that you have an AWS account, you want to launch an EC2 virtual server instance on which you can run Keras. Launching an instance is as easy as selecting the image to load and starting the virtual server. Thankfully there is already an image available that has almost everything we need it is called the **Deep Learning AMI (Amazon Linux)** and was created and is maintained by Amazon. Let's launch it as an instance.

- 1. Login to your AWS console if you have not already.
<https://console.aws.amazon.com/console/home>

Amazon Web Services



Figure C.3: AWS Console

- 2. Click on EC2 for launching a new virtual server.
- 3. Select *US West Oregon* from the drop-down in the top right hand corner. This is important otherwise you will not be able to find the image we plan to use.
- 4. Click the *Launch Instance* button.
- 5. Click *Community AMIs*. An AMI is an Amazon Machine Image. It is a frozen instance of a server that you can select and instantiate on a new virtual server.

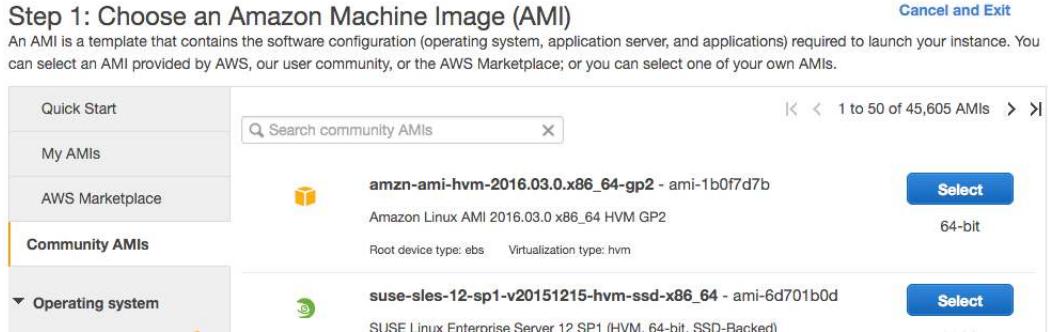


Figure C.4: Community AMIs

- 6. Enter Deep Learning AMI in the *Search community AMIs* search box and press enter.



Figure C.5: Select a Specific AMI

- 7. Click *Select* to choose the AMI in the search result.
- 8. Now you need to select the hardware on which to run the image. Scroll down and select the p3.2xlarge hardware (I used to recommend g2 or g3 instances and p2 instances, but the p3 instances² are newer and faster). This includes a Tesla V100 GPU that we can use to significantly increase the training speed of our models. It also includes 8 CPU Cores, 61GB of RAM and 16GB of GPU RAM. Note: using this instance will cost approximately \$3 USD/hour.

	Compute Optimized	General Purpose	Memory Optimized	Dedicated	Storage Optimized	GPU Accelerated	File Storage
<input checked="" type="checkbox"/>	GPU Instances	g2.2xlarge	8	15	1 x 60 (SSD)	Yes	High
<input type="checkbox"/>	CPU Instances	g2.2xlarge	8	15	1 x 60 (SSD)	No	Medium

Figure C.6: Select g2.2xlarge Hardware

- 9. Click *Review and Launch* to finalize the configuration of your server instance.
- 10. Click the *Launch* button.
- 11. Select Your Key Pair.

If you have a key pair because you have used EC2 before, select *Choose an existing key pair* and choose your key pair from the list. Then check *I acknowledge....* If you do not have a key pair, select the option *Create a new key pair* and enter a *Key pair name* such as keras-keypair. Click the *Download Key Pair* button.

²<https://aws.amazon.com/ec2/instance-types/p3/>

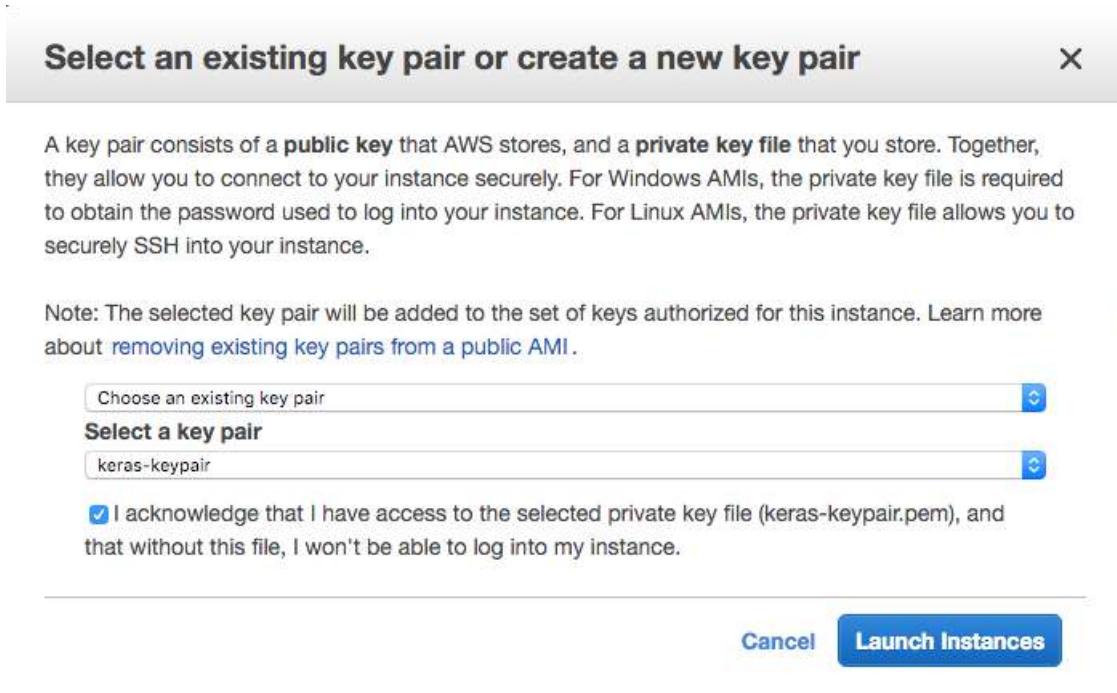


Figure C.7: Select Your Key Pair

- 12. Open a Terminal and change directory to where you downloaded your key pair.
- 13. If you have not already done so, restrict the access permissions on your key pair file. This is required as part of the SSH access to your server. For example, open a terminal on your workstation and type:

```
cd Downloads  
chmod 600 keras-aws-keypair.pem
```

Listing C.1: Change Permissions of Your Key Pair File.

- 14. Click *Launch Instances*. If this is your first time using AWS, Amazon may have to validate your request and this could take up to 2 hours (often just a few minutes).
- 15. Click *View Instances* to review the status of your instance.

Description		Status Checks	Monitoring	Tags
Instance ID	i-0852e21f4779bb063			
Instance state	running			
Instance type	g2.2xlarge			
Elastic IPs				
Availability zone	us-west-2b			
Security groups	launch-wizard-2, view inbound rules			
Scheduled events	No scheduled events			
AMI ID	Deep Learning AMI AmazonLinux - 2.0 (ami-dfb13ebf)			
Platform	-			
IAM role	-			
Key pair name	test-aws-deep			
Owner	959845963779			
Launch time	March 22, 2017 at 8:13:53 AM UTC+11 (less than one hour)			
Termination protection	False			
Lifecycle	normal			
Monitoring	basic			
Alarm status	None			
Kernel ID	-			
RAM disk ID	-			
Placement group	-			
Virtualization	hvm			
Reservation	r-01d15becdf2de436d			
AMI launch index	0			
Tenancy	default			
Host ID	-			
Affinity	-			
State transition reason	-			
State transition reason message	-			
Public DNS (IPv4)	ec2-54-186-97-77.us-west-2.compute.amazonaws.com			
IPv4 Public IP	54.186.97.77			
IPv6 IPs	-			
Private DNS	ip-172-31-45-13.us-west-2.compute.internal			
Private IPs	172.31.45.13			
Secondary private IPs				
VPC ID	vpc-7d09db18			
Subnet ID	subnet-ddc962b8			
Network interfaces	eth0			
Source/dest. check	True			
EBS-optimized	False			
Root device type	ebs			
Root device	/dev/xvda			
Block devices	/dev/xvda			

Figure C.8: Review Your Running Instance

Your server is now running and ready for you to log in.

C.4 Login, Configure and Run

Now that you have launched your server instance, it is time to log in and start using it.

- 1. Click *View Instances* in your Amazon EC2 console if you have not done so already.
- 2. Copy the *Public IP* (down the bottom of the screen in Description) to your clipboard. In this example my IP address is 54.186.97.77. **Do not use this IP address, it will not work as your server IP address will be different.**
- 3. Open a Terminal and change directory to where you downloaded your key pair. Login to your server using SSH, for example:

```
ssh -i keras-aws-keypair.pem ec2-user@54.186.97.77
```

Listing C.2: Log-in To Your AWS Instance.

- 4. If prompted, type **yes** and press enter.

You are now logged into your server.

```
=====
| ( | / Deep Learning AMI for Amazon Linux
| \ |
=====

The README file for the AMI ----- /home/ec2-user/src/README.md
Tests for deep learning frameworks ----- /home/ec2-user/src/bin
=====

[ec2-user@ip-172-31-45-13 ~]$ 
```

Figure C.9: Log in Screen for Your AWS Server

The instance will ask what Python environment you wish to use. I recommend using:

- **TensorFlow(+Keras2) with Python3 (CUDA 9.0 and Intel MKL-DNN)**

You can activate this virtual environment by typing:

```
source activate tensorflow_p36
```

Listing C.3: Activate the appropriate virtual environment.

You are now free to run your code.

C.5 Build and Run Models on AWS

This section offers some tips for running your code on AWS.

C.5.1 Copy Scripts and Data to AWS

You can get started quickly by copying your files to your running AWS instance. For example, you can copy the examples provided with this book to your AWS instance using the `scp` command as follows:

```
scp -i keras-aws-keypair.pem -r src ec2-user@54.186.97.77:~/
```

Listing C.4: Example for Copying Sample Code to AWS.

This will copy the entire `src/` directory to your home directory on your AWS instance. You can easily adapt this example to get your larger datasets from your workstation onto your AWS instance. Note that Amazon may impose charges for moving very large amounts of data in and out of your AWS instance. Refer to Amazon documentation for relevant charges.

C.5.2 Run Models on AWS

You can run your scripts on your AWS instance as per normal:

```
python filename.py
```

Listing C.5: Example of Running a Python script on AWS.

You are using AWS to create large neural network models that may take hours or days to train. As such, it is a better idea to run your scripts as a background job. This allows you to close your terminal and your workstation while your AWS instance continues to run your script. You can easily run your script as a background process as follows:

```
nohup /path/to/script >/path/to/script.log 2>&1 < /dev/null &
```

Listing C.6: Run Script as a Background Process.

You can then check the status and results in your `script.log` file later.

C.6 Close Your EC2 Instance

When you are finished with your work you must close your instance. Remember you are charged by the amount of time that you use the instance. It is cheap, but you do not want to leave an instance on if you are not using it.

- 1. Log out of your instance at the terminal, for example you can type:

```
exit
```

Listing C.7: Log-out of Server Instance.

- 2. Log in to your AWS account with your web browser.
- 3. Click EC2.
- 4. Click *Instances* from the left-hand side menu.

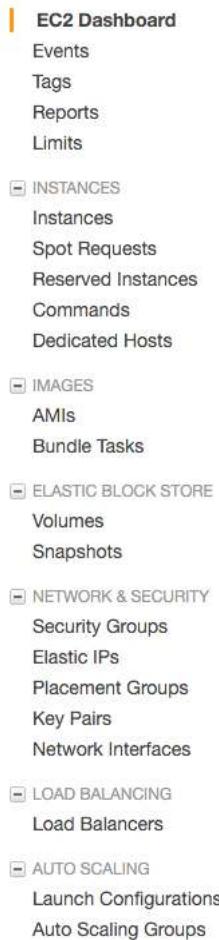


Figure C.10: Review Your List of Running Instances

- 5. Select your running instance from the list (it may already be selected if you only have one running instance).
- 6. Click the *Actions* button and select *Instance State* and choose *Terminate*. Confirm that you want to terminate your running instance.

It may take a number of seconds for the instance to close and to be removed from your list of instances.

C.7 Tips and Tricks for Using Keras on AWS

Below are some tips and tricks for getting the most out of using Keras on AWS instances.

- **Design a suite of experiments to run beforehand.** Experiments can take a long time to run and you are paying for the time you use. Make time to design a batch of experiments to run on AWS. Put each in a separate file and call them in turn from another script. This will allow you to answer multiple questions from one long run, perhaps overnight.

- **Always close your instance at the end of your experiments.** You do not want to be surprised with a very large AWS bill.
- **Try spot instances for a cheaper but less reliable option.** Amazon sell unused time on their hardware at a much cheaper price, but at the cost of potentially having your instance closed at any second. If you are learning or your experiments are not critical, this might be an ideal option for you. You can access spot instances from the *Spot Instance* option on the left hand side menu in your EC2 web console.

C.8 Further Reading

Below is a list of resources to learn more about AWS and developing deep learning models in the cloud.

- An introduction to Amazon Elastic Compute Cloud (EC2).
<http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html>
- An introduction to Amazon Machine Images (AMI).
<http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html>
- Deep Learning AMI (Amazon Linux) on the AMI Marketplace.
<https://aws.amazon.com/marketplace/pp/B077GF11NF>
- P3 EC2 Instances.
<https://aws.amazon.com/ec2/instance-types/p3/>

C.9 Summary

In this lesson you discovered how you can develop and evaluate your large deep learning models in Keras using GPUs on the Amazon Web Service. You learned:

- Amazon Web Services with their Elastic Compute Cloud offers an affordable way to run large deep learning models on GPU hardware.
- How to setup and launch an EC2 server for deep learning experiments.
- How to update the Keras version on the server and confirm that the system is working correctly.
- How to run Keras experiments on AWS instances in batch as background tasks.

Part IX

Conclusions

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come. You now know:

- About the promise of neural networks and deep learning methods in general for computer vision problems.
- How to load and prepare image data, such as photographs, for modeling using best-of-breed Python libraries.
- How specialized layers for image data work, including 1D and 2D convolutions, max and average pooling, and intuitions for the impact that each layer has on input data.
- How to configure convolutional layers, including aspects such as filter size, stride, and pooling.
- How key modeling innovations for convolutional neural networks work and how to implement them from scratch, such as VGG blocks, inception models, and resnet modules.
- How to develop, tune, evaluate and make predictions with convolutional neural networks on standard benchmark computer vision datasets for image classification, such as Fashion-MNIST and CIFAR-10.
- How to develop, tune, evaluate, and make predictions with convolutional neural networks on entirely new datasets for image classification, such as satellite photographs and photographs of pets.
- How to use techniques such as pre-trained models, transfer learning and image augmentation to accelerate and improve model development.
- How to use pre-trained models and develop new models for object recognition tasks, such as object localization and object detection in photographs, using techniques like R-CNN and YOLO.
- How to use deep learning models for face recognition tasks, such as face identification and face verification in photographs, using techniques like Google's FaceNet and Oxford's VGGFace.

Don't make light of this. You have come a long way in a short amount of time. You have developed the important and valuable set of skills for developing deep learning neural network models for computer vision applications. You can now confidently:

- Load, prepare and scale image data ready for modeling.
- Develop effective convolutional neural network models quickly from scratch.
- Harness world-class pre-trained models on new problems.

The sky's the limit.

Thank You!

I want to take a moment and sincerely thank you for letting me help you start your journey toward deep learning for computer vision. I hope you keep learning and have fun as you continue to master machine learning.

Jason Brownlee
2019

Licenses

Licensed Images

- Opera House and Ferries, Ed Dunens, CC BY 2.0.
<https://www.flickr.com/photos/blachswan/36102705716/>
- The Sydney Harbour Bridge, Bernard Spragg. NZ Follow, Public domain.
<https://www.flickr.com/photos/volvob12b/27318376851/>
- Bondi Beach, Sydney, Isabell Schulz, CC BY-SA 2.0.
<https://www.flickr.com/photos/isapisa/45545118405/>
- Hungary-0132 - A Mustang...!, Dennis Jarvis, CC BY-SA 2.0.
<https://www.flickr.com/photos/archer10/7296236992/>
- Blue Car, Bill Smith, CC BY 2.0.
<https://www.flickr.com/photos/byzantiumbooks/25158850275/>
- Feathered Friend, AndYaDontStop, CC BY 2.0.
<https://www.flickr.com/photos/thenovys/3854468621>
- Phillip Island, Penguin Parade, VIC4., Larry Koester, CC BY 2.0.
<https://www.flickr.com/photos/larrywkoester/45435718605/>
- dog, Justin Morgan, CC BY-SA 2.0.
<https://www.flickr.com/photos/jmorgan/5164287/>
- Safari, Boegh, CC BY-SA 2.0.
<https://www.flickr.com/photos/boegh/5676993427/>
- This is the elephant who charged us, Mandy Goldberg, CC BY 2.0.
<https://www.flickr.com/photos/viewfrom52/2081198423/>
- College Students, CollegeDegrees360, CC BY-SA 2.0.
<https://www.flickr.com/photos/83633410@N07/7658261288/>
- Sequoia HS Swim Team 2008, Bob n Renee, CC BY 2.0.
<https://www.flickr.com/photos/bobnrenee/2370369784/>
- Sharon Stone Cannes 2013 2, Georges Biard, CC BY-SA 3.0.
https://en.wikipedia.org/wiki/File:Sharon_Stone_Cannes_2013_2.jpg
- Channing Tatum, Gage Skidmore, CC BY-SA 3.0.
https://en.wikipedia.org/wiki/File:Channing_Tatum_by_Gage_Skidmore_3.jpg

- Sharon Stone 2, Rita Molnar, CC BY-SA 2.5.
https://en.wikipedia.org/wiki/File:Sharon_Stone..jpg
- Sharon Stone 3, Gage Skidmore, CC BY-SA 3.0.
https://en.wikipedia.org/wiki/File:Sharon_Stone_by_Gage_Skidmore_3.jpg

Licensed Code

- keras-yolo3, Huynh Ngoc Anh, MIT License.
<https://github.com/experiencor/keras-yolo3>