TensorFlow 2.0

Pocket Primer

TensorFlow 2.0
Pocket Primer

Oswald Campesato

Mercury Learning and Information
Dulles, Virginia
Boston, Massachusetts
New Delhi

Publisher: David Pallai
Mercury Learning and Information
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
800-232-0223

O. Campesato. *TensorFlow 2.0. Pocket Primer.*
ISBN: 978-1-68392-460-9

Library of Congress Control Number: 2019941748

192021321 This book is printed on acid-free paper in the United States of America.

I'd like to dedicate this book to my parents –
may this bring joy and happiness into their lives.

CONTENTS

PREFACE

WHAT IS THE GOAL?

The goal of this book is to introduce TensorFlow 2 fundamentals for basic machine learning algorithms in TensorFlow. It is intended to be a fast-paced introduction to various "core" features of TensorFlow, with code samples that cover deep learning and TensorFlow. The material in the chapters illustrates how to solve a variety of tasks using TensorFlow, after which you can do further reading to deepen your knowledge.

This book provides more detailed code samples than those that are found in intermediate and advanced TensorFlow books. Although it contains some basic code samples in TensorFlow, some familiarity with the software will be helpful.

The book will also save you the time required to search for code samples, which is a potentially time-consuming process. In any case, if you're not sure whether or not you can absorb the material in this book, glance through the code samples to get a feel for the level of complexity. At the risk of stating the obvious, please keep in mind the following point: *you will not become an expert in TensorFlow by reading this book*.

WHAT WILL I LEARN FROM THIS BOOK?

The first chapter contains TensorFlow code samples that illustrate very simple TensorFlow functionality, followed by a chapter whose code samples illustrate an assortment of built-in APIs. The third chapter delves into the TensorFlow Dataset, with a plethora of code samples that illustrate how to use "lazy" operators in conjunction with datasets. The fourth chapter discusses linear regression and the fifth chapter covers logistic regression. If you think that you'll struggle significantly with the code in the first two chapters, then an "absolute beginners" type of book is recommended to prepare you for this one.

Another point: although Jupyter is popular, all the code samples in this book are Python scripts. However, you can quickly learn about the useful features of Jupyter through various online tutorials. In addition, it's worth looking at Google Colaboratory, which is entirely online and is based on Jupyter notebooks, along with free GPU usage.

WHY DOES THIS BOOK INCLUDE TF 1.X MATERIAL?

If you are new to TensorFlow, then feel free to skip the TF 1.x content, particularly if you are starting with a new project involving TensorFlow and you don't have any TF 1.x. However, as this book goes to print, the vast majority of existing TensorFlow code is TF 1.x code, which is massive when you consider all the companies that are using TensorFlow. Hence, many people who are working with TF 1.x also need to learn how to convert TF 1.x to TF 2.

Almost all the TF 1.x material (including the section regarding the upgrade script from TF 1.x to TF 2) is limited to the second half of Chapter 1. Keep in mind another detail: even if you plan to learn only TF 2, you might be faced with a task that involves upgrading from TF 1.x to TF 2, and now you'll have some potentially useful information regarding TF 1.x in this book.

THE TF 1.X AND TF 2.0 BOOKS: HOW ARE THEY DIFFERENT?

TensorFlow 2 uses eager execution whereas TensorFlow 1.x uses deferred execution, which means that the coding styles are significantly different. TF 2.0 also introduces new features, such as generators (which are decorated Python functions), that are discussed in that book.

In some cases, TF 1.x and TF 2 contain the same functionality that is implemented using different APIs. For example, `tf.data.Dataset` in TF 1.x uses iterators (there are four main types) to iterate through datasets, whereas `tf.data.Dataset` in TF 2 uses generators. The TF 2.0 book contains both types of code samples for `tf.data.Dataset` code samples (with the primary focus on TF 2.0 coding style).

WHY ISN'T KERAS IN ITS OWN CHAPTER IN THIS BOOK?

The answer is straightforward: this book introduces TensorFlow 2 from the perspective of people who are interested in machine learning. Consequently, Keras is introduced on an "as-needed" topic. For example, Chapter 4 contains a section about Keras in the context of linear regression. Chapter 5 contains a Keras-based code sample in the context of classifiers (specifically for logistic regression). The appendix also contains some Keras-based code samples for advanced topics.

For the same reason, Chapter 5 is devoted to classifiers in machine learning, and the Keras and TF 2 material is discussed in the second half of the chapter. The extent to which this mixture appeals to you depends on your objectives regarding TensorFlow 2 and machine learning.

HOW MUCH KERAS KNOWLEDGE IS NEEDED FOR THIS BOOK?

The answer depends on the extent to which you become involved in machine learning: there are essentially four options available, which are discussed as follows.

Option #1: if you are not interested in Keras, you can skip the last example in Chapter 4 and Chapter 5, as well as the appendix: even so, there is still plenty of TF 2 content in this book.
Option #2: if you only want to learn enough details about Keras to work with linear regression, there is a very simple example in Chapter 4 that follows a "bare bones" section regarding Keras.
Option #3: if you also want to learn about Keras and logistic regression, there is an example in Chapter 5. This example requires some theoretical knowledge involving activation functions, optimizers, and cost functions, all of which are discussed in the first half of Chapter 5.
Option #4: if you want to go even further and also learn about Keras and deep learning, the appendix discusses some of the underpinnings of MLPs, CNNs, RNNs, and LSTMs.

Please keep in mind that Keras is well-integrated into TensorFlow 2 (in the `tf.keras` namespace), and it provides a layer of abstraction over "pure" TensorFlow that will enable you to develop prototypes more quickly.

If you have never worked with Keras, you'll probably enjoy the experience, and if need be, you can read some introductory online tutorials in preparation for the Keras-based content in this book. Regardless of your knowledge level, if you decide to skip the Keras-related content for now, *eventually* you do need to learn Keras in order to fully master TensorFlow 2.

DO I NEED TO LEARN THE THEORY PORTIONS OF THIS BOOK?

Once again, the answer depends on the extent to which you plan to become involved in machine learning. In addition to creating a model, you will use various algorithms to see which ones provide the level of accuracy (or some other metric) that you need for your project. If you fall short, the theoretical aspects of machine learning can help you perform a "forensic" analysis of your model and your data, and ideally assist in determining how to improve your model.

You can acquire a cursory understanding of TensorFlow 2 from the material in this book; delving further into TF 2 depends on your tasks and career goals.

HOW WERE THE CODE SAMPLES CREATED?

The code samples in this book were created and tested using the TensorFlow `tf-nightly-2.0-preview` (from 4/7/2019) on a MacBook Pro with OS X 10.12.6 (macOS Sierra). Regarding their content: the code samples are derived primarily from the author for his deep learning and TensorFlow graduate course. In some cases there are code samples that incorporate short sections of code from discussions in online forums. The key point to remember is that the code samples follow the "Four Cs": they must be Clear, Concise, Complete, and Correct to the extent that it's possible to do so, given the size of this book.

WHAT ARE THE TECHNICAL PREREQUISITES FOR THIS BOOK?

You need some familiarity with Python, and also need to know how to launch Python code from the command line (in a Unix-like environment for Mac users). In addition, a mixture of basic linear algebra (vectors and matrices), probability/statistics (mean, median, standard deviation), and basic concepts in calculus (such as derivatives) will help you learn the material in this book.

Some knowledge of NumPy and Matplotlib is also helpful, and the assumption is that you are familiar with basic functionality (such as `NumPy` arrays). For example, Chapter 2 contains a code sample that invokes the `tf.range()` API, which is similar to the `NumPy linspace()` API; however, the `NumPy linspace()` API is not explained in the code (so you need to look up the details of this API if it's unfamiliar). As another example, in Chapter 3 a TF 2 `Dataset` is described as being analogous to a `Pandas DataFrame`; however, Pandas APIs are not explained in this book.

One other prerequisite is important for understanding the code samples in the appendix: some familiarity with neural networks, which includes the concept of hidden layers and activation functions (even if you don't fully understand them). Knowledge of cross entropy is also helpful for some of the code samples.

Also keep in mind that TensorFlow provides a vast assortment of APIs, some of which are discussed in the code samples in the book chapters. While it's possible for you to "pick up" the purpose of the more intuitive APIs by reading the online documentation, that's only true for the basic TensorFlow APIs. Consequently, you probably won't really understand how to "tweak" the values of their parameters and why they are needed until you work with them in TensorFlow code samples. *In other words, if you read TensorFlow code samples containing APIs that you do not understand, in many cases it's not enough to repeatedly read the code samples.*

A more efficient approach is to learn about the purpose of the TensorFlow APIs by reading small code samples that clearly illustrate the purpose of those APIs, after which you can read more complex TensorFlow code samples.

WHAT ARE THE NONTECHNICAL PREREQUISITES FOR THIS BOOK?

Although the answer to this question is more difficult to quantify, it's very important to have a strong desire to learn TensorFlow and machine learning, along with the motivation and discipline to read and understand the code samples.

Even the non-trivial TensorFlow APIs can be a challenge to understand the first time you encounter them, so be prepared to read the code samples several times. The latter requires persistence when learning TensorFlow, and whether or not you have enough persistence is something that you need to decide for yourself.

WHICH TOPICS ARE EXCLUDED?

The chapters in this book do not cover CNNs (Convolutional Neural Networks), RNNs (Recurrent Neural Networks), or LSTMs (Long Short Term Memory). However, these topics are introduced in the appendix, in a somewhat cursory fashion, which is to say that the appendix is not a substitute for taking a deep learning course.

You will not find in-depth details about TensorFlow layers and estimators (but they are lightly discussed). Keep in mind that online searches on Stackoverflow will often involve solutions employing TF 1.x, whereas solutions for TF 2 will be less common.

HOW DO I SET UP A COMMAND SHELL?

If you are a Mac user, there are three ways to do so. The first method is to use `Finder` to navigate to `Applications > Utilities` and then double-click on the `Utilities` application. Next, if you already have a command shell available, you can launch a new command shell by typing the following command:

```
open /Applications/Utilities/Terminal.app
```

A second method for Mac users is to open a new command shell on a MacBook from a command shell that is already visible simply by clicking `command+n` in that command shell, and your Mac will launch another command shell.

If you are a PC user, you can install Cygwin (open source https://cygwin.com/), which simulates `bash` commands, or use another toolkit such as `MKS` (a commercial product). Please read the online documentation that describes the download and installation process. Note that custom aliases are not automatically set if they are defined in a file other than the main start-up file (such as .bash_login).

COMPANION FILES

All the code samples and figures in this book may be obtained by writing to the publisher at info@merclearning.com.

WHAT ARE THE "NEXT STEPS" AFTER FINISHING THIS BOOK?

The answer to this question varies widely, mainly because the answer depends heavily on your objectives. The best answer is to try out a new tool or technique from the book on a problem or task you care about, professionally or personally. Precisely what that might be depends on who you are, as the needs of a data scientist, manager, student, or developer are all different. In addition, keep what you learned in mind as you tackle new challenges.

If you have reached the limits of what you have learned here and want to get further technical depth regarding TensorFlow, there are various online resources and literature describing more complex features of TensorFlow.

CHAPTER 1

INTRODUCTION TO TENSORFLOW 2

Welcome to TensorFlow 2! This chapter introduces you to various features of TensorFlow 2 (abbreviated as TF 2), as well as some of the TF 2 tools and projects that are covered under the TF 2 "umbrella." You will see TF 2 code samples that illustrate new TF 2 features (such as `tf.GradientTape` and the `@tf.function` decorator), plus an assortment of code samples that illustrate how to write code "the TF 2 way."

Despite the simplicity of many topics in this chapter, they provide you with a foundation for TF 2. This chapter also prepares you for Chapter 2, which delves into frequently used TF 2 APIs that you will encounter in other chapters of this book.

Keep in mind that the TensorFlow 1.x releases are considered legacy code after the production release of TF 2. Google will provide only security-related updates for TF 1.x (i.e., no new code development) and support TensorFlow 1.x for at least another year beyond the initial production release of TF 2. For your convenience, TensorFlow provides a conversion script to facilitate the automatic conversion of TensorFlow 1.x code to TF 2 code in many cases (details provided later in this chapter).

As you saw in the Preface, this chapter contains several sections regarding TF 1.x, all of which are placed near the end of this chapter. If you do not have TF 1.x code, obviously these sections are optional (and they are labeled as such).

The first part of this chapter briefly discusses some TF 2 features and some of the tools that are included under the TF 2 "umbrella." The second section of this chapter shows you how to write TF 2 code involving TF constants and TF variables.

The third section digresses a bit: you will learn about the new TF 2 Python function decorator `@tf.function` that is used in many code samples in this chapter. Although this decorator is not always required, it's important to become comfortable with this feature, and there are some nonintuitive caveats regarding its use that are discussed in this section.

The fourth section of this chapter shows you how to perform typical arithmetic operations in TF 2, how to use some of the built-in TF 2 functions, and how to calculate trigonometric values. If you need to perform scientific calculations, see the code samples that pertain to the type of precision that you can achieve with floating point numbers in TF 2. This section also shows you how to use `for` loops and how to calculate exponential values.

The fifth section contains TF 2 code samples involving arrays, such as creating an identity matrix, a constant matrix, a random uniform matrix, and a truncated normal matrix, along with an explanation about the difference between a truncated matrix and a random matrix. This section also shows you how to multiply second-order tensors in TF 2 and how to convert Python arrays to second-order tensors in TF 2. The sixth section contains code samples that illustrate how to use some of the new features of TF 2, such as `tf.GradientTape`.

Although the TF 2 code samples in this book use Python 3.x, it's possible to modify the code samples in order to run under Python 2.7. Also make note of the following convention in this book (and only this book): TF 1.x files have a "tf_" prefix and TF 2 files have a "tf2_" prefix.

With all that in mind, the next section discusses a few details of TF 2, its architecture, and some of its features.

WHAT IS TF 2?

TF 2 is an open source framework from Google that is the newest version of TensorFlow. The TF 2 framework is a modern framework that's well-suited for machine learning and deep learning, and it's available through an Apache license. Interestingly, TensorFlow surprised many people, perhaps even members of the TF team, in terms of the creativity and plethora of use cases for TF in areas such as art, music, and medicine. For a variety of reasons, the TensorFlow team created TF 2 with the goal of consolidating the TF APIs, eliminating duplication of APIs, enabling rapid prototyping, and making debugging an easier experience.

There is good news if you are a fan of Keras: improvements in TF 2 are partially due to the adoption of Keras as part of the core functionality of TF 2. In fact, TF 2 extends and optimizes Keras so that it can take advantage of all the advanced features in TF 2.

If you work primarily with deep learning models (CNNs, RNNs, LSTMs, and so forth), you'll probably use some of the classes in the `tf.keras` namespace, which is the implementation of Keras in TF 2. Moreover, `tf.keras.layers` provides many standard layers for neural networks. As you'll see later, there are several ways to define Keras-based models, via the `tf.keras.Sequential` class, a functional style definition, and via a subclassing technique. Alternatively, you can still use lower-level operations and automatic differentiation if you wish to do so.

Furthermore, TF 2 removes duplicate functionality, provides a more intuitive syntax across APIs, and also compatibility throughout the TF 2 ecosystem. TF 2 even provides a backward compatibility module called `tf.compat.v1` (which does not include `tf.contrib`), and a conversion script `tf_upgrade_v2` to help users migrate from TF 1.x to TF 2.

Another significant change in TF 2 is eager execution as the default mode (not deferred execution), with new features such as the `@tf.function` decorator and TF 2 privacy-related features. Here is a condensed list of some TF 2 features and related technologies:

- support for `tf.keras`: a specification for high-level code for ML and DL
- tensorflow.js v1.0: TF in modern browsers
- TensorFlow Federated: an open source framework for ML and decentralized data
- ragged tensors: nested variable-length (“uneven”) lists
- TensorFlow Probability: probabilistic models combined with deep learning
- Tensor2Tensor: a library of DL models and datasets

TF 2 also supports a variety of programming languages and hardware platforms, including:

- Support for Python, Java, C++
- Desktop, server, mobile device (TF Lite)
- CPU/GPU/TPU support
- Linux and Mac OS X support
- VM for Windows

Navigate to the TF 2 home page, where you will find links to many resources for TF 2: *https://www.tensorflow.org*

TF 2 Use Cases

TF 2 is designed to solve tasks that arise in a plethora of use cases, some of which are listed here:

- Image recognition
- Computer vision
- Voice/sound recognition
- Time series analysis
- Language detection
- Language translation
- Text-based processing
- Handwriting recognition

The preceding list of use cases can be solved in TF 1.x as well as TF 2, and in the latter case, the code tends to be simpler and cleaner compared to its TF 1.x counterpart.

TF 2 Architecture: The Short Version

TF 2 is written in C++ and supports operations involving primitive values and tensors (discussed later). The default execution mode for TF 1.x is *deferred execution* whereas TF 2 uses *eager execution* (think "immediate mode"). Although TF 1.4 introduced eager execution, the vast majority of TF 1.x code samples that you will find online use deferred execution.

TF 2 supports arithmetic operations on tensors (i.e., multidimensional arrays with enhancements) as well as conditional logic, "for" loops, and "while" loops. Although it's possible to switch between eager execution mode and deferred mode in TF 2, all the code samples in this book use eager execution mode.

Data visualization is handled via TensorBoard (discussed in Chapter 2) that is included as part of TF 2. As you will see in the code samples in this book, TF 2 APIs are available in Python and can therefore be embedded in Python scripts.

So, enough already with the high-level introduction: let's learn how to install TF 2, which is the topic of the next section.

TF 2 Installation

Install TensorFlow by issuing the following command from the command line:

```
pip install tensorflow==2.0.0-beta0
```

When a production release of TF 2 is available, you can issue the following command from the command line (which will be the most current version of TF 2):

```
pip install --upgrade tensorflow
```

If you want to install a specific version (let's say version 1.13.1) of TensorFlow, type the following command:

```
pip install --upgrade tensorflow==1.13.1
```

You can also downgrade the installed version of TensorFlow. For example, if you have installed version 1.13.1 and you want to install version 1.10, specify the value 1.10 in the preceding code snippet. TensorFlow will uninstall your current version and install the version that you specified (i.e., 1.10).

As a sanity check, create a Python script with the following three lines of code to determine the version number of TF that is installed on your machine:

```
import tensorflow as tf
print("TF Version:",tf.__version__)
print("eager execution:",tf.executing_eagerly())
```

Launch the preceding code and you ought to see something similar to the following output:

```
TF version: 2.0.0-beta0
eager execution: True
```

As a simple example of TF 2 code, place this code snippet in a text file:

```
import tensorflow as tf
print("1 + 2 + 3 + 4 =", tf.reduce_sum([1, 2, 3, 4]))
```

Launch the preceding code from the command line and you should see the following output:

```
1 + 2 + 3 + 4 = tf.Tensor(10, shape=(), dtype=int32)
```

TF 2 and the Python REPL

In case you aren't already familiar with the Python `REPL` (read-eval-print-loop), it's accessible by opening a command shell and then typing the following command:

```
python
```

As a simple illustration, access TF 2-related functionality in the `REPL` by importing the TF 2 library as follows:

```
>>> import tensorflow as tf
```

Now check the version of TF 2 that is installed on your machine with this command:

```
>>> print('TF version:',tf.__version__)
```

The output of the preceding code snippet is shown here (the number that you see depends on which version of TF 2 that you installed):

```
TF version: 2.0.0-beta0
```

Although the `REPL` is useful for short code blocks, the TF 2 code samples in this book are Python scripts that you can launch with the Python executable.

OTHER TF 2-BASED TOOLKITS

In addition to providing support for TF 2-based code on multiple devices, TF 2 provides the following toolkits:

- TensorBoard for visualization (included as part of TensorFlow)
- TensorFlow Serving (hosting on a server)
- TensorFlow Hub

- TensorFlow Lite (for mobile applications)
- Tensorflow.js (for Web pages and NodeJS)

TensorBoard is a graph visualization tool that runs in a browser. Launch TensorBoard from the command line as follows: open a command shell and type the following command to access a saved TF graph in the subdirectory `/tmp/abc` (or a directory of your choice):

```
tensorboard –logdir /tmp/abc
```

Note that there are two consecutive dashes ("-") that precede the `logdir` parameter in the preceding command. Now launch a browser session and navigate to this URL: `localhost:6006`

After a few moments you will see a visualization of the TF 2 graph that was created in your code and then saved in the directory `/tmp/abc`.

TensorFlow Serving is a cloud-based, flexible, high-performance serving system for ML models that is designed for production environments. TensorFlow Serving makes it easy to deploy new algorithms and experiments, while keeping the same server architecture and APIs. More information is here:

https://www.TF 2.org/serving/

TensorFlow Lite was specifically created for mobile development (both Android and iOS). Please keep in mind that TensorFlow Lite supersedes TF 2 Mobile, which was an earlier SDK for developing mobile applications. TensorFlow Lite (which also exists for TF 1.x) supports on-device ML inference with low latency and a small binary size. Moreover, TensorFlow Lite supports hardware acceleration with the Android Neural Networks API. More information about TensorFlow Lite is here:

https://www.tensorflow.org/lite/

A more recent addition is `tensorflow.js`, which provides JavaScript APIs to access TensorFlow in a Web page. The `tensorflow.js` toolkit was previously called `deeplearning.js`. You can also use `tensorflow.js` with NodeJS. More information about `tensorflow.js` is here:

https://js.tensorflow.org/

TF 2 EAGER EXECUTION

TF 2 eager execution mode makes TF 2 code much easier to write compared to TF 1.x code (which used deferred execution mode). You might be surprised to discover that TF introduced "eager execution" as an alternative to deferred execution in version 1.4.1, but this feature was vastly underutilized. With TF 1.x code, TensorFlow creates a dataflow graph that consists of (a) a set of `tf.Operation` objects that represent units of computation, and (b) `tf.Tensor` objects that represent the units of data that flow between operations.

On the other hand, TF 2 evaluates operations immediately without instantiating a `Session` object or a creating a graph. Operations return concrete values instead of creating a computational graph. TF 2 eager execution is based on Python control flow instead of graph control flow. Arithmetic operations are simpler and intuitive, as you will see in code samples later in this chapter. Moreover, TF 2 eager execution mode simplifies the debugging process. However, keep in mind that there isn't a 1:1 relationship between a graph and eager execution.

TF 2 TENSORS, DATA TYPES, AND PRIMITIVE TYPES

In simplified terms, a TF 2 tensor is an n-dimensional array that is similar to a `NumPy ndarray`. A TF 2 tensor is defined by its dimensionality, as illustrated here:

```
scalar number:         a zeroth-order tensor
vector:                a first-order tensor
matrix:                a second-order tensor
3-dimensional array: a 3rd order tensor
```

The next section discusses some of the data types that are available in TF 2, followed by a section that discusses TF 2 primitive types.

TF 2 Data Types

TF 2 supports the following data types (similar to the supported data types in TensorFlow 1.x):

- `tf.float32`
- `tf.float64`
- `tf.int8`
- `tf.int16`
- `tf.int32`
- `tf.int64`
- `tf.uint8`
- `tf.string`
- `tf.bool`

The data types in the preceding list are self-explanatory: two floating point types, four integer types, one unsigned integer type, one string type, and one Boolean type. As you can see, there is a 32-bit and a 64-bit floating point type, and integer types that range from 8-bit through 64-bit.

TF 2 Primitive Types

TF 2 supports `tf.constant()` and `tf.Variable()` as primitive types. Notice the capital "V" in `tf.Variable()`: this indicates a TF 2 class (which is not the case for a lowercase initial letter such as `tf.constant()`).

A TF 2 *constant* is an immutable value, and a simple example is shown here:

```
aconst = tf.constant(3.0)
```

A TF 2 *variable* is a "trainable value" in a TF 2 graph. For example, the slope m and y-intercept b of a best-fitting line for a dataset consisting of points in the Euclidean plane are two examples of trainable values. Some examples of TF variables are shown here:

```
b = tf.Variable(3, name="b")
x = tf.Variable(2, name="x")
z = tf.Variable(5*x, name="z")

W = tf.Variable(20)
lm = tf.Variable(W*x + b, name="lm")
```

Notice that b, x, and z are defined as TF variables. In addition, b and x are initialized with numeric values, whereas the value of the variable z is an expression that depends on the value of x (which equals 2).

CONSTANTS IN TF 2

Here is a short list of some properties of TF 2 constants:

- initialized during their definition
- cannot change their value ("immutable")
- can specify their name (optional)
- the type is required (ex: tf.float32)
- are not modified during training

Listing 1.1 displays the contents of tf2_constants1.py, which illustrates how to assign and print the values of some TF 2 constants.

LISTING 1.1: tf2_constants1.py

```
import tensorflow as tf

scalar = tf.constant(10)
vector = tf.constant([1,2,3,4,5])
matrix = tf.constant([[1,2,3],[4,5,6]])
cube   = tf.constant([[[1],
[2],[3]],[[4],[5],[6]],[[7],[8],[9]]])

print(scalar.get_shape())
print(vector.get_shape())
print(matrix.get_shape())
print(cube.get_shape())
```

Listing 1.1 contains four `tf.constant()` statements that define TF 2 tensors of dimension 0, 1, 2, and 3, respectively. The second part of Listing 1.1 contains four `print()` statements that display the shape of the four TF 2 constants that are defined in the first section of Listing 1.1. The output from Listing 1.1 is here:

```
()
(5,)
(2, 3)
(3, 3, 1)
```

Listing 1.2 displays the contents of `tf2_constants2.py`, which illustrates how to assign values to TF 2 constants and then print those values.

LISTING 1.2: tf2_constants2.py

```
import tensorflow as tf

x = tf.constant(5,name="x")
y = tf.constant(8,name="y")

@tf.function
def calc_prod(x, y):
  z = 2*x + 3*y
  return z

result = calc_prod(x, y)
print('result =',result)
```

Listing 1.2 defines a "decorated" (shown in bold) Python function `calc_prod()` with TF 2 code that would otherwise be included in a TF 1.x `tf.Session()` code block. Specifically, `z` would be included in a `sess.run()` statement, along with a `feed_dict` that provides values for `x` and `y`. Fortunately, a decorated Python function in TF 2 makes the code look like "normal" Python code.

VARIABLES IN TF 2

TF 2.0 eliminates global collections and their associated APIs, such as `tf.get_variable`, `tf.variable_scope`, and `tf.initializers.global_variables`. Whenever you need a `tf.Variable` in TF 2, construct and initialize it directly, as shown here:

```
tf.Variable(tf.random.normal([2, 4])
```

Listing 1.3 displays the contents of `tf2_variables.py`, which illustrates how to compute values involving TF constants and variables in a `with` code block.

LISTING 1.3: tf2_variables.py

```
import tensorflow as tf

v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
print("v.value():", v.value())
print("")
print("v.numpy():", v.numpy())
print("")

v.assign(2 * v)
v[0, 1].assign(42)
v[1].assign([7., 8., 9.])
print("v:",v)
print("")

try:
  v[1] = [7., 8., 9.]
except TypeError as ex:
  print(ex)
```

Listing 1.3 defines a TF 2 variable `v` and prints its value. The next portion of Listing 1.3 updates the value of `v` and prints its new value. The last portion of Listing 1.3 contains a `try/except` block that attempts to update the value of `v[1]`. The output from Listing 1.3 is here:

```
v.value(): tf.Tensor(
[[1. 2. 3.]
 [4. 5. 6.]], shape=(2, 3), dtype=float32)

v.numpy(): [[1. 2. 3.]
 [4. 5. 6.]]

v: <tf.Variable 'Variable:0' shape=(2, 3) dtype=float32,
numpy=
array([[ 2., 42.,  6.],
       [ 7.,  8.,  9.]], dtype=float32)>

'ResourceVariable' object does not support item assignment
```

This concludes the quick tour involving TF 2 code that contains various combinations of TF constants and TF variables. The next few sections delve into more details regarding the TF primitive types that you saw in the preceding sections.

THE `tf.rank()` API

The *rank* of a TF 2 tensor is the dimensionality of the tensor, whereas the *shape* of a tensor is the number of elements in each dimension. Listing 1.4 displays the contents of `tf2_rank.py`, which illustrates how to find the rank of TF 2 tensors.

LISTING 1.4: tf2_rank.py

```
import tensorflow as tf # tf2_rank.py

A = tf.constant(3.0)
B = tf.fill([2,3], 5.0)
C = tf.constant([3.0, 4.0])

@tf.function
def show_rank(x):
  return tf.rank(x)

print('A:',show_rank(A))
print('B:',show_rank(B))
print('C:',show_rank(C))
```

Listing 1.4 contains familiar code for defining the TF constant `A`, followed by the TF tensor `B`, which is a 2x3 tensor in which every element has the value 5. The TF tensor `C` is a 1x2 tensor with the values 3.0 and 4.0.

The next code block defines the decorated Python function `show_rank()`, which returns the rank of its input variable. The final section invokes `show_rank()` with `A` and then with `B`. The output from Listing 1.4 is here:

```
A: tf.Tensor(0, shape=(), dtype=int32)
B: tf.Tensor(2, shape=(), dtype=int32)
C: tf.Tensor(1, shape=(), dtype=int32)
```

THE `tf.shape()` API

The *shape* of a TF 2 tensor is the number of elements in each dimension of a given tensor.

Listing 1.5 displays the contents of `tf2_getshape.py`, which illustrates how to find the shape of TF 2 tensors.

LISTING 1.5: tf2_getshape.py

```
import tensorflow as tf

a = tf.constant(3.0)
print("a shape:",a.get_shape())

b = tf.fill([2,3], 5.0)
print("b shape:",b.get_shape())

c = tf.constant([[1.0,2.0,3.0], [4.0,5.0,6.0]])
print("c shape:",c.get_shape())
```

Listing 1.5 contains the definition of the TF constant `a` whose value is 3.0. Next, the TF variable `b is initialized as a 2x3 tensor whose six values are all 5`, followed by the constant `c` whose value is

`[[1.0,2.0,3.0],[4.0,5.0,6.0]]`. The three `print()` statements display the values of `a`, `b`, and `c`. The output from Listing 1.5 is here:

```
a shape: ()
b shape: (2, 3)
c shape: (2, 3)
```

Shapes that specify a 0-D Tensor (scalar) are numbers (9, -5, 2.34, and so forth), [], and (). As another example, Listing 1.6 displays the contents of `tf2_shapes.py`, which contains an assortment of tensors and their shapes.

LISTING 1.6: tf2_shapes.py

```
import tensorflow as tf

list_0 = []
tuple_0 = ()
print("list_0:",list_0)
print("tuple_0:",tuple_0)

list_1 = [3]
tuple_1 = (3)
print("list_1:",list_1)
print("tuple_1:",tuple_1)

list_2 = [3, 7]
tuple_2 = (3, 7)
print("list_2:",list_2)
print("tuple_2:",tuple_2)

any_list1  = [None]
any_tuple1 = (None)
print("any_list1:",any_list1)
print("any_tuple1:",any_tuple1)

any_list2 = [7,None]
any_list3 = [7,None,None]
print("any_list2:",any_list2)
print("any_list3:",any_list3)
```

Listing 1.6 contains simple lists and tuples of various dimensions in order to illustrate the difference between these two types. The output from Listing 1.6 is probably what you would expect and it's shown here:

```
list_0: []
tuple_0: ()
list_1: [3]
tuple_1: 3
list_2: [3, 7]
tuple_2: (3, 7)
any_list1: [None]
```

```
any_tuple1: None
any_list2: [7, None]
any_list3: [7, None, None]
```

VARIABLES IN TF 2 (REVISITED)

TF 2 variables can be updated during backward error propagation (also called "backprop," which is discussed later in this book). TF 2 variables can also be saved and then restored at a later point in time. The following list contains some properties of TF 2 variables:

- initial value is optional
- must be initialized before graph execution
- updated during training
- constantly recomputed
- they hold values for weights and biases
- in-memory buffer (saved/restored from disk)

Here are some simple examples of TF 2 variables:

```
b = tf.Variable(3, name='b')
x = tf.Variable(2, name='x')
z = tf.Variable(5*x, name="z")

W = tf.Variable(20)
lm = tf.Variable(W*x + b, name="lm")
```

Notice that the variables `b`, `x`, and `W` specify constant values, whereas the variables `z` and `lm` specify expressions that are defined in terms of other variables. If you are familiar with linear regression, you undoubtedly noticed that the variable `lm` ("linear model") defines a line in the Euclidean plane. Other properties of TF 2 variables are listed as follows:

- a tensor that's updateable via operations
- exist outside the context of `session.run`
- like a "regular" variable
- holds the learned model parameters
- variables can be shared (or non-trainable)
- used for storing/maintaining state
- internally stores a persistent tensor
- you can read/modify the values of the tensor
- multiple workers see the same values for `tf.Variables`
- the best way to represent shared, persistent state manipulated by your program

TF 2 also provides the method `tf.assign()` in order to modify values of TF 2 variables; be sure to read the relevant code sample later in this chapter so that you learn how to use this API correctly.

TF 2 Variables versus Tensors

Keep in mind the following distinction between TF variables and TF tensors: TF *variables* represent your model's trainable parameters (e.g., weights and biases of a neural network), whereas TF *tensors* represent the data fed into your model and the intermediate representations of that data as it passes through your model.

In the next section, you will learn about the `@tf.function` "decorator" for Python functions and how it can improve performance.

WHAT IS `@tf.function` IN TF 2?

TF 2 introduced the `@tf.function` "decorator" for Python functions that defines a graph and performs session execution: it's sort of a "successor" to `tf.Session()` in TF 1.x. Since graphs can still be useful, `@tf.function` transparently converts Python functions into functions that are "backed" by graphs. This decorator also converts tensor-dependent Python control flow into TF control flow, and also adds control dependencies to order read and write operations to a TF 2 state. Remember that `@tf.function` works best with TF 2 operations instead of `NumPy` operations or Python primitives.

In general, you won't need to decorate functions with @tf.function; use it to decorate high-level computations, such as one step of training, or the forward pass of a model.

Although TF 2 eager execution mode facilitates a more intuitive user interface, this ease-of-use can be at the expense of decreased performance. Fortunately, the `@tf.function` decorator is a technique for generating graphs in TF 2 code that execute more quickly than eager execution mode.

The performance benefit depends on the types of operations that are performed: matrix multiplication does not benefit from the use of `@tf.function`, whereas optimizing a deep neural network can benefit from `@tf.function`.

How Does `@tf.function` Work?

Whenever you decorate a Python function with `@tf.function`, TF 2 automatically builds the function in graph mode. If a Python function that is decorated with `@tf.function` invokes other Python functions that are not decorated with `@tf.function`, then the code in those "non-decorated" Python functions will also be included in the generated graph.

Another point to keep in mind is that a `tf.Variable` in eager mode is actually a "plain" Python object: this object is destroyed when it's out of scope. On the other hand, a `tf.Variable` object defines a persistent object if the function is decorated via `@tf.function`. In this scenario, eager mode is disabled and the `tf.Variable` object defines a node in a persistent TF 2 graph. Consequently, a function that works in eager mode without annotation can fail when it is decorated with `@tf.function`.

A Caveat about `@tf.function` in TF 2

If constants are defined *before* the definition of a decorated Python function, you can print their values inside the function using the Python `print()` function. On the other hand, if constants are defined *inside* the definition of a decorated Python function, you can print their values inside the function using the TF 2 `tf.print()` function. Consider this code block:

```
import tensorflow as tf

a = tf.add(4, 2)

@tf.function
def compute_values():
  print(a) # 6

compute_values()

# output:
# tf.Tensor(6, shape=(), dtype=int32)
```

As you can see, the correct result is displayed (shown in bold). However, if you define constants *inside* a decorated Python function, the output contains types and attributes but *not* the execution of the addition operation. Consider the following code block:

```
import tensorflow as tf

@tf.function
def compute_values():
  a = tf.add(4, 2)
  print(a)

compute_values()

# output:
# Tensor("Add:0", shape=(), dtype=int32)
```

The zero in the preceding output is part of the tensor name and not an outputted value. Specifically, `Add:0` is output zero of the `tf.add()` operation. Any additional invocation of `compute_values()` prints nothing. If you want actual results, one solution is to return a value from the function, as shown here:

```
import tensorflow as tf

@tf.function
def compute_values():
  a = tf.add(4, 2)
  return a
```

```
result = compute_values()
print("result:", result)
```

The output from the preceding code block is here:

```
result: tf.Tensor(6, shape=(), dtype=int32)
```

A second solution involves the TF `tf.print()` function instead of the Python `print()` function, as shown in bold in this code block:

```
@tf.function
def compute_values():
  a = tf.add(4, 2)
  tf.print(a)
```

A third solution is to cast the numeric values to Tensors if they do not affect the shape of the generated graph, as shown here:

```
import tensorflow as tf

@tf.function
def compute_values():
  a = tf.add(tf.constant(4), tf.constant(2))
  return a

result = compute_values()
print("result:", result)
```

The `tf.print()` Function and Standard Error

There is one more detail to remember: the Python `print()` function "sends" output to something called "standard output" that is associated with a file descriptor whose value is 1; on the other hand, `tf.print()` sends output to "standard error" that is associated with a file descriptor whose value is 2. In programming languages such as C, only errors are sent to standard error, so keep in mind that the behavior of `tf.print()` differs from the convention regarding standard out and standard error. The following code snippets illustrate this difference:

```
python3 file_with_print.py    1>print_output
python3 file_with_tf.print.py 2>tf.print_output
```

If your Python file contains both `print()` and `tf.print()`, you can capture the output as follows:

```
python3 both_prints.py 1>print_output 2>tf.print_output
```

However, keep in mind that the preceding code snippet might also redirect *real* error messages to the file `tf.print_output`.

WORKING WITH `@tf.function` IN TF 2

The preceding section explained how the output will differ depending on whether you use the Python `print()` function versus the `tf.print()` function in TF 2 code, where the latter function also sends output to standard error instead of standard output.

This section contains several examples of the `@tf.function` decorator in TF 2 to show you some nuances in behavior that depend on where you define constants and whether you use the `tf.print()` function or the Python `print()` function. Also keep in mind the comments in the previous section regarding `@tf.function`, as well as the fact that you don't need to use `@tf.function` in all your Python functions.

An Example without `@tf.function`

Listing 1.7 displays the contents of `tf2_simple_function.py`, which illustrates how to define a Python function with TF 2 code.

LISTING 1.7: tf2_simple_function.py

```
import tensorflow as tf

def func():
  a = tf.constant([[10,10],[11.,1.]])
  b = tf.constant([[1.,0.],[0.,1.]])
  c = tf.matmul(a, b)
  return c

print(func().numpy())
```

The code in Listing 1.7 is straightforward: a Python function `func()` defines two TF 2 constants, computes their product, and returns that value.

Since TF 2 works in eager mode by default, the Python function `func()` is treated as a "normal" function. Launch the code and you will see the following output:

```
[[20. 30.]
 [22. 3.]]
```

An Example with `@tf.function`

Listing 1.8 displays the contents of `tf2_at_function.py`, which illustrates how to define a decorated Python function with TF code.

LISTING 1.8: tf2_at_function.py

```
import tensorflow as tf

@tf.function
def func():
  a = tf.constant([[10,10],[11.,1.]])
  b = tf.constant([[1.,0.],[0.,1.]])
  c = tf.matmul(a, b)
  return c

print(func().numpy())
```

Listing 1.8 defines a decorated Python function: the rest of the code is identical to Listing 1.7. However, because of the `@tf.function` annotation, the Python `func()` function is "wrapped" in a `tensorflow.python.eager.def_function.Function` object. The Python function is assigned to the `.python_function` property of the object.

When `func()` is invoked, the graph construction begins. Only the Python code is executed, and the behavior of the function is traced so that TF 2 can collect the required data to construct the graph. The output is shown here:

```
[[20. 30.]
 [22.  3.]]
```

Overloading Functions with `@tf.function`

If you have worked with programming languages such as Java and C++, you are already familiar with the concept of "overloading" a function. If this term is new to you, the idea is simple: an overloaded function is a function that can be invoked with different data types. For example, you can define an overloaded "add" function that can add two numbers as well as "add" (i.e., concatenate) two strings.

If you're curious, overloaded functions in various programming languages are implemented via "name mangling," which means that the signature (the parameters and their data types for the function) is appended to the function name in order to generate a unique function name. This happens "under the hood," which means that you don't need to worry about the implementation details.

Listing 1.9 displays the contents of `tf2_overload.py`, which illustrates how to define a decorated Python function that can be invoked with different data types.

LISTING 1.9: tf2_overload.py

```
import tensorflow as tf

@tf.function

def add(a):
  return a + a

print("Add 1:             ", add(1))
print("Add 2.3:           ", add(2.3))
print("Add string tensor:", add(tf.constant("abc")))
```

```
c = add.get_concrete_function(tf.TensorSpec(shape=None,
dtype=tf.string))
c(a=tf.constant("a"))
```

Listing 1.9 defines a decorated Python function `add()`, which is preceded by a `@tf.function` decorator. This function can be invoked by passing an integer, a decimal value, or a TF 2 tensor, and the correct result is calculated. Launch the code and you will see the following output:

```
Add 1:              tf.Tensor(2, shape=(), dtype=int32)
Add 2.3:            tf.Tensor(4.6, shape=(), dtype=float32)
Add string tensor: tf.Tensor(b'abcabc', shape=(),
dtype=string)

c: <tensorflow.python.eager.function.ConcreteFunction
object at 0x1209576a0>
```

What Is AutoGraph in TF 2?

`AutoGraph` refers to the conversion from Python code to its graph representation, which is a significant new feature in TF 2. In fact, `AutoGraph` is automatically applied to functions that are decorated with `@tf.function`; this decorator creates callable graphs from Python functions.

`AutoGraph` transforms a subset of Python syntax into its portable, high-performance and language agnostic graph representation, thereby bridging the gap between TF 1.x and TF 2.0. In fact, `AutoGraph` allows you to inspect its auto-generated code with this code snippet. For example, if you define a Python function called `my_product()`, you can inspect its auto-generated code with this snippet:

```
print(tf.autograph.to_code(my_product))
```

In particular, the Python `for/while` construct is implemented in TF 2 via `tf.while_loop` (`break` and `continue` are also supported). The Python `if` construct is implemented in TF 2 via `tf.cond`. The "`for _ in dataset`" is implemented in TF 2 via `dataset.reduce`.

`AutoGraph` also has some rules for converting loops. A `for` loop is converted if the iterable in the loop is a tensor, and a `while` loop is converted if the `while` condition depends on a tensor. If a loop is converted, it will be dynamically "unrolled" with `tf.while_loop`, as well as the special case of a `for x in tf.data.Dataset` (the latter is transformed into `tf.data.Dataset.reduce`). If a loop is not converted, it will be statically unrolled.

`AutoGraph` supports control flow that is nested arbitrarily deep, so you can implement many types of ML programs. Check the online documentation for more information regarding `AutoGraph`.

ARITHMETIC OPERATIONS IN TF 2

Listing 1.10 displays the contents of `tf2_arithmetic.py`, which illustrates how to perform arithmetic operations in TF 2.

LISTING 1.10: tf2_arithmetic.py

```
import tensorflow as tf

@tf.function # replace print() with tf.print()
def compute_values():
  a = tf.add(4, 2)
  b = tf.subtract(8, 6)
  c = tf.multiply(a, 3)
  d = tf.math.divide(a, 6)

  print(a) # 6
  print(b) # 2
  print(c) # 18
  print(d) # 1

compute_values()
```

Listing 1.10 defines the decorated Python function `compute_values()` with simple code for computing the sum, difference, product, and quotient of two numbers via the `tf.add()`, `tf.subtract()`, `tf.multiply()`, and the `tf.math.divide()` APIs, respectively. The four `print()` statements display the values of `a`, `b`, `c`, and `d`. The output from Listing 1.10 is here:

```
tf.Tensor(6,   shape=(), dtype=int32)
tf.Tensor(2,   shape=(), dtype=int32)
tf.Tensor(18,  shape=(), dtype=int32)
tf.Tensor(1.0, shape=(), dtype=float64)
```

CAVEATS FOR ARITHMETIC OPERATIONS IN TF 2

As you can probably surmise, you can also perform arithmetic operations involving TF 2 constants and variables. Listing 1.11 displays the contents of `tf2_const_var.py`, which illustrates how to perform arithmetic operations involving a TF 2 constant and a variable.

LISTING 1.11: tf2_const_var.py

```
import tensorflow as tf

v1 = tf.Variable([4.0, 4.0])
c1 = tf.constant([1.0, 2.0])
```

```
diff = tf.subtract(v1,c1)
print("diff:",diff)
```

Listing 1.11 computes the difference of the TF variable `v1` and the TF constant `c1`, and the output is shown here:

```
diff: tf.Tensor([3. 2.], shape=(2,), dtype=float32)
```

However, if you update the value of `v1` and then print the value of `diff`, it will *not* change. You must reset the value of `diff`, just as you would in other imperative programming languages.

Listing 1.12 displays the contents of `tf2_const_var2.py`, which illustrates how to perform arithmetic operations involving a TF 2 constant and a variable.

LISTING 1.12: tf2_const_var2.py

```
import tensorflow as tf

v1 = tf.Variable([4.0, 4.0])
c1 = tf.constant([1.0, 2.0])

diff = tf.subtract(v1,c1)
print("diff1:",diff.numpy())

# diff is NOT updated:
v1.assign([10.0, 20.0])
print("diff2:",diff.numpy())

# diff is updated correctly:
diff = tf.subtract(v1,c1)
print("diff3:",diff.numpy())
```

Listing 1.12 recomputes the value of `diff` in the final portion of Listing 1.11, after which it has the correct value. The output is shown here:

```
diff1: [3. 2.]
diff2: [3. 2.]
diff3: [9. 18.]
```

TF 2 AND BUILT-IN FUNCTIONS

Listing 1.13 displays the contents of `tf2_math_ops.py`, which illustrates how to perform additional arithmetic operations in a TF graph.

LISTING 1.13: tf2_math_ops.py

```
import tensorflow as tf

PI = 3.141592

@tf.function # replace print() with tf.print()
```

```
def math_values():
  print(tf.math.divide(12,8))
  print(tf.math.floordiv(20.0,8.0))
  print(tf.sin(PI))
  print(tf.cos(PI))
  print(tf.math.divide(tf.sin(PI/4.), tf.cos(PI/4.)))

math_values()
```

Listing 1.13 contains a hard-coded approximation for `PI`, followed by the decorated Python function `math_values()` with five `print()` statements that display various arithmetic results. Note in particular the third output value is a very small number (the correct value is zero). The output from Listing 1.13 is here:

```
1.5
tf.Tensor(2.0,           shape=(), dtype=float32)
tf.Tensor(6.2783295e-07, shape=(), dtype=float32)
tf.Tensor(-1.0,          shape=(), dtype=float32)
tf.Tensor(0.99999964,    shape=(), dtype=float32)
```

Listing 1.14 displays the contents of `tf2_math-ops_pi.py`, which illustrates how to perform arithmetic operations in TF 2.

LISTING 1.14: tf2_math_ops_pi.py

```
import tensorflow as tf
import math as m

PI = tf.constant(m.pi)

@tf.function # repłace print() with tf.print()
def math_values():
  print(tf.math.divide(12,8))
  print(tf.math.floordiv(20.0,8.0))
  print(tf.sin(PI))
  print(tf.cos(PI))
  print(tf.math.divide(tf.sin(PI/4.), tf.cos(PI/4.)))

math_values()
```

Listing 1.14 is almost identical to the code in Listing 1.13: the only difference is that Listing 1.14 specifies a hard-coded value for PI, whereas Listing 1.14 assigns `m.pi` to the value of PI. As a result, the approximated value is one decimal place closer to the correct value of zero. The output from Listing 1.14 is here; notice how the output format differs from Listing 1.13 due to the Python `print()` function:

```
1.5
tf.Tensor(2.0,          shape=(), dtype=float32)
tf.Tensor(-8.742278e-08, shape=(), dtype=float32)
tf.Tensor(-1.0,          shape=(), dtype=float32)
tf.Tensor(1.0,           shape=(), dtype=float32)
```

CALCULATING TRIGONOMETRIC VALUES IN TF

Listing 1.15 displays the contents of `tf2_trig_values.py`, which illustrates how to compute values involving trigonometric functions in TF 2.

LISTING 1.15: tf2_trig_values.py

```
import tensorflow as tf
import math as m

PI = tf.constant(m.pi)

a = tf.cos(PI/3.)
b = tf.sin(PI/3.)
c = 1.0/a # sec(60)
d = 1.0/tf.tan(PI/3.) # cot(60)

@tf.function # this decorator is okay
def math_values():
  print("a:",a)
  print("b:",b)
  print("c:",c)
  print("d:",d)

math_values()
```

Listing 1.14 is straightforward: there are several of the same TF 2 APIs that you saw in Listing 1.13. In addition, Listing 1.14 contains the `tf.tan()` API, which computes the tangent of a number (in radians). The output from Listing 1.14 is here:

```
a: tf.Tensor(0.49999997, shape=(), dtype=float32)
b: tf.Tensor(0.86602545, shape=(), dtype=float32)
c: tf.Tensor(2.0000002,  shape=(), dtype=float32)
d: tf.Tensor(0.57735026, shape=(), dtype=float32)
```

CALCULATING EXPONENTIAL VALUES IN TF 2

Listing 1.15 displays the contents of `tf2_exp_values.py`, which illustrates how to compute values involving additional trigonometric functions in TF 2.

LISTING 1.15: tf2_exp_values.py

```
import tensorflow as tf

a  = tf.exp(1.0)
```

```
b  = tf.exp(-2.0)
s1 = tf.sigmoid(2.0)
s2 = 1.0/(1.0 + b)
t2 = tf.tanh(2.0)

@tf.function # this decorator is okay
def math_values():
  print('a: ', a)
  print('b: ', b)
  print('s1:', s1)
  print('s2:', s2)
  print('t2:', t2)

math_values()
```

Listing 1.15 starts with the TF 2 APIs `tf.exp()`, `tf.sigmoid()`, and `tf.tanh()` that compute the exponential value of a number, the sigmoid value of a number, and the hyperbolic tangent of a number, respectively. The output from Listing 1.15 is here:

```
a:  tf.Tensor(2.7182817,  shape=(), dtype=float32)
b:  tf.Tensor(0.13533528, shape=(), dtype=float32)
s1: tf.Tensor(0.880797,   shape=(), dtype=float32)
s2: tf.Tensor(0.880797,   shape=(), dtype=float32)
t2: tf.Tensor(0.9640276,  shape=(), dtype=float32)
```

WORKING WITH STRINGS IN TF 2

Listing 1.16 displays the contents of `tf2_strings.py`, which illustrates how to work with strings in TF 2.

LISTING 1.16: tf2_strings.py

```
import tensorflow as tf

x1 = tf.constant("café")
print("x1:",x1)
tf.strings.length(x1)
print("")

len1 = tf.strings.length(x1, unit="UTF8_CHAR")
len2 = tf.strings.unicode_decode(x1, "UTF8")

print("len1:",len1.numpy())
print("len2:",len2.numpy())
print("")

# String arrays
x2 = tf.constant(["Café", "Coffee", "caffè", "咖啡"])
print("x2:",x2)
print("")
```

```
len3 = tf.strings.length(x2, unit="UTF8_CHAR")
print("len2:",len3.numpy())
print("")

r = tf.strings.unicode_decode(x2, "UTF8")
print("r:",r)
```

Listing 1.16 defines the TF 2 constant `x1` as a string that contains an accent mark. The first `print()` statement displays the first three characters of `x1`, followed by a pair of hexadecimal values that represent the accented "e" character. The second and third `print()` statements display the number of characters in `x1`, followed by the `UTF8` sequence for the string `x1`.

The next portion of Listing 1.16 defines the TF 2 constant `x2` as a first-order TF 2 tensor that contains four strings. The next `print()` statement displays the contents of `x2`, using `UTF8` values for characters that contain accent marks.

The final portion of Listing 1.16 defines `r` as the `Unicode` values for the characters in the string `x2`. The output from Listing 1.14 is here:

```
x1: tf.Tensor(b'caf\xc3\xa9', shape=(), dtype=string)

len1: 4
len2: [ 99  97 102 233]

x2: tf.Tensor([b'Caf\xc3\xa9' b'Coffee' b'caff\xc3\xa8' b'\
xe5\x92\x96\xe5\x95\xa1'], shape=(4,), dtype=string)

len2: [4 6 5 2]

r: <tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102,
102, 101, 101], [99, 97, 102, 102, 232], [21654, 21857]]>
```

Chapter 2 contains a complete code sample with more examples of a `RaggedTensor` in TF 2.

WORKING WITH TENSORS AND OPERATIONS IN TF 2

Listing 1.17 displays the contents of `tf2_tensors_operations.py`, which illustrates how to use various operators with tensors in TF 2.

LISTING 1.17: tf2_tensors_operations.py

```
import tensorflow as tf

x = tf.constant([[1., 2., 3.], [4., 5., 6.]])

print("x:", x)
print("")
print("x.shape:", x.shape)
print("")
print("x.dtype:", x.dtype)
print("")
print("x[:, 1:]:", x[:, 1:])
print("")
print("x[..., 1, tf.newaxis]:", x[..., 1, tf.newaxis])
print("")
print("x + 10:", x + 10)
print("")
print("tf.square(x):", tf.square(x))
print("")
print("x @ tf.transpose(x):", x @ tf.transpose(x))

m1 = tf.constant([[1., 2., 4.], [3., 6., 12.]])
print("m1:                ", m1)
print("m1 + 50:           ", m1 + 50)
print("m1 * 2:            ", m1 * 2)
print("tf.square(m1):     ", tf.square(m1))
```

Listing 1.17 defines the TF tensor x that contains a 2x3 array of real numbers. The bulk of the code in Listing 1.17 illustrates how to display properties of x by invoking `x.shape` and `x.dtype`, as well as the TF function `tf.square(x)`. The output from Listing 1.17 is here:

```
x: tf.Tensor(
[[1. 2. 3.]
 [4. 5. 6.]], shape=(2, 3), dtype=float32)

x.shape: (2, 3)

x.dtype: <dtype: 'float32'>

x[:, 1:]: tf.Tensor(
[[2. 3.]
 [5. 6.]], shape=(2, 2), dtype=float32)

x[..., 1, tf.newaxis]: tf.Tensor(
[[2.]
 [5.]], shape=(2, 1), dtype=float32)

x + 10: tf.Tensor(
[[11. 12. 13.]
 [14. 15. 16.]], shape=(2, 3), dtype=float32)

tf.square(x): tf.Tensor(
[[ 1.  4.  9.]
 [16. 25. 36.]], shape=(2, 3), dtype=float32)

x @ tf.transpose(x): tf.Tensor(
[[14. 32.]
 [32. 77.]], shape=(2, 2), dtype=float32)
```

```
m1:              tf.Tensor(
[[1. 2. 4.]
 [3. 6. 12.]], shape=(2, 3), dtype=float32)

m1 + 50:         tf.Tensor(
[[51. 52. 54.]
 [53. 56. 62.]], shape=(2, 3), dtype=float32)

m1 * 2:          tf.Tensor(
[[ 2.  4.  8.]
 [ 6. 12. 24.]], shape=(2, 3), dtype=float32)

tf.square(m1):   tf.Tensor(
[[  1.   4.  16.]
 [  9.  36. 144.]], shape=(2, 3), dtype=float32)
```

SECOND-ORDER TENSORS IN TF 2 (1)

Listing 1.18 displays the contents of `tf2_elem2.py`, which illustrates how to define a second-order TF tensor and access elements in that tensor.

LISTING 1.18: tf2_elem2.py

```
import tensorflow as tf

arr2 = tf.constant([[1,2],[2,3]])

@tf.function
def compute_values():
  print('arr2: ',arr2)
  print('[0]:  ',arr2[0])
  print('[1]:  ',arr2[1])

compute_values()
```

Listing 1.18 contains the TF constant `arr1` that is initialized with the value `[[1,2],[2,3]]`. The three `print()` statements display the value of `arr1`, the value of the element whose index is 1, and the value of the element whose index is `[1,1]`. The output from Listing 1.18 is here:

```
arr2:   tf.Tensor(
[[1 2]
 [2 3]], shape=(2, 2), dtype=int32)
[0]:   tf.Tensor([1 2], shape=(2,), dtype=int32)
[1]:   tf.Tensor([2 3], shape=(2,), dtype=int32)
```

SECOND-ORDER TENSORS IN TF 2 (2)

Listing 1.19 displays the contents of `tf2_elem3.py`, which illustrates how to define a second-order TF 2 tensor and access elements in that tensor.

LISTING 1.19: tf2_elem3.py

```
import tensorflow as tf

arr3 = tf.constant([[[1,2],[2,3]],[[3,4],[5,6]]])

@tf.function # replace print() with tf.print()
def compute_values():
  print('arr3:   ',arr3)
  print('[1]:    ',arr3[1])
  print('[1,1]:  ',arr3[1,1])
  print('[1,1,0]:',arr3[1,1,0])

compute_values()
```

Listing 1.19 contains the TF constant `arr3` that is initialized with the value `[[[1,2],[2,3]],[[3,4],[5,6]]]`. The four `print()` statements display the value of `arr3`, the value of the element whose index is 1, the value of the element whose index is `[1,1]`, and the value of the element whose index is `[1,1,0]`. The output from Listing 1.19 (adjusted slightly for display purposes) is here:

```
arr3:    tf.Tensor(
[[[1 2]
  [2 3]]

 [[3 4]
  [5 6]]], shape=(2, 2, 2), dtype=int32)
[1]:     tf.Tensor(
[[3 4]
 [5 6]], shape=(2, 2), dtype=int32)
[1,1]:   tf.Tensor([5 6], shape=(2,), dtype=int32)
[1,1,0]: tf.Tensor(5, shape=(), dtype=int32)
```

MULTIPLYING TWO SECOND-ORDER TENSORS IN TF

Listing 1.20 displays the contents of `tf2_mult.py`, which illustrates how to multiply second-order tensors in TF 2.

LISTING 1.20: tf2_mult.py

```
import tensorflow as tf

m1 = tf.constant([[3., 3.]])   # 1x2
m2 = tf.constant([[2.],[2.]]) # 2x1
p1 = tf.matmul(m1, m2)        # 1x1

@tf.function
def compute_values():
  print('m1:',m1)
  print('m2:',m2)
  print('p1:',p1)

compute_values()
```

Listing 1.20 contains two TF constants `m1` and `m2` that are initialized with the values `[[3., 3.]]` and `[[2.],[2.]]`. Due to the nested square brackets, `m1` has shape `1x2`, whereas `m2` has shape `2x1`. Hence, the product of `m1` and `m2` has shape `(1,1)`.

The three `print()` statements display the values of `m1`, `m2`, and `p1`. The output from Listing 1.20 is here:

```
m1: tf.Tensor([[3. 3.]], shape=(1, 2), dtype=float32)
m2: tf.Tensor(
[[2.]
 [2.]], shape=(2, 1), dtype=float32)
p1: tf.Tensor([[12.]], shape=(1, 1), dtype=float32)
```

CONVERT PYTHON ARRAYS TO TF TENSORS

Listing 1.21 displays the contents of `tf2_convert_tensors.py`, which illustrates how to convert a Python array to a TF 2 tensor.

LISTING 1.21: tf2_convert_tensors.py

```
import tensorflow as tf
import numpy as np

x1 = np.array([[1.,2.],[3.,4.]])
x2 = tf.convert_to_tensor(value=x1, dtype=tf.float32)

print ('x1:',x1)
print ('x2:',x2)
```

Listing 1.21 is straightforward, starting with an `import` statement for TensorFlow and one for `NumPy`. Next, the `x_data` variable is a `NumPy` array, and `x` is a TF tensor that is the result of converting `x_data` to a TF tensor. The output from Listing 1.21 is here:

```
x1: [[1. 2.]
 [3. 4.]]
x2: tf.Tensor(
[[1. 2.]
 [3. 4.]], shape=(2, 2), dtype=float32)
```

Conflicting Types in TF 2

Listing 1.22 displays the contents of `tf2_conflict_types.py`, which illustrates what happens when you try to combine incompatible tensors in TF 2.

LISTING 1.22: tf2_conflict_types.py

```
import tensorflow as tf

try:
  tf.constant(1) + tf.constant(1.0)
except tf.errors.InvalidArgumentError as ex:
  print(ex)

try:
  tf.constant(1.0, dtype=tf.float64) + tf.constant(1.0)
except tf.errors.InvalidArgumentError as ex:
  print(ex)
```

Listing 1.22 contains two `try/except` blocks. The first block adds two constants 1 and 1.0, which are compatible. The second block attempts to add the value 1.0 that's declared as a `tf.float64` with 1.0, which are not compatible tensors. The output from Listing 1.22 is here:

```
cannot compute Add as input #1(zero-based) was expected to
be a int32 tensor but is a float tensor [Op:Add] name: add/
cannot compute Add as input #1(zero-based) was expected
to be a double tensor but is a float tensor [Op:Add] name:
add/
```

DIFFERENTIATION AND `tf.GradientTape` IN TF 2

Automatic differentiation (i.e., calculating derivatives) is useful for implementing ML algorithms such as back propagation for training various types of NNs (Neural Networks). During eager execution, the TF 2 context manager `tf.GradientTape` traces operations for computing gradients. This context manager provides a `watch()` method for specifying a tensor that will be differentiated (in the mathematical sense of the word).

The `tf.GradientTape` context manager records all forward-pass operations on a "tape." Next, it computes the gradient by "playing" the tape backward, and then discards the tape after a single gradient computation. Thus, a `tf.GradientTape` can only compute one gradient: subsequent invocations throw a runtime error. Keep in mind that the `tf.GradientTape` context manager only exists in eager mode.

Why do we need the `tf.GradientTape` context manager? Consider deferred execution mode, where we have a graph in which we know how nodes are connected. The gradient computation of a function is performed in two steps: (a) backtracking from the output to the input of the graph, and (b) computing the gradient to obtain the result.

By contrast, in eager execution the only way to compute the gradient of a function using automatic differentiation is to construct a graph. After constructing the graph of the operations executed within the `tf.GradientTape` context manager on some "watchable" element (such as a variable), we can instruct the tape to compute the required gradient. If you want a more detailed explanation, the `tf.GradientTape` documentation page contains an example that explains how and why tapes are needed.

The default behavior for `tf.GradientTape` is to "play once and then discard." However, it's possible to specify a persistent tape, which means that the values are persisted and therefore the tape can be "played" multiple times. The next section contains several examples of `tf.GradientTape`, including an example of a persistent tape.

EXAMPLES OF `tf.GradientTape`

Listing 1.23 displays the contents of `tf2_gradient_tape1.py`, which illustrates how to invoke `tf.GradientTape` in TF 2. This example is one of the simplest examples of using `tf.GradientTape` in TF 2.

LISTING 1.23: tf2_gradient_tape1.py

```
import tensorflow as tf

w = tf.Variable([[1.0]])

with tf.GradientTape() as tape:
  loss = w * w

grad = tape.gradient(loss, w)
print("grad:",grad)
```

Listing 1.23 defines the variable `w`, followed by a `with` statement that initializes the variable `loss` with the expression `w*w`. Next, the variable `grad` is initialized with the derivative that is returned by the tape, and then evaluated with the current value of `w`.

As a reminder, if we define the function `z = w*w`, then the first derivative of `z` is the term `2*w` , and when this term is evaluated with the value of 1.0 for `w`, the result is 2.0. Launch the code in Listing 1.23 and you will see the following output:

```
grad: tf.Tensor([[2.]], shape=(1, 1), dtype=float32)
```

Using the `watch()` Method of `tf.GradientTape`

Listing 1.24 displays the contents of `tf2_gradient_tape2.py`, which also illustrates the use of `tf.GradientTape` with the `watch()` method in TF 2.

LISTING 1.24: tf2_gradient_tape2.py

```
import tensorflow as tf

x = tf.constant(3.0)
```

```
with tf.GradientTape() as g:
  g.watch(x)
  y = 4 * x * x

dy_dx = g.gradient(y, x)
```

Listing 1.24 contains a similar `with` statement as Listing 1.23, but this time a `watch()` method is also invoked to "watch" the tensor `x`. As you saw in the previous section, if we define the function `y = 4*x*x`, then the first derivative of `y` is the term `8*x`; when the latter term is evaluated with the value `3.0` for `x`, the result is 24.0.

Launch the code in Listing 1.24 and you will see the following output:

```
dy_dx: tf.Tensor(24.0, shape=(), dtype=float32)
```

Using Nested Loops with `tf.GradientTape`

Listing 1.25 displays the contents of `tf2_gradient_tape3.py`, which also illustrates how to define nested loops with `tf.GradientTape` in order to calculate the first and the second derivative of a tensor in TF 2.

LISTING 1.25: tf2_gradient_tape3.py

```
import tensorflow as tf

x = tf.constant(4.0)
with tf.GradientTape() as t1:
  with tf.GradientTape() as t2:
    t1.watch(x)
    t2.watch(x)
    z = x * x * x
  dz_dx = t2.gradient(z, x)
d2z_dx2 = t1.gradient(dz_dx, x)

print("First  dz_dx:  ",dz_dx)
print("Second d2z_dx2:",d2z_dx2)

x = tf.Variable(4.0)
with tf.GradientTape() as t1:
  with tf.GradientTape() as t2:
    z = x * x * x
  dz_dx = t2.gradient(z, x)
d2z_dx2 = t1.gradient(dz_dx, x)

print("First  dz_dx:  ",dz_dx)
print("Second d2z_dx2:",d2z_dx2)
```

The first portion of Listing 1.25 contains a nested loop, where the outer loop calculates the first derivative and the inner loop calculates the second derivative of the term `x*x*x` when `x` equals 4. The second portion of Listing

1.25 contains another nested loop that produces the same output with slightly different syntax.

In case you're a bit rusty regarding derivatives, the next code block shows you a function `z`, its first derivative `z'`, and its second derivative `z''`:

```
z   = x*x*x
z'  = 3*x*x
z'' = 6*x
```

When we evaluate `z`, `z'`, and `z''` with the value 4.0 for `x`, the result is 64.0, 48.0, and 24.0, respectively. Launch the code in Listing 1.25 and you will see the following output:

```
First  dz_dx:   tf.Tensor(48.0, shape=(), dtype=float32)
Second d2z_dx2: tf.Tensor(24.0, shape=(), dtype=float32)
First  dz_dx:   tf.Tensor(48.0, shape=(), dtype=float32)
Second d2z_dx2: tf.Tensor(24.0, shape=(), dtype=float32)
```

Other Tensors with `tf.GradientTape`

Listing 1.26 displays the contents of `tf2_gradient_tape4.py`, which illustrates how to use `tf.GradientTape` in order to calculate the first derivative of an expression that depends on a 2x2 tensor in TF 2.

LISTING 1.26: tf2_gradient_tape4.py

```
import tensorflow as tf

x = tf.ones((3, 3))

with tf.GradientTape() as t:
  t.watch(x)
  y = tf.reduce_sum(x)
  print("y:",y)
  z = tf.multiply(y, y)
  print("z:",z)
  z = tf.multiply(z, y)
  print("z:",z)

# the derivative of z with respect to y
dz_dy = t.gradient(z, y)
print("dz_dy:",dz_dy)
```

In Listing 1.26, `y` equals the sum of the elements in the 3x3 tensor `x`, which is 9.

Next, `z` is assigned the term `y*y` and then multiplied again by `y`, so the final expression for `z` (and its derivative) is here:

```
z  = y*y*y
z' = 3*y*y
```

When z' is evaluated with the value 9 for y, the result is 3*9*9, which equals 243. Launch the code in Listing 1.26 and you will see the following output (slightly reformatted for readability):

```
y: tf.Tensor(9.0,        shape=(), dtype=float32)
z: tf.Tensor(81.0,       shape=(), dtype=float32)
z: tf.Tensor(729.0,      shape=(), dtype=float32)
dz_dy: tf.Tensor(243.0, shape=(), dtype=float32)
```

A Persistent Gradient Tape

Listing 1.27 displays the contents of `tf2_gradient_tape5.py`, which illustrates how to define a persistent gradient tape with `tf.GradientTape` in order to calculate the first derivative of a tensor in TF 2.

LISTING 1.27: tf2_gradient_tape5.py

```
import tensorflow as tf

x = tf.ones((3, 3))

with tf.GradientTape(persistent=True) as t:
  t.watch(x)
  y = tf.reduce_sum(x)
  print("y:",y)
  w = tf.multiply(y, y)
  print("w:",w)
  z = tf.multiply(y, y)
  print("z:",z)
  z = tf.multiply(z, y)
  print("z:",z)

# the derivative of z with respect to y
dz_dy = t.gradient(z, y)
print("dz_dy:",dz_dy)
dw_dy = t.gradient(w, y)
print("dw_dy:",dw_dy)
```

Listing 1.27 is almost the same as Listing 1.26: the new sections are displayed in bold. Note that w is the term `y*y` and therefore the first derivative w "is `2*y`. Hence, the values for w and w" are 81 and 18, respectively, when they are evaluated with the value 9.0. Launch the code in Listing 1.27 and you will see the following output (slightly reformatted for readability), where the new output is shown in bold:

```
y: tf.Tensor(9.0,        shape=(), dtype=float32)
w: tf.Tensor(81.0,       shape=(), dtype=float32)
z: tf.Tensor(81.0,       shape=(), dtype=float32)
z: tf.Tensor(729.0,      shape=(), dtype=float32)
```

```
dz_dy: tf.Tensor(243.0, shape=(), dtype=float32)
dw_dy: tf.Tensor(18.0,  shape=(), dtype=float32)
```

This concludes the portion of the chapter that discusses new features of TF 2. The remaining sections discuss migration of TF 1.x code to TF 2.

MIGRATING TF 1.X CODE TO TF 2 CODE (OPTIONAL)

If you do not have any TF 1.x code, this section is optional, yet it might be worth skimming through the material, just in case you need to migrate some code from TF 1.x to TF 2 at some point in the future. In brief, the major changes involve streamlined namespaces, eager execution, no global variables, and functions instead of sessions.

The TF 1.x libraries `tf.app`, `tf.logging`, and `tf.flags` are not available in TF 2. The most significant change is the removal of `tf.contrib` from TF 2: check the contents of `tf.experiment`, which might be the new "home" for code that was previously in `tf.contrib`.

Furthermore, since the `tf.Session` class has been removed, its "run" method has also been removed. There are other simplifications that will become apparent as you read the samples in this book.

The APIs in TF 2 look quite different from TF 1.x, and they have a more "Python-like" style. Some TensorFlow 1.x APIs are not available in TF 2, including `equal()`, `eval()`, `name_scope()`, `reduce_sum()`, and `summary.scalar()`.

In order to migrate from TF 1.x to TF 2, remove the graph definition, the session execution, variables initialization, variable sharing via scopes, as well as any `tf.control_dependencies`. There are several techniques for converting older TF 1.x code, as described in the following subsections.

Two Conversion Techniques from TF 1.x to TF 2

The simplest option (let's call it option #1) is to launch the upgrade/conversion script that performs an initial pass to convert your TF 1.x code to TF 2. This script inserts `tf.compat.v1` endpoints to access placeholders, sessions, collections, and other TF 1.x functionality.

By contrast, option #2 involves "pure" TF 2 functionality, which is not the case for the conversion script; hence, option #2 is recommended, and the details are discussed in the next section.

NOTE *Do* not *make manual upgrade-related changes to TF 1.x code before launching the conversion script, which expects TF 1.x syntax (it fails if you make manual updates).*

CONVERTING TO PURE TF 2 FUNCTIONALITY

The most commonly required steps for converting TF 1.x code to pure TF 2 code are listed here:

1. replace `tf.Session.run` calls with a Python function
2. `feed_dict` and `tf.placeholders` become function arguments
3. fetches become the function's return value
4. add a `@tf.function` decorator to the Python function
5. use `tf.Variable` instead of `tf.get_variable`

Additional conversion steps include: combining `tf.data.Dataset` and `@tf.function`, and using Keras layers and models (discussed in Chapter 4) to manage variables.

Converting Sessions to Functions

Let's look at an example of TF 1.x and its TF 2 counterpart, such as the following code snippet from TensorFlow 1.x:

```
outputs = session.run(f(placeholder),feed_
dict={placeholder:input})
```

The equivalent code in TF 2 is here, where `f` is a decorated Python function:

```
outputs = f(input)
```

Combine `tf.data.Dataset` and `@tf.function`

Chapter 3 is devoted to TF 2 `tf.data.Dataset` code samples, so this section will make more sense after you have read that chapter. When iterating over training data that fits in memory, use regular Python iteration. Otherwise, use `tf.data.Dataset` to stream training data from the disk. Datasets are iterables (not iterators), and work the same as other Python iterables in eager mode.

You can fully utilize dataset asynchronous prefetching/streaming features by wrapping your code in `@tf.function()`, which replaces Python iteration with the equivalent graph operations using `AutoGraph`.

Use Keras Layers and Models to Manage Variables

When possible, use Keras layers and models to manage variables, because they recursively collect dependent variables. This functionality facilitates the handling of local variables.

In TF 2, Keras layers and models inherit from `tf.train.Checkpointable` and are also integrated with `@tf.function`; this integration allows you to directly checkpoint or export `SavedModels` from Keras objects. If you are familiar with Keras, you'll be interested to know that the Keras `.fit()` API is not required in order to leverage these integrations. Chapters 4 and 5 contain Keras-based code samples for linear regression and logistic regression, respectively.

THE TENSORFLOW UPGRADE SCRIPT (OPTIONAL)

TensorFlow provides an upgrade script `tf_upgrade_v2` that is included in the most recent TF 1.x installations (such as TF 1.12).

Note that the tf_upgrade_v2 upgrade script is currently only available through the pip *command for installing TF 1.13 or higher (and nightly TF 2 builds): it's not available via the* pip3 *command.*

In order to create a TF 1.x environment that does not overlap with a TF 2 environment, you can use the `conda` command (which is part of the `Anaconda` distribution that is freely available).

Another option is to use the `virtual_env` command. Check the online documentation for instructions regarding either of these two commands.

NOTE *Do* not *make manual upgrade modifications to TF 1.x scripts, because those changes can cause the upgrade script to fail (because it assumes that your code contains TF 1.x syntax).*

When you have everything ready, the following command converts the TF 1.x code in `oldtf.py` to the Python script `newtf.py` that contains TF 2 code:

```
tf_upgrade_v2 --infile oldtf.py --outfile newtf.py
```

The preceding command creates the Python script `newtf.py` containing the upgraded TF 2 code. In addition, this utility generates the text file `report.txt` that contains a list of errors (if any) that the upgrade script cannot fix.

The upgrade can also upgrade an entire directory tree, as shown here:

```
# upgrade the .py files and copy all the other files to the
outtree
tf_upgrade_v2 --intree oldcode --outtree newcode-upgraded
```

As a variation, you can invoke this upgrade script on a directory tree and also keep the upgraded Python scripts in the same directory, as shown here:

```
# just upgrade the .py files
tf_upgrade_v2 --intree coolcode --outtree coolcode-upgraded
--copyotherfiles False
```

SUMMARY

This chapter introduced you to TF 2, a very brief view of its architecture, and some of the tools that are part of the TF 2 "family." Then you learned how to write basic Python scripts containing TF 2 code with TF constants and variables. You also learned how to perform arithmetic operations and also some built-in TF functions.

Next, you learned how to calculate trigonometric values, how to use for loops, and how to calculate exponential values. You also saw how to perform various operations on second-order TF 2 tensors.

In addition, you saw code samples that illustrate how to use some of the new features of TF 2, such as the `@tf.function` decorator and `tf.GradientTape`. Finally, you learned how to make some common changes when migrating TF 1.x code to TF 2 code.

CHAPTER 2

USEFUL TF 2 APIs

This chapter focuses on TF 2 APIs that will be useful for various tasks in your TF 2 code. Although an entire chapter devoted to APIs seems rather dry, there is a simple reason for doing so: you need *all* the APIs in this chapter if you continue learning about TensorFlow beyond this book. In addition, this "one-stop" chapter makes it easier for you to find these TF 2 APIs. At a minimum, please skim through the material in this chapter to make note of the TF 2 APIs that are discussed.

The first part of this chapter briefly discusses some tensor operations (such as multiplying tensors) and also how to create `for` loops and `while` loops in TensorFlow. Recall from Chapter 1 that TF 2 uses eager execution as the default execution, whereas TF 1.x uses deferred execution.

The second part of this chapter contains a collection of TF 2 code samples that show you how to use various APIs that are commonly used in machine learning. Specifically, you will see how to use the `tf.random_normal()` API for generating random numbers (which is useful for initializing the weights of edges in neural networks).

You will see examples of the `tf.argmax()` API for finding the index of each row (or column) that contains the maximum value in each row (or column), which is used for calculating the accuracy of the training process involving various algorithms. In addition, you will learn about the `tf.range()` API, which is similar to the `NumPy` `linspace()` API.

The third portion of this chapter discusses another set of TF 2 APIs, including `reduce_mean()` and `equal()`, both of which are useful for calculating the accuracy of a trained neural network (in conjunction with `tf.argmax()`). You will also learn about the `truncated_normal()` API, which is a variant of the `tf.random_normal()` API, and the `one_hot()` API for encoding data in a particular fashion (i.e., the digit 1 in one position and zero in all other positions of a vector).

One of the most frequently used APIs is `reshape()`, which you will see in any TF 2 code that involves training a CNN (Convolutional Neural Network). After you have completed this section of the chapter, navigate to the following URL that contains a massive collection of TF 2 APIs: *https://www.tensorflow.org/api_docs/python/tf*

The last section of this chapter is about launching TensorBoard from the command line. You will also learn about Google Colaboratory, which is a fully online Jupyter-based environment, and also how to launch TensorBoard in a Jupyter notebook in Google Colaboratory.

TF 2 TENSOR OPERATIONS

TF 2 supports many arithmetic operations on TensorFlow tensors, such as adding, multiplying, dividing, and subtracting tensors. The preceding operations are performed on an element-by-element basis of two tensors. For example, adding two 2x2 tensors involves four additions, whereas adding two 4x4 tensors involves sixteen additions.

The TF 2 `tf.argmax()` API enables you to find the maximum value of each row (or each column) of a two-dimensional TF 2 tensor. This API is used while training CNNs (Convolutional Neural Networks) as part of the calculation of the number of images (in the case of MNIST) that are correctly identified during the training phase. The TF 2 `tf.argmin()` is similar to the `tf.argmax()` API, except that minimum values are found instead of maximum values.

TF 2 provides statistical methods such as `tf.reduce_mean()` and `tf.random_normal()` for calculating the mean of a set of numbers and randomly selecting numbers from a normal distribution. The `tf.truncated_normal()` API is similar to the `tf.random_normal()` API, with the added constraint that the selected numbers must be in a specified range (which is specified by you).

Now let's look at the code samples in the next two sections that show you simple examples of a `for` loop and a `while` loop in TF 2.

USING `for` LOOPS IN TF 2

Listing 2.1 displays the contents of `tf2_forloop1.py`, which illustrates how to create a simple `for` loop in a TF 2 graph.

LISTING 2.1: tf2_forloop1.py

```
import tensorflow as tf

x = tf.Variable(0, name='x')

for i in range(5):
  x = x + 1
  print("x:",x)
```

As you can see, Listing 2.1 contains simple Python code (except for the declaration of the tensor `x`). Listing 2.1 initializes the TF 2 variable `x` with the value 0, followed by a `for` loop that iterates through the values 1 through 5. During each iteration of the loop, the variable `x` is incremented by 1 and its value is printed. The output from Listing 2.1 is here:

```
1
2
3
4
5
```

USING `while` LOOPS IN TF 2

Listing 2.2 displays the contents of `tf2_while_loop.py`, which illustrates how to create a `while` loop in TF 2.

LISTING 2.2: tf2_while_loop.py

```
import tensorflow as tf

a = tf.constant(12)

while not tf.equal(a, 1):
  if tf.equal(a % 2, 0):
    a = a / 2
  else:
    a = 3 * a + 1
  print(a)
```

Listing 2.2 defines the TF 2 constant `a` whose value is 12. The next portion of Listing 2.2 is a `while` loop that contains an `if/else` statement. If the value of `a` is even, then `a` is replaced by half its value. If `a` is odd, then its value is tripled and incremented by 1. Launch the code in Listing 2.2 and you will see the following output:

```
tf.Tensor(6.0, shape=(), dtype=float64)
tf.Tensor(3.0, shape=(), dtype=float64)
tf.Tensor(10.0,shape=(), dtype=float64)
tf.Tensor(5.0, shape=(), dtype=float64)
tf.Tensor(16.0,shape=(), dtype=float64)
tf.Tensor(8.0, shape=(), dtype=float64)
tf.Tensor(4.0, shape=(), dtype=float64)
tf.Tensor(2.0, shape=(), dtype=float64)
tf.Tensor(1.0, shape=(), dtype=float64)
```

TF 2 OPERATIONS WITH RANDOM NUMBERS

TF 2 provides APIs for generating random numbers, such as the TF 2 `tf.random_normal()` API that generates random numbers from a normal distribution. Listing 2.3 displays the contents of `tf2_normal_dist.py`, which illustrates how to use the `tf.random_normal()` method in TF 2.

LISTING 2.3: tf2_normal_dist.py

```
import tensorflow as tf

# normal distribution:
w = tf.Variable(tf.random_normal([784, 10], stddev=0.01))

# mean of an array:
b = tf.Variable([10,20,30,40,50,60],name='t')

print("w: ",w)
print("b: ",tf.reduce_mean(input_tensor=b))
```

Listing 2.3 defines the TF 2 variables `w` (initialized with random values) and `b` (initialized with hard-coded values). The TF 2 variable `w` has dimensions 784x10 and b is a 1x6 tensor. Launch the code in Listing 2.3 and you will see the output shown here:

```
w:  <tf.Variable 'Variable:0' shape=(784, 10)
dtype=float32, numpy=
array([[ 0.00407915, -0.00796624, -0.01256408, ...,
         0.01846658,
        -0.00702356,  0.02048219],
       [ 0.00358197, -0.00531838, -0.01946299, ...,
         0.00724312,
         0.00584369,  0.00208779],
       [-0.00771784,  0.00230517, -0.00738808, ...,
        -0.01874011,
        -0.00284803, -0.00042984],
       ...,
       [ 0.00850285,  0.00289324,  0.00047594, ...,
        -0.0062794 ,
        -0.01276   , -0.01168498],
       [ 0.00468423,  0.00165335,  0.00315462, ...,
        -0.01164965,
        -0.00566464, -0.00804143],
       [-0.00787143,  0.00773228, -0.00716571, ...,
         0.00040842,
         0.00976203,  0.00791298]], dtype=float32)>
b:  tf.Tensor(35, shape=(), dtype=int32)
```

Listing 2.4 displays the contents of `random_normal.py`, which illustrates how to use the `tf.random_normal()` method in TF 2.

LISTING 2.4: tf2_random_normal.py

```
import tensorflow as tf
```

```
# initialize a 6x3 array of random numbers:
values = {'weights':tf.Variable(tf.random_normal([6,3]))}

print("values:")
print(values['weights'])
```

Listing 2.4 initializes the `values` variable with the element `weights`, which is initialized as a TF 2 variable that comprises a 6x3 tensor containing randomly selected values from a normal distribution. The output from launching the code in Listing 2.4 is here:

```
<tf.Variable 'Variable:0' shape=(6, 3) dtype=float32, numpy=
array([[-0.9062903 , -0.20363109,  0.46733373],
       [ 1.3933249 ,  1.0044192 ,  0.4911133 ],
       [-1.1827736 , -1.7746108 ,  0.17291453],
       [-0.17107153, -0.15072137, -0.7849119 ],
       [-0.5893343 , -1.8309714 , -0.42436346],
       [ 0.49252385,  0.04508299,  1.1422006 ]],
dtype=float32)>
```

Listing 2.5 displays the contents of `tf2_array1.py`, which illustrates how to convert a `NumPy` array to a TF 2 tensor.

LISTING 2.5: tf2_array1.py

```
import tensorflow as tf
import numpy as np

import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'

# create a Python array:
array_1d = np.array([1.3, 1, 4.0, 23.5])
tf_tensor = tf.convert_to_tensor(value=array_1d, dtype=tf.
float64)

print(tf_tensor)
print(tf_tensor[0])
print(tf_tensor[2])
```

Listing 2.5 defines the `NumPy` variable `array_1d` that is initialized as an array of four real numbers. Next, the TF 2 variable `tf_tensor` is assigned the result of converting the `NumPy` variable `array_1d` to a TF 2 tensor. The output from launching the code in Listing 2.5 is here:

```
tf.Tensor([ 1.3  1.   4.  23.5], shape=(4,), dtype=float64)
tf.Tensor(1.3, shape=(), dtype=float64)
tf.Tensor(4.0, shape=(), dtype=float64)
```

TF 2 TENSORS AND MAXIMUM VALUES

The TF 2 `tf.argmax()` API determines the index values containing maximum values on a row-wise basis or on a column-wise basis for a TF 2 tensor. Just to be sure you understand the previous statement: the TF 2 `tf.argmax()` API determines the *index* values that contain maximum values and *not* the actual maximum values in those index positions.

As a trivial example, the array `[10,20,30]` contains a minimum value of 10 in index position 0 and a maximum value of 30 in index position 2. Consequently, the TF 2 `tf.argmax()` API returns the value 2, whereas the TF 2 `tf.argmin()` API returns the value 0.

Listing 2.6 displays the contents of `tf2_row_max.py`, which illustrates how to find the maximum value on a row-wise basis in a TF 2 tensor.

LISTING 2.6: tf2_row_max.py

```
import tensorflow as TF 2

# initialize an array of arrays:
a = [[1,2,3], [30,20,10], [40,60,50]]
b = tf.Variable(a, name='b')

print("b: ",tf.argmax(b,1))
```

Listing 2.6 defines the Python variable `a` as a 3x3 array of integers. Next, the variable `b` is initialized as a TF 2 variable that is based on the contents of the Python variable `a`. The output is shown here:

```
b:  tf.Tensor([2 0 1], shape=(3,), dtype=int64)
```

Notice that `tf.argmax()` in Listing 2.6 specifies the value 1 (shown in bold): this indicates that you want the indexes containing *row-wise* maximum values. On the other hand, specify the value 0 if you want the indexes containing *column-wise* maximum values.

THE TF 2 `range()` API

Listing 2.7 displays the contents of `tf2_range1.py`, which illustrates how to use the TF 2 `range()` API, which generates a range of numbers between an initial value and a final value, where consecutive values differ by the same constant.

LISTING 2.7: tf2_range1.py

```
import tensorflow as tf
```

```
a1 = tf.range(3, 18, 3)
a2 = tf.range(0, 8, 2)
a3 = tf.range(-6, 6, 3)
a4 = tf.range(-10, 10, 4)

print('a1:',a1)
print('a2:',a2)
print('a3:',a3)
print('a4:',a4)
```

Listing 2.7 defines the TF 2 variable `a1` that is the set of numbers between 3 (inclusive) and 18 (exclusive), where each number is 3 larger than its predecessor. Similarly, the variables `a2`, `a3`, and `a4` are defined with ranges that have a different start value, and end value, and increment value. The output from launching the code in `tf2_range1.py` is here:

```
a1: tf.Tensor([3    6  9 12 15], shape=(5,), dtype=int32)
a2: tf.Tensor([0    2  4 6],     shape=(4,), dtype=int32)
a3: tf.Tensor([-6  -3  0 3],     shape=(4,), dtype=int32)
a4: tf.Tensor([-10 -6 -2 2 6],   shape=(5,), dtype=int32)
```

OPERATIONS WITH NODES

Listing 2.8 displays the contents of `tf2_addnodes.py`, which illustrates how to add two nodes in TF 2.

LISTING 2.8: tf2_addnodes.py

```
import tensorflow as TF 2

a1 = tf.Variable(7,  tf.float32)
a2 = tf.Variable(13, tf.float32)
a3 = a1 + a2

@tf.function
def compute_values(a1, a2):
  return a1 + a2

result = compute_values(a1, a2)
print("a1 + a2 =",result)
```

Listing 2.8 defines `a3` as the sum of `a1` and `a2`. The next portion of Listing 2.8 defines the decorated Python function `compute_values()`, which computes the sum of its two arguments and returns that sum. The output from launching the code in Listing 2.8 is here:

```
a1 + a2 = tf.Tensor(20, shape=(), dtype=int32)
```

THE `tf.size()`, `tf.shape()`, AND `tf.rank()` APIs

These three TF 2 APIs are somewhat related, so they are included in the same section for easy reference. First of all, the `tf.size()` API returns the number of elements in a TF 2 tensor. Here is a simple example:

```
t = tf.constant([[[1,1,1],[2,2,2]],[[3,3,3],[4,4,4]]])
tf.size(t)  # 12
```

In essence, ignore the square brackets and count the number of elements in order to determine the answer.

Second, the `tf.shape()` API returns the shape of a TF 2 tensor, which is the number of elements in each "dimension." Here is a simple example:

```
t = tf.constant([[[1,1,1],[2,2,2]],[[3,3,3],[4,4,4]]])
tf.shape(t)  # [2, 2, 3]
```

Third, the `tf.rank()` API returns the number of indices required to uniquely select each element of the tensor. The rank is also known as the "order," "degree," or "ndims." Here is a simple example:

```
# the shape of tensor 't' is [2, 2, 3]
t = tf.constant([[[1,1,1],[2,2,2]],[[3,3,3],[4,4,4]]])
# the rank of t is 3
```

Note that the rank of a tensor does *not* equal the rank of a matrix: the latter equals the number of linearly independent rows in that matrix.

You might initially think that the rank in the preceding example is 4 instead of 3 because it's easy to overlook the number of nested square brackets. An easier way to display the preceding TF 2 tensor is shown here:

```
[
  [
    [1,1,1],
    [2,2,2]
  ],
  [
    [3,3,3],
    [4,4,4]
  ]
]
```

As you can see in the preceding layout, the rank of the tensor is 3 because you need to "traverse" 3 levels in order to uniquely identify each element of the tensor.

THE `tf.reduce_prod()` AND `tf.reduce_sum()` APIs

Listing 2.9 displays the contents of `tf2_reduce_prod.py`, which illustrates how to invoke the TF 2 `reduce_prod()` and `reduce_sum()`

APIs for multiplying and adding, respectively, the numeric elements in a TF 2 tensor.

LISTING 2.9: tf2_reduce_prod.py

```
import tensorflow as tf

x = tf.constant([100,200,300], name="x")
y = tf.constant([1,2,3], name="y")

sum_x  = tf.reduce_sum(x, name="sum_x")
prod_y = tf.reduce_prod(y, name="prod_y")
div_xy = tf.math.divide(sum_x, prod_y, name="div_xy")
# 'div' is deprecated in favor of operator or tf.math.divide

print("sum_x: ",sum_x)
print("prod_y:",prod_y)
print("div_xy:",div_xy)

sum_x:  tf.Tensor(600, shape=(), dtype=int32)
prod_y: tf.Tensor(6,   shape=(), dtype=int32)
div_xy: tf.Tensor(100, shape=(), dtype=int32)
```

Listing 2.9 defines the TF 2 constants `x` and `y`, followed by three variables whose values are based on three TF 2 APIs. Specifically, `sum_x` equals the value of invoking the `tf.reduce_sum()` API with the TF 2 constant `x`, which equals the sum of the numeric elements of `x`.

Next, `prod_y` equals the value of invoking the `tf.reduce_prod()` API with the TF 2 constant `y`, which equals the product of the numeric elements of `y`. Finally, `div_xy` equals the ratio of `sum_x` and `prod_y`. The output from launching the code in Listing 2.9 is here:

```
sum_x:  tf.Tensor(600, shape=(), dtype=int32)
prod_y: tf.Tensor(6, shape=(), dtype=int32)
div_xy: tf.Tensor(100, shape=(), dtype=int32)
```

THE `tf.reduce_mean()` API

Listing 2.10 displays the contents of `tf2_reduce_mean.py`, which illustrates how to invoke the `reduce_mean()` API in TF 2.

LISTING 2.10: tf2_reduce_mean.py

```
import tensorflow as tf

x = tf.constant([100,200,300], name='x')
y = tf.constant([1,2,3], name='y')

sum_x  = tf.reduce_sum(x, name="sum_x")
prod_y = tf.reduce_prod(y, name="prod_y")
mean   = tf.reduce_mean([sum_x,prod_y], name="mean")
```

```
print("sum_x: ",sum_x)
print("prod_y:",prod_y)
print("mean:  ",mean)

sum_x:  tf.Tensor(600, shape=(), dtype=int32)
prod_y: tf.Tensor(6,   shape=(), dtype=int32)
mean:   tf.Tensor(303, shape=(), dtype=int32)
```

Listing 2.10 defines the TF 2 constants `x` and `y`, followed by three variables whose values are based on three APIs in TF 2. Specifically, `sum_x` equals the value of invoking the `tf.reduce_sum()` API with the TF 2 constant `x`, which equals the sum of the numeric elements of `x`.

Next, `prod_y` equals the value of invoking the `tf.reduce_prod()` API with the TF 2 constant `y`, which equals the product of the numeric elements of `y`. Finally, `mean` equals the sum of `sum_x` and `prod_y`. The output from launching the code in Listing 2.10 is here:

```
sum_x:  tf.Tensor(600, shape=(), dtype=int32)
prod_y: tf.Tensor(6,   shape=(), dtype=int32)
mean:   tf.Tensor(303, shape=(), dtype=int32)
```

THE `tf.random_normal()` API (1)

The TF 2 `tf.random_normal()` API returns a set of values from a normal distribution with mean 0 and standard deviation 1. Listing 2.11 displays the contents of `tf2_random_normal.py`, which illustrates how to invoke the TF 2 `tf.random_normal()` API in a Python script.

LISTING 2.11: tf2_random_normal.py

```
import tensorflow as tf

# initialize a 6x3 2nd order tensor of random numbers:
values = {'weights':tf.Variable(tf.random.normal([6,3]))}

print("values:")
print(values['weights'])
```

Listing 2.11 defines the TF 2 variable `values`, which is a 6x3 second-order tensor of random numbers. The output from launching the code in Listing 2.11 is here:

```
values:
<tf.Variable 'Variable:0' shape=(6, 3) dtype=float32,
numpy=
array([[ 1.6026226 ,  0.8578084 , -0.4129617 ],
       [-1.2773342 ,  0.00630822, -0.26294807],
       [-0.6857447 ,  0.8162317 , -1.3068705 ],
       [ 0.8561586 ,  0.4733295 , -0.01647461],
       [-0.87976044, -0.7573596 ,  1.1681179 ],
       [ 0.6858091 ,  0.9455758 ,  0.67297345]],
dtype=float32)>
```

THE TF 2 `random_normal()` API (2)

The previous section showed you how to use the TF 2 `random_normal()` API. Listing 2.12 displays the contents of `tf2_random_normal2.py`, which illustrates how to use the `random_normal()` API in conjunction with a `NumPy` array.

LISTING 2.12: tf2_random_normal2.py

```
import tensorflow as tf

for i in range(3):
  x_train = tf.random.normal((1,), mean=5, stddev=2.0)
  y_train = x_train * 2 + 3
  print("x_train:",x_train)
print("-----------------\n")

for i in range(3):
  x_train = tf.random.normal((2,), mean=5, stddev=2.0)
  y_train = x_train * 2 + 4
  print("x_train:",x_train)
print("-----------------\n")

for i in range(3):
  x_train = tf.random.normal((3,), mean=5, stddev=2.0)
  y_train = x_train * 2 + 6
  print("x_train:",x_train)
print("-----------------\n")
```

Listing 2.12 contains three `for` loops, all of which initialize the variable `x_train` by invoking the `tf.random_normal()` API with the mean equal to 5 and the `stddev` equal to 2.0. Moreover, all three loops define the `y_train` variable as a linear combination of the `x_train` values.

The first parameter of the `tf.random_normal()` API specifies the shape of the set of random numbers. This parameter is set to `(1,)`, `(2,)`, and `(3,)` in the three `for` loops, which means that there will be one, two, and three columns of output, respectively. The output from Listing 2.12 is here:

```
x_train: tf.Tensor([7.1610246], shape=(1,), dtype=float32)
x_train: tf.Tensor([4.7292676], shape=(1,), dtype=float32)
x_train: tf.Tensor([3.34873],   shape=(1,), dtype=float32)
-----------------
```

```
x_train: tf.Tensor([6.7025995 4.178926 ], shape=(2,),
dtype=float32)
x_train: tf.Tensor([8.426264  6.4971704], shape=(2,),
dtype=float32)
x_train: tf.Tensor([2.7849288 7.8707666], shape=(2,),
dtype=float32)
-----------------

x_train: tf.Tensor([8.499574  3.6422663 7.9269   ],
shape=(3,), dtype=float32)
x_train: tf.Tensor([4.3513556 1.2529728 6.7783537],
shape=(3,), dtype=float32)
x_train: tf.Tensor([6.3333287 1.5062737 3.0980983],
shape=(3,), dtype=float32)
-----------------
```

THE `tf.truncated_normal()` API

The `tf.truncated_normal()` API produces a set of random values from a *truncated* normal distribution, which differs from the `tf.random_normal()` API in terms of the interval from which random values are selected. First, visualize a regular normal distribution whose mean is close to 0. Second, mentally "chop off" the values that are more than 2 standard deviations from the mean, which results in a "truncated" normal distribution.

This truncated interval is the interval from which random numbers are selected. Specifically, a random number is generated, and that number is included in the "result" set *only if it's inside the "truncated" normal distribution*.

However, if the randomly chosen value lies *outside* the truncated normal distribution, regenerate the value (and do so as often as necessary) until it's *inside* the truncated normal distribution, after which the number is included in the "result" set.

Perhaps an analogy would be helpful here. Suppose you toss a fair die and only record the numbers that are 2, 3, or 4, which is to say that you ignore a 1, 5, or 6 whenever that number is displayed. In addition, suppose that you want 100 occurrences of 2, 3, or 4. In this situation, you are almost guaranteed that you must toss the die more than 100 times (it's possible to toss a die that returns only a 2, 3, or 4 in the first 100 tosses, but the probability of doing so is practically zero). A function that returns the desired values is somewhat analogous to the `tf.truncated_normal()` function.

One other detail: the `tf.truncated_normal()` API is useful because it helps to prevent (or at least reduce) saturation that can occur with the sigmoid function: neurons stop "learning" if saturation occurs.

THE `tf.reshape()` API

Listing 2.13 displays the contents of `tf2_reshape.py`, which illustrates how to invoke the TF 2 `reshape()` APIs in order to create TF 2 tensors with different shapes.

LISTING 2.13: tf2_reshape.py

```
import tensorflow as tf

x = tf.constant([[2,5,3,-5],[0,3,-2,5],[4,3,5,3]])

print("shape:  ",tf.shape(input=x))
print("shape 1:",tf.reshape(x, [6,2]))
print("shape 2:",tf.reshape(x, [3,4]))
```

Listing 2.13 defines the TF 2 constant `x` as a TF 2 tensor with shape (3,4) that consists of 12 integers (some are positive and some are negative). *We can reshape the variable* `x` *as long as the product of the new row size and column size equals 12.*

Hence, the allowable pairs of values for rows and columns are: 1 and 12, 2 and 6, 3 and 4, 4 and 3, 6 and 2, and also 12 and 1. The output from launching the code in Listing 2.13 is here:

```
shape:   tf.Tensor([3 4], shape=(2,), dtype=int32)
shape 1: tf.Tensor(
[[ 2  5]
 [ 3 -5]
 [ 0  3]
 [-2  5]
 [ 4  3]
 [ 5  3]], shape=(6, 2), dtype=int32)
shape 2: tf.Tensor(
[[ 2  5  3 -5]
 [ 0  3 -2  5]
 [ 4  3  5  3]], shape=(3, 4), dtype=int32)
```

THE tf.range() API

Listing 2.14 displays the contents of `tf2_range.py`, which illustrates how to invoke the TF 2 `tf.range()` APIs to generate a range of numeric values. If you are familiar with `NumPy`, the TF 2 `tf.range()` API is similar to the `NumPy linspace()` API.

LISTING 2.14: tf2_range.py

```
import tensorflow as tf

a1 = tf.range(3, 18, 3)
a2 = tf.range(0, 8, 2)
a3 = tf.range(-6, 6, 3)
a4 = tf.range(-10, 10, 4)

print('a1:',a1)
print('a2:',a2)
print('a3:',a3)
print('a4:',a4)
```

Listing 2.14 defines `a1`, `a2`, `a3`, and `a4` by invoking the `tf.range()` API with different numeric triples so that you can see some of the possibilities with the `tf.range()` API. The output from launching the code in Listing 2.14 is here:

```
a1: tf.Tensor([3    6   9 12 15], shape=(5,), dtype=int32)
a2: tf.Tensor([0    2   4  6],    shape=(4,), dtype=int32)
a3: tf.Tensor([-6  -3   0  3],    shape=(4,), dtype=int32)
a4: tf.Tensor([-10 -6 -2  2 6],  shape=(5,), dtype=int32)
```

THE tf.equal() API (1)

Listing 2.15 displays the contents of `tf2_equal.py`, which illustrates how to invoke the TF 2 `equal()` API as well as the TF 2 `not_equal()` API to determine whether or not two TF 2 tensors are equal.

LISTING 2.15: tf2_equal.py

```
import tensorflow as tf

x1  = tf.constant([0.9, 2.5, 2.3, -4.5])
x2  = tf.constant([1.0, 2.0, 2.0, -4.0])
eq  = tf.equal(x1,x2)
neq = tf.not_equal(x1,x2)

print('x1: ',x1)
print('x2: ',x2)
print('eq: ',eq)
print('neq:',neq)
```

Listing 2.15 defines the TF 2 constants `x1` and `x2` as one-dimensional constants. Next, the variable `eq` is defined by performing an element-by-element comparison of `x1` and `x2`, and the result of the comparison is a one-dimensional tensor of Boolean values. The output from launching the code in Listing 2.15 is here:

```
x1:  tf.Tensor([0.9 2.5 2.3 -4.5], shape=(4,), dtype=float32)
x2:  tf.Tensor([1.  2.  2. -4.],   shape=(4,), dtype=float32)
eq:  tf.Tensor([False False False False], shape=(4,),
dtype=bool)
neq: tf.Tensor([True  True  True  True], shape=(4,),
dtype=bool)
```

THE tf.equal() API (2)

Listing 2.16 displays the contents of `tf2_equal2.py`, which also illustrates how to invoke the TF 2 `equal()` API.

LISTING 2.16: tf2_equal2.py

```
import tensorflow as tf
import numpy as np

x1 = tf.constant([0.9, 2.5, 2.3, -4.5])
x2 = tf.constant([1.0, 2.0, 2.0, -4.0])
x3 = tf.Variable(x1)

print('x1:',x1)
print('x2:',x2)
print('r3:',tf.round(x3))
print('eq:',tf.equal(x1,x3))
```

Listing 2.16 is straightforward: there are three TF 2 constants, `x1`, `x2`, and `x3`, which contain an assortment of positive and negative decimal values. Their values are displayed by the four `print()` statements. Notice that the fourth `print()` statement displays a tensor of Boolean values that are based on an element-by-element comparison of the elements of `x1` and `x3`. The output from launching the code in Listing 2.16 is here:

```
x1: tf.Tensor([0.9  2.5  2.3 -4.5],  shape=(4,),
dtype=float32)
x2: tf.Tensor([1.  2.  2. -4.],      shape=(4,),
dtype=float32)
r3: tf.Tensor([1.  2.  2. -4.],      shape=(4,),
dtype=float32)
eq: tf.Tensor([True True True True], shape=(4,), dtype=bool)
```

THE tf.argmax() API (1)

As you learned earlier in this chapter, the TF 2 `argmax()` API returns the *index* position of a tensor of values that contains the maximum value in a tensor (*not* the actual maximum value). Listing 2.17 displays the contents of `tf2_argmax.py`, which illustrates how to invoke the TF 2 `argmax()` API.

LISTING 2.17: tf2_argmax.py

```
import tensorflow as tf
import numpy as np

x1 = tf.constant([3.9, 2.1, 2.3, -4.0])
x2 = tf.constant([1.0, 2.0, 5.0, -4.2])

print('x1:',x1)
print('x2:',x2)
print('a1:',tf.argmax(input=x1, axis=0))
print('a2:',tf.argmax(input=x2, axis=0))
```

Listing 2.17 defines the TF 2 constants x1 and x2, which are one-dimensional tensors that contain positive and negative decimal values. The first pair of print() statements displays the contents of x1 and x2, followed by the index positions of the maximum values in x1 and x2. The output from launching the code in Listing 2.17 is here:

```
x1: tf.Tensor([ 3.9  2.1  2.3 -4. ], shape=(4,),
dtype=float32)
x2: tf.Tensor([ 1.   2.   5.  -4.2], shape=(4,),
dtype=float32)
a1: tf.Tensor(0, shape=(), dtype=int64)
a2: tf.Tensor(2, shape=(), dtype=int64)
```

THE tf.argmax() API (2)

Listing 2.18 displays the contents of tf2_argmax2.py, which illustrates another example of invoking the TF 2 argmax() API.

LISTING 2.18: tf2_argmax2.py

```
import tensorflow as tf

# initialize array of arrays:
arr1 = [[1,2,3], [30,20,10], [40,60,50]]
b = tf.Variable(arr1, name='b')

print("index of max values in b: ",tf.argmax(input=b,axis=1))
```

Listing 2.18 defines the 3x3 array arr1 that contains integer values, followed by the definition of the TF 2 variable b. The print() statement displays the index position of each row of arr1 that contains the maximum value for that row. The output from launching the code in Listing 2.18 is here:

```
index of max values in b:  tf.Tensor([2 0 1], shape=(3,),
dtype=int64)
```

THE tf.argmax() API (3)

Listing 2.19 displays the contents of tf2_argmax3.py with another example of invoking the TF 2 argmax() API, this time involving two 3x3 NumPy arrays.

LISTING 2.19: tf2_argmax3.py

```
import tensorflow as tf
import numpy as np

x = np.array([[31, 23,  4, 54],
              [18,  3, 25,  0],
              [28, 14, 33, 22],
              [17, 12,  5, 81]])

y = np.array([[31, 23,  4, 24],
              [18,  3, 25,  0],
              [28, 14, 33, 22],
              [17, 12,  5, 11]])

print('xmax:', tf.argmax(input=x,axis=1))
print('ymax:', tf.argmax(input=y,axis=1))
print('equal:',tf.equal(x,y))
```

Listing 2.19 defines the 3x3 `NumPy` arrays `x` and `y` that contain integer values. The `print()` statement displays the index of the maximum value for each *row* of `x`, followed by another `print()` statement that displays the index of the maximum value for each *row* of `y`.

The third `print()` statement displays a tensor of Boolean values that are the result of performing an element-by-element comparison of the elements of `x` and `y` to determine which pairs contain equal values. The output from launching the code in Listing 2.19 is here:

```
xmax: tf.Tensor([3 2 2 3], shape=(4,), dtype=int64)
ymax: tf.Tensor([0 2 2 0], shape=(4,), dtype=int64)

equal: tf.Tensor(
[[ True  True  True False]
 [ True  True  True  True]
 [ True  True  True  True]
 [ True  True  True False]], shape=(4, 4), dtype=bool)
```

COMBINING `tf.argmax()` AND `tf.equal()` APIs

Listing 2.20 displays the contents of `tf2_argmax_equal.py`, which illustrates how to invoke the TF 2 `equal()` API with the TF 2 `tf.argmax()` API.

LISTING 2.20: tf2_argmax_equal.py

```
import tensorflow as tf
import numpy as np

pred = np.array([[31,  23,  4, 24, 27, 34],
                 [18,  3,  25,  0,  6, 35],
                 [28,  14, 33, 22, 20,  8],
                 [13,  30, 21, 19,  7,  9],
                 [16,  1,  26, 32,  2, 29],
                 [17,  12, 5,  11, 10, 15]])

y =    np.array([[31,  23,  4, 24, 27, 14],
                 [18,  3,  25,  0,  6, 35],
                  [28,  14, 33, 22, 20,  8],
                  [13,  30, 21, 19,  7,  9],
                  [16,  1,  26, 32,  2, 29],
                  [17,  12,  5, 11, 10, 15]])

prediction = tf.equal(tf.argmax(input=pred,axis=1),tf.
argmax(input=y,axis=1))
accuracy = tf.reduce_mean(input_tensor=tf.cast(prediction,
tf.float32))

print("prediction:",prediction)
print("accuracy:  ",accuracy)
```

Listing 2.20 defines two `NumPy` two-dimensional arrays of integers. The variable `prediction` contains the indexes of the maximum row values of `x` and `y`. Next, the variable `accuracy` compares the index values in the `prediction` variable to determine how often they are equal. The result (after multiplying by 100) gives us the percentage of occurrences of equal index positions.

In Listing 2.20, the maximum value in each of the rows 2 through 6 of `x` are in the same position as the maximum value for rows 2 through 6 of `y`. However, the index of the maximum value in row 1 of `x` is 5, whereas the index of the maximum value in row 1 of `y` is 0 (i.e., the index values do not match). Hence, the index values only match in 5 of the 6 rows, which equals the fraction 5/6 that equals the decimal value 0.8333333 (rounded to six decimal places), which in turn is a percent value of 83.33333%.

This code sample is very helpful for understanding the logic (which is identical to this code sample) for calculating the accuracy of the training and testing portion of CNNs that are trained for the purpose of correctly identifying images. The output from launching the code in Listing 2.20 is here:

```
prediction: tf.Tensor([False  True  True  True  True
True], shape=(6,), dtype=bool)
accuracy:   tf.Tensor(0.8333333, shape=(), dtype=float32
```

COMBINING `tf.argmax()` AND `tf.equal()` APIs (2)

Listing 2.21 displays the contents of `tf2_argmax_equal2.py`, which illustrates how to invoke the TF 2 `equal()` API with the TF 2 `argmax()` API.

LISTING 2.21: tf2_argmax_equal2.py

```
import tensorflow as tf
import numpy as np

# predictions from our model:
pred = np.array([[0.1, 0.03, 0.2, 0.05, 0.02, 0.6],
                 [0.5, 0.04, 0.2, 0.06, 0.10, 0.1],
                 [0.2, 0.04, 0.5, 0.06, 0.10, 0.1]])

# true values from our labeled data:
y_vals = np.array([[0,  0,  0,  0,  0,  1],
                   [1,  0,  0,  0,  0,  0],
                   [0,  0,  1,  0,  0,  0]])

print("argmax(pred,1):  ", tf.argmax(input=pred,axis=1))
print("argmax(y_vals,1):", tf.argmax(input=y_vals,axis=1))

prediction = tf.equal(tf.argmax(input=pred, axis=1),tf.
argmax(input=y_vals, axis=1))

accuracy = tf.reduce_mean(input_tensor=tf.cast(prediction,
tf.float32))

print("prediction:",prediction)
print("accuracy:",accuracy)
```

Listing 2.21 contains code that is very similar to Listing 2.20: the main difference is that Listing 2.21 contains integer values for `x` and `y`, whereas the `NumPy` arrays `pred` and `y_vals` in Listing 2.21 contain decimal values that are between 0 and 1. The output from launching the code in Listing 2.21 is here:

```
argmax(pred,1):   tf.Tensor([5 0 2], shape=(3,), dtype=int64)
argmax(y_vals,1): tf.Tensor([5 0 2], shape=(3,), dtype=int64)
prediction: tf.Tensor([True  True  True], shape=(3,),
dtype=bool)
accuracy:   tf.Tensor(1.0, shape=(), dtype=float32)
```

THE `tf.map_fn()` API

Although Chapter 3 contains more information about lazy operators, this section contains a basic introduction to the `tf.map_fn()` API. In essence, this API is similar to the `map()` API: both APIs take an array of numbers and then “send” every number in the array to a function that is called a *lambda expression*. Note that you specify the array as well as the function.

For example, suppose you want to double every number in the array [1,2,3]. A common solution involves a loop that creates a new array whose values are twice their corresponding values in the initial array (or you could update the initial array “in place” by doubling each value).

An easier way to accomplish the same task involves the `tf.map_fn()` API. Listing 2.22 displays the contents of `tf2_map_function.py`, which illustrates how to invoke the TF 2 `tf.map_fn()` API in order to perform various operations on arrays of numbers, such as squaring every number in an array.

LISTING 2.22: tf2_map_function.py

```
import tensorflow as tf
import numpy as np

elems = np.array([1, 2, 3, 4, 5])

doubles = tf.map_fn(lambda x: 2 * x, elems)
print("doubles:",doubles)
# [2, 4, 6, 8, 10]

squares = tf.map_fn(lambda x: x * x, elems)
print("squares:",squares)
# [1, 4, 9, 16, 25]

elems = (np.array([1, 2, 3]), np.array([-1, 1, -1]))
neg_pos = tf.map_fn(lambda x: x[0] * x[1], elems, dtype=tf.
int64)
print("neg_pos:",neg_pos)
# [-1, 2, -3]

elems = np.array([1, 2, 3])
pos_neg = tf.map_fn(lambda x: (x, -x), elems, dtype=(tf.
int64, tf.int64))
print("pos_neg:",pos_neg)
```

Listing 2.22 contains a `NumPy` array `elems`, followed by four code blocks, each of which involves a lambda expression, all of which are displayed here:

```
lambda x: 2 * x
lambda x: x * x
lambda x: x[0] * x[1]
lambda x: (x, -x)
```

As you can see, the *first* lambda expression computes `2*x`, where `x` is a number in the `NumPy` array `elems`, whereas the *second* lambda expression computes `x*x`, where `x` is also a number from the `NumPy` array `elems`. Note that both lambda expressions execute independently of each other, which means that they both process every element in the `NumPy` array `elems`.

Examine the first two output lines as follows to convince yourself that the output consists of the doubled values and the squared values, respectively, of the numbers in the `NumPy` array `elems`. Now look at the other two lambda expressions to determine the resulting output, which you can confirm by inspecting the following output. The complete output from launching the code in Listing 2.22 is here:

```
doubles: tf.Tensor([2 4 6  8 10],    shape=(5,), dtype=int64)
squares: tf.Tensor([1 4 9 16 25 36], shape=(6,), dtype=int64)
neg_pos: tf.Tensor([-1  2 -3], shape=(3,), dtype=int64)
```

```
pos_neg: (<tf.Tensor: id=206,  shape=(3,), dtype=int64,
numpy=array([1, 2, 3])>, <tf.Tensor: id=207,   shape=(3,),
dtype=int64, numpy=array([-1, -2, -3])>)
```

WHAT IS A ONE-HOT ENCODING?

This section briefly describes how to create a "one-hot" encoding for categorical (i.e., nonnumerical) data. Before we perform a one-hot encoding, keep in mind that a feature in a dataset that contains nonnumerical values is called *categorical* or *nominal* data.

A one-hot encoding "maps" nonnumerical feature values into a corresponding set of numeric values, which is often required (in fact, it's always required when dealing with convolutional neural networks). The term one-hot encoding involves the conversion of each nonnumerical value into a vector that contains a single 1 (and zeroes elsewhere).

For example, suppose that we have a color variable whose values are `red`, `green`, or `blue`. A one-hot encoding of this color variable happens to look like a 3x3 identity matrix, as shown here:

```
red, green, blue
1,      0,     0
0,      1,     0
0,      0,     1
```

Now suppose that you have a dataset with six rows of data whose color values are `red`, `green`, `blue`, `red`, `green`, and `blue`. Then the six rows would contain the following values (let's ignore the values of the other elements of these six rows):

```
1, 0, 0
0, 1, 0
0, 0, 1
1, 0, 0
0, 1, 0
0, 0, 1
```

THE TF `one_hot()` API

Listing 2.23 displays the contents of `tf2_onehot2.py`, which illustrates how to use the TF 2 `one_hot()` API with a tensor.

LISTING 2.23: tf2_onehot2.py

```
import tensorflow as tf


   idx = tf.constant([2, 0, -1, 0])
   target = tf.one_hot(idx, 3, 2, 0)
```

```
@tf.function
def compute_values():
  print(idx)
  print(target)

compute_values()
```

Listing 2.23 starts by defining the variable `idx` based on a TF 2 constant that is a one-dimensional TF 2 tensor. Notice that the second and fourth elements in the TF 2 tensor are equal, which means that their one-hot encoding will be the same. The next portion of Listing 2.23 defines `target`, which will contain the one-hot encoded values for `idx`. Next, the `compute_values` function prints the TF 2 variable `idx` and then prints the contents of `target`, which is a 4x3 tensor. The output from Listing 2.23 is here:

```
tf.Tensor([ 2  0 -1  0], shape=(4,), dtype=int32)

tf.Tensor(
[[0 0 2]
 [2 0 0]
 [0 0 0]
 [2 0 0]], shape=(4, 3), dtype=int32)
```

OTHER USEFUL TF 2 APIs

In addition to the TF 2 APIs that you have seen in this chapter, you will also encounter the following APIs, whose names are intuitive. This section contains short code blocks that illustrate the syntax for these APIs, and you can find more detailed information in the online documentation.

The `tf.zeros()` API initializes a tensor with all zeroes, as shown here:

```
import tensorflow as tf
zeroes = tf.zeros([2, 3])
print("zeroes:",zeros)
```

The output from the preceding code block is a 2x3 second-order tensor containing all zeroes, as shown here:

```
zeroes: tf.Tensor(
 [[0. 0. 0.]
  [0. 0. 0.]], shape=(2, 3), dtype=float32)
```

The `tf.ones()` API initializes a tensor with all ones, as shown here:

```
import tensorflow as tf
ones = tf.ones ([2, 3])
print("ones:",ones)
```

The output from the preceding code block is a 2x3 second-order tensor containing all ones, as shown here:

```
ones: tf.Tensor(
[[1. 1. 1.]
 [1. 1. 1.]], shape=(2, 3), dtype=float32)
```

The `tf.fill()` API initializes a tensor with a specified numeric or string value, as shown here:

```
import tensorflow as tf
nines = tf.fill(dims=[2, 3], value=9)
pizza = tf.fill(dims=[2, 3], value="pizza")

print("nines:",nines)
print("pizza:",pizza)
```

The output from the preceding pair of `print()` statements is shown here:

```
nines: tf.Tensor(
 [[9 9 9]
  [9 9 9]], shape=(2, 3), dtype=int32)
pizza: tf.Tensor(
 [[b'pizza' b'pizza' b'pizza']
  [b'pizza' b'pizza' b'pizza']], shape=(2, 3), dtype=string)
```

The `tf.unique()` API finds the unique numbers or strings (duplicate values are ignored) in a TF 2 tensor, as shown here:

```
import tensorflow as tf

x = tf.constant([1, 1, 2, 4, 4, 4, 7, 8, 8])
val, idx = tf.unique(x)
y = tf.constant(['a','a','b','b','c','c'])
val2, idx2 = tf.unique(y)

print("val: ",val)
print("idx: ",idx)
print("val2:",val2)
print("idx2:",idx2)
```

The output from the preceding four `print()` statements is shown here:

```
val:  tf.Tensor([1 2 4 7 8], shape=(5,), dtype=int32)
idx:  tf.Tensor([0 0 1 2 2 2 3 4 4], shape=(9,), dtype=int32)
val2: tf.Tensor([b'a' b'b' b'c'], shape=(3,), dtype=string)
idx2: tf.Tensor([0 0 1 1 2 2], shape=(6,), dtype=int32)
```

The `tf.where()` API determines the location of a matching number (if any). For example, the following code block finds the location of the numbers 3 and 5 in the variable `t1`:

```
import tensorflow as tf

t1 = tf.constant([[1, 2, 3], [4, 5, 6]])
t2 = tf.where(tf.equal(t1, 3))
t3 = tf.where(tf.equal(t1, 5))

print("t1:",t1)
print("t2:",t2)
print("t3:",t3)
```

The output from the preceding three `print()` statements is shown here:

```
t1: tf.Tensor(
 [[1 2 3]
  [4 5 6]], shape=(2, 3), dtype=int32)
t2: tf.Tensor([[0 2]], shape=(1, 2), dtype=int64)
t3: tf.Tensor([[1 1]], shape=(1, 2), dtype=int64)
```

In the preceding code block, notice that `t1` has dimensions 2x3; the number 3 appears in position 3 (which has index 2) of the first element (which has index 0). Hence, the result is an element that contains the one-dimensional tensor [0 2]. Similarly, the number 5 appears in `t1` in position 2 (which has index 1) of the second element (which has index 1). Hence, the result is an element that contains the one-dimensional tensor [1 1].

SAVE AND RESTORE TF 2 VARIABLES

Listing 2.24 displays the contents of `tf2_save_restore.py`, which illustrates how to save and restore TF 2 variables.

LISTING 2.24: tf2_save_restore.py

```
import tensorflow as tf

x = tf.Variable(10.)
#checkpoint = tf.train.Checkpoint()
checkpoint = tf.train.Checkpoint(x=x)
print("x:",x)  # => 10.0

# Assign a new value to x and save
x.assign(3.)
print("x:",x)  # => 3.0
checkpoint_path = './ckpt/'
checkpoint.save(checkpoint_path)

# Change the variable after saving.
x.assign(25.)
print("x:",x)  # => 25.0
```

```
# Restore values from the checkpoint
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_
path))
print("x:",x)  # => 3.0
```

Listing 2.24 contains the TF 2 variable `x` with the value 10 and initializes the TF 2 variable `checkpoint` (which has type `tf.train.Checkpoint`) to "track" the value of `x`.

Checkpoints capture the exact value of all parameters (`tf.Variable` objects) used by a model. Since checkpoints do not contain any description of the computation defined by the model, typically they are only useful when source code that will use the saved parameter values is available. After you finish reading this code sample, try using the commented out code snippet for the variable checkpoint and compare the difference in the output.

The next code snippet in Listing 2.24 assigns the value 3 to `x`, after which the `checkpoint.save()` code snippet creates the following directory structure:

```
./ckpt
./ckpt/-1.data-00000-of-00001
./ckpt/-1.index
./ckpt/checkpoint
```

Notice how the next code snippet assigns the value 25 to the variable `x`, but when the code `checkpoint.restore()` is invoked, `x` is restored to its "saved" value. Launch the code in Listing 2.24 and you will see the following output, and notice the sequence of values for the variable `x` (the values are highlighted in bold):

```
x: <tf.Variable 'Variable:0' shape=() dtype=float32,
numpy=10.0>
x: <tf.Variable 'Variable:0' shape=() dtype=float32,
numpy=3.0>
x: <tf.Variable 'Variable:0' shape=() dtype=float32,
numpy=25.0>
x: <tf.Variable 'Variable:0' shape=() dtype=float32,
numpy=3.0>
```

TENSORFLOW RAGGED CONSTANTS AND TENSORS

As you probably know, every element in a "regular" multidimensional tensor has the same dimensions. For example, a 2x3 second-order tensor contains two rows and three columns: each row is a 1x3 vector, and each column is a 2x1 vector. As another example, a 2x3x4 tensor contains two 3x4 tensors (and the same logic applies to each 3x4 tensor).

On the other hand, a *ragged constant* is a set of elements that have different lengths. You can think of ragged constants as a generalization of "regular" datasets.

Listing 2.25 displays the contents of `tf2_ragged_tensors1.py`, which illustrates how to define a ragged dataset and then iterate through its contents.

LISTING 2.25: tf2_ragged_tensors1.py

```
import tensorflow as tf

digits = tf.ragged.constant([[3, 1, 4, 1], [], [5, 9, 2],
[6], []])
words = tf.ragged.constant([["Bye", "now"], ["thank",
"you", "again", "sir"]])

print(tf.add(digits, 3))
print(tf.reduce_mean(digits, axis=1))
print(tf.concat([digits, [[5, 3]]], axis=0))
print(tf.tile(digits, [1, 2]))
print(tf.strings.substr(words, 0, 2))
```

Listing 2.25 defines two ragged constants `digits` and `words` consisting of integers and strings, respectively. The remaining portion of Listing 2.25 consists of five `print()` statements that apply various operations to these two datasets and then display the results.

The first `print()` statement adds the value 3 to every number in the `digits` dataset, and the second `print()` statement computes the row-wise average of the elements of the `digits` dataset because `axis=1` (whereas `axis=0` performs column-wise operations).

The third `print()` statement appends the element `[[5,3]]` to the `digits` dataset, and performs this operation in a column-wise fashion (because `axis=0`). The fourth `print()` statement "doubles" each non-empty element of the `digits` dataset. Finally, the fifth `print()` statement extracts the first two characters from every string in the `words` dataset. The output from launching the code in Listing 2.25 is here:

```
<tf.RaggedTensor [[6, 4, 7, 4], [], [8, 12, 5], [9], []]>
tf.Tensor([2.25              nan 5.33333333 6.
nan], shape=(5,), dtype=float64)
<tf.RaggedTensor [[3, 1, 4, 1], [], [5, 9, 2], [6], [],
[5, 3]]>
<tf.RaggedTensor [[3, 1, 4, 1, 3, 1, 4, 1], [], [5, 9, 2,
5, 9, 2], [6, 6], []]>
<tf.RaggedTensor [[b'By', b'no'], [b'th', b'yo', b'ag',
b'si']]>
```

Listing 2.26 displays the contents of `tf2_ragged_tensors2.py`, which illustrates how to define a ragged tensor in TF 2.

LISTING 2.26: tf2_ragged_tensors2.py

```
import tensorflow as tf

x1 = tf.RaggedTensor.from_row_splits(
        values=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        row_splits=[0, 5, 10])
print("x1:",x1)

x2 = tf.RaggedTensor.from_row_splits(
        values=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        row_splits=[0, 4, 7, 10])
print("x2:",x2)

x3 = tf.RaggedTensor.from_row_splits(
        values=[1, 2, 3, 4, 5, 6, 7, 8],
        row_splits=[0, 4, 4, 7, 8, 8])
print("x3:",x3)
```

Listing 2.26 defines the TF 2 ragged tensors `x1`, `x2`, and `x3` that are based on the integers from 1 to 10 inclusive. The `values` parameter specifies a set of values that will be "split" into a set of vectors, using the numbers in the `row_splits` parameter for the start index and the end index of each vector.

For example, `x1` specifies `row_splits` with the value `[0,5,10]` whose values are used as index positions in order to create two vectors: the vector whose values are from *index 0 through index 4* of `x1`, and the vector whose values are from *index 5 through index 9* of `x1`. The contents of those two vectors are `[1, 2, 3, 4, 5]` and `[6, 7, 8, 9, 10]`, respectively (see the output as follows).

As another example, `x2` specifies `row_splits` with the value `[0,4,7,10]`, which determines three vectors: the vector whose values are from *index 0 through index 3* of `x1`, the vector whose values are from *index 4 through index 6* of `x1`, and the vector whose values are from *index 7 through index 9* of `x1`. The contents of those two vectors are `[1,2,3,4]`, `[5,6,7]`, and `[8, 9, 10]`, respectively (see the output as follows).

You can perform a similar analysis for `x3`, keeping mind that the vector whose start index and end index are `[4,4]` is an empty vector. The output from launching the code in Listing 2.26 is here:

```
x1: <tf.RaggedTensor [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]]>
x2: <tf.RaggedTensor [[1, 2, 3, 4], [5, 6, 7], [8, 9, 10]]>
x3: <tf.RaggedTensor [[1, 2, 3, 4], [], [5, 6, 7], [8], []]>
```

If you want to generate a list of values, invoke the `to_list()` operator. For instance, suppose you define `x4` as follows:

```
x4 = tf.RaggedTensor.from_row_splits(
        values=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
        row_splits=[0, 5, 10]).to_list()
print("x4:",x4)
```

The output from the preceding code snippet is here (which you can compare with the output for `x1` in the preceding output block):

```
x4: [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]]
```

You can also create higher-dimensional ragged tensors in TF 2. For example, the following code snippet creates a two-dimensional ragged tensor in TF 2:

```
RaggedTensor.from_nested_row_splits(
     flat_values=[3,1,4,1,5,9,2,6],
     nested_row_splits=([0,3,3,5], [0,4,4,7,8,8])).to_list()
```

The preceding code snippet generates the following output:

```
[[[3, 1, 4, 1], [], [5, 9, 2]], [], [[6], []]]
```

WHAT IS A TFRecord?

A `TFRecord` is a file that describes the data required during the training phase and the testing phase of a model. There are two protocol buffer message types available for a `TFRecord`: the `Example` message type and the `SequenceExample` message type. These protocol buffer message types enable you to arrange data as a map from string keys to values that are lists of integers, 32-bit floats, or bytes.

The data in a `TFRecord` is "wrapped" inside a `Feature` class. In addition, each feature is stored in a key value pair, where the key corresponds to the title that is allotted to each feature. These titles are used later for extracting the data from `TFRecord`. The created dictionary is passed as input to a `Feature` class. Finally, the features object is passed as input to an `Example` class that is appended to the `TFRecord`. The preceding process is repeated for every type of data that is stored in `TFRecord`.

The `TFRecord` file format is a record-oriented binary format that you can use for training data. In addition, the `tf.data.TFRecordDataset` class enables you to stream over the contents of one or more `TFRecord` files as part of an input pipeline.

You can store any type of data, including images, in the `tf.train.Example` format. However, you specify the mechanism for arranging the data into serialized bytes, as well as reconstructing the original format.

A Simple TFRecord

Listing 2.27 displays the contents of `tf2_record1.py`, which illustrates how to define a `TFRecord` in TF 2.

LISTING 2.27: tf2_record1.py

```
import tensorflow as tf

data = b"pasta"
simple1 = tf.train.Example(features=tf.train.
Features(feature={
  'my_ints':  tf.train.Feature(int64_list=tf.train.
Int64List(value=[2, 5])),
  'my_float': tf.train.Feature(float_list=tf.train.
FloatList(value=[3.6])),
  'my_bytes': tf.train.Feature(bytes_list=tf.train.
BytesList(value=[data]))
}))

print("my_ints:",  simple1.features.feature['my_ints'].
int64_list.value)
print("my_floats:",simple1.features.feature['my_float'].
float_list.value)
print("my_bytes:", simple1.features.feature['my_bytes'].
bytes_list.value)

#print("simple1:",simple1)
```

Listing 2.27 contains the definition of the variable `simple1` that is an instance of the `tf.train.Example` class. The `simple1` variable defines a record consisting of the fields `my_ints`, `my_floats`, and `my_bytes` that are of type `Int64List`, `FloatList`, and `ByteList`, respectively.

The final portion of Listing 2.27 contains `print()` statements that display the values of various elements in the `simple1` variable, as shown here:

```
('my_ints:', [2L, 5L])
('my_floats:', [3.5999999046325684])
('my_bytes:', [b'pasta'])
```

WHAT ARE `tf.layers`?

The `tf.layers` namespace contains an assortment of classes for the layers in Neural Networks, including DNNs (Dense Neural Networks) and CNNs (Convolutional Neural Networks). Some of the more common classes in the `tf.layers` namespace are listed as follows (and are discussed in more detail in the appendix):

- BatchNormalization: Batch normalization layer
- Conv2D: 2D convolution layer (e.g., spatial convolution over images)
- Dense: Densely connected layer class
- Dropout: Applies Dropout to the input
- Flatten: Flattens an input tensor while preserving the batch axis (axis 0)
- Layer: Base layer class
- MaxPooling2D: Max pooling layer for 2D inputs (e.g., images)

For example, a minimalistic CNN starts with a "triple" that consists of a `Conv2D` layer, followed by `ReLU` (Rectified Linear Unit) activation function, and then a `MaxPooling2D` layer. If you see this triple appear a second time, followed by two consecutive `Dense` layers and then a softmax activation function, it's known as "LeNet."

A bit of trivia: in the late 1990s, when people deposited checks at an automated bank machine, LeNet scanned the contents of those checks to determine the digits of the check amount (of course, customers had to confirm that the number determined by LeNet was correct). LeNet had an accuracy rate around 90%, which is a very impressive result for such a simple Convolutional Neural Network!

WHAT IS TENSORBOARD?

TensorBoard is very powerful data and graph visualization tool that provides a great deal of useful information as well as debugging support. TensorBoard is part of the TensorFlow distribution, so you don't need to perform a separate installation.

TensorBoard has a background thread that loads event data from event files that are in the directory that you specify with "—logdir" when you launch TensorBoard from the command line. Data from event files is loaded into memory because it's more efficient than querying data from files.

TensorBoard itself is an extensible Web server with a plug-in architecture, which is the mechanism for adding dashboards. You access TensorBoard through a Polymer-based Web component framework in a Web browser session. The Web application involves a mix of JavaScript and TypeScript. In addition, D3.js, dagre.js, and three.js are used for the visualizations.

TensorBoard supports multiple dashboards for scalars, graph, histograms, images, and so forth. TensorBoard enables you to analyze data based on a specific "run" and also by "tag," which enables you to perform analysis of the manner in which your data changes over time.

TensorBoard provides a "writer" for saving the contents of a TensorFlow graph to a file in a directory (that is specified by you). In addition, TensorBoard provides various APIs in order to insert the values of variables in a TensorBoard visualization.

In order to view the contents of a TF 2 graph in TensorBoard, open a command shell, navigate to the parent directory of the directory that contains graph-related files (let's pretend its name is `tf_log_files`), and launch the following command:

```
tensorboard –logdir=./tf_log_files
```

Next, launch a Chrome browser and navigate to this URL:

```
localhost:6006
```

When you see the TensorFlow graph rendered in your browser, use your mouse to resize the graph, and double-click on nodes to "drill down" and find more information about each node.

TensorBoard provides support for CPUs, GPUs, and TPUs, in conjunction with TF 1.x and TF 2, with one exception: TensorBoard currently does not support TPUs with TF 2. You can follow this issue for updates on the support status:

https://github.com/tensorflow/tensorflow/issues/24412

TF 2 with TensorBoard

This section contains some useful links that provide more detailed and instructive information regarding TensorBoard with TF 2 as well as TF 1.x. Navigate to the following link if you want to see an example of TF 2 and TensorBoard in a Jupyter notebook:

https://colab.research.google.com/github/tensorflow/tensorboard/blob/master/docs/r2/get_started.ipynb#scrollTo=XKUjdIoV87um

You can download the preceding Jupyter notebook and also a Python file that contains the same code as the Jupyter notebook; in the latter case you also need to "comment out" the so-called magic commands in Jupyter.

If the Keras code in the preceding Jupyter notebook is unfamiliar to you, read the Keras-based code samples that are discussed later in this book and then the code in this Jupyter notebook will make more sense.

If you have not upgraded to TF 2 yet, navigate to this link for more information about TensorBoard with TensorFlow 1.x:

https://www.tensorflow.org/guide/summaries_and_tensorboard

Other tips and how-to information about TensorBoard is available here:

https://github.com/TF 2 2/tensorboard/blob/master/README.md#my-tensorboard-isnt-showing-any-data-whats-wrong

Some information about TF 2 with Keras and how to write image summaries for TensorBoard is here:

https://stackoverflow.com/questions/55421290/tensorflow-2-0-keras-how-to-write-image-summaries-for-tensorboard/55754700#55754700

A video about TensorBoard during the TF Summit (2019):

https://www.youtube.com/watch?v=xM8sO33x_OU&list=PLQY2H8rRoyvzoUYI26kHmKSJBedn3SQuB&index=11&t=0s

The appendix contains a brief section about TensorFlow Graphics, which supports 3D effects in TensorBoard.

TensorBoard Dashboards

TensorBoard supports a variety of dashboards, some of which are listed as follows with a brief description of their functionality:

- Scalar Dashboard
- Histogram Dashboard
- Distribution Dashboard
- Image Dashboard
- Audio Dashboard
- Text Dashboard

The Scalar Dashboard visualizes scalar statistics that vary over time (e.g., the loss values of a model). The Histogram Dashboard visualizes data recorded via the `tf.summary.histogram` API and displays how the statistical distribution of a Tensor has varied over time. The charts display temporal "slices" of data, where each slice is a histogram of the tensor at a given step.

The Distribution Dashboard also displays histogram data (via the `tf.summary.histogram` API) and shows high-level statistics on a distribution. Each line on the chart represents a percentile in the distribution over the data. Moreover, the percentiles can also be viewed as standard deviation boundaries on a normal distribution.

Finally, the Image Dashboard, Audio Dashboard, and Text Dashboard display PNGs, audio files, and text, respectively.

There are a few things to keep in mind. First, TensorBoard expects a single events file, which is to say that multiple summary writers involve multiple events files. In the case of a distributed TensorFlow instance, designate one worker as the "chief" that is responsible for all summary processing. Second, if data appears to overlap with itself, you might have multiple executions of TensorFlow that wrote to the same log directory.

The tf.summary API

The tf.summary API is the primary way for "serving up" data from event files to TensorBoard. This API also assists in displaying log metrics and prediction details. TF 1.x has a `tf.summary` module that will be replaced by a new API for TensorBoard in TF 2 that differs as follows:

- The data-format-specific parts will be defined in `tensorboard.summary`
- The generated summary events will use a more extensible "wire format"
- The write-side code will use the V2 summary-writing API

Navigate to the following URL for more information about TensorBoard in TF 2:

`https://www.tensorflow.org/tensorboard/r2/get_started`

In addition, the following URL contains an example of profiling training metrics for a Keras-based model:

`https://www.tensorflow.org/tensorboard/r2/tensorboard_profiling_keras`

GOOGLE COLABORATORY

Depending on the hardware, GPU-based TF 2 code is typically at least fifteen times faster than CPU-based TF 2 code. However, the cost of a good GPU can be a significant factor. Although NVIDIA provides GPUs, those consumer-based GPUs are not optimized for multi-GPU support (which *is* supported by TF 2).

Fortunately Google Colaboratory is an affordable alternative that provides free GPU support, and also runs as a `Jupyter` notebook environment. In addition, Google Colaboratory executes your code in the cloud and involves zero configuration, and it's available here:

`https://colab.research.google.com/notebooks/welcome.ipynb`

This `Jupyter` notebook is suitable for training simple models and testing ideas quickly. Google Colaboratory makes it easy to upload local files, install software in `Jupyter` notebooks, and even connect Google Colaboratory to a `Jupyter` runtime on your local machine.

Some of the supported features of Colaboratory include TF 2 execution with GPUs, visualization using Matplotlib, and the ability to save a copy of your Google Colaboratory notebook to Github by using `File > Save a copy to GitHub`.

Moreover, you can load any .ipynb on GitHub just by adding the path to the URL `colab.research.google.com/github/` (see the Colaboratory website for details).

Google Colaboratory has support for other technologies such as HTML and SVG, enabling you to render SVG-based graphics in notebooks that are in Google Colaboratory. One point to keep in mind: any software that you install in a Google Colaboratory notebook is only available on a per-session basis: if you log out and log in again, you need to perform the same installation steps that you performed during your earlier Google Colaboratory session.

As mentioned earlier, there is one other *very* nice feature of Google Colaboratory: you can execute code on a GPU for up to twelve hours per day for free. This free GPU support is extremely useful for people who don't have a suitable GPU on their local machine (which is probably the majority of users), and now they launch TF 2 code to train neural networks in less than twenty or thirty minutes that would otherwise require multiple hours of CPU-based execution time.

In case you're interested, you can launch TensorBoard inside a Google Colaboratory notebook with the following command (replace the specified directory with your own location):

```
%tensorboard --logdir /logs/images
```

Keep in mind the following details about Google Colaboratory. First, whenever you connect to a server in Google Colaboratory, you start what's known as a *session*. You can execute the code in a session with either a GPU or a TPU without any cost to you, and you can execute your code without any time limit for your session. However, if you select the GPU option for your session, *only the first twelve hours of GPU execution time are free*. Any additional GPU time during that same session incurs a small charge (see the website for those details).

The other point to keep in mind is that any software that you install in a Jupyter notebook during a given session will *not* be saved when you exit that session. For example, the following code snippet installs `TFLearn` in a Jupyter notebook:

```
!pip install tflearn
```

When you exit the current session, and at some point later you start a new session, you need to install `TFLearn` again, as well as any other software (such as Github repositories) that you also installed in any previous session.

Incidentally, you can also run TF 2 code and TensorBoard in Google Colaboratory, with support for CPUs and GPUs (and support for TPUs will be available later). Navigate to this link for more information:

https://www.tensorflow.org/tensorboard/r2/tensorboard_in_notebooks

OTHER CLOUD PLATFORMS

GCP (Google Cloud Platform) is a cloud-based service that enables you to train TF 2 code in the cloud. GCP provides Deep Learning DL images (similar in concept to Amazon AMIs) that are available here:

https://cloud.google.com/deep-learning-vm/docs

The preceding link provides documentation and also a link to DL images based on different technologies, including TF 2 and PyTorch, with GPU and CPU versions of those images. Along with support for multiple versions of Python, you can work in a browser session or from the command line.

GCP SDK

Install GCloud SDK on a Mac-based laptop by downloading the software at this link:

https://cloud.google.com/sdk/docs/quickstart-macos

You will also receive USD 300 dollars worth of credit (over one year) if you have never used Google cloud.

This concludes the material for this chapter, and if you want to learn more about any of the features that you have seen, perform an Internet search for additional tutorials.

SUMMARY

In this chapter, you learned about some TF 2 features such as eager execution, which has a more Python-like syntax than "regular" TensorFlow syntax, tensor operations (such as multiplying tensors), and also how to create `for` loops and `while` loops in TF 2.

Next, you saw how to use the TF 2 `tf.random_normal()` API for generating random numbers (which is useful for initializing the weights of edges in neural networks), followed by the `tf.argmax()` API for finding the index of each row (or column) that contains the maximum value in each row (or column), which is used for calculating the accuracy of the training process involving various algorithms. You also saw the `tf.range()` API, which is similar to the `NumPy` `linspace()` API.

In addition, you learned about the TF 2 `reduce_mean()` and `equal()` APIs, both of which are involved in calculating the accuracy of the training of a neural network (in conjunction with `tf.argmax()`). Next, you saw the TensorFlow `truncated_normal()` API, which is a variant of the `tf.random_normal()` API, and the TF 2 `one_hot()` API for encoding data in a particular fashion. Moreover, you learned about the TF 2 `reshape()` API, which you will see in any TF 2 code that involves training a CNN (Convolutional Neural Network).

In the second half of this chapter you were introduced to TensorBoard, which is a very powerful visualization tool that is part of the TensorFlow distribution. You saw some code samples that invoke TensorBoard APIs alongside other TF 2 APIs in order to augment the TensorFlow graph with supplemental information that was rendered in TensorBoard in a Web browser. Finally, you got an introduction to Google Colaboratory, which is a fully online Jupyter-based environment.

CHAPTER 3

TF 2 DATASETS

This chapter discusses the TF 2 `tf.data.Dataset` namespace and the classes therein that support a rich set of operators for processing very large datasets (i.e., datasets that are too large to fit in memory). You will learn about so-called lazy operators (such as `filter()` and `map()`) that you can invoke via "method chaining" to extract a desired subset of data from a dataset. In addition, you'll learn about TF 2 `Estimators` (in the `tf.estimator` namespace) and TF 2 `layers` (in the `tf.keras.layers` namespace).

Please note that the word "dataset" in this chapter refers to a TF 2 class in the `tf.data.Dataset` namespace. Such a dataset acts as a "wrapper" for actual data, where the latter can be a CSV file or some other data source. This chapter does not cover TF 2 built-in datasets of "pure" data, such as `MNIST`, `CIFAR`, and `IRIS`, except for cases in which they are part of code samples that involve TF 2 lazy operators.

Familiarity with lambda expressions (discussed later) and Functional Reactive Programming will be very helpful for this chapter. In fact, the code samples chapter will be very straightforward if you already have experience with `Observables` in `RxJS`, `RxAndroid`, `RxJava`, or some other environment that involves lazy execution.

The first part of this chapter briefly introduces you to TF 2 `Datasets` and *lambda expressions*, along with some simple code samples. You will learn about `iterators` that work with TF 1.x `tf.data.Datasets`, and also TF 2 *generators* (which are Python functions with a `@tf.function` decorator).

The second part of this chapter discusses `TextLineDatasets` that are very convenient for working with text files. As explained previously, the TF 2 code samples in this section use TF 2 generators instead of iterators (which work with TF 1.x).

The third part of this chapter discusses various lazy operators, such as `filter()`, `map()`, and `batch()` operators, and also briefly describes how they work (and when you might need to use them). You'll also learn *method chaining* for combining these operators, which results in powerful code combinations that can significantly reduce the complexity of your TF 2 code.

The final portion of the chapter briefly discusses TF 2 estimators in the `tf.estimator` namespace (which are a layer of abstraction above `tf.keras.layers`), as well as TF 2 `layers` that provide an assortment of classes for DNNs (Dense Neural Networks) and CNNs (Convolutional Neural Networks) that are discussed in the appendix.

THE TF 2 tf.data.DatasetS

Before we delve into this topic, we need to make sure that the following distinction is clear: a "dataset" contains rows of data (often in a flat file), where the columns are called "features" and the rows represent an "instance" of the dataset. By contrast, a TF 2 `Dataset` refers to a class in the `tf.data.Dataset` namespace that acts like a "wrapper" around a "regular" dataset that contains rows of data.

You can also think of a TF 2 `Dataset` as being analogous to a `Pandas DataFrame`. Again, if you are familiar with `Observables` in Angular (or something similar), you can perform a quick knowledge transfer as you learn about TF 2 `Datasets`.

TF 2 `tf.data.Datasets` are well-suited for creating asynchronous and optimized data pipelines. In brief, the TF 2 `Dataset` API loads data from the disk (both images and text), applies optimized transformations, creates batches, and sends the batches to the GPU. In fact, the TF 2 `Dataset` API is well-suited for better GPU utilization. In addition, use `tf.functions` in TF 2.0 to fully utilize dataset asynchronous prefetching/streaming features.

According to the TF 2 documentation: "A Dataset can be used to represent an input pipeline as a collection of elements (nested structures of tensors) and a 'logical plan' of transformations that act on those elements."

A TF 2 `tf.data.Dataset` is designed to handle very large datasets. A TF 2 `Dataset` can also represent an input pipeline as a collection of elements (i.e., a nested structure of tensors), along with a "logical plan" of transformations that act on those elements. For example, you can define a TF 2 `Dataset` that initially contains the lines of text in a text file, then extract the lines of text that start with a "`#`" character, and then display only the first three matching lines. Creating this pipeline is easy: create a TF 2 `Dataset` and then chain the lazy operators `filter()` and `take()`, which is similar to an example that you will see later in this chapter.

Creating a Pipeline

Think of a dataset as a pipeline that starts with a source, which can be a `NumPy` array, tensors in memory, or some other source. If the source involves tensors, use `tf.data.Dataset.from_tensors()` to combine the input, otherwise use `tf.data.Dataset.from_tensor_slices()` if you want a separate row for each input tensor. On the other hand, if the input data is located on disk in a `TFRecord` format (which is recommended), construct a `tf.data.TFRecordDataset`.

The difference between the first two APIs is shown as follows:

```
#combine the input into one element => [[1, 2], [3, 4]]
t1 = tf.constant([[1, 2], [3, 4]])
ds1 = tf.data.Dataset.from_tensors(t1)

#a separate element for each item: [1, 2], [3, 4]
t2 = tf.constant([[1, 2], [3, 4]])
ds2 = tf.data.Dataset.from_tensor_slices(t2)
for item in ds1:
  print("1item:",item)

print("--------------")

for item in ds2:
  print("2item:",item)
```

The output from the preceding code block is here:

```
1item: tf.Tensor(
[[1 2]
 [3 4]], shape=(2, 2), dtype=int32)
--------------
2item: tf.Tensor([1 2], shape=(2,), dtype=int32)
2item: tf.Tensor([3 4], shape=(2,), dtype=int32)
```

The TF 2 `from_tensors()` API also requires compatible dimensions, which means that the following code snippet causes an error:

```
# exception: ValueError: Dimensions 10 and 9 are not
compatible
ds1 = tf.data.Dataset.from_tensor_slices(
    (tf.random_uniform([10, 4]), tf.random_
uniform([9])))
```

On the other hand, the TF 2 `from_tensor_slices()` API does not have a compatibility restriction, so the following code snippet works correctly:

```
ds2 = tf.data.Dataset.from_tensors(
    (tf.random_uniform([10, 4]), tf.random_uniform([9])))
```

Another situation in which there are differences in these two APIs involves the use of lists, as shown here:

```
ds1 = tf.data.Dataset.from_tensor_slices(
    [tf.random_uniform([2, 3]), tf.random_uniform([2, 3])])
```

```
ds2 = tf.data.Dataset.from_tensors(
    [tf.random_uniform([2, 3]), tf.random_uniform([2, 3])])

print(ds1) # shapes: (2, 3)
print(ds2) # shapes: (2, 2, 3)
```

In the preceding code block, the TF 2 `from_tensors()` API creates a 3D tensor whose shape is (2,2,3), whereas the TF 2 `from_tensor_slices()` API merges the input tensor and produces a tensor whose shape is (2,3).

As a further illustration of these two APIs, consider the following code block:

```
import tensorflow as tf

ds1 = tf.data.Dataset.from_tensor_slices(
    (tf.random.uniform([3, 2]), tf.random.uniform([3])))

ds2 = tf.data.Dataset.from_tensors(
    (tf.random.uniform([3, 2]), tf.random.uniform([3])))

print('-----------------------------')
for i, item in enumerate(ds1):
  print('elem1: ' + str(i + 1), item[0], item[1])

print('-----------------------------')
for i, item in enumerate(ds2):
  print('elem2: ' + str(i + 1), item[0], item[1])
print('-----------------------------')
```

Launch the preceding code and you will see the following output:

```
-----------------------------
elem1: 1 tf.Tensor([0.965013  0.8327141], shape=(2,),
dtype=float32) tf.Tensor(0.03369963, shape=(),
dtype=float32)
elem1: 2 tf.Tensor([0.2875235  0.11409616], shape=(2,),
dtype=float32) tf.Tensor(0.05131495, shape=(),
dtype=float32)
elem1: 3 tf.Tensor([0.08330548 0.13498652], shape=(2,),
dtype=float32) tf.Tensor(0.3145547, shape=(),
dtype=float32)
-----------------------------

elem2: 1 tf.Tensor(
[[0.9139079  0.13430142]
 [0.9585271  0.58751714]
 [0.4501326  0.8380357 ]], shape=(3, 2), dtype=float32)
tf.Tensor([0.00776255 0.2655964  0.61935973], shape=(3,),
dtype=float32)
-----------------------------
```

Basic Steps for TF 2 `Datasets`

Perform the following three steps in order to create and process the contents of a TF 2 `Dataset`:

1. Create or import data
2. Define a generator (Python function)
3. Consume the data

There are many ways to populate a TF 2 `Dataset` from multiple sources. For simplicity, the code samples in the first part of this chapter perform the following steps: start by creating a TF 2 `Dataset` instance with an initialized `NumPy` array of data; second, define a Python function in order to iterate through the TF 2 `Dataset`; and third, access the elements of the dataset (and in some cases, supply those elements to a TF 2 model).

As you saw earlier in this chapter, keep in mind that TF 1.x combines `Datasets` with *iterators*, whereas TF 2 uses *generators* with `Datasets`. TF 2 uses generators because eager execution (the default execution mode for TF 2) does not support iterators.

A Simple TF 2 `tf.data.Dataset`

Listing 3.1 displays the contents of `tf2_numpy_dataset.py`, which illustrates how to create a very basic TF 2 `tf.data.Dataset` from a `NumPy` array of numbers. Although this code sample is minimalistic, it's the initial code block that appears in other code samples in this chapter.

LISTING 3.1: tf2_numpy_dataset.py

```
import tensorflow as tf
import numpy as np

x = np.arange(0, 10)

# make a dataset from a numpy array
ds = tf.data.Dataset.from_tensor_slices(x)
```

Listing 3.1 contains two familiar `import` statements and then initializes the variable `x` as a `NumPy` array with the integers from 0 through 9 inclusive. The variable `ds` is initialized as a TF 2 `Dataset` that's based on the contents of the variable `x`.

Note that nothing else happens in Listing 3.1, and no output is generated: later you will see more meaningful code samples involving TF 2 `Datasets`.

WHAT ARE LAMBDA EXPRESSIONS?

In brief, a *lambda expression* is an anonymous function. Use lambda expressions to define local functions that can be passed as arguments, returned as the value of function calls, or used as "one-off" function definitions.

Informally, a lambda expression takes an input variable and performs some type of operation (specified by you) on that variable. For example, here's a "bare bones" lambda expression that adds the number 1 to an input variable `x`:

```
lambda x: x + 1
```

The term on the left of the ":" is `x`, and it's just a formal variable name that acts as the input (you can replace x with another string that's convenient for you). The term on the right of the ":" is `x+1`, which simply increments the value of the input `x`.

As another example, the following lambda expression doubles the value of the input parameter:

```
lambda x: 2*x
```

You can also define a lambda expression in a valid TF 2 code snippet, as shown here (`ds` is a TF 2 `Dataset` that is defined elsewhere):

```
ds.map(lambda x: x + 1)
```

Even if you are unfamiliar with TF 2 `Datasets` or the `map()` operator, you can still understand the preceding code snippet. Later in this chapter you'll see other examples of lambda expressions that are used in conjunction with lazy operators.

The next section contains a complete TF 2 code sample that illustrates how to define a generator (which is a Python function) that adds the number 1 to the elements of a TF 2 `Dataset`.

WORKING WITH GENERATORS IN TF 2

Listing 3.2 displays the contents of `tf2_plusone.py`, which illustrates how to use a lambda expression to add the number 1 to the elements of a TF 2 `Dataset`.

LISTING 3.2: tf2_plusone.py

```
import tensorflow as tf
import numpy as np

x = np.arange(0, 10)

def gener():
  for i in x:
    yield (i+1)

ds = tf.data.Dataset.from_generator(gener, (tf.int64))

#for value in ds.take(len(x)):
for value in ds:
  print("1value:",value)
```

```
for value in ds.take(2*len(x)):
  print("2value:",value)
```

Listing 3.2 initializes the variable `x` as a `NumPy` array consisting of the integers from 0 through 9 inclusive. Next, the variable `ds` is initialized as a TF 2 `Dataset` that is created from the Python function `gener()`, which returns the input value incremented by 1. Notice that the Python function `gener()` does *not* have a `@tf.function()` decorator: even so, this function is treated as a generator because it's specified as such in the `from_generator()` API.

The next portion of Listing 3.2 contains two `for` loops that iterate through the elements of `ds` and display their values. Since the first `for` loop does not specify the number of elements in `ds`, that `for` loop will process all the numbers in `ds`.

Here's an important detail regarding generators in TF 2: *they only emit a single value when they are invoked*. This means that the `for` loop in the Python `gener()` function does *not* execute ten times: it executes only *once* when it is invoked, and then it "waits" until the `gener()` function is invoked again.

In case it's helpful, you can think of the `gener()` function as a "writer" that prints a single value to a pipe, and elsewhere there is some code that acts like a "reader" that reads a data value from the pipe. The code that acts as a reader is the first `for` loop that is reproduced here:

```
for value in ds:
  print("1value:",value)
```

How does the preceding code block invoke the `gener()` function when it doesn't even appear in the code? The answer is simple: the preceding code block indirectly invokes the `gener()` function because it's specified in the definition of ds, as shown here in bold:

```
ds = tf.data.Dataset.from_generator(gener, (tf.int64))
```

To summarize, each time that the preceding `for` loop executes, it invokes the Python `gener()` function, which in turn prints a value and then "waits" until it is invoked again.

The second `for` loop also acts as a "reader," and this time the code invokes the `take()` operator (it will "take" data from the dataset) that specifies *twice* the length of the `NumPy` array `x`. Why would anyone specify a length that is greater than the number of elements in the underlying array? There may be various reasons (perhaps it was accidental), so it's good to know what will happen in this situation (see if you can correctly guess the result). The output from launching the code in Listing 3.2 is here:

```
1value: tf.Tensor(1,  shape=(), dtype=int64)
1value: tf.Tensor(2,  shape=(), dtype=int64)
1value: tf.Tensor(3,  shape=(), dtype=int64)
1value: tf.Tensor(4,  shape=(), dtype=int64)
```

```
1value: tf.Tensor(5,  shape=(), dtype=int64)
1value: tf.Tensor(6,  shape=(), dtype=int64)
1value: tf.Tensor(7,  shape=(), dtype=int64)
1value: tf.Tensor(8,  shape=(), dtype=int64)
1value: tf.Tensor(9,  shape=(), dtype=int64)
1value: tf.Tensor(10, shape=(), dtype=int64)
2value: tf.Tensor(1,  shape=(), dtype=int64)
2value: tf.Tensor(2,  shape=(), dtype=int64)
2value: tf.Tensor(3,  shape=(), dtype=int64)
2value: tf.Tensor(4,  shape=(), dtype=int64)
2value: tf.Tensor(5,  shape=(), dtype=int64)
2value: tf.Tensor(6,  shape=(), dtype=int64)
2value: tf.Tensor(7,  shape=(), dtype=int64)
2value: tf.Tensor(8,  shape=(), dtype=int64)
2value: tf.Tensor(9,  shape=(), dtype=int64)
2value: tf.Tensor(10, shape=(), dtype=int64)
```

WHAT ARE ITERATORS? (OPTIONAL)

As you saw earlier in this chapter, iterators are used with `Datasets` in TF 1.x code, so if you only work with TF 2, consider this section as optional.

An iterator bears some resemblance to a "cursor" in other languages, which is to say that an iterator is something that "points" to a row of data in a dataset. By way of analogy, if you have a linked list of items, an iterator is analogous to a pointer that "points" to the first element in the list, and each time you move the pointer to the next item in the list, you are "advancing" the iterator. Working with datasets and iterators involves the following sequence of steps:

1. create a dataset
2. create an iterator (see next section)
3. "point" the iterator to the dataset
4. print the contents of the current item
5. "advance" the iterator to the next item
6. go to step 4) if there are more items

Notice that step 6 in the preceding list specifies "if there are more items," which you can handle via a `try/except` block (shown later in this chapter) when the iterator goes beyond the last item in the dataset. This technique is very useful because it obviates the need to know the number of items in a dataset. TF 1.x provides several types of iterators, as discussed in the next section.

TF 1.x Iterators (optional)

If you are working exclusively with TF 2, then this section is optional. If you are working with TensorFlow 1.x, it's probably useful to know that TF 1.x supports four types of iterators, as listed here:

1. One shot
2. Initializable

3. Reinitializable
4. Feedable

A *one-shot iterator* can iterate only once through a dataset. After we reach the end of the dataset, the iterator will no longer yield elements; instead, it will raise an `Exception`. For example, if `dx` is an instance of `tf.Dataset`, then the following code snippet defines a one-shot iterator:

```
iterator = dx.make_one_shot_iterator()
```

An *initializable iterator* can be dynamically updated: invoke its initializer operation and pass new data via the parameter `feed_dict`. If `dx` is an instance of `tf.Dataset`, then the following code snippet defines a reusable iterator:

```
iterator = dx.make_initializable_iterator()
```

A *reinitializable iterator* can be initialized from a different `Dataset`. This type of iterator is very useful for training datasets that require some additional transformation, such as shuffling their contents.

A *feedable iterator* allows you to select from different iterators: this type of iterator is essentially a "selector" to select an iterator from a collection of iterators.

Keep in mind that initializable iterators are not supported in eager mode: the alternative is to use generators.

This concludes the section regarding iterators in TF 1.x. The next section contains a code sample that illustrates how to concatenate two TF 2 `Datasets`.

CONCATENATING TF 2 `tf.Data.DatasetS`

Listing 3.3 displays the contents of `tf2_concatenate.py`, which illustrates how to concatenate two TF 2 `Datasets`.

LISTING 3.3: tf2_concatenate.py

```
import tensorflow as tf
import numpy as np

x1 = np.array([1,2,3,4,5])
x2 = np.array([6,7,8,9,10])

ds1 = tf.data.Dataset.from_tensor_slices(x1)
ds2 = tf.data.Dataset.from_tensor_slices(x2)
ds3 = ds1.concatenate(ds2)

try:
  for value in ds3.take(20):
    print("value:",value)
```

```
except tf.errors.OutOfRangeError:
  pass
```

Listing 3.3 contains two `NumPy` arrays `x1` and `x2`, followed by the TF 2 `Datasets` `ds1` and `ds2` that act as "containers" for `x1` and `x2`, respectively. Next, the dataset `ds3` is defined as the concatenation of `ds1` and `ds2`.

The next portion of Listing 3.3 is a `try/except` block that contains a `for` loop in order to display the contents of `ds3`. The output from launching the code in Listing 3.4 is here:

```
ds3 value: tf.Tensor(1, shape=(), dtype=int64)
ds3 value: tf.Tensor(2, shape=(), dtype=int64)
ds3 value: tf.Tensor(3, shape=(), dtype=int64)
ds3 value: tf.Tensor(4, shape=(), dtype=int64)
ds3 value: tf.Tensor(5, shape=(), dtype=int64)
ds3 value: tf.Tensor(6, shape=(), dtype=int64)
ds3 value: tf.Tensor(7, shape=(), dtype=int64)
ds3 value: tf.Tensor(8, shape=(), dtype=int64)
ds3 value: tf.Tensor(9, shape=(), dtype=int64)
ds3 value: tf.Tensor(10, shape=(), dtype=int64)
```

One other point to keep in mind: different structures *cannot* be concatenated. For example, consider the variables `y1` and `y2`:

```
# y1 = { (8, 9), (10, 11), (12, 13) }
# y2 = { 14.0, 15.0, 16.0 }
```

If you create a TF 2 `Dataset` from `y1` and `y2`, the resulting datasets cannot be concatenated to `ds1`.

THE TF 2 reduce() OPERATOR

The TF 2 `reduce()` operator performs a reduction on its input until a single value is produced. For example, you can use the `reduce()` operator to add all the numbers in an array. Listing 3.4 displays the contents of `tf2_reduce.py`, which illustrates how to use the `reduce()` API in TF 2.

LISTING 3.4: tf2_reduce.py

```
import tensorflow as tf
import numpy as np

x1 = tf.data.Dataset.range(8).reduce(np.int64(0),lambda x,
_: x + 1)
x2 = tf.data.Dataset.range(8).reduce(np.int64(0),lambda x,
y: x + y)

print("x1:",x1)
print("x2:",x2)
```

Listing 3.4 defines the variables x1 and x2 as instances of tf.data.Dataset, which in turn is based on the digits from 0 to 7 inclusive. Notice that x1 and x2 specify different lambda expressions. The lambda expression for x1 returns its input value incremented by one. Since the largest number in the input set of values is 7, the last output value is 8.

On the other hand, x2 defines a lambda expression that returns the sum of two consecutive input values. The initial sum is 0, so the final output equals the sum of the numbers 1, 2, . . . , and 7, which equals 28. The output from launching the code in Listing 3.4 is here:

```
x1: tf.Tensor(8,  shape=(), dtype=int64)
x2: tf.Tensor(28, shape=(), dtype=int64)
```

WORKING WITH GENERATORS IN TF 2

Earlier in the chapter you were introduced to TF 2 *generators*, which are a Python function (for our code samples, let's just name this function gener()) that works somewhat like a "pipe": you read a single value each time that the gener() function is invoked. You can also think of a TF 2 generator as a function that "emits" one value when the function is invoked. [If you are familiar with the Go programming language, this is essentially the same as a "channel."]

After emitting the last available value, the "pipe" no longer returns any values. Contrary to what you might expect, no error message is displayed when the "pipe" is empty.

Now that you understand the underlying behavior of a generator in TF 2, let's look at the following code snippet (which you've seen already) that shows you how to define a TF 2 tf.data.Dataset that involves a generator:

```
ds = tf.data.Dataset.from_generator(gener, (tf.int64))
```

If you read the previous code snippet in English, it is as follows: "the Dataset ds obtains its values from the Python function gener() that emits a value of type tf.int64." If you iterate through the values of ds via a for loop, the gener() function is invoked and it will "yield" a single value. Hence, the number of times your code iterates through the values of ds equals the number of times that the gener() function is invoked.

Listing 3.5 displays the contents of tf2_generator1.py, which illustrates how to define a generator in TF 2 that "yields" a value that is three times its input value.

LISTING 3.5: tf2_generator1.py

```
import tensorflow as tf
import numpy as np

x = np.arange(0, 10)
```

```
def gener():
  for i in x:
    yield (3*i)

ds = tf.data.Dataset.from_generator(gener, (tf.int64))

for value in ds.take(len(x)):
  print("value:",value)

for value in ds.take(2*len(x)):
  print("value:",value)
```

Listing 3.5 contains the NumPy variable x that contains the digits from 0 to 9 inclusive. The next portion of Listing 3.4 defines the Python function gener() that contains a for loop that iterates through the values in x. Notice that it's not necessary to specify a @tf.function decorator, because the definition of ds specifies the Python function gener() as a generator.

However, recall that the yield keyword performs a parsimonious operation: it "yields" only a single value. In this example, the variable i ranges from 0 to 9, but the first invocation of gener() returns only the value 3*0 because i equals 0.

The next invocation of gener() returns the value 3*1 because i equals 1. Each subsequent invocation of gener() returns the sequence of values 3*2, 3*3, . . . , 3*9. In a sense, the for loop in the gener() function is a "stateful" loop in the sense that it "remembers" the current value of i during subsequent invocations of the gener() function.

The output from launching the code in Listing 3.5 is here:

```
value: tf.Tensor(0,  shape=(), dtype=int64)
value: tf.Tensor(3,  shape=(), dtype=int64)
value: tf.Tensor(6,  shape=(), dtype=int64)
value: tf.Tensor(9,  shape=(), dtype=int64)
value: tf.Tensor(12, shape=(), dtype=int64)
value: tf.Tensor(15, shape=(), dtype=int64)
value: tf.Tensor(18, shape=(), dtype=int64)
value: tf.Tensor(21, shape=(), dtype=int64)
value: tf.Tensor(24, shape=(), dtype=int64)
value: tf.Tensor(27, shape=(), dtype=int64)
value: tf.Tensor(0,  shape=(), dtype=int64)
value: tf.Tensor(3,  shape=(), dtype=int64)
value: tf.Tensor(6,  shape=(), dtype=int64)
value: tf.Tensor(9,  shape=(), dtype=int64)
value: tf.Tensor(12, shape=(), dtype=int64)
value: tf.Tensor(15, shape=(), dtype=int64)
value: tf.Tensor(18, shape=(), dtype=int64)
value: tf.Tensor(21, shape=(), dtype=int64)
value: tf.Tensor(24, shape=(), dtype=int64)
value: tf.Tensor(27, shape=(), dtype=int64)
```

THE TF 2 filter() OPERATOR (1)

The `filter()` operator uses Boolean logic to "filter" the elements in an array in order to determine which elements satisfy the Boolean condition. As an analogy, if you hold a piece of smoked glass in front of your eyes, the glass will "filter out" a portion of the light spectrum. A filter in TF 2 performs an analogous function: it generally results in a subset of the original set. [A filter that returns every input element is technically possible, but it's also pointless.]

As a simple example, suppose that we have a `NumPy` array `[1,2,3,4]` and we want to select only the *even* numbers in this array. The result is [2,4], whose contents are a subset of the original array. Listing 3.6 displays the contents of `tf2_filter.py`, which illustrates how to use the `filter()` operator in TF 2.

LISTING 3.6: tf2_filter.py

```
import tensorflow as tf
import numpy as np

#def filter_fn(x):
#  return tf.reshape(tf.not_equal(x % 2, 1), [])

x = np.array([1,2,3,4,5,6,7,8,9,10])

ds = tf.data.Dataset.from_tensor_slices(x)
ds = ds.filter(lambda x: tf.reshape(tf.not_equal(x%2,1),
[]))
#ds = ds.filter(filter_fn)

for value in ds:
  print("value:",value)
```

Listing 3.6 initializes the variable `x` as a `NumPy` array consisting of the integers from 1 through 10 inclusive. Next, the variable `ds` is initialized as a TF 2 `Dataset` that is created from the contents of the variable `x`. The next code snippet invokes the `filter()` operator, inside of which a lambda expression returns only even numbers because of this expression:

```
tf.not_equal(x%2,1)
```

The next portion of Listing 3.6 is a `for` loop that iterates through the elements of the dataset `ds`. The output from launching the code in Listing 3.6 is here:

```
value: tf.Tensor(2,  shape=(), dtype=int64)
value: tf.Tensor(4,  shape=(), dtype=int64)
value: tf.Tensor(6,  shape=(), dtype=int64)
```

```
value: tf.Tensor(8,  shape=(), dtype=int64)
value: tf.Tensor(10, shape=(), dtype=int64)
```

THE TF 2 filter() OPERATOR (2)

Listing 3.7 displays the contents of `tf2_filter2.py`, which illustrates another example of the `filter()` operator in TF 2.

LISTING 3.7: tf2_filter2.py

```
import tensorflow as tf
import numpy as np

ds = tf.data.Dataset.from_tensor_slices([1,2,3,4,5])
ds = ds.filter(lambda x: x < 4) # [1,2,3]

print("First iteration:")
for value in ds:
  print("value:",value)

# "tf.math.equal(x, y)" is required for equality comparison
def filter_fn(x):
  return tf.math.equal(x, 1)

ds = ds.filter(filter_fn)

print("Second iteration:")
for value in ds:
  print("value:",value)
```

Listing 3.7 defines the variable `ds` as a TF 2 `Dataset` that is created from the array `[1,2,3,4,5]`. The next code snippet invokes the `filter()` operator, inside of which a lambda expression returns numbers that are less than 4. The `for` loop prints the numbers in the `ds` variable, which consist of the “filtered” list of digits 1, 2, and 3.

The next portion of Listing 3.7 is the decorated Python function `filter_fn()` that is specified as part of the new definition of `ds`, as shown here:

```
ds = ds.filter(filter_fn)
```

The preceding code snippet executes the decorated Python function `filter_fn()` in the *second* `for` loop in Listing 3.7. The output from launching the code in Listing 3.7 is here:

```
First iteration:
value: tf.Tensor(1, shape=(), dtype=int32)
value: tf.Tensor(2, shape=(), dtype=int32)
value: tf.Tensor(3, shape=(), dtype=int32)
Second iteration:
value: tf.Tensor(1, shape=(), dtype=int32)
```

THE TF 2 batch() OPERATOR (1)

The `batch(n)` operator processes a "batch" of n elements during each iteration. Listing 3.8 displays the contents of `tf2_batch1.py`, which illustrates how to use the `batch()` operator in TF 2.

LISTING 3.8: tf2_batch1.py

```
import tensorflow as tf
import numpy as np

x = np.arange(0, 34)
ds = tf.data.Dataset.from_tensor_slices(x).batch(3)

for value in ds:
  print("value:",value)
```

Listing 3.8 initializes the variable `x` as a `NumPy` array consisting of the integers from 0 through 33 inclusive (note that this array contains 34 numbers). Next, the variable `ds` is initialized as a TF 2 `Dataset` that is a container for the contents of the variable `x`. Notice that the definition of `x` involves method chaining by "tacking on" the `batch(3)` operator as part of the definition of `ds`.

The final portion of Listing 3.8 contains a loop that iterates through the elements of the dataset `ds`. Now launch the code in Listing 3.8 to see the output in its entirety, as shown here:

```
tf.Tensor([0 1 2],    shape=(3,), dtype=int64)
tf.Tensor([3 4 5],    shape=(3,), dtype=int64)
tf.Tensor([6 7 8],    shape=(3,), dtype=int64)
tf.Tensor([ 9 10 11], shape=(3,), dtype=int64)
tf.Tensor([12 13 14], shape=(3,), dtype=int64)
tf.Tensor([15 16 17], shape=(3,), dtype=int64)
tf.Tensor([18 19 20], shape=(3,), dtype=int64)
tf.Tensor([21 22 23], shape=(3,), dtype=int64)
tf.Tensor([24 25 26], shape=(3,), dtype=int64)
tf.Tensor([27 28 29], shape=(3,), dtype=int64)
tf.Tensor([30 31 32], shape=(3,), dtype=int64)
tf.Tensor([33],       shape=(1,), dtype=int64)
```

THE TF 2 batch() OPERATOR (2)

Listing 3.9 displays the contents of `tf2_generator2.py`, which illustrates how to use a generator function to display "batches" of numbers.

LISTING 3.9: tf2_generator2.py

```
import tensorflow as tf
import numpy as np
```

```
x = np.arange(0, 12)

def gener():
  i = 0
  while(i < len(x/3)):
    yield (i, i+1, i+2)
    i += 3

ds = tf.data.Dataset.from_generator(gener, (tf.int64,tf.
int64,tf.int64))

third = int(len(x)/3)
for value in ds.take(third):
  print("value:",value)
```

Listing 3.9 initializes the variable `x` as a `NumPy` array consisting of the integers from 0 through 12 inclusive. The Python function `gener()` returns a "triple" of three consecutive numbers from the `NumPy` array x. Since the next code snippet invokes the `from_generator()` API with the `gener()` function, the latter is treated as a generator (you saw an example of this behavior earlier in this chapter).

The final portion of Listing 3.9 contains a `for` loop that iterates through the elements of `ds`, printing three consecutive values during each `print()` statement. The output from launching the code in Listing 3.9 is here:

```
value: (<tf.Tensor: id=34, shape=(), dtype=int64, numpy=0>,
<tf.Tensor: id=35, shape=(), dtype=int64, numpy=1>, <tf.
Tensor: id=36, shape=(), dtype=int64, numpy=2>)
value: (<tf.Tensor: id=40, shape=(), dtype=int64, numpy=3>,
<tf.Tensor: id=41, shape=(), dtype=int64, numpy=4>, <tf.
Tensor: id=42, shape=(), dtype=int64, numpy=5>)
value: (<tf.Tensor: id=46, shape=(), dtype=int64, numpy=6>,
<tf.Tensor: id=47, shape=(), dtype=int64, numpy=7>, <tf.
Tensor: id=48, shape=(), dtype=int64, numpy=8>)
value: (<tf.Tensor: id=52, shape=(), dtype=int64, numpy=9>,
<tf.Tensor: id=53, shape=(), dtype=int64, numpy=10>, <tf.
Tensor: id=54, shape=(), dtype=int64, numpy=11>)
```

The companion files contains `tf2_generator1.py` and `tf2_generator3.py`, which illustrate variations of the preceding code sample. Experiment with the code by changing the hard-coded values and then see if you can correctly predict the output.

THE TF 2 `map()` OPERATOR (1)

The `map()` operator is often defined as a projection, and while this is technically correct, the actual behavior might not be clear. Here's the basic idea: when you provide a list or an array of values as input for the `map()` operator, this operator "applies" a lambda expression to each input element.

For example, the lambda expression `lambda x: x*2` returns twice its input value `x`, whereas the lambda expression `lambda x: x/2` returns half its input value `x`. In both lambda expressions the input list and the output list have the same number of elements. In many cases the values in the two lists are different, but there are many exceptions. For example, the lambda expression `lambda x: x%2` returns the value 0 for even numbers and the value 1 for odd numbers, so the output consists of two distinct numbers, whereas the input list can be arbitrarily large. Listing 3.10 displays the contents of `tf2_map.py`, which illustrates a complete example of the `map()` operator in TF 2.

LISTING 3.10: tf2_map.py

```
import tensorflow as tf
import numpy as np

x = np.array([[1],[2],[3],[4]])
ds = tf.data.Dataset.from_tensor_slices(x)
ds = ds.map(lambda x: x*2)

for value in ds:
  print("value:",value)
```

Listing 3.10 initializes the variable `x` as a `NumPy` array consisting of four elements, where each element is a 1x1 array consisting of the numbers 1, 2, 3, and 4. Next, the variable `ds` is initialized as a TF 2 `Dataset` that is created from the contents of the variable `x`. Notice how `ds.map()` then defines a lambda expression that doubles each input value (which takes the value of each integer from 1 to 4) in this example.

The final portion of Listing 3.10 contains a `for` loop that iterates through the elements of `ds` and displays their values. The output from launching the code in Listing 3.10 is here:

```
value: tf.Tensor([2], shape=(1,), dtype=int64)
value: tf.Tensor([4], shape=(1,), dtype=int64)
value: tf.Tensor([6], shape=(1,), dtype=int64)
value: tf.Tensor([8], shape=(1,), dtype=int64)
```

THE TF 2 `map()` OPERATOR (2)

Listing 3.11 displays the contents of `tf2_map2.py`, which illustrates two techniques for defining a dataset, as well as how to invoke multiple occurrences of the `map()` operator in TF 2.

LISTING 3.11: tf2_map2.py

```
import tensorflow as tf
import numpy as np
```

```
# a simple Numpy array
x = np.array([[1],[2],[3],[4]])

# make a dataset from a Numpy array
dataset = tf.data.Dataset.from_tensor_slices(x)

# METHOD #1: THE LONG WAY
# a lambda expression to double each value
#dataset = dataset.map(lambda x: x*2)
# a lambda expression to add one to each value
#dataset = dataset.map(lambda x: x+1)
# a lambda expression to cube each value
#dataset = dataset.map(lambda x: x**3)

# METHOD #2: A SHORTER WAY
dataset = dataset.map(lambda x: x*2).map(lambda x: x+1).
map(lambda x: x**3)

for value in ds:
  print("value:",value)
```

Listing 3.11 initializes the variable x as a NumPy array consisting of four elements, where each element is a 1x1 array consisting of the numbers 1, 2, 3, and 4. Next, the variable dataset is initialized as a TF 2 Dataset that is created from the contents of the variable x.

The next portion of Listing 3.11 is a “commented out” code block that consists of three lambda expressions, followed by a code snippet (shown in bold) that uses method chaining in order to produce a more compact way of invoking the same three lambda expressions:

```
dataset = dataset.map(lambda x: x*2).map(lambda x: x+1).
map(lambda x: x**3)
```

The preceding code snippet transforms each input value by first doubling the value, then adding one to the output from the first lambda expression, and then cubing the output from the second lambda expression.

Although method chaining is a concise way to combine operators, invoking many lazy operators in a single (very long) line of code can also become difficult to understand, whereas writing code using the “longer way” would be easier to debug.

A suggestion: start with each lazy operator in a separate line of code, and after you are satisfied that the individual results are correct, *then* use method chaining to combine the operators into a single line of code (perhaps up to a maximum of four or five lazy operators).

The final portion of Listing 3.11 contains a for loop that iterates through the transformed values and displays their values. The output from launching the code in Listing 3.11 is here:

```
value: tf.Tensor([27],  shape=(1,), dtype=int64)
value: tf.Tensor([125], shape=(1,), dtype=int64)
```

```
value: tf.Tensor([343], shape=(1,), dtype=int64)
value: tf.Tensor([729], shape=(1,), dtype=int64)
```

THE TF 2 flatmap() OPERATOR (1)

In addition to the TF 2 `map()` operator, TF 2 also supports the TF 2 `flat_map()` operator. However, the TF 2 `map()` and TF 2 `flat_map()` operators expect functions with different signatures. Specifically, `map()` takes a function that maps a single element of the input dataset to a single new element, whereas `flat_map()` takes a function that maps a single element of the input dataset to a `Dataset` of elements.

Listing 3.12 displays the contents of `tf2_flatmap1.py`, which illustrates how to use the `flatmap()` operator in TF 2.

LISTING 3.12: tf2_flatmap1.py

```
import tensorflow as tf
import numpy as np

x = np.array([[1,2,3], [4,5,6], [7,8,9]])

ds = tf.data.Dataset.from_tensor_slices(x)
ds.flat_map(lambda x: tf.data.Dataset.from_tensor_
slices(x))

for value in ds.take(3):
  print("value:",value)
```

Listing 3.12 initializes the variable `x` as a `NumPy` array consisting of three elements, where each element is a 1x3 array of numbers. Next, the variable `ds` is initialized as a TF 2 `Dataset` that is a container for the contents of the variable `x`.

The final portion of Listing 3.12 contains a `for` loop that iterates through the elements of `dataset` and displays their values. Once again, note that the `try/except` block is unnecessary, even if the `take()` method specifies a number that is greater than the number of elements in `ds`. The output from launching the code in Listing 3.12 is here:

```
value: tf.Tensor([1 2 3], shape=(3,), dtype=int64)
value: tf.Tensor([4 5 6], shape=(3,), dtype=int64)
value: tf.Tensor([7 8 9], shape=(3,), dtype=int64)
```

THE TF 2 `flatmap()` OPERATOR (2)

The code in the previous section works fine, but there is a hard-coded value 3 in the code block that displays the elements of the dataset. The code sample in this section removes the hard-coded value.

Listing 3.13 displays the contents of `tf2_flatmap2.py`, which illustrates how to use the `flatmap()` operator in TF 2 and then iterate through the elements of the dataset.

LISTING 3.13: tf2_flatmap2.py

```
import tensorflow as tf
import numpy as np

x = np.array([[1,2,3], [4,5,6], [7,8,9]])

ds = tf.data.Dataset.from_tensor_slices(x)
ds.flat_map(lambda x: tf.data.Dataset.from_tensor_
slices(x))

for value in ds:
  print("value:",value)
```

Listing 3.13 initializes the variable `x` as a `NumPy` array consisting of three elements, where each element is a 1x3 array of numbers. Next, the variable `ds` is initialized as a TF 2 `Dataset` that is created from the contents of the variable `x`.

The final portion of Listing 3.13 iterates through the elements of `ds` and displays their values. The `for` loop iterates through the elements of `ds`. The output from launching the code in Listing 3.13 is the same as the output from Listing 3.12:

```
value: tf.Tensor([1 2 3], shape=(3,), dtype=int64)
value: tf.Tensor([4 5 6], shape=(3,), dtype=int64)
value: tf.Tensor([7 8 9], shape=(3,), dtype=int64)
```

THE TF 2 flat_map() AND filter() OPERATORS

Listing 3.14 displays the contents of `comments.txt`, and Listing 3.15 displays the contents of `tf2_flatmap_filter.py`, which illustrates how to use the `filter()` operator in TF 2.

LISTING 3.14: comments.txt

```
#this is file line #1
#this is file line #2
this is file line #3
this is file line #4
#this is file line #5
```

LISTING 3.15: tf2_flatmap_filter.py

```
import tensorflow as tf
```

```
filenames = ["comments.txt"]

ds = tf.data.Dataset.from_tensor_slices(filenames)

# 1) Use Dataset.flat_map() to transform each file
#    as a separate nested ds, then concatenate their
#    contents sequentially into a single "flat" ds
# 2) Skip the first line (header row)
# 3) Filter out lines beginning with "#" (comments)

ds = ds.flat_map(
    lambda filename: (
      tf.data.TextLineDataset(filename)
      .skip(1)
      .filter(lambda line:
tf.not_equal(tf.strings.substr(line,0,1),"#"))))

for value in ds.take(2):
  print("value:",value)
```

Listing 3.15 defines the variable `filenames` as an array of text filenames, which in this case consists of just one text file named `comments.txt` (whose contents are shown in Listing 3.14). Next, the variable `dataset` is initialized as a TF 2 `Dataset` that contains the contents of `comments.txt`.

The next section of Listing 3.15 is a comment block that explains the purpose of the subsequent code block that defines the variable `ds`. As you can see, `ds` involves a small set of operations that are executed via method chaining in order to perform various transformations on the contents of the variable `ds`.

Specifically, the `flat_map()` operator "flattens" whatever is returned by the nested lambda expression, which involves several transformations. The first transformation involves passing each input filename, one at a time, to the `tf.data.TextLineDataset` class. The second transformation skips the first line of text from the current input file. The third transformation invokes a `filter()` operator that specifies another lambda expression with conditional logic, as shown here:

```
tf.not_equal(tf.strings.substr(line,0,1),"#"))
```

The preceding code snippet returns the current line of text (from the currently processed text file) if and only if the character in the first position of the line of text is not the character "#"; otherwise, nothing is returned (i.e., the line of text is skipped). These transformations can be summarized as follows: "for each input file, skip the first line, and print any subsequent lines that do not start with the character #."

The final portion of Listing 3.15 prints two lines of output, which might seem anticlimactic after defining such a fancy set of transformations! Launch the code in Listing 3.15 and you will see the following output:

```
value: tf.Tensor(b'this is file line #3 ', shape=(),
dtype=string)
value: tf.Tensor(b'this is file line #4 ', shape=(),
dtype=string)
```

THE TF 2 repeat() OPERATOR

The `repeat(n)` operator simply repeats its input values `n` times. Listing 3.16 displays the contents of `tf2_repeat.py`, which illustrates how to use the `repeat()` operator in TF 2.

LISTING 3.16: tf2_repeat.py

```
import tensorflow as tf

ds = tf.data.Dataset.from_tensor_slices(tf.range(4))
ds = ds.repeat(2)

for value in ds.take(20):
  print("value:",value)
```

Listing 3.16 initializes the variable `ds1` as a TF 2 `Dataset` that is created from the integers between 0 and 3 inclusive. The next code snippet "tacks on" the `repeat()` operator to `ds`, which has the effect of appending the contents of `ds` to itself. Hence, ds contains eight numbers: the numbers from 0 through 3 inclusive, and again the numbers 0 through 3 inclusive.

The final portion of Listing 3.16 contains a `for` loop that iterates through the elements of the dataset `ds`. Although the `take()` method specifies the number 20, the loop is only executed 8 times because the `repeat()` operator specifies the value 2. The output from launching the code in Listing 3.16 is here:

```
value: tf.Tensor(0, shape=(), dtype=int32)
value: tf.Tensor(1, shape=(), dtype=int32)
value: tf.Tensor(2, shape=(), dtype=int32)
value: tf.Tensor(3, shape=(), dtype=int32)
value: tf.Tensor(0, shape=(), dtype=int32)
value: tf.Tensor(1, shape=(), dtype=int32)
value: tf.Tensor(2, shape=(), dtype=int32)
value: tf.Tensor(3, shape=(), dtype=int32)
```

THE TF 2 take() OPERATOR

The `take(n)` operator "takes" `n` input values. Listing 3.17 displays the contents of `tf2_take.py`, which illustrates another example of the `take()` operator in TF 2.

LISTING 3.17: tf2_take.py

```
import tensorflow as tf

ds = tf.data.Dataset.from_tensor_slices(tf.range(8))
ds = ds.take(5)

for value in ds.take(20):
  print("value:",value)
```

Listing 3.17 initializes the variable `ds1` as a TF 2 `Dataset` that is created from the integers between 0 and 7 inclusive. The next code snippet "tacks on" the `take()` operator to `ds`, which has the effect of limiting the output to the first five integers.

The final portion of Listing 3.17 contains a `for` loop that iterates through the elements of the dataset `ds`. See the code in the preceding section for an explanation of the how the output is generated. The output from launching the code in Listing 3.17 is here:

```
value: tf.Tensor(0, shape=(), dtype=int32)
value: tf.Tensor(1, shape=(), dtype=int32)
value: tf.Tensor(2, shape=(), dtype=int32)
value: tf.Tensor(3, shape=(), dtype=int32)
value: tf.Tensor(4, shape=(), dtype=int32)
```

COMBINING THE TF 2 `map()` AND `take()` OPERATORS

Listing 3.18 displays the contents of `tf2_map_take.py`, which illustrates how to use method chaining in order to invoke the `map()` operator three times, using three different lambda expressions, followed by the `take()` operator in TF 2.

LISTING 3.18: tf2_map_take.py

```
import tensorflow as tf
import numpy as np

x = np.array([[1],[2],[3],[4]])

# make a ds from a numpy array
ds = tf.data.Dataset.from_tensor_slices(x)
ds = ds.map(lambda x: x*2).map(lambda x: x+1).map(lambda x:
x**3)

for value in ds.take(4):
  print("value:",value)
```

Listing 3.18 initializes the variable `x` as a `NumPy` array consisting of four elements, where each element is a 1x1 array consisting of the numbers 1, 2, 3, and 4. Next, the variable `dataset` is initialized as a TF 2 `Dataset` that is created from the contents of the variable `x`. The next portion of Listing 3.18 involves three lambda expressions that are shown in bold and reproduced here:

```
ds = ds.map(lambda x: x*2).map(lambda x: x+1).map(lambda x:
x**3)
```

The preceding code snippet transforms each input value by first doubling the value, then adding one to the first result, and then cubing the second result.

The final portion of Listing 3.18 "takes" the first four elements from the variable `dataset` and displays their contents, as shown here:

```
value: tf.Tensor([27],  shape=(1,), dtype=int64)
value: tf.Tensor([125], shape=(1,), dtype=int64)
value: tf.Tensor([343], shape=(1,), dtype=int64)
value: tf.Tensor([729], shape=(1,), dtype=int64)
```

COMBINING THE TF 2 `zip()` AND `batch()` OPERATORS

Listing 3.19 displays the contents of `tf2_zip_batch.py`, which illustrates how to combine the `zip()` and `batch()` operators in TF 2.

LISTING 3.19: tf2_zip_batch.py

```
import tensorflow as tf

ds1 = tf.data.Dataset.range(100)
ds2 = tf.data.Dataset.range(0, -100, -1)
ds3 = tf.data.Dataset.zip((ds1, ds2))
ds4 = ds3.batch(4)

for value in ds.take(10):
  print("value:",value)
```

Listing 3.19 initializes the variables `ds1`, `ds2`, `ds3`, and `ds4` as TF 2 `Datasets` that are created successively starting from `ds1`, which contains the integers between 0 and 99 inclusive. The variable `ds2` is initialized via the `range()` operator that starts from 0 and is decreased to -99, and the variable `ds3` is initialized via the `zip()` operator that processes two elements at a time, in a pairwise fashion. Next, the variable `ds3` is initialized by invoking the `batch()` operator on the variable `ds3`. The final portion of Listing 3.19 prints three lines of "batched" output, as shown here:

```
value: (<tf.Tensor: id=20, shape=(4,), dtype=int64,
numpy=array([0, 1, 2, 3])>, <tf.Tensor: id=21, shape=(4,),
dtype=int64, numpy=array([ 0, -1, -2, -3])>)
value: (<tf.Tensor: id=24, shape=(4,), dtype=int64,
numpy=array([4, 5, 6, 7])>, <tf.Tensor: id=25, shape=(4,),
dtype=int64, numpy=array([-4, -5, -6, -7])>)
```

```
value: (<tf.Tensor: id=28, shape=(4,), dtype=int64,
numpy=array([ 8,  9, 10, 11])>, <tf.Tensor: id=29,
shape=(4,), dtype=int64, numpy=array([ -8,  -9, -10,
-11])>)
value: (<tf.Tensor: id=32, shape=(4,), dtype=int64,
numpy=array([12, 13, 14, 15])>, <tf.Tensor: id=33,
shape=(4,), dtype=int64, numpy=array([-12, -13, -14,
-15])>)
value: (<tf.Tensor: id=36, shape=(4,), dtype=int64,
numpy=array([16, 17, 18, 19])>, <tf.Tensor: id=37,
shape=(4,), dtype=int64, numpy=array([-16, -17, -18,
-19])>)
value: (<tf.Tensor: id=40, shape=(4,), dtype=int64,
numpy=array([20, 21, 22, 23])>, <tf.Tensor: id=41,
shape=(4,), dtype=int64, numpy=array([-20, -21, -22,
-23])>)
value: (<tf.Tensor: id=44, shape=(4,), dtype=int64,
numpy=array([24, 25, 26, 27])>, <tf.Tensor: id=45,
shape=(4,), dtype=int64, numpy=array([-24, -25, -26,
-27])>)
value: (<tf.Tensor: id=48, shape=(4,), dtype=int64,
numpy=array([28, 29, 30, 31])>, <tf.Tensor: id=49,
shape=(4,), dtype=int64, numpy=array([-28, -29, -30,
-31])>)
value: (<tf.Tensor: id=52, shape=(4,), dtype=int64,
numpy=array([32, 33, 34, 35])>, <tf.Tensor: id=53,
shape=(4,), dtype=int64, numpy=array([-32, -33, -34,
-35])>)
value: (<tf.Tensor: id=56, shape=(4,), dtype=int64,
numpy=array([36, 37, 38, 39])>, <tf.Tensor: id=57,
shape=(4,), dtype=int64, numpy=array([-36, -37, -38,
-39])>)
```

For your convenience, here is a slightly more condensed and clearer version of the output from Listing 3.19:

```
[ 0, 1, 2, 3],     [ 0, -1, -2, -3]
[ 4, 5, 6, 7],     [-4, -5, -6, -7]
[ 8,  9, 10, 11], [ -8,  -9, -10, -11]
[12, 13, 14, 15], [-12, -13, -14, -15]
[16, 17, 18, 19], [-16, -17, -18, -19]
[20, 21, 22, 23], [-20, -21, -22, -23]
[24, 25, 26, 27], [-24, -25, -26, -27]
[28, 29, 30, 31], [-28, -29, -30, -31]
[32, 33, 34, 35], [-32, -33, -34, -35]
[36, 37, 38, 39], [-36, -37, -38, -39]
[40, 41, 42, 43], [-40, -41, -42, -43]
[44, 45, 46, 47], [-44, -45, -46, -47]
. . . .
[96, 97, 98, 99], [-96, -97, -98, -99]
```

COMBINING THE TF 2 `zip()` AND `take()` OPERATORS

The `zip()` operator processes two elements at a time, in a pairwise fashion. Think of two lines of people waiting at the entrance to a movie theater with double doors. After opening the doors, a "pair" of people—one from each "line"—enters the theater.

Listing 3.20 displays the contents of `tf2_zip_take.py`, which illustrates how to combine the `zip()` and `take()` operators in TF 2.

LISTING 3.20: tf2_zip_take.py

```
import tensorflow as tf
import numpy as np

x = np.arange(0, 10)
y = np.arange(1, 11)

# create dataset objects from the arrays
dx = tf.data.Dataset.from_tensor_slices(x)
dy = tf.data.Dataset.from_tensor_slices(y)

# zip the two datasets together
d2 = tf.data.Dataset.zip((dx, dy)).batch(3)

for value in d2.take(8):
  print("value:",value)
```

Listing 3.20 initializes the variables `x` and `y` as a range of integers from 0 to 9 and from 1 to 10, respectively. Next, the variables `dx` and `dy` are initialized as TF 2 `Datasets` that are created from the contents of the variables `x` and `y`, respectively.

The next code snippet defines the variable `d2` as a TF 2 `Dataset` that combines the elements from `dx` and `dy` in a pairwise fashion via the `zip()` operator, as shown here:

```
d2 = tf.data.Dataset.zip((dx, dy)).batch(3)
```

Notice how method chaining is performed by "tacking on" the `batch(3)` operator as part of the definition of `dcomb`.

The final portion of Listing 3.20 contains a loop that executes fifteen times, and during each iteration the loop prints the current contents of the variable `iterator`. Each line of output consists of two "blocks" of numbers, where a block consists of three consecutive integers. The output from launching the code in Listing 3.20 is here:

```
value: (<tf.Tensor: id=16, shape=(3,), dtype=int64,
numpy=array([0, 1, 2])>, <tf.Tensor: id=17, shape=(3,),
dtype=int64, numpy=array([1, 2, 3])>)
```

```
value: (<tf.Tensor: id=20, shape=(3,), dtype=int64,
numpy=array([3, 4, 5])>, <tf.Tensor: id=21, shape=(3,),
dtype=int64, numpy=array([4, 5, 6])>)
value: (<tf.Tensor: id=24, shape=(3,), dtype=int64,
numpy=array([6, 7, 8])>, <tf.Tensor: id=25, shape=(3,),
dtype=int64, numpy=array([7, 8, 9])>)
value: (<tf.Tensor: id=28, shape=(1,), dtype=int64,
numpy=array([9])>, <tf.Tensor: id=29, shape=(1,),
dtype=int64, numpy=array([10])>)
```

TF 2 tf.data.Datasets and Random Numbers

Listing 3.21 displays the contents of `tf2_generator3.py`, which illustrates how to create a TF 2 `Dataset` with random numbers.

LISTING 3.21: tf2_generator3.py

```
import tensorflow as tf
import numpy as np

x = np.random.sample((8,2))
size = x.shape[0]

def gener():
  for i in range(0,size):
    yield (x[i][0], x[i][1])

ds = tf.data.Dataset.from_generator(gener, (tf.float64,tf.
float64))

for value in ds:
  print("value:",value)
```

Listing 3.21 initializes the variable `x` as a `NumPy` array consisting of 100 rows and 2 columns of randomly generated numbers. Next, the variable `ds` is initialized as a TF 2 `Dataset` that is created from the contents of the variable `x`.

The next portion of Listing 3.21 defines the Python function `gener()`, which is a generator, for the same reason that has been discussed in previous code samples. The final portion of Listing 3.21 prints the first line of transformed data, as shown here:

```
value: (<tf.Tensor: id=32, shape=(), dtype=float64,
numpy=0.20591749665857995>, <tf.Tensor: id=33, shape=(),
dtype=float64, numpy=0.5990477322965386>)
value: (<tf.Tensor: id=36, shape=(), dtype=float64,
numpy=0.4384201871832957>, <tf.Tensor: id=37, shape=(),
dtype=float64, numpy=0.516920941899825б>)
```

```
value: (<tf.Tensor: id=40, shape=(), dtype=float64,
numpy=0.587374875326609>, <tf.Tensor: id=41, shape=(),
dtype=float64, numpy=0.8141864916735249>)
value: (<tf.Tensor: id=44, shape=(), dtype=float64,
numpy=0.05471699195088109>, <tf.Tensor: id=45, shape=(),
dtype=float64, numpy=0.806596986559444>)
value: (<tf.Tensor: id=48, shape=(), dtype=float64,
numpy=0.887837922295610 6>, <tf.Tensor: id=49, shape=(),
dtype=float64, numpy=0.9533861033011681>)
value: (<tf.Tensor: id=52, shape=(), dtype=float64,
numpy=0.4504035573049521>, <tf.Tensor: id=53, shape=(),
dtype=float64, numpy=0.6303139480618501>)
value: (<tf.Tensor: id=56, shape=(), dtype=float64,
numpy=0.84588294357816>, <tf.Tensor: id=57, shape=(),
dtype=float64, numpy=0.916291642540712>)
value: (<tf.Tensor: id=60, shape=(), dtype=float64,
numpy=0.8851826544276614>, <tf.Tensor: id=61, shape=(),
dtype=float64, numpy=0.6337544549532578>)
```

TF 2, MNIST, and tf.data.Dataset

In addition to creating a dataset from NumPy arrays of data or from Pandas Dataframes, you can create a dataset from existing datasets. For example, Listing 3.22 displays the contents of tf2_mnist.py, which illustrates how to create a tf.data.Dataset from the MNIST dataset.

LISTING 3.22: tf2_mnist.py

```
import tensorflow as tf

train, test = tf.keras.datasets.mnist.load_data()
mnist_x, mnist_y = train

print("mnist_x.shape:",mnist_x.shape)
print("mnist_y.shape:",mnist_y.shape)

mnist_ds = tf.data.Dataset.from_tensor_slices(mnist_x)
#print(mnist_ds)

for value in mnist_ds:
  print("value:",value)
```

Listing 3.22 initializes the variables train and test from the MNIST dataset, and then initializes the variables mnist_x and mnist_y from the train variable. The next code snippet initializes the mnist_ds variable as a tf.data.Dataset that is created from the mnist_x variable. The next portion of Listing 3.22 contains a for loop that iterates through the elements in mnist_ds.

The complete output from launching the code in Listing 3.22 is very lengthy, and you can see the full output by launching this code sample from the command line.

The next block shows you the shape of `mnist_x` and `mnist_y`, followed by a portion of the data (i.e., the pixel values) in the first image contained in the `MNIST` dataset.

```
mnist_x.shape: (60000, 28, 28)
mnist_y.shape: (60000,)

value: tf.Tensor(
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   3  18
18  18 126 136
  175  26 166 255 247 127   0   0   0   0]
 [  0   0   0   0   0   0   0   0  30  36  94 154 170 253
253 253 253 253
  225 172 253 242 195  64   0   0   0   0]
// output omitted for brevity
[  0   0   0   0  55 172 226 253 253 253 253 244 133  11
0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0 136 253 253 253 212 135 132  16   0   0
0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0
    0   0   0   0   0   0   0   0   0   0]], shape=(28,
28), dtype=uint8)
```

If you launch the code in Listing 3.22 from the command line, you will see the complete set of 784 (=28 x 28) pixel values.

WORKING WITH THE TFDS PACKAGE IN TF 2

The `tensorflow_datasets` package (`tfds`) contains utilities for loading predefined datasets. Keep in mind that these are datasets that contain data and are not to be confused with `tf.data.Dataset`. Listing 3.23 displays the contents of `tfds.py`, which illustrates how to display the list of available built-in datasets in TF 2 by means of the `tfds` package.

LISTING 3.23: tfds.py

```
import tensorflow as tf
import tensorflow_datasets as tfds

# See available datasets
print(tfds.list_builders())

# Construct a tf.data.Dataset
ds = tfds.load(name="mnist", split=tfds.Split.TRAIN)

# Build your input pipeline
ds = ds.shuffle(1024).batch(32).prefetch(tf.data.
experimental.AUTOTUNE)

for features in ds.take(1):
  image, label = features["image"], features["label"]
```

Listing 3.23 contains a `print()` statement that displays the complete list of built-in datasets in TF 2. The variable ds is initialized as the training-related data in the `MNIST` dataset. The next code snippet uses method chaining to invoke three operators: first the `shuffle()` operator (to shuffle the input data), then the `batch()` operator to specify 32 rows per batch, and then the `prefetch()` method to select the first batch of data. The final code block is a `for` loop that "takes" only the first row of data from `ds`. The output from launching the code in Listing 3.23 is here:

```
['bair_robot_pushing_small', 'cats_vs_dogs', 'celeb_a',
'celeb_a_hq', 'cifar10', 'cifar100', 'coco2014', 'diabetic_
retinopathy_detection', 'dummy_dataset_shared_generator',
'dummy_mnist', 'fashion_mnist', 'image_label_folder',
'imagenet2012', 'imdb_reviews', 'lm1b', 'lsun', 'mnist',
'moving_mnist', 'nsynth', 'omniglot', 'open_images_v4',
'quickdraw_bitmap', 'squad', 'starcraft_video', 'svhn_
cropped', 'tf_flowers', 'wmt_translate_ende', 'wmt_
translate_enfr']
```

As you can see, the previous output contains some well-known datasets, including CIFAR10, CIFAR100, MNIST, and FASHION_MNIST (among others).

The CIFAR10 Dataset and TFDS in TF 2

Listing 3.24 displays the contents of tfds-cifar10.py, which illustrates how to perform some processing on the CIFAR10 dataset and use lambda expressions and the map() operator to train the datasets.

LISTING 3.24: tfds-cifar10.py

```
import tensorflow as tf
import tensorflow_datasets as tfds

loader = tfds.load("cifar10", as_supervised=True)
train, test = loader["train"], loader["test"]

train = train.map(
  lambda image, label: (tf.image.convert_image_dtype(image,
tf.float32), label)
).cache().map(
  lambda image, label: (tf.image.random_flip_left_
right(image), label)
).map(
  lambda image, label: (tf.image.random_contrast(image,
lower=0.0, upper=1.0), label)
).shuffle(100).batch(64).repeat()
```

The code in this section is from the following stackoverflow post (which contains additional details):

https://stackoverflow.com/questions/55141076/how-to-apply-data-augmentation-in-tensorflow-2-0-after-tfds-load

WORKING WITH tf.estimator

The first subsection introduced as follows is useful if you have some experience with well-known machine learning algorithms using a Python library such as scikit-learn. You will see a list of the TF 2 classes that are similar to their Python-based counterparts in machine learning, with classes for regression tasks and classes for classification tasks.

The second subsection contains a list of TF 2 classes that are relevant for defining CNNs (Convolutional Neural Networks) in TF 2.

If you are new to machine learning then this section will have limited value to you right now, but you can still learn what's available in TF 2 (perhaps for future reference).

What Are TF 2 Estimators?

The `tf.estimator` namespace contains an assortment of classes that implement various algorithms that are available in machine learning, such as boosted trees, DNN classifiers, DNN regressors, linear classifiers, and linear regressors.

The estimator-related classes `DNNRegressor`, `LinearRegressor`, and `DNNLinearCombinedRegressor` are for regression tasks, whereas the classes `DNNClassifier`, `LinearClassifier`, and `DNNLinearCombinedClassifier` are for classification tasks. A more extensive list of estimator classes (with very brief descriptions) is listed as follows:

- BoostedTreesClassifier: A Classifier for TF 2 Boosted Trees models
- BoostedTreesRegressor: A Regressor for TF 2 Boosted Trees models
- CheckpointSaverHook: Saves checkpoints every N steps or seconds
- DNNClassifier: A classifier for TF 2 DNN models
- DNNEstimator: An estimator for TF 2 DNN models with user-specified head
- DNNLinearCombinedClassifier: An estimator for TF 2 Linear and DNN joined classification models
- DNNLinearCombinedRegressor: An estimator for TF 2 Linear and DNN joined models for regression
- DNNRegressor: A regressor for TF 2 DNN models
- Estimator: Estimator class to train and evaluate TF 2 models
- LinearClassifier: Linear classifier model
- LinearEstimator: An estimator for TF 2 linear models with user-specified head
- LinearRegressor: An estimator for TF 2 Linear regression problems

All estimator classes are in the `tf.estimator` namespace, and all estimator classes inherit from the `tf.estimator.Estimator` class. Read the online documentation for the details of the preceding classes as well as online tutorials for relevant code samples.

OTHER TF 2 NAMESPACES

In addition to the classes and namespaces that are mentioned in previous sections, TF 2 provides various other useful namespaces, including the following:

- tf.data (contains `tf.data.Dataset`)
- tf.keras (Keras-based functionality)
- tf.linalg (linear algebra)
- tf.lite (for mobile applications)
- tf.losses (cost functions)
- tf.math (mathematical functions)

- tf.nn (neural networks)
- tf.random (random values)
- tf.saved_model
- tf.test
- tf.train
- tf.version

The `tf.data` namespace contains the `tf.data.Dataset` namespace, which contains classes that are discussed in the first half of this chapter; the `tf.linalg` namespace contains an assortment of classes that perform operations in linear algebra; and the `tf.lite` namespace contains classes for mobile application development.

The `tf.keras` namespace contains functionality that is relevant to anyone who wants to work with Keras-based code. In particular, `tf.keras` contains many important namespaces, including the following:

- tf.keras.layers (Activation, Dense, Dropout, etc.)
- tf.keras.models (Sequential and Functional)
- tf.keras.optimizers (algorithms for cost functions)

The appendix contains code samples that involve classes from each of the three namespaces in the preceding list.

SUMMARY

This chapter introduced you to TF 2 `Datasets` that are well-suited for processing the contents of "normal" size datasets as well datasets that are too large to fit in memory. You saw how to define a lambda expression and use that expression in a TF 2 `Dataset`.

Next, you learned about various "lazy operators," including `batch()`, `filter()`, `flatmap()`, `map()`, `take()`, and `zip()`, and how to use them to define a subset of the data in a TF 2 `Dataset`. You also learned how to use TF 2 generators in order to iterate through the elements of a TF 2 `Dataset`.

Next, you learned how to create a TF 2 `Dataset` from a CSV file and then display its contents. Then you got a brief introduction to the `tf.estimator` namespace, which contains an assortment of classes that implement various algorithms, such as boosted trees, DNN classifiers, DNN regressors, linear classifiers, and linear regressors.

Finally, you learned about various other important aspects of TF 2, such as the `tf.keras.layers` namespace that contains an assortment of classes for DNNs (Dense Neural Networks) and CNNs (Convolutional Neural Networks).

CHAPTER 4

LINEAR REGRESSION

This chapter introduces linear regression, which is a well-known algorithm in machine learning. You'll learn some important aspects and assumptions regarding linear regression, and some statistical quantities for determining how well a model represents a dataset. You will see code examples that involve Python and `NumPy` code (often using the `NumPy linspace()` API), as well as code samples involving TF 2 code.

The first part of this chapter briefly discusses the basic concepts involved in linear regression. Although linear regression was developed more than 200 years ago, this technique is still one of the "core" techniques for solving (albeit simple) problems in statistics and machine learning. This section introduces "Mean Squared Error" (MSE) for finding a best-fitting line for data points in a 2D plane (or a hyperplane for higher dimensions).

The second section in this chapter contains very simple graphs of lines, scatterplots, and a quadratic plot in the plane (skip them if they are familiar). The third section discusses regularization, ML and feature scaling, and data normalization versus standardization.

The fourth section discusses various metrics for measuring models, such as R-Squared and its limitations, the confusion matrix, and accuracy versus precision versus recall. You will also learn about other useful statistical terms, such as RSS, TSS, F1 score, and p-value.

The fifth section shows you how to calculate the MSE value manually for a small dataset in the 2D plane. The sixth section discusses linear regression in conjunction with TF 2 `estimators` (in the `tf.estimator` namespace) that provides "canned" APIs for various algorithms. You will also see an example of solving linear regression using the TF 2 `LinearRegressor()` class that is also a TF 2 estimator.

The final section contains a Keras-based code sample to train a model in order to solve a task in linear regression. Although the Keras code is minimal (only a single layer), you do need to have some understanding of some Keras APIs, some of which are discussed briefly in this section.

As you will soon see, roughly two-thirds of this chapter discusses linear regression and topics pertaining to machine learning, and the TF 2 code samples are in the final third of this chapter. The amount of material that you read is obviously your choice, and although you can skip many of the theoretical concepts in this chapter, eventually you will need to learn them if you intend to deepen your knowledge of machine learning.

WHAT IS LINEAR REGRESSION?

In simplified terms, linear regression in the 2D plane attempts to determine the best-fitting line that "represents" a dataset. A best-fitting line minimizes the distance of that line from the points in the dataset. There is no correlation between best-fitting line and the number of points in the dataset that actually lie on the best-fitting line. In addition, linear regression differs from curve-fitting: the latter typically involves finding a polynomial that actually passes through points in a dataset.

In fact, 2D datasets consist of points that are often "scattered" in such a way that they cannot be points on a polynomial curve, regardless of how large a value that you choose for the degree of the polynomial. The reason is due to an important limiting factor: a polynomial can *never* intersect more than one point on any vertical line in the Euclidean plane. Since a dataset in the plane can contain *many* points that lie on the same vertical line, a different polynomial must be found to intersect each of the points on such a line.

Incidentally, the same property holds for continuous as well as noncontinuous functions in the plane: they can intersect *at most one* point of any vertical line in the plane. In fact, the definition of a function (which includes polynomials) states the following: if a function intersects two points `(x1,y1)` and `(x2,y2)` that have the same x value, then those two points must have the same y value. That is to say, if `x1=x2` then `y1=y2` must be true. Note that there is no such restriction for points that have the same y value because such points lie on a horizontal line (which is a function).

Now perform the following thought experiment: consider a scatter plot with many points in the plane that are sort of "clustered" in a tilted and elongated cloud-like shape. For such a dataset, a best-fitting line will probably intersect only a limited number of points. In fact, it's even possible that a best-fitting line doesn't intersect *any* of the points in the dataset.

One other scenario: suppose a dataset contains a set of 2D points that lie on the same line. For instance, let's suppose that the x values are in the set `{1,2,3,...,10}` and also that the y values are in the set `{2,4,6,...,20}`. Then the equation of the best-fitting line is `y=2*x+0`. In this scenario, all the points are *collinear*, which is to say that they lie on the same line.

Linear Regression versus Curve-Fitting

In some situations, it's possible to determine the maximum degree of a polynomial that fits a set of points in the 2D plane. For instance, suppose we have a set of `n` points of the form `(x,y)`. Let's also make the assumption that no pair of points have the same `x` value; this means that no two points lie on the same vertical line. In this situation, there is a polynomial of degree at most `n-1` that passes through those n points (if you are really interested, you can find a mathematical proof of this statement in online articles).

For example, a non-vertical line in the 2D plane is a polynomial of degree one, and it intersects any pair of points in the 2D plane (as long as the pair of points are not on a vertical line). For any triple of points that are not collinear in the plane (i.e., they do not all lie on the same line), there is a polynomial of degree two (also known as a quadratic polynomial) that passes through those three points. If you have 100 such points, then there is a polynomial of degree at most 99 that passes through all those points.

The good news is that sometimes a lower degree polynomial is available. For instance, consider the set of 100 points in which the `x` value equals the `y` value: in this case, the line `y = x` (which is a polynomial of degree one) passes through all 100 points.

Notice that the preceding paragraphs mentioned "a set of points" along with a set of assumptions. In general, a dataset of points in a 2D plane might not satisfy all those assumptions.

However, keep in mind that the extent to which a line "represents" a set of points in the plane depends on how closely those points can be approximated by a line, which is measured by the *variance* of the points (the variance is a statistical quantity). The more collinear the points, the smaller the variance; conversely, the more "spread out" the points are, the larger the variance.

What Is Multivariate Analysis?

Multivariate analysis generalizes the equation of a line in the Euclidean plane to higher dimensions, and it's called a *hyper plane* instead of a line. The generalized equation has the following form:

```
y = w1*x1 + w2*x2 + . . . + wn*xn + b
```

In the case of 2D linear regression, you only need to find the value of the slope (`m`) and the y-intercept (`b`), whereas in multivariate analysis you need to find the values for `w1, w2, . . ., wn`. Note that multivariate analysis is a term from statistics, and in machine learning it's often referred to as "generalized linear regression."

Keep in mind that most of the code samples in this book that pertain to linear regression involve 2D points in the Euclidean plane.

WHEN ARE SOLUTIONS EXACT IN MACHINE LEARNING?

Although statistics-based solutions provide closed-form solutions for linear regression, neural networks generally provide *approximate* solutions. This is due to the fact that machine learning algorithms involve a sequence of approximations that "converges" to optimal values, which means that machine learning algorithms produce estimates of the exact values. For example, the slope m and y-intercept b of a best-fitting line for a set of points in a 2D plane have a closed-form solution in statistics, but they can only be approximated via machine learning algorithms (exceptions do exist, but they are rare situations).

Keep in mind that even though a closed-form solution for "traditional" linear regression provides an exact value for both m and b, sometimes you can only use an approximation of the exact value. For instance, suppose that the slope m of a best-fitting line equals the square root of 3 and the y-intercept b is the square root of 2. If you plan to use these values in source code, you can only work with an approximation of these two numbers. In the same scenario, a neural network computes approximations for m and b, regardless of whether or not the exact values for m and b are irrational, fractional, or integer values. However, machine learning algorithms are better suited for complex, nonlinear, multidimensional datasets, which is beyond the capacity of linear regression.

As a simple example, suppose that the closed-form solution for a linear regression problem produces integer or rational values for both m and b. Specifically, let's suppose that a closed-form solution yields the values 2.0 and 1.0 for the slope and y-intercept, respectively, of a best-fitting line. The equation of the line looks like this:

```
y = 2.0 * x + 1.0
```

However, the corresponding solution from training a neural network might produce the values 2.0001 and 0.9997 for the slope m and the y-intercept b, respectively, as the values of m and b for a best-fitting line. Always keep this point in mind, especially when you are training a neural network.

CHALLENGES WITH LINEAR REGRESSION

Linear regression models are very powerful and simpler than their alternatives. However, issues can arise because of various factors, and an accurate analysis of these issues can be difficult. Here is a list of potential problems that can arise:

- Nonlinear data
- Nonconstant variance of error terms (heteroscedasticity)
- Correlation of error terms
- Collinearity
- Outliers

Nonlinear Data

Linear regression is based on the assumption that a line is an accurate model for the data. If you suspect that the data is not sufficiently linear, use residual plots to check for nonlinearity. Recall that the residual values are the differences between the y-coordinate of each point and the y-coordinate of its corresponding point on the estimated line: `ei = yi - y_i`.

Nonconstant Variance of Error Terms

Various terms in linear models, including standard errors and confidence intervals, rely on the assumption that error terms have constant variance. Sometimes a transformation of the dependent variable `Y` can reduce heteroscedasticity. Two examples of transformation functions are `log(Y)` or `sqrt(Y);` however, you can specify other functions as well (just make sure that they are monotonically nondecreasing functions).

Correlation of Error Terms

A correlation among the error terms results in estimated standard errors that tend to underestimate the true standard errors. In addition, confidence and prediction intervals are correspondingly narrower. Error correlations can occur in consecutive time periods for time series data.

Collinearity

Collinearity refers to variables that are very close to each other, and it can be difficult to distinguish the individual effects of collinear variables. Moreover, collinearity reduces the accuracy of the estimates of the regression coefficients. Inspect the correlation matrix of the predictors for relatively large elements, which indicates a pair of highly correlated variables, and hence collinearity in the data.

However, collinearity can exist among more than two variables (multicollinearity), and collinearity among those variables is not detectable in the correlation matrix. Collinearity can be problematic in regression models, especially when there is a high degree of correlation between two or more variables.

Fortunately, there are techniques for addressing collinearity, such as PCA (Principal Component Analysis), LDA (Linear Discriminant Analysis), SVD (Singular Value Decomposition), and various other techniques. However, these techniques are beyond the scope of this book (search online and you will find many articles).

Outliers and Anomalies

Outliers are “unusual” or unexpected data points, and while you might be tempted to ignore them, it’s not always possible to do so. For instance, a stock market crash is an outlier, and it most likely contains important information, so it’s advisable to retain such data.

Although an outlier might not significantly affect the MSE value, it can have a more significant effect on the RSE value. Residual plots can help you detect outliers in a dataset.

Anomalies are outliers that cannot be ignored, which is to say that anomalies are more serious than outliers. Thus, anomalies are outliers, but outliers are not necessarily anomalies. This means that anomalies form a subset of outliers. The earlier example of a stock market crash in a stock-related dataset is an example of an outlier that is also an anomaly.

By contrast, suppose that your credit card purchases are always within a fifty-mile radius, and suddenly a purchase in a different state or country appears on your credit card. That purchase is an outlier because it's much different from all your other purchases; however, it's not necessarily a fraudulent transaction (you might be on a business trip or on vacation), which means that this purchase is not automatically an anomaly.

OTHER TYPES OF REGRESSION

Linear regression finds the best-fitting line that "represents" a dataset, but what happens if a line in the plane is not a good fit for the dataset? This is a very important question when you work with datasets. Some alternatives to linear regression include quadratic equations, cubic equations, or higher-degree polynomials. However, these alternatives involve trade-offs, as we'll discuss later.

Another possibility is a sort of hybrid approach that involves a piecewise linear function that comprises a set of line segments. If contiguous line segments are connected, then it's a piecewise linear continuous function; otherwise it's a piecewise linear discontinuous function.

Thus, given a set of points in the plane, regression involves addressing the following questions:

1. What type of curve fits the data well? How do we know?
2. Does another type of curve fit the data better?
3. What does "best fit" mean?

One way to check if a line fits the data involves a visual check, but this approach does not work for data points that are higher than two dimensions. Moreover, this is a subjective decision, and some sample datasets are displayed later in this chapter. By visual inspection of a dataset, you might decide that a quadratic or cubic (or even higher degree) polynomial has the potential of being a better fit for the data. However, visual inspection is probably limited to points in a 2D plane or in three dimensions.

Let's defer the nonlinear scenario and let's make the assumption that a line would be a good fit for the data. There is a well-known technique for finding the "best-fitting" line for such a dataset called Mean Squared Error (MSE), and we'll discuss it later in this chapter.

The next section provides a quick review of linear equations in the plane, along with some images that illustrate examples of linear equations.

WORKING WITH LINES IN THE PLANE

This section starts with basic examples involving lines in a 2D plane. If you are comfortable with this topic, feel free to skip this section and proceed to the next section.

In case you don't remember, here is a general equation for a line in the Euclidean plane (except for vertical lines):

```
y = m*x + b
```

The value of `m` is the slope of the line and the value of `b` is the y-intercept (i.e., the place where the line intersects the y-axis).

If need be, you can use a more general equation that can also represent vertical lines, as shown here:

```
a*x + b*y + c = 0
```

However, we won't be working with vertical lines, so we'll stick with the first formula.

Figure 4.1 displays three horizontal lines whose equations (from top to bottom) are `y = 3`, `y = 0`, and `y = -3`, respectively.

Figure 4.2 displays two slanted lines whose equations are `y = x` and `y = -x`.

Figure 4.3 displays two slanted parallel lines whose equations are `y = 2*x` and `y = 2*x + 3`.

Figure 4.4 displays a piecewise linear graph consisting of connected line segments.

Now let's turn our attention to generating quasi-random data using a `NumPy` API, and then we'll plot the data using Matplotlib.

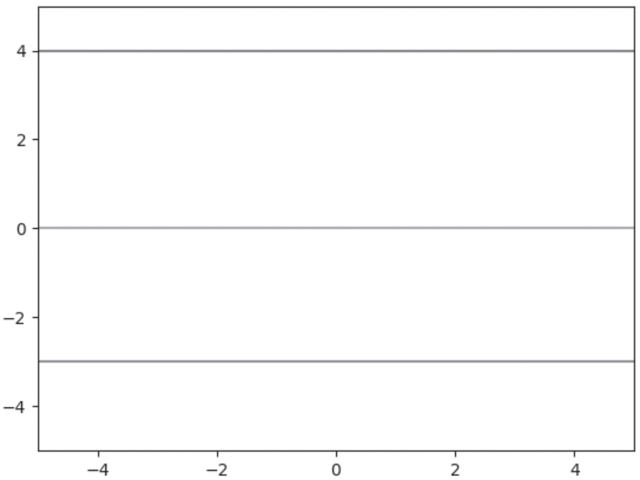

FIGURE 4.1. A graph of three horizontal line segments.

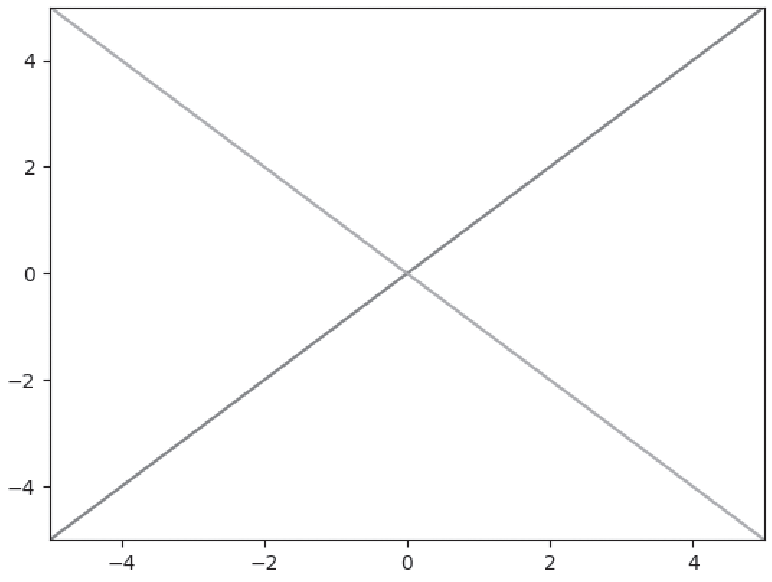

FIGURE 4.2. A graph of two diagonal line segments.

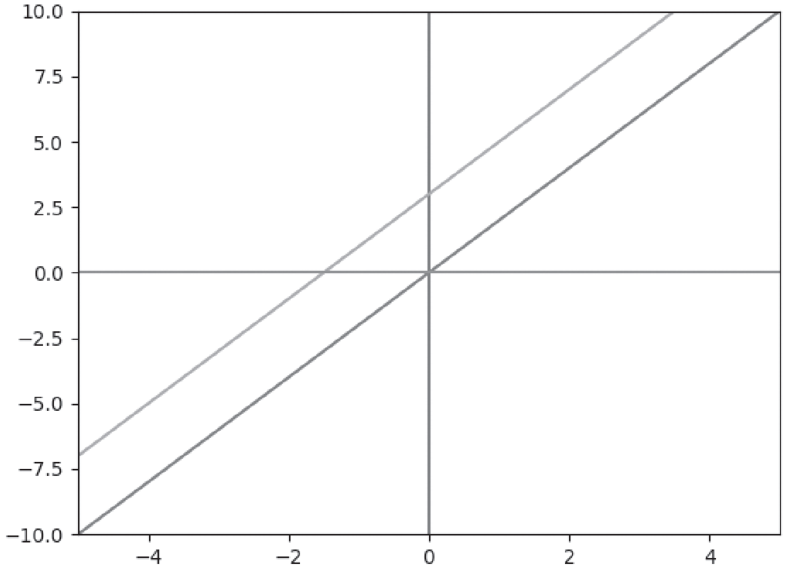

FIGURE 4.3. A graph of two slanted parallel line segments.

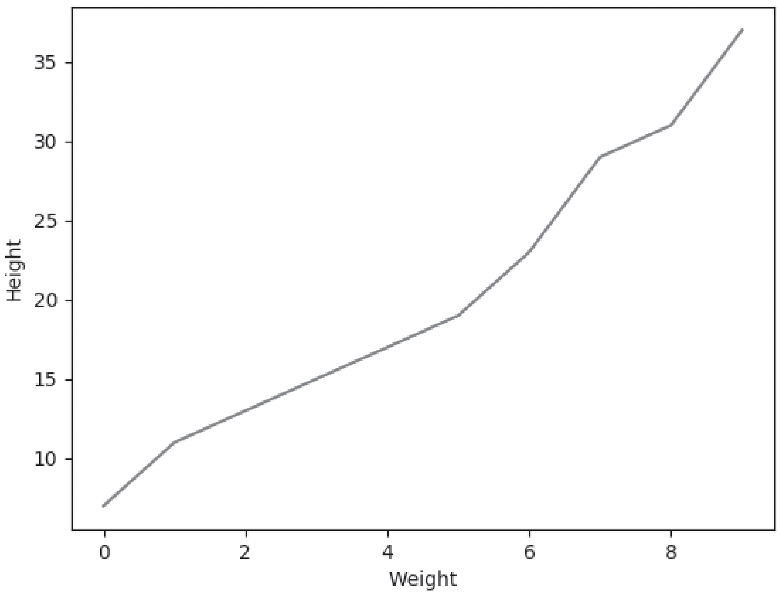

FIGURE 4.4. A piecewise linear graph of line segments.

SCATTER PLOTS WITH `NumPy` AND `Matplotlib` (1)

Listing 4.1 displays the contents of `np_plot.py`, which illustrates how to use the `NumPy` `randn()` API to generate a dataset and then the `scatter()` API in `Matplotlib` to plot the points in the dataset.

One detail to note is that all the adjacent horizontal values are equally spaced, whereas the vertical values are based on a linear equation plus a "perturbation" value. This "perturbation technique" (which is not a standard term) is used in other code samples in this chapter in order to add a slightly randomized effect when the points are plotted. The advantage of this technique is that the best-fitting values for `m` and `b` are known in advance, and therefore we do not need to guess their values.

LISTING 4.1: np_plot.py

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

x = np.random.randn(15,1)
y = 2.5*x + 5 + 0.2*np.random.randn(15,1)

print("x:",x)
print("y:",y)
```

```
plt.scatter(x,y)
plt.show()
```

Listing 4.1 contains two `import` statements and then initializes the `NumPy` array `x` with fifteen random numbers between 0 and 1. Next, the `NumPy` array `y` is defined in two parts: the first part is a linear equation `2.5*x + 5` and the second part is a "perturbation" value that is based on a random number. Thus, the array variable `y` simulates a set of values that closely approximates a line segment.

This technique is used in code samples that simulate a line segment, and then the training portion approximates the values of `m` and `b` for the best-fitting line. Obviously, we already *know* the equation of the best-fitting line: the purpose of this technique is to compare the trained values for the slope `m` and y-intercept `b` with the known values (which in this case are 2.5 and 5). A partial output from Listing 4.1 is here:

```
x: [[-1.42736308]
 [ 0.09482338]
 [-0.45071331]
 [ 0.19536304]
 [-0.22295205]
 // values omitted for brevity
y: [[1.12530514]
 [5.05168677]
 [3.93320782]
 [5.49760999]
 [4.46994978]
 // values omitted for brevity
```

Figure 4.5 displays a scatter plot of points based on the values of `x` and `y`.

Why the "Perturbation Technique" Is Useful

The code sample in this section initializes a dataset with points that are defined in the Python array variables `X` and `Y`:

```
X = [0,0.12,0.25,0.27,0.38,0.42,0.44,0.55,0.92,1.0]
Y = [0,0.15,0.54,0.51, 0.34,0.1,0.19,0.53,1.0,0.58]
```

If you need to find the best-fitting line for the preceding dataset, how would you estimate the values for the slope `m` and the y-intercept `b`? In most cases, you probably cannot guess their values. On the other hand, the "perturbation technique" enables you to "jiggle" the points on a line whose value for the slope `m` (and optionally the value for the y-intercept `b`) is specified in advance.

Keep in mind that the "perturbation technique" only works when you introduce small random values that do not result in different values for `m` and `b`.

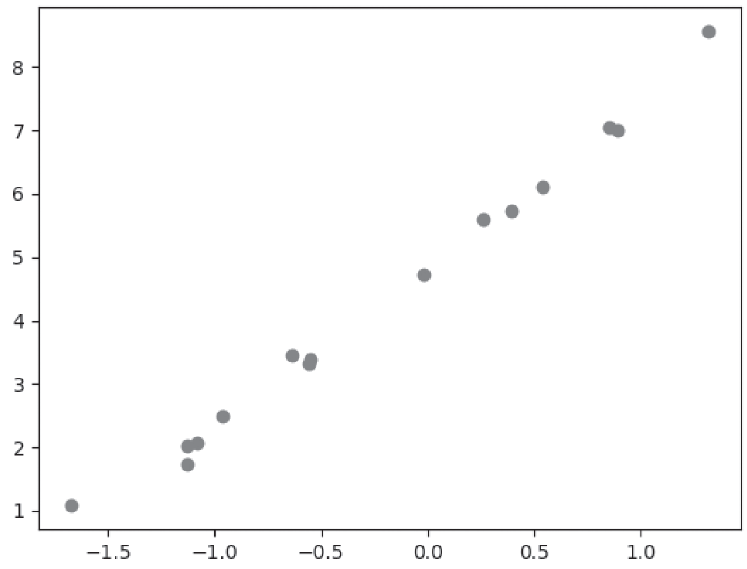

FIGURE 4.5. A scatter plot of points for a line segment.

SCATTER PLOTS WITH `NumPy` AND `Matplotlib` (2)

The code in Listing 4.1 in the previous section assigned random values to the variable `x`, whereas a hard-coded value is assigned to the slope `m`. The `y` values are a hard-coded multiple of the `x` values, plus a random value that is calculated via the "perturbation technique." Hence, we do not know the value of the y-intercept `b`.

In this section the values for `trainX` are based on the `np.linspace()` API, and the values for `trainY` involve the "perturbation technique" that is described in the previous section.

The code in this example simply prints the values for `trainX` and `trainY`, which correspond to data points in the Euclidean plane. Listing 4.2 displays the contents of `np_plot2.py`, which illustrates how to simulate a linear dataset in `NumPy`.

LISTING 4.2: np_plot2.py

```
import numpy as np

x_data = np.linspace(-1, 1, 11)
y_data = 4*x_data + np.random.randn(*x_data.shape)*0.5

print("x_data: ",x_data)
print("y_data: ",y_data)
```

Listing 4.2 initializes the `NumPy` variable `x_data` via the `NumPy` `linspace()` API, followed by the `NumPy` variable `y_data` that is defined in two parts. The first part is the linear term `4*x_data` and the second part involves the "perturbation technique" that is a randomly generated number. The output from Listing 4.2 is here:

```
x_data:  [-1.  -0.8 -0.6 -0.4 -0.2  0.   0.2  0.4  0.6  0.8
1. ]
y_data:  [-3.60147459 -2.66593108 -2.26491189 -1.65121314
-0.56454605  0.22746004 0.86830728  1.60673482  2.51151543
3.59573877  3.05506056]
```

The purpose of this code sample is merely to generate and display a set of randomly generated numbers. Later in this chapter we will use this code as a starting point for an actual linear regression task.

The next section contains an example that is similar to Listing 4.2, using the same "perturbation technique" to generate a set of points that approximates a quadratic equation instead of a line segment.

A QUADRATIC SCATTER PLOT WITH `NumPy` AND `Matplotlib`

Listing 4.3 displays the contents of `np_plot_quadratic.py`, which illustrates how to plot a quadratic function in the plane.

LISTING 4.3: np_plot_quadratic.py

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

#see what happens with this set of values:
#x = np.linspace(-5,5,num=100)

x = np.linspace(-5,5,num=100)[:,None]
y = -0.5 + 2.2*x +0.3*x**2 + 2*np.random.randn(100,1)
print("x:",x)

plt.plot(x,y)
plt.show()
```

Listing 4.3 initializes the `NumPy` variable `x` with the values that are generated via the `np.linspace()` API, which in this case is a set of 100 equally spaced decimal numbers between -5 and 5. Notice the snippet `[:,None]` in the initialization of `x`, which results in an array of elements, each of which is an array consisting of a single number.

The array variable `y` is defined in two parts: the first part is a quadratic equation `-0.5 + 2.2*x +0.3*x**2` and the second part is a "perturbation" value that is based on a random number (similar to the code in Listing 4.1). Thus, the array variable y simulates a set of values that approximates a quadratic equation. The output from Listing 4.3 is here:

```
x:
[[-5.        ]
 [-4.8989899 ]
 [-4.7979798 ]
 [-4.6969697 ]
 [-4.5959596 ]
 [-4.49494949]
 // values omitted for brevity
 [ 4.8989899 ]
 [ 5.        ]]
```

Figure 4.6 displays a scatter plot of points based on the values of x and y, which have an approximate shape of a quadratic equation.

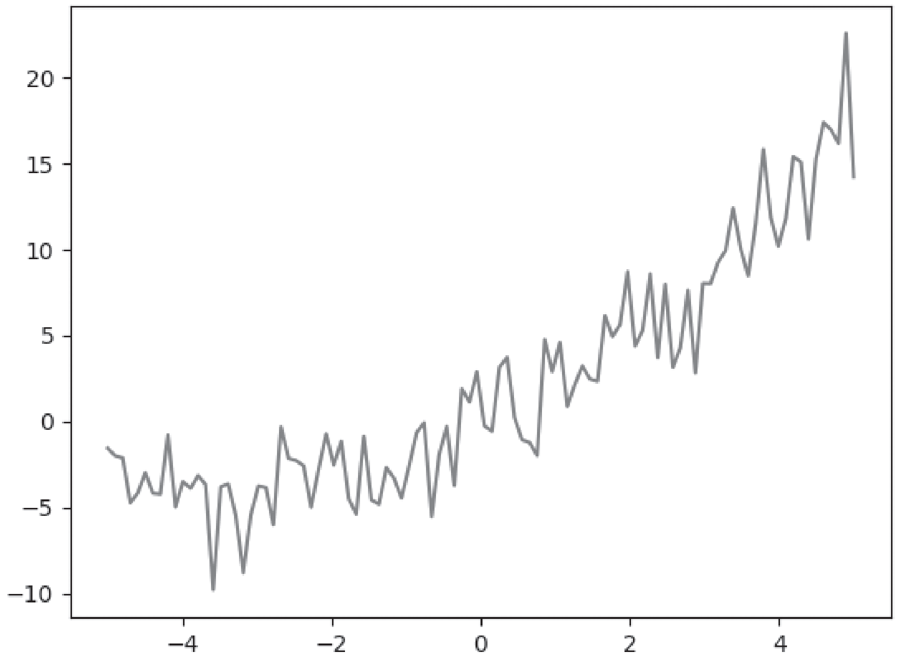

FIGURE 4.6. A scatter plot of points for a quadratic equation.

THE MEAN SQUARED ERROR (MSE) FORMULA

Figure 4.8 displays the formula for the MSE. In plain English, the MSE is the sum of the squares of the difference between an actual y value and the predicted y value (this difference is called a "residual value"), divided by the number of points. Notice that the predicted y value is the y value that each point would have if that point were actually on the best-fitting line.

Although the MSE is popular for linear regression, there are other error types available, some of which are discussed briefly in the next section.

A List of Error Types

Although we will only discuss MSE for linear regression in this book, there are other types of formulas that you can use for linear regression, some of which are listed here:

- MSE
- RMSE
- RMSPROP
- MAE

The MSE is the basis for the preceding error types. For example, RMSE is "Root Mean Squared Error," which is the square root of the MSE.

On the other hand, MAE is "Mean Absolute Error," which is the sum of *the absolute value of the differences of the y terms* (*not* the square of the differences of the y terms), which is then divided by the number of terms.

The RMSProp optimizer utilizes the magnitude of recent gradients to normalize the gradients. Maintain a moving average over the RMS (root mean squared) gradients, and then divide that term by the current gradient.

Although it's easier to compute the derivative of MSE, it's also true that MSE is more susceptible to outliers, whereas MAE is less susceptible to outliers. The reason is simple: a squared term can be significantly larger than the absolute value of a term. For example, if a difference term is 10, then a squared term of 100 is added to the MSE, whereas only 10 is added to the MAE. Similarly, if a difference term is -20, then a squared term 400 is added to the MSE, whereas only 20 (which is the absolute value of -20) is added to the MAE.

Nonlinear Least Squares

When predicting housing prices, where the dataset contains a wide range of values, techniques such as linear regression or random forests can cause the model to overfit the samples with the highest values in order to reduce quantities such as mean absolute error.

In this scenario you probably want an error metric, such as relative error, that reduces the importance of fitting the samples with the largest values. This technique is called *nonlinear least squares*, which may use a log-based transformation of labels and predicted values.

The next section discusses regularization, which is an important yet optional topic if you are primarily interested in TF 2 code. If you plan to become proficient in machine learning, you will need to learn about regularization.

WHAT IS REGULARIZATION?

Regularization helps to solve overfitting problems, which occur when a model performs well on training data but poorly on validation or test data. Regularization solves this problem by adding a penalty term to the cost function, thereby controlling the model complexity with this penalty term. Regularization is generally useful for:

1. large number of variables
2. low ratio of (# observations)/(# of variables)
3. high multi-collinearity

There are two main types of regularization: L1 Regularization (which is related to MAE, or the absolute value of differences) and L2 Regularization (which is related to MSE, or the square of differences). In general, L2 performs better than L1 and L2 is efficient in terms of computation.

Machine Learning and Feature Scaling

Feature scaling standardizes the range of features of data. This step is performed during the data preprocessing step, in part because gradient descent benefits from feature scaling.

The assumption is that the data conforms to a standard normal distribution, and standardization involves subtracting the mean and dividing by the standard deviation for every data point, which results in a N(0,1) normal distribution.

Data Normalization vs. Standardization

Data normalization is a linear scaling technique. Let's assume that a dataset has the values `{X1, X2, . . . , Xn}` along with the following terms:

```
Minx = minimum of Xi values
Maxx = maximum of Xi values
```

Now calculate new `Xi` values as follows:

```
Xi = (Xi – Minx)/[Maxx – Minx]
```

The new `Xi` values are now scaled so that they are between 0 and 1.

THE BIAS-VARIANCE TRADE-OFF

Bias in machine learning can be due to an error from wrong assumptions in a learning algorithm. High bias might cause an algorithm to miss relevant relations between features and target outputs (underfitting).

Prediction bias can occur because of "noisy" data, an incomplete feature set, or a biased training sample.

Error due to bias is the difference between the expected (or average) prediction of your model and the correct value that you want to predict. Repeat the model-building process multiple times, gather new data each time, and also run an analysis to produce a new model. The resulting models have a range of predictions, because the underlying data sets have a degree of randomness. Bias measures the extent to which the predictions for these models deviate from the correct value.

Variance in machine learning is the expected value of the squared deviation from the mean. High variance can/might cause an algorithm to model the random noise in the training data, rather than the intended outputs (aka overfitting).

Adding parameters to a model increases its complexity, increases the variance, and decreases the bias. Dealing with bias and variance is dealing with underfitting and overfitting.

Error due to variance is the variability of a model prediction for a given data point. As before, repeat the entire model-building process, and the variance is the extent to which predictions for a given point vary among different "instances" of the model.

METRICS FOR MEASURING MODELS

One of the most frequently used metrics is R-squared, in which R-squared measures how close the data is to the fitted regression line (regression coefficient). R-squared is always between 0% and 100%. The value 0% indicates that the model explains none of the variability of the response data around its mean. The value 100% indicates that the model explains all the variability of the response data around its mean. In general, a higher R-squared indicates a better model.

Limitations of R-Squared

Although high R-squared values are preferred, they are not necessarily always good values. Similarly, low R-squared values are not always bad. An R-squared value for predicting human behavior is often less than 50%. Moreover, R-squared cannot determine whether the coefficient estimates and predictions are biased. In addition, R-squared does not indicate whether a regression model is adequate. Thus, it's possible to have a low R-squared value for a good model, or a high R-squared value for a poorly fitting model. Evaluate R-squared values in conjunction with residual plots, other model statistics, and subject area knowledge.

Confusion Matrix

In its simplest form, a confusion matrix (also called an error matrix) is a type of contingency table with two rows and two columns that contains the number of false positives, false negatives, true positives, and true negatives. The four entries in a 2x2 confusion matrix can be labeled as follows:

```
TP: True Positive
FP: False Positive
TN: True Negative
FN: False Negative
```

The diagonal values of the confusion matrix are correct, whereas the off-diagonal values are incorrect predictions. In general a lower FP value is better than an FN value. For example, an FP indicates that a healthy person was incorrectly diagnosed with a disease, whereas an FN indicates that an unhealthy person was incorrectly diagnosed as healthy.

Accuracy vs. Precision vs. Recall

A 2x2 confusion matrix has four entries that represent the various combinations of correct and incorrect classifications. Given the definitions in the preceding section, the definitions of precision, accuracy, and recall are given by the following formulas:

```
precision = TP/(TN + FP)
accuracy  = (TP + TN)/[P + N]
recall    = TP/[TP + FN]
```

Accuracy is an unreliable metric because it yields misleading results in unbalanced data sets. When the numbers of observations in different classes are significantly different, it gives equal importance to both false positive and false negative classifications. For example, declaring cancer as benign is worse than incorrectly informing patients that they are suffering from cancer. Unfortunately, accuracy won't differentiate between these two cases.

OTHER USEFUL STATISTICAL TERMS

Machine learning relies on a number of statistical quantities in order to assess the validity of a model, some of which are listed here:

- `RSS`
- `TSS`
- `R^2`
- `F1 score`
- `p-value`

The definitions of `RSS`, `TSS`, and `R^2` are shown as follows, where `y^` is the y-coordinate of a point on a best-fitting line and `y_` is the mean of the y-values of the points in the dataset:

```
RSS = sum of squares of residuals (y - y^)**2
TSS = total sum of squares         (y - y_)**2
R^2 = 1 - RSS/TSS
```

What Is an F1 Score?

The F1 score is a measure of the accuracy of a test, and it's defined as the harmonic mean of precision and recall. Here are the relevant formulas, where `p` is the precision and `r` is the recall:

```
p = (# of correct positive results)/(# of all positive
results)
r = (# of correct positive results)/(# of all relevant
samples)

F1-score  = 1/[((1/r) + (1/p))/2]
          = 2*[p*r]/[p+r]
```

The best value of an F1 score is 1 and the worst value is 0. Keep in mind that an F1 score tends to be used for categorical classification problems, whereas the R^2 value is typically used for regression tasks (such as linear regression).

What Is a p-value?

The p-value is used to reject the null hypothesis if the p-value is small enough (< 0.005), which indicates a higher significance. Recall that the null hypothesis states that there is no correlation between a dependent variable (such as y) and an independent variable (such as x). The threshold value for p is typically 1% or 5%.

There is no straightforward formula for calculating p-values, which are values that are always between 0 and 1. In fact, p-values are statistical quantities to evaluate the null hypothesis, and they are calculated using p-value tables or spreadsheet/statistical software.

WORKING WITH DATASETS

There are several aspects of working with datasets that contain data (i.e., not the `tf.data.Dataset` class in Chapter 3), such as selecting training data versus test data, and also performing cross-validation on data. More details are provided in the subsequent sections.

Training Data Versus Test Data

A training set is a subset of a dataset that is used to train a model, whereas a test set is a subset to test the trained model. Ensure the following for your test sets:

1. the set is large enough to yield statistically meaningful results
2. it's representative of the data set as a whole
3. never train on test data
4. never test on training data

What Is Cross-Validation?

The purpose of cross-validation is to test a model with non-overlapping test sets, which is performed in the following manner:

Step 1) split the data into k subsets of equal size
Step 2) select one subset for testing and the others for training
Step 3) repeat step 2) for the other k-1 subsets

This process is called `k-fold cross-validation`, and the overall error estimate is the average of the error estimates.

A standard method for evaluation involves ten-fold cross-validation. Extensive experiments have shown that ten subsets is the best choice to obtain an accurate estimate. In fact, you can repeat ten-fold cross-validation ten times and compute the average of the results, which helps to reduce the variance.

The next section contains several code samples, the first of which involves calculating the MSE manually, followed by an example that uses `NumPy` formulas to perform the calculations. Finally, we'll look at a TF 2 example for calculating the MSE.

CALCULATING THE MSE MANUALLY

This section contains two line graphs, both of which contain a line that approximates a set of points in a scatter plot.

Figure 4.7 displays a line segment that approximates a scatter plot of points (some of which intersect the line segment).

Figure 4.8 displays a set of points and a line that is a potential candidate for best-fitting line for the data. The MSE for the line in Figure 4.7 is computed as follows:

```
MSE = (-2)*(-2) + 2*2 = 8
```

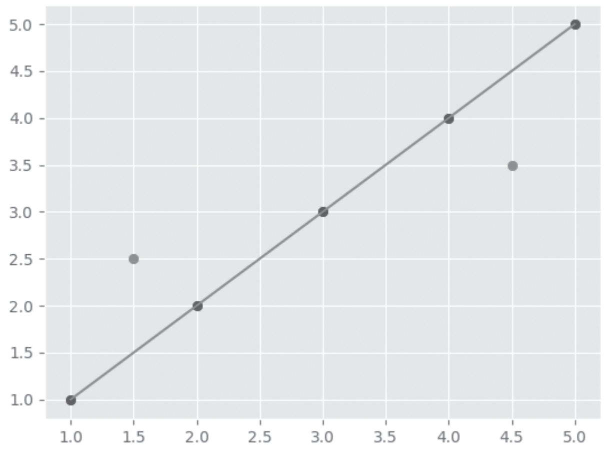

FIGURE 4.7. A line graph that approximates points of a scatter plot.

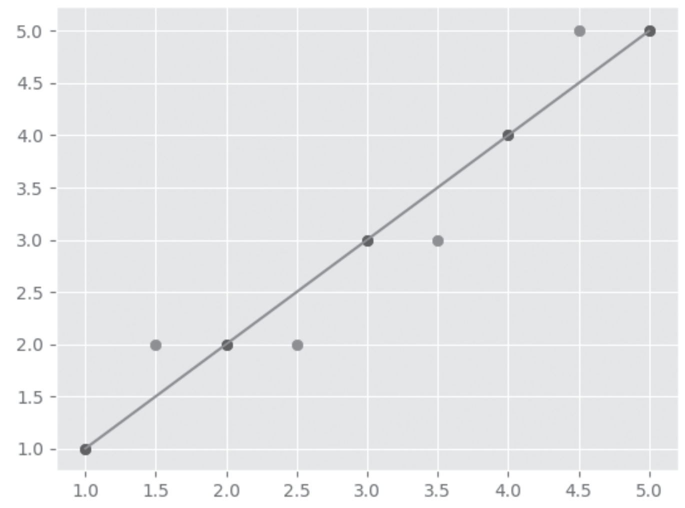

FIGURE 4.8. A line graph that approximates points of a scatter plot.

Now look at Figure 4.10, which also displays a set of points and a line that is a potential candidate for best-fitting line for the data.

The MSE for the line in Figure 4.8 is computed as follows:

```
MSE = 1*1 + (-1)*(-1) + (-1)*(-1) + 1*1 = 4
```

Thus, the line in Figure 4.8 has a smaller MSE than the line in Figure 4.7, which might have surprised you (or did you guess correctly?).

In these two figures we calculated the MSE easily and quickly, but in general it's significantly more difficult. For instance, if we plot ten points in the Euclidean plane that do not closely fit a line, with individual terms that involve non-integer values, we would probably need a calculator.

A better solution involves `NumPy` functions, such as the `np.linspace()` API, as discussed in the next section.

SIMPLE 2D DATA POINTS IN TF 2

Listing 4.4 displays the contents of `basic_linear1.py`, which calculates the y-coordinates of 2D points based on simulated data for the x-coordinates (but linear regression is not performed in this code sample).

LISTING 4.4: basic_linear1.py

```
import tensorflow as tf

W = tf.Variable([.5], dtype=tf.float32)
b = tf.Variable([-1], dtype=tf.float32)
x = tf.Variable([0],  dtype=tf.float32)
```

```
@tf.function
def compute_values(x):
  lm = W*x + b
  return lm

for x in range(4):
  val = compute_values(x+1)
  print("val:", val)
```

Listing 4.4 contains the variables `W`, `b`, and `x`, which are combined in the decorated Python function `compute_values()` to calculate a linear combination of values. Finally, a loop with a `print()` statement generates the output, as shown here:

```
val: tf.Tensor([-0.5],shape=(1,), dtype=float32)
val: tf.Tensor([0.],  shape=(1,), dtype=float32)
val: tf.Tensor([0.5], shape=(1,), dtype=float32)
val: tf.Tensor([1.],  shape=(1,), dtype=float32)
```

The preceding four tensor values are computed by invoking the decorated Python function `compute_values()` with the `x` values 1, 2, 3, and 4. Now that we know how to generate `(x,y)` values for a linear equation, let's learn how to calculate the MSE, which is discussed in the next section.

TF2, `tf.GradientTape()`, AND LINEAR REGRESSION

The code sample in this section shows you how to perform linear regression with `tf.GradientTape()`, which supersedes the TF 1.x code style that involves `tf.Session()`.

Listing 4.5 displays the contents of `tf2_linreg_tape.py`, which illustrates how to use `tf.GradientTape()` in order to train the values for the slope `m` and intercept `b` of a best-fitting line in the Euclidean plane.

LISTING 4.5: tf2_linreg_tape.py

```
import tensorflow as tf

step    = 20
rows    = 100
slope   = 0.4
bias    = 1.5

x_train = tf.random.uniform(shape=(rows,))
perturb = tf.random.normal(shape=(len(x_train),),
stddev=0.01)
y_train = slope * x_train + bias + perturb

# initial values for slope 'm' and bias 'b'
m = tf.Variable(0.)
b = tf.Variable(0.)
```

```
# predict the y value based on a value for x
def predict_y_value(x):
  y = m * x + b
  return y

# loss = RSS = residual sum of squares
#      = sum of squares of difference
#        between predicted and true values
def squared_error(y_pred, y_true):
  return tf.reduce_mean(tf.square(y_pred - y_true))

loss = squared_error(predict_y_value(x_train), y_train)
print("Initial loss:", loss.numpy())

######################################
# backward error propagation requires:
# a loss function (squared_error)
# an optimizer    (tape.gradient)
# a value for the learning rate
#####################################

learning_rate = 0.05
steps = 200

for i in range(steps):
  with tf.GradientTape() as tape:
    predictions = predict_y_value(x_train)
    loss = squared_error(predictions, y_train)

  gradients = tape.gradient(loss, [m, b])

  m.assign_sub(gradients[0] * learning_rate)
  b.assign_sub(gradients[1] * learning_rate)

  if(i % step) == 0:
    print("Step %d, Loss %f" % (i, loss.numpy()))

# display trained values for slope m and bias b
print ("m: %f, b: %f" % (m.numpy(), b.numpy()))
```

Listing 4.5 starts with the initialization of four variables, followed by a code block that initializes the variables x_train and x_train, along with the "perturbation" technique that you have seen in previous code samples. The next two lines of code initialize the two trainable variables m (the slope) and b (the bias) with the value 0. When the code finishes execution, you will see the calculated value for m and b, both of which are close to the values of slope and bias that are initialized in the beginning of this code sample.

The next portion of Listing 4.5 is the Python function predict_y_value() that calculates (and returns) the value of y based on the value of x. This function is invoked when we calculate the value of loss later in the code.

Next, the Python function squared_error() is the loss function, which is essentially the MSE (Mean Squared Error) that is discussed earlier in this chapter. This function takes the predicted y values and the initial y values as input in order to compute the current MSE value.

Next, the value of loss is determined by first invoking the Python function predict_y_value, and then passing the result of that invocation to the Python function squared_error.

Now that the initializations and Python functions are defined, let's look at the training loop in the next portion of Listing 4.5. This loop calculates the predictions variable and the loss variable, as shown here:

```
with tf.GradientTape() as tape:
  predictions = predict_y_value(x_train)
  loss = squared_error(predictions, y_train)

gradients = tape.gradient(loss, [m, b])
```

Notice how the tape variable (an instance of tf.GradientTape()) calculates the new gradient values based on the current values of loss, m, and b. The gradients variable contains the partial derivative of the loss function loss with respect to the variables m and b, which are the only two variables that we need to update for linear regression in the Euclidean plan.

Specifically, gradients[0] is the partial derivative of the loss variable with respect to m, and gradients[1] is the partial derivative of the loss variable with respect to b. In the case of a linear regression task in n dimensions, the gradients variable is a 1xn vector whose elements are partial derivatives for each of those n dimensions.

The next section of Listing 4.5 uses the values in the gradients array to update the values of m and b via a very simple calculation, as shown here:

```
m.assign_sub(gradients[0] * learning_rate)
b.assign_sub(gradients[1] * learning_rate)
```

The preceding code snippet calculates the new value of b by subtracting the quantity **gradients**[0]*learning_rate from the current value of m, and then updates b by subtracting the quantity **gradients**[1]*learning_rate from the current value of b.

The final portion of Listing 4.5 periodically displays the values of the loss variable, and the last code snippet displays the trained values for m and b. Launch the code in Listing 4.5 and you will see the following output:

```
Initial loss: 2.9317048
Step 0, Loss 2.931705
Step 20, Loss 0.018575
Step 40, Loss 0.005255
Step 60, Loss 0.004075
Step 80, Loss 0.003194
Step 100, Loss 0.002508
Step 120, Loss 0.001974
Step 140, Loss 0.001558
Step 160, Loss 0.001234
Step 180, Loss 0.000982
m: 0.498547, b: 1.447338
```

As you can see from the preceding output, the trained values for `m` and `b` are 0.498547 and 1.447338, respectively, which are reasonably close to the initial values of 0.4 and 1.5 for the `slope` and `bias`, respectively.

If you increase the value of steps from 200 to 500, the trained values for `m` and `b` are 0.410014 and 1.494226, respectively, which are considerably closer to the initial values of 0.4 and 1.5 for the `slope` and `bias`, respectively.

WORKING WITH KERAS

If you are already comfortable with Keras, you can skim this section to learn about the new namespaces and what they contain, and then proceed to the next section that contains details for creating a Keras-based model.

If you are new to Keras, you might be wondering why this section is included in this chapter. First, Keras is well-integrated into TF 2, and it's in the `tf.keras` namespace. Second, Keras is well-suited for defining models to solve a myriad of tasks, such as linear regression and logistic regression, as well as deep learning tasks involving CNNs, RNNs, and LSTMs that are discussed in the appendix.

The next several subsections contain lists of bullet items for various Keras-related namespaces, and they will be very familiar if you have worked with TF 1.x. If you are new to TF 2, you'll see examples of some of Keras-related classes in subsequent code samples.

Working with Keras Namespaces in TF 2

TF 2 provides the `tf.keras` namespace, which in turn contains the following namespaces:

- `tf.keras.layers`
- `tf.keras.models`
- `tf.keras.optimizers`
- `tf.keras.utils`
- `tf.keras.regularizers`

The preceding namespaces contain various layers in Keras models, different types of Keras models, optimizers (Adam et al.), utility classes, and regularizers (such as L1 and L2), respectively.

Currently there are three ways to create Keras-based models:

- The Sequential API
- The Functional API
- The Model API

The Keras-based code samples in this book use primarily the Sequential API (it's the most intuitive and straightforward). The `Sequential` API enables you to specify a list of layers, most of which are defined in the `tf.keras.layers` namespace (discussed later).

The Keras-based models that use the functional API involve specifying layers that are passed as function-like elements in a "pipeline-like" fashion. Although the functional API provides some additional flexibility, you will probably use the `Sequential` API to define Keras-based models if you are a TF 2 beginner.

The model-based API provides the greatest flexibility, and it involves defining a Python class that encapsulates the semantics of your Keras model. This class is a subclass of the `tf.keras.model.Model` class, and you must implement the two methods `__init__` and `call` in order to define a Keras model in this subclass.

Perform an online search for more details regarding the Functional API and the Model API.

Working with the `tf.keras.layers` Namespace

The most common (and also the simplest) Keras-based model is the `Sequential()` class that is in the `tf.keras.models` namespace. This model is comprised of various layers that belong to the `tf.keras.layers` namespace, as shown here:

- `tf.keras.layers.Conv2D()`
- `tf.keras.layers.MaxPooling2D()`
- `tf.keras.layers.Flatten()`
- `tf.keras.layers.Dense()`
- `tf.keras.layers.Dropout()`
- `tf.keras.layers.BatchNormalization()`
- `tf.keras.layers.embedding()`
- `tf.keras.layers.RNN()`
- `tf.keras.layers.LSTM()`
- `tf.keras.layers.Bidirectional (ex: BERT)`

The `Conv2D()` and `MaxPooling2D()` classes are used in Keras-based models for CNNs, which are discussed in the appendix. Generally speaking, the next six classes in the preceding list can appear in models for CNNs as well as models for machine learning. The `RNN()` class is for simple RNNS and the LSTM class is for LSTM-based models. The `Bidirectional()` class is a bidirectional LSTM that you will often see in models for solving NLP (Natural Language Processing) tasks. Two very important NLP frameworks that use bidirectional architectures were released as open source (on GitHub) in 2018: ELMo from Facebook and BERT from Google.

Working with the tf.keras.activations Namespace

Machine learning and deep learning models require activation functions. For Keras-based models, the activation functions are in the `tf.keras.activations` namespace, some of which are listed here:

- `tf.keras.activations.relu`
- `tf.keras.activations.selu`
- `tf.keras.activations.linear`
- `tf.keras.activations.elu`
- `tf.keras.activations.sigmoid`
- `tf.keras.activations.softmax`
- `tf.keras.activations.softplus`
- `tf.keras.activations.tanh`
- `Others` …

The `ReLU/SELU/ELU` activation functions are closely related, and they often appear in ANNs (Artificial Neural Networks) and CNNs. Before the `relu()` function became popular, the `sigmoid()` and `tanh()` functions were used in ANNs and CNNs. However, they are still important and they are used in various gates in GRUs and LSTMs. The `softmax()` function is typically used in the pair of layers consisting of the rightmost hidden layer and the output layer.

Working with the `tf.keras.datasets` Namespace

For your convenience, TF 2 provides a set of built-in datasets in the `tf.keras.datasets` namespace, some of which are listed here:

- `tf.keras.datasets.boston_housing`
- `tf.keras.datasets.cifar10`
- `tf.keras.datasets.cifar100`
- `tf.keras.datasets.fashion_mnist`
- `tf.keras.datasets.imdb`
- `tf.keras.datasets.mnist`
- `tf.keras.datasets.reuters`

The preceding datasets are popular for training models with small datasets. The `mnist` dataset and `fashion_mnist` dataset are both popular when training CNNs, whereas the `boston_housing` dataset is popular for linear regression. The `Titanic` dataset is also popular for linear regression, but it's not currently supported as a default dataset in the `tf.keras.datasets` namespace.

Working with the `tf.keras.experimental` Namespace

The `contrib` namespace in TF 1.x has been deprecated in TF 2, and its "successor" is the `tf.keras.experimental` namespace, which contains the following classes (among others):

- `tf.keras.experimental.CosineDecay`
- `tf.keras.experimental.CosineDecayRestarts`
- `tf.keras.experimental.LinearCosineDecay`
- `tf.keras.experimental.NoisyLinearCosineDecay`
- `tf.keras.experimental.PeepholeLSTMCell`

If you are a beginner, you probably won't use any of the classes in the preceding list. Although the `PeepholeLSTMCell` class is a variation of the LSTM class, there are limited use cases for this class.

Working with Other tf.keras Namespaces

TF 2 provides a number of other namespaces that contain useful classes, some of which are listed here:

```
• tf.keras.callbacks      (early stopping)
• tf.keras.optimizers     (Adam et al)
• tf.keras.regularizers   (L1 and L2)
• tf.keras.utils          (to_categorical)
```

The `tf.keras.callbacks` namespace contains a class that you can use for "early stopping," which is to say that it's possible to terminate the training process if there is insufficient reduction in the cost function in two successive iterations.

The `tf.keras.optimizers` namespace contains the various optimizers that are available for working in conjunction with cost functions, which includes the popular Adam optimizer.

The `tf.keras.regularizers` namespace contains two popular regularizers: L1 regularizer (also called `LASSO` in machine learning) and the L2 regularizer (also called the `Ridge` regularizer in machine learning). L1 is for MAE (Mean Absolute Error) and L2 is for MSE (Mean Squared Error). Both of these regularizers act as "penalty" terms that are added to the chosen cost function in order to reduce the "influence" of features in a machine learning model. Note that `LASSO` can drive values to zero, with the result that features are actually eliminated from a model, and hence it is related to something called *feature selection* in machine learning.

The `tf.keras.utils` namespace contains an assortment of functions, including the `to_categorical()` function for converting a class vector into a binary class.

Although there are other namespaces in TF 2, the classes listed in all the preceding subsections will probably suffice for the majority of your tasks if you are a beginner in TF 2 and machine learning.

TF 2 Keras versus "Standalone" Keras

The original Keras is actually a specification, with various "backend" frameworks such as TensorFlow, Theano, and CNTK. Currently Keras standalone does not support TF 2, whereas the implementation of Keras in `tf.keras` has been optimized for performance.

Keras standalone will live in perpetuity in the keras.io package, which is discussed in detail at the Keras website: keras.io.

Now that you have a high-level view of the TF 2 namespaces for Keras and the classes that they contain, let's find out how to create a Keras-based model, which is the subject of the next section.

CREATING A KERAS-BASED MODEL

A Keras model generally involves at least the following sequence of steps:

- Specify a dataset (if necessary, convert data to numeric data)
- Split the dataset into training data and test data (usually 80/20 split)
- Define the Keras model (the `tf.keras.models.Sequential()` API)
- Compile the Keras model (the `compile()` API)
- Train (fit) the Keras model (the `fit()` API)
- Make a prediction (the `prediction()` API)

Note that the preceding bullet items skip some steps that are part of a real Keras model, such as evaluating the Keras model on the test data, as well as dealing with issues such as overfitting.

The first bullet item states that you need a dataset, which can be as simple as a CSV file with 100 rows of data (and just 3 columns). In general, a dataset is substantially larger: it can be a file with 1,000,000 rows of data and 10,000 columns in each row. We'll look at a concrete dataset in a subsequent section.

Next, a Keras model is in the `tf.keras.models` namespace, and the simplest (and also very common) Keras model is `tf.keras.models.Sequential`. In general, a Keras model contains layers that are in the `tf.keras.layers` namespace, such as `tf.keras.Dense` (which means that two adjacent layers are completely connected).

The activation functions that are referenced in Keras layers are in the `tf.nn` namespace, such as the `tf.nn.ReLU` for the ReLU activation function.

Here's a code block of the Keras model that's described in the preceding paragraphs (which covers the first four bullet points):

```
import tensorflow as tf

model = tf.keras.models.Sequential([
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
])
```

We have three more bullet items to discuss, starting with the compilation step. Keras provides a `compile()` API for this step, an example of which is here:

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Next we need to specify a training step, and Keras provides the `fit()` API (as you can see, it's not called `train()`), an example of which is here:

```
model.fit(x_train, y_train, epochs=5)
```

The final step is the prediction, and Keras provides the `predict()` API, an example of which is here:

```
pred = model.predict(x)
```

Listing 4.6 displays the contents of `tf2_basic_keras.py`, which combines the code blocks in the preceding steps into a single code sample.

LISTING 4.6: tf2_basic_keras.py

```
import tensorflow as tf

# NOTE: we need the train data and test data

model = tf.keras.models.Sequential([
  tf.keras.layers.Dense(1, activation=tf.nn.relu),
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Listing 4.6 contains no new code, and we've essentially glossed over some of the terms such as the optimizer (an algorithm that is used in conjunction with a cost function), the loss (the type of loss function), and the metrics (how to evaluate the efficacy of a model).

The explanations for these details cannot be condensed into a few paragraphs (alas), but the good news is that you can find a plethora of detailed online blog posts that discuss these terms. The appendix contains additional Keras-based code samples involving advanced topics.

KERAS AND LINEAR REGRESSION

This section contains a simple example of creating a Keras-based model in order to solve a task involving linear regression: given a positive number representing kilograms of pasta, predict its corresponding price. Listing 4.7 displays the contents of `pasta.csv` and Listing 4.8 displays the contents of `tf2_pasta.py` that performs this task.

LISTING 4.7: pasta.csv

```
weight,price
5,30
10,45
15,70
20,80
25,105
30,120
35,130
40,140
50,150
```

LISTING 4.8: tf2_pasta.py

```
import tensorflow as tf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# price of pasta per kilogram
df = pd.read_csv("pasta.csv")

weight = df['weight']
price  = df['price']

model = tf.keras.models.Sequential([
   tf.keras.layers.Dense(units=1,input_shape=[1])
])

# MSE loss function and Adam optimizer
model.compile(loss='mean_squared_error',
              optimizer=tf.keras.optimizers.Adam(0.1))

# train the model
history = model.fit(weight, price, epochs=100, verbose=False)

# graph the # of epochs versus the loss
plt.xlabel('Number of Epochs')
plt.ylabel("Loss Values")
plt.plot(history.history['loss'])
plt.show()
```

```
print("Cost for 11kg:",model.predict([11.0]))
print("Cost for 45kg:",model.predict([45.0]))
```

Listing 4.8 initializes the `Pandas Dataframe` `df` with the contents of the CSV file `pasta.csv`, and then initializes the variables `weight` and `cost` with the first and second columns, respectively, of `df`.

The next portion of Listing 4.8 defines a Keras-based model that consists of a single `Dense` layer. This model is compiled and trained, and then a graph is displayed that shows the "number of epochs" on the horizontal axis and the corresponding value of the loss function for the vertical axis. Launch the code in Listing 4.8 and you will see the following output:

```
Cost for 11kg: [[41.727108]]
Cost for 45kg: [[159.02121]]
```

Figure 4.9 displays a graph of epochs versus loss during the training process.

The next section introduces the `tf.estimator` namespace, followed by an example of using a TF estimator with a TF Dataset.

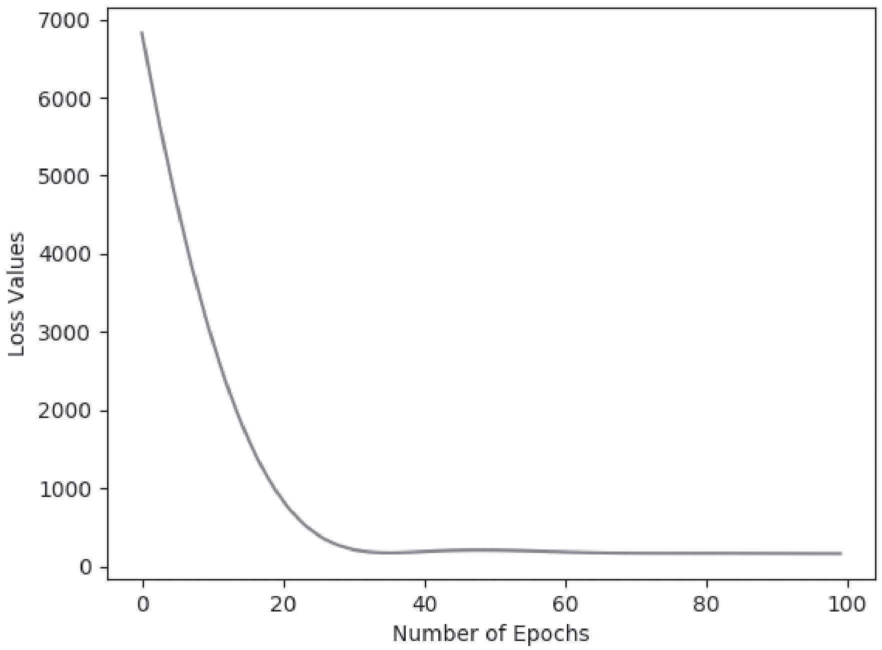

FIGURE 4.9. A graph of epochs versus loss.

WORKING WITH `tf.estimator`

Estimators are a layer above `tf.keras.layers`, which means that estimators provide a layer of abstraction. Estimator-based models run on CPUs, GPUs, or TPUs without model changes. Moreover, estimator-based models run locally or in a distributed environment without model changes.

The estimator classes "live" in the `tf.estimator` namespace. Estimators exist for an assortment of classes that implement various algorithms in machine learning, such as boosted trees, DNN classifiers, DNN regressors, linear classifiers, and linear regressors.

Every estimator has a model function that constructs graphs for training, evaluation, and prediction. Whenever you create a custom `Estimator`, you must define the model function (they are already defined for existing `Estimators`).

The estimator-related classes `DNNRegressor`, `LinearRegressor`, and `DNNLinearCombinedRegressor` are for regression tasks, whereas the classes `DNNClassifier`, `LinearClassifier`, and `DNNLinearCombinedClassifier` are for classification tasks. A more extensive list of estimator classes (with very brief descriptions) is listed as follows:

- BoostedTreesClassifier: A classifier for Tensorflow Boosted Trees models
- BoostedTreesRegressor: A regressor for Tensorflow Boosted Trees models
- CheckpointSaverHook: Saves checkpoints every N steps or seconds
- DNNClassifier: A classifier for TensorFlow DNN models
- DNNEstimator: An estimator for TensorFlow DNN models with user-specified head
- DNNLinearCombinedClassifier: An estimator for TensorFlow Linear and DNN joined classification models
- DNNLinearCombinedRegressor: An estimator for TensorFlow Linear and DNN joined models for regression
- DNNRegressor: A regressor for TensorFlow DNN models
- Estimator: Estimator class to train and evaluate TensorFlow models
- LinearClassifier: Linear classifier model
- LinearEstimator: An estimator for TensorFlow Linear models with user-specified head
- LinearRegressor: An estimator for TensorFlow Linear regression problems

All estimator classes are in the `tf.estimator` namespace, and all the estimator classes inherit from the `tf.estimator.Estimator` class. Read the online documentation for the details of the preceding classes as well as online tutorials for relevant code samples.

SUMMARY

This chapter introduced you to linear regression and a brief description of how to calculate a best-fitting line for a dataset of points in the Euclidean plane. You saw how to perform linear regression using `NumPy` in order to initialize arrays with data values, along with a "perturbation" technique that introduces some randomness for the y values. This technique is useful because you will know the correct values for the slope and y-intercept of the best-fitting line, which you can then compare with the trained values.

In addition, you learned about concepts such as regularization, ML and feature scaling, and data normalization versus standardization. Then you were introduced to metrics for measuring models, such as R-Squared and its limitations, the confusion matrix, and accuracy versus precision versus recall. Moreover, you learned about other useful statistical terms, such as RSS, TSS, F1 score, and p-value.

You then learned how to perform linear regression in code samples that involve TF 2. Furthermore, you learned about TF estimators and how they provide implementations of various algorithms, such as linear regression (`LinearRegressor`) and linear classification (`LinearClassifier`). You also saw code samples involving the `LinearRegressor` class for training TF 2 models to perform linear regression.

Finally, you got a very condensed introduction to Keras, with a description of some of its more important namespaces, along with a Keras-based code sample for solving a task involving linear regression.

CHAPTER 5

WORKING WITH CLASSIFIERS

This chapter presents numerous classification algorithms in machine learning. This includes algorithms such as the `kNN` (k Nearest Neighbor) algorithm, logistic regression (despite its name it *is* a classifier), decision trees, random forests, SVMs, and Bayesian classifiers. The emphasis on algorithms is intended to introduce you to machine learning, which includes a tree-based code sample that relies on `scikit-learn`. The latter portion of this chapter contains TF 2 code samples and Keras-based code samples for standard datasets.

Due to space constraints, this chapter does not cover other well-known algorithms such as linear discriminant analysis and the kMeans algorithm (for unsupervised learning and clustering). However, there are many online tutorials available that discuss these and other algorithms in machine learning.

With the preceding points in mind, the first section of this chapter briefly discusses the classifiers that are mentioned in the introductory paragraph. The second section of this chapter provides an overview of activation functions, which will be very useful if you decide to learn about deep neural networks. In this section you will learn how and why they are used in neural networks. This section also contains a list of the TensorFlow APIs for activation functions, followed by a description of some of their merits.

The third section introduces logistic regression, along with a code sample that involves logistic regression and TensorFlow. Logistic regression relies on the sigmoid function, which is also used in RNNs (recurrent neural networks) and LSTMs (long short term memory).

The fourth part of this chapter contains a code sample involving TensorFlow, logistic regression, and the MNIST dataset. This code sample relies on an understanding of other code samples that are discussed in Chapter 2 (the names of those code samples are provided in the description of the code sample).

The final portion of this chapter contains Keras-based code samples that illustrate how to perform "early stopping" and how to define a custom Python class to handle various events during the Keras training life cycle.

In order to give you some context, classifiers are one of three major types of algorithms: regression algorithms (such as linear regression in Chapter 4), classification algorithms (discussed in this chapter), and clustering algorithms (such as kMeans, which is not discussed in this book).

Another point: the section pertaining to activation functions does involve a basic understanding of hidden layers in a neural network. Depending on your comfort level, you might benefit from reading some preparatory material before diving into this section (there are many articles available online).

WHAT IS CLASSIFICATION?

Given a dataset that contains observations whose class membership is known, classification is the task of determining the class to which a new data point belongs. Classes refer to categories and are also called targets or labels. For example, spam detection in email service providers involves binary classification (only two classes). The `MNIST` dataset contains a set of images where each image is a single digit, which means there are ten labels. Some applications in classification include credit approval, medical diagnosis, and target marketing.

What Are Classifiers?

In the previous chapter, you learned that linear regression uses supervised learning in conjunction with numeric data: the goal is to train a model that can make numeric predictions (e.g., the price of stock tomorrow, the temperature of a system, its barometric pressure, and so forth). By contrast, classifiers use supervised learning in conjunction with nonnumerical classes of data: the goal is to train a model that can make categorical predictions.

For instance, suppose that each row in a dataset is a specific wine, and each column pertains to a specific wine feature (tannin, acidity, and so forth). Suppose further that there are five classes of wine in the dataset: for simplicity, let's label them A, B, C, D, and E. Given a new data point, which is to say a new row of data, a classifier attempts to determine the label for the new wine.

Some of the classifiers in this chapter can perform categorical classification and also make numeric predictions (i.e., they can be used for regression as well as classification).

Common Classifiers

Some of the most popular classifiers for machine learning are listed here (in no particular order):

- linear classifiers
- kNN (k Nearest Neighbor)

- logistic regression
- decision trees
- random forests
- SVMs (Support Vector Machines)
- Bayesian classifiers
- CNNs (Convolutional Neural Networks)

Keep in mind that different classifiers have different advantages and disadvantages, which often involves a trade-off between complexity and accuracy, similar to algorithms in fields that are outside of AI.

In the case of deep learning, CNNs perform image classification, which makes them classifiers (they can also be used for audio and text processing). The subsequent sections provide a brief description of the ML classifiers that are listed in the previous list.

WHAT ARE LINEAR CLASSIFIERS?

A linear classifier separates a dataset into two classes. A linear classifier is a line for 2D points, a plane for 3D points, and a hyperplane (a generalization of a plane) for higher dimensional points.

Linear classifiers are often the fastest classifiers, so they are often used when the speed of classification is of high importance. Linear classifiers usually work well when the input vectors are sparse (i.e., mostly zero values) or when the number of dimensions is large.

WHAT IS KNN?

The kNN ("k Nearest Neighbor") algorithm is a classification algorithm. In brief, data points that are "near" each other are classified as belonging to the same class. When a new point is introduced, it's added to the class of the majority of its nearest neighbor. For example, suppose that k equals 3, and a new data point is introduced. Look at the class of its three nearest neighbors: let's say they are A, A, and B. Then by majority vote, the new data point is labeled as a data point of class A.

The kNN algorithm is essentially a heuristic and not a technique with complex mathematical underpinnings, and yet it's still an effective and useful algorithm. Try the kNN algorithm if you want to use a simple algorithm, or when you believe that the nature of your dataset is highly unstructured. The kNN algorithm can produce highly nonlinear decisions despite being very simple.

Note that kNN is often used in search applications where you are looking for "similar" items; that is, when your task is some form of "find items similar to this one."

Measure similarity by creating a vector representation of the items, and then compare the vectors using an appropriate distance metric (such as Euclidean distance).

Some concrete examples of a kNN search include searching for semantically similar documents.

How to Handle a Tie in kNN

An odd value for k is less likely to result in a tie vote, but it's not impossible. For example, suppose that k equals 7, and when a new data point is introduced, the labels of its seven nearest neighbors belong to the set {A,B,A,B,A,B,C}. As you can see, there is no majority vote, because there are three points in class A, three points in class B, and one point in class C.

There are several techniques for handling a tie, as listed here:

- Assign higher weights to closer points
- Increase the value of k until a winner is determined
- Decrease the value of k until a winner is determined
- Randomly select one class

If you reduce k until it equals 1, it's still possible to have a tie vote: there might be two points that are equally distant from the new point, so you need a mechanism for deciding which of those two points to select as the 1-neighbor.

If there is a tie between classes A and B, then randomly select either class A or class B. Another variant is to keep track of the "tie" votes, and alternate round-robin style to ensure a more even distribution.

WHAT ARE DECISION TREES?

Decision trees are another type of classification algorithm that involves a tree-like structure. Keep in mind that a "generic" tree is constructed using conditional logic. As a simple illustration, suppose that a dataset contains a set of numbers representing ages of people, and let's also suppose that the first number is 50. This number is chosen as the root of the tree, and all numbers that are smaller than 50 are added on the left branch of the tree, whereas all numbers that are greater than 50 are added on the right branch of the tree.

For example, suppose the sequence of numbers is {50, 25, 70, 40}. Then we can construct a tree as follows: 50 is the root node; 25 is the left child of 50; 70 is the right child of 50; and 40 is the right child of 20. Each additional numeric value that we add to this dataset is processed to determine which direction to proceed ("left or right") at each node in the tree.

Listing 5.1 displays a portion of the dataset `partial_wine.csv`, which contains two features and a label column (there are three classes). The total row count for this dataset is 178.

LISTING 5.1: partial_wine.csv

```
Alcohol, Malic acid, class
14.23,1.71,1
13.2,1.78,1
```

```
13.16,2.36,1
14.37,1.95,1
13.24,2.59,1
14.2,1.76,1
```

Listing 5.2 displays the contents of `tree_classifier.py`, which uses a decision tree in order to train a model on the dataset `partial_wine.csv`.

LISTING 5.2: tree_classifier.py

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

# Importing the dataset
dataset = pd.read_csv('partial_wine.csv')
X = dataset.iloc[:, [0, 1]].values
y = dataset.iloc[:, 2].values

# split the dataset into a training set and a test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size = 0.25, random_state = 0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# ====> INSERT YOUR CLASSIFIER CODE HERE <====
from sklearn.tree import DecisionTreeClassifier
classifier = DecisionTreeClassifier(criterion='entropy',ran
dom_state=0)
classifier.fit(X_train, y_train)
# ====> INSERT YOUR CLASSIFIER CODE HERE <====

# predict the test set results
y_pred = classifier.predict(X_test)

# generate the confusion matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print("confusion matrix:")
print(cm)
```

Listing 5.3 contains some `import` statements and then populates the Pandas `DataFrame` variable `dataset` with the contents of the CSV file `partial_wine.csv`. Next, the variable `x` is initialized with the first two columns (and all the rows) of `dataset`, and the variable `y` is initialized with the third column (and all the rows) of `dataset`.

Next, the variables X_train, X_test, y_train, and y_test are populated with data from X and y using a 75/25 split proportion. Notice that the variable sc (which is an instance of the StandardScalar class) performs a scaling operation on the variables X_train and X_test.

The code block shown in bold in Listing 5.3 is where we create an instance of the DecisionTreeClassifier class and then train the instance with the data in the variables X_train and X_test.

The next portion of Listing 5.3 populates the variable y_pred with a set of predictions that are generated from the data in the X_test variable. The last portion of Listing 5.3 creates a confusion matrix based on the data in y_test and the predicted data in y_pred.

Remember that the diagonal elements of a confusion matrix are the correct predictions (such as true positive and true negative); all the other cells contain a numeric value that specifies the number of predictions that are incorrect (such as false positive and false negative).

Now launch the code in Listing 5.3 and you will see the following output for the confusion matrix in which there are thirty-six correct predictions and nine incorrect predictions (with an accuracy of 80%):

```
confusion matrix:
[[13  1  2]
 [ 0 17  4]
 [ 1  1  6]]
```

There is a total of forty-five entries in the preceding 3x3 matrix, and the diagonal entries are correctly identified labels. Hence, the accuracy is 36/45 = 0.80.

WHAT ARE RANDOM FORESTS?

Random forests are a generalization of decision trees: this classification algorithm involves multiple trees (the number is specified by you). If the data involves making a numeric prediction, the average of the predictions of the trees is computed. If the data involves a categorical prediction, the mode of the predictions of the trees is determined.

By way of analogy, random forests operate in a manner similar to financial portfolio diversification: the goal is to balance the losses with higher gains. Random forests use a “majority vote” to make predictions, which operates under the assumption that selecting the majority vote is more likely to be correct (and more often) than any individual prediction from a single tree.

You can easily modify the code in Listing 5.3 to use a random forest by replacing the two lines shown in bold with the following code:

```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators = 10,
criterion='entropy', random_state = 0)
```

Make this code change, launch the code, and examine the confusion matrix to compare its accuracy with the accuracy of the decision tree in Listing 5.3.

WHAT ARE SVMS?

Support vector machines involve a supervised ML algorithm and can be used for classification or regression problems. SVMs can work with nonlinearly separable data as well as linearly separable data. An SVM uses a technique called the "kernel trick" to transform data and then finds an optimal boundary. The data points are "transformed" into a higher dimension in order to find a linear separation of the transformed data.

The key idea involves finding a hyperplane that best divides a dataset into two classes. SVMs are more common in classification tasks than regression tasks. Some common use cases for SVMs include:

- text classification tasks: category assignment
- detecting spam/sentiment analysis
- used for image recognition: aspect-based recognition and color-based classification
- handwritten digit recognition (postal automation)

Trade-offs of SVMs

Although SVMs are extremely powerful, there are trade-offs involved. Some of the advantages of SVMs are listed here:

- high accuracy
- works well on smaller, cleaner datasets
- can be more efficient because it uses a subset of training points
- an alternative to CNNs in cases of limited datasets
- captures more complex relationships between data points

There are some disadvantages of SVMs that are listed here:

- not suited to larger datasets: training time can be high
- less effective on noisier datasets with overlapping classes
- SVMs involve more parameters than decision trees and random forests

Now modify Listing 5.3 to use an SVM by replacing the two lines shown in bold with the following two lines shown in bold:

```
from sklearn.svm import SVC
classifier = SVC(kernel = 'linear', random_state = 0)
```

You now have an SVM-based model, simply by making the previous code update! Make the code change, launch the modified code, and examine the confusion matrix in order to compare its accuracy with the accuracy of the decision tree model and the random forest model earlier in this chapter.

WHAT IS BAYESIAN INFERENCE?

Bayesian inference is an important technique in statistics that involves statistical inference and Bayes's theorem to update the probability for a hypothesis as more information becomes available. Bayesian inference is often called "Bayesian probability," and it's important in dynamic analysis of sequential data.

Bayes's Theorem

Given two sets A and B, let's define the following numeric values (all of them are between 0 and 1):

```
P(A)  = probability of being in set A
P(B)  = probability of being in set B
P(Both) = probability of being in A intersect B
P(A|B) = probability of being in A (given you're in B)
P(B|A) = probability of being in B (given you're in A)
```

Given the preceding definitions, the following formulas are also true:

```
P(A|B) = P(Both)/P(B) (#1)
P(B|A) = P(Both)/P(A) (#2)
```

Multiply the preceding pair of equations by the term that appears in the denominator and we get these equations:

```
P(B)*P(A|B) = P(Both) (#3)
P(A)*P(B|A) = P(Both) (#4)
```

Now set the left sides of equations #3 and #4 equal to each another, and that gives us this equation:

```
P(B)*P(A|B) = P(A)*P(B|A) (#5)
```

Divide both sides of #5 by `P(B)` and we get this well-known equation:

```
P(A|B) = P(A)*P(A|B)/P(B) (#6)
```

Some Bayesian Terminology

In the previous section, we derived the following relationship:

```
P(h|d) = (P(d|h) * P(h)) / P(d)
```

Each of the four terms in the preceding equation has a name, as discussed in the following.

First, the *posterior probability* is `P(h|d)`, which is the probability of hypothesis h given the data d.

Second, `P(d|h)` is the probability of data d given that the hypothesis h was true.

Third, the *prior probability* of `h` is `P(h)`, which is the probability of hypothesis `h` being true (regardless of the data).

Finally, `P(d)` is the probability of the data (regardless of the hypothesis).

We are interested in calculating the posterior probability of P(h|d) from the prior probability p(h) with P(D) and P(d|h).

What Is MAP?

The maximum a posteriori (MAP) hypothesis is the hypothesis with the highest probability, which is the maximum probable hypothesis. This can be written as follows:

```
MAP(h) = max(P(h|d))
```

or:

```
MAP(h) = max((P(d|h) * P(h)) / P(d))
```

or:

```
MAP(h) = max(P(d|h) * P(h))
```

Why Use Bayes's Theorem?

Bayes's Theorem describes the probability of an event based on the prior knowledge of the conditions that might be related to the event. If we know the conditional probability, we can use Bayes rule to find out the reverse probabilities. The previous statement is the general representation of the Bayes rule.

WHAT IS A BAYESIAN CLASSIFIER?

A Naive Bayes (NB) classifier is a probabilistic classifier inspired by the Bayes theorem. An NB classifier assumes the attributes are conditionally independent, and it can work even when this assumption is not true. This assumption greatly reduces computational cost, and it's a simple algorithm to implement that only requires linear time. Moreover, an NB classifier is easily scalable to larger datasets, and good results are obtained in most cases. Other advantages of an NB classifier include:

- can be used for binary and multiclass classification
- provides different types of NB algorithms
- good choice for text classification problems
- a popular choice for spam email classification
- can be easily trained on small datasets

As you can probably surmise, NB classifiers do have some disadvantages, as listed in the following:

- all features are assumed unrelated
- it cannot learn relationships between features
- it can suffer from “the zero probability problem”:

The “zero probability problem” refers to the case when the conditional probability is zero for an attribute, and it fails to give a valid prediction. However, it can be fixed explicitly using a Laplacian estimator.

Types of Naive Bayes Classifiers

Naive Bayes classifiers consist of “probabilistic classifiers” that are based on applying Bayes’s theorem with strong (naive) independence assumptions among the features. Naive Bayes classifiers are highly scalable, requiring a number of parameters that is linear in the number of variables (features/predictors) in a learning problem. Maximum-likelihood training is performed by evaluating a closed-form expression that requires linear time, which is more efficient than other types of classifiers.

There are three major types of NB classifiers. A *Gaussian Naive Bayes* classifier is used in classification, and the assumption is that features follow a normal distribution. A *Multinomial Naive Bayes* classifier involves 1xn feature vectors representing the frequencies of events that have been generated. Each feature vector contains data for a histogram whose elements equal the number of times that an event was observed in an instance. This type of event model is used for document classification. A *Bernoulli Naive Bayes* classifier is suitable for binary feature vectors. An example is the Bag of Words (BoW) algorithm for text classification where 0 and 1 represent absence or occurrence of a word, respectively, in a document.

TRAINING CLASSIFIERS

Some common techniques for training classifiers are listed here:

- holdout method
- k-fold cross-validation

The *holdout method* is the most common method, which starts by dividing the dataset into two partitions called train and test (80% and 20%, respectively). The train set is used for training the model, and the test data tests its predictive power.

The *k-fold cross-validation* technique is used to verify that the model is not over-fitted. The dataset is randomly partitioned into k mutually exclusive subsets, where each partition has equal size. One partition is for testing and the other partitions are for training. Iterate throughout the whole of the k folds.

EVALUATING CLASSIFIERS

Whenever you select a classifier for a dataset, it's obviously important to evaluate the accuracy of that classifier. Some common techniques for evaluating classifiers are listed here:

- precision and recall
- ROC curve (receiver operating characteristics)

Precision and recall are discussed in Chapter 4 and are reproduced here for your convenience. Recall the following definitions from Chapter 4:

```
TP = the number of true positive results
FP = the number of false positive results
TN = the number of true negative results
FN = the number of false negative results
```

Then the definitions of precision, accuracy, and recall are given by the following formulas:

```
precision = TP/(TN + FP)
accuracy  = (TP + TN)/[P + N]
recall    = TP/[TP + FN]
```

The *ROC curve (receiver operating characteristics)* is used for visual comparison of classification models that shows the trade-off between the true positive rate and the false positive rate. The area under the ROC curve is a measure of the accuracy of the model. When a model is closer to the diagonal, it is less accurate, and the model with perfect accuracy will have an area of 1.0.

The ROC curve plots true positive rate versus false positive rate. Another type of curve is the PR curve that plots precision versus recall. When dealing with highly skewed datasets (strong class imbalance), precision-recall (PR) curves give better results.

Later in this chapter you will see many of the Keras-based classes (located in the `tf.keras.metrics` namespace) that correspond to common statistical terms, which include some of the terms in this section.

This concludes the portion of the chapter pertaining to statistical terms and techniques for measuring the validity of a dataset. Now let's look at activation functions in machine learning, which is discussed in the next section.

WHAT ARE ACTIVATION FUNCTIONS?

The following is a one-sentence description: an activation function is a nonlinear function that introduces nonlinearity into a neural network, thereby preventing a "consolidation" of the hidden layers in the neural network. Specifically, suppose that every pair of adjacent layers in a neural network involves just a matrix transformation and no activation function. *Such a network is a* *linear system, which means that its layers can be consolidated into a much smaller system.*

Notice that the weights of the edges that connect the input layer with the first hidden layer can be represented by a matrix: let's call it `W1`. Next, the weights of the edges that connect the first hidden layer with the second hidden layer can also be represented by a matrix: let's call it `W2`. Repeat this process until we reach the edges that connect the final hidden layer with the output layer: let's call this matrix `Wk`. Since we do not have an activation function, we can simply multiply the matrices `W1`, `W2`, …, `Wk` together and produce one matrix: let's call it `W`. We have now replaced the original neural network with an equivalent neural network that contains one input layer, a single matrix of weights `W`, and an output layer. In other words, we no longer have our original multilayered neural network!

Fortunately, we can prevent the previous scenario from happening when we specify an activation function between every pair of adjacent layers. In other words, *an activation function at each layer prevents this "matrix consolidation."* Hence, we can maintain all the intermediate hidden layers during the process of training the neural network.

For simplicity, let's assume that we have the same activation function between every pair of adjacent layers (we'll remove this assumption shortly). The process for using an activation function in a neural network is described as follows:

1. start with an input vector `x1` of numbers
2. multiply `x1` by the matrix of weights W1 that represents the edges that connect the input layer with the first hidden layer: the result is a new vector `x2`
3. "apply" the activation function to each element of `x2` to create another vector `x3`

Now repeat steps 2 and 3, except that we use the "starting" vector x3 and the weights matrix `W2` for the edges that connect the first hidden layer with the second hidden layer (or just the output layer if there is only one hidden layer).

After completing the preceding process, we have "preserved" the neural network, which means that it can be trained on a dataset. One other thing: instead of using the same activation function at each step, you can replace each activation function by a different activation function (the choice is yours).

Why Do We Need Activation Functions?

The previous section outlines the process for transforming an input vector from the input layer and then through the hidden layers until it reaches the output layer. The purpose of activation functions in neural networks is vitally important, so it's worth repeating here: activation functions "maintain" the structure of neural networks and prevent them from being reduced to an input layer and an output layer. In other words, if we specify a nonlinear activation function between every pair of consecutive layers, then the neural network cannot be replaced with a neural network that contains fewer layers.

Without a nonlinear activation function, we simply multiply a weight matrix for a given pair of consecutive layers with the output vector that is produced from the previous pair of consecutive layers. We repeat this simple multiplication until we reach the output layer of the neural network.

How Do Activation Functions Work?

If this is the first time you have encountered the concept of an activation function, it's probably confusing, so here's an analogy that might be helpful. Suppose you're driving your car late at night and there's nobody else on the highway. You can drive at a constant speed for as long as there are no obstacles (stop signs, traffic lights, and so forth). On the other hand, suppose you drive into the parking lot of a large grocery store. When you approach a speed bump you must slow down, cross the speed bump, and increase speed again, and repeat this process for every speed bump, and also slow down for other vehicles and pedestrians.

Think of the nonlinear activation functions in a neural network as the counterpart to the speed bumps: you simply cannot maintain a constant speed, which (by analogy) means that you cannot first multiply all the weight matrices together and "collapse" them into a single weight matrix. Another analogy involves a road with multiple toll booths: you must slow down, pay the toll, and then resume driving until you reach the next toll booth. These are only analogies (and hence imperfect) to help you understand the need for nonlinear activation functions.

COMMON ACTIVATION FUNCTIONS

Although there are many activation functions (and you can define your own if you know how to do so), here is a list of common activation functions, followed by brief descriptions:

- Sigmoid
- Tanh
- ReLU
- ReLU6
- ELU
- SELU

The `sigmoid` activation function is based on Euler's constant e, with a range of values between 0 and 1, and its formula is shown here:

```
1/[1+e^(-x)]
```

The tanh activation function is also based on Euler's constant e, and its formula is shown here:

```
[e^x - e^(-x)]/[e^x+e^(-x)]
```

One way to remember the preceding formula is to note that the numerator and denominator have the same pair of terms: they are separated by a "-" sign in the numerator and a "+" sign in the denominator. The tanh function has a range of values between -1 and 1.

The ReLU (rectified linear unit) activation function is straightforward: if x is negative then ReLU(x) is 0; for all other values of x, ReLU(x) equals x. ReLU6 is specific to TensorFlow, and it's a variation of ReLU(x): the additional constraint is that ReLU(x) equals 6 when x >= 6 (hence its name).

ELU is the exponential linear unit, and it's the exponential "envelope" of ReLU, which replaces the two linear segments of ReLU with an exponential activation function that is differentiable for all values of x (including x = 0).

SELU is an acronym for scaled exponential linear unit, and it's slightly more complicated than the other activation functions (and used less frequently). For a thorough explanation of these and other activation functions (along with graphs that depict their shape), navigate to the following Wikipedia link:

https://en.wikipedia.org/wiki/Activation_function

The preceding link provides a long list of activation functions as well as their derivatives.

Activation Functions in Python

Listing 5.4 displays contents of the file activations.py, which contains the formulas for various activation functions.

LISTING 5.4: activations.py

```
import numpy as np

# Python sigmoid example:
z = 1/(1 + np.exp(-np.dot(W, x)))

# Python tanh example:
z = np.tanh(np.dot(W,x))

# Python ReLU example:
z = np.maximum(0, np.dot(W, x))
```

Listing 5.4 contains Python code that uses NumPy methods in order to define a sigmoid function, a tanh function, and a ReLU function. Figure 5.1 displays a graph of each of the activation functions in Listing 5.4.

TF 2 (in addition to other frameworks) provides implementations for many activation functions, which saves you the time and effort from writing your own

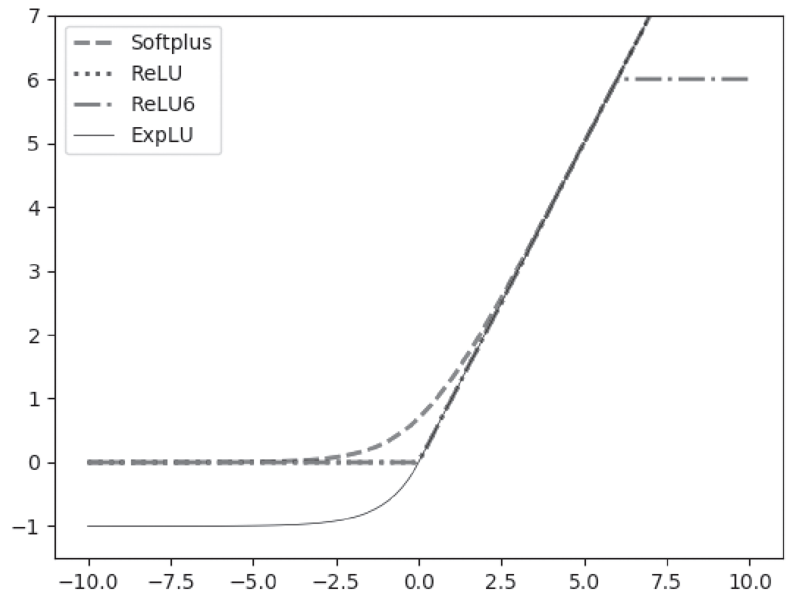

FIGURE 5.1. TensorFlow activation functions.

implementation of activation functions. The list of TF 2 API activation functions from Chapter 4 are reproduced here for your convenience:

- `tf.keras.activations.relu`
- `tf.keras.activations.selu`
- `tf.keras.activations.linear`
- `tf.keras.activations.elu`
- `tf.keras.activations.sigmoid`
- `tf.keras.activations.softmax`
- `tf.keras.activations.softplus`
- `tf.keras.activations.tanh`

The following subsections provide additional information regarding some of the activation functions in the preceding list. Keep the following point in mind: for simple neural networks, use `ReLU` as your first preference.

THE ReLU AND ELU ACTIVATION FUNCTIONS

Currently `ReLU` is often the "preferred" activation function: previously the preferred activation function was `tanh` (and before `tanh` it was `sigmoid`). `ReLU` behaves close to a linear unit and provides the best training accuracy and validation accuracy.

ReLU is like a switch for linearity: it's "off" if you don't need it, and its derivative is 1 when it's active, which makes ReLU the simplest of all the current activation functions. Note that the second derivative of the function is 0 everywhere (but undefined at the origin): it's a very simple function that simplifies optimization. In addition, the gradient is large whenever you need large values, and it never "saturates" (i.e., it does not shrink to zero on the positive horizontal axis).

Rectified linear units and generalized versions are based on the principle that linear models are easier to optimize. Use the ReLU activation function or one of its related alternatives (discussed later).

The Advantages and Disadvantages of ReLU

The following list contains the advantages of the ReLU activation function:

- does not saturate in the positive region
- very efficient in terms of computation
- models with ReLU typically converge faster than those with other activation functions

However, ReLU does have a disadvantage when the activation value of a ReLU neuron becomes 0: then the gradients of the neuron will also be 0 during back-propagation. You can mitigate this scenario by judiciously assigning the values for the initial weights as well as the learning rate.

ELU

ELU is an acronym for *exponential linear unit* that is based on ReLU: the key difference is that ELU is differentiable at the origin (ReLU is a continuous function but *not* differentiable at the origin). However, keep in mind several points. First, ELUs trade computational efficiency for "immortality" (immunity to dying): read the following paper for more details: arxiv.org/abs/1511.07289. Secondly, ReLUs are still popular and preferred over ELU because the use of ELU introduces an additional new hyper-parameter.

SIGMOID, SOFTMAX, AND HARDMAX SIMILARITIES

The sigmoid activation function has a range in (0,1), and it saturates and "kills" gradients. Unlike the tanh activation function, sigmoid outputs are not zero-centered. In addition, both sigmoid and softmax (discussed later) are discouraged for vanilla feed forward implementation (see Chapter 6 of the online book by Ian Goodfellow et al.). However, the sigmoid activation function is still used in LSTMs (specifically for the forget gate, input gate, and the output gate), GRUs (gated recurrent units), and probabilistic models. Moreover, some autoencoders have additional requirements that preclude the use of piecewise linear activation functions.

Softmax

The softmax activation function maps the values in a dataset to another set of values that are between 0 and 1, and whose sum equals 1. Thus, softmax creates a probability distribution. In the case of image classification with convolutional neural networks, the softmax activation function "maps" the values in the final hidden layer (often abbreviated as "FC") to the ten neurons in the output layer. The index of the position that contains the largest probability is matched with the index of the number 1 in the one-hot encoding of the input image. If the index values are equal, then the image has been classified; otherwise, it's considered a mismatch.

Softplus

The softplus activation function is a smooth (i.e., differentiable) approximation of the ReLU activation function. Recall that the origin is the only non-differentiable point of the ReLU function, which is "smoothed" by the softmax activation, whose equation is here:

```
f(x) = ln(1 + e^x)
```

Tanh

The tanh activation function has a range of values in the interval (-1,1), whereas the sigmoid function has a range of values in the interval (0,1). Both of these activations saturate, but unlike the sigmoid neuron the tanh output is zero-centered. Therefore, in practice the tanh nonlinearity is always preferred to the sigmoid nonlinearity.

The sigmoid and tanh activation functions appear in LSTMs (sigmoid for the three gates and tanh for the internal cell state) as well as GRUs (gated recurrent units) during the calculations pertaining to input gates, forget gates, and output gates (discussed in more detail in the next chapter).

SIGMOID, SOFTMAX, AND HARDMAX DIFFERENCES

This section briefly discusses some of the differences among these three functions. First, the sigmoid function is used for binary classification in the logistic regression model, as well as for the gates in LSTMs and GRUs. The sigmoid function is used as an activation function while building neural networks, but keep in mind that the sum of the probabilities is *not* necessarily equal to 1.

Second, the softmax function generalizes the sigmoid function: it's used for multi-classification in the logistic regression model. The softmax function is the activation function for the "fully connected layer" in CNNs, which is the rightmost hidden layer and the output layer. Unlike in the sigmoid function, the sum of the probabilities *must* equal 1. You can use either the sigmoid function or softmax for binary (n=2) classification.

Third, the so-called "`hardmax`" function assigns 0 or 1 to output values (similar to a step function). For example, suppose that we have three classes `{c1, c2, c3}` whose scores are `[1, 7, 2]`, respectively. The `hardmax` probabilities are `[0, 1, 0]`, whereas the `softmax` probabilities are `[0.1, 0.7, 0.2]`. Notice that the sum of the `hardmax` probabilities is 1, which is also true of the sum of the `softmax` probabilities. However, the `hardmax` probabilities are all or nothing, whereas the `softmax` probabilities are analogous to receiving "partial credit."

TF 2 AND THE SIGMOID ACTIVATION FUNCTION

Listing 5.5 displays the contents of `tf2_activation_functions.py`, which illustrates how to create a TensorFlow graph that involves seven activation functions, including the `sigmoid` function for logistic regression.

LISTING 5.5: tf2_activation_functions.py

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

# ReLU activation
print(tf.nn.relu([-3., 3., 10.]))
y_relu = tf.nn.relu(x_vals)

# ReLU-6 activation
print(tf.nn.relu6([-3., 3., 10.]))
y_relu6 = tf.nn.relu6(x_vals)

# Sigmoid activation
print(tf.nn.sigmoid([-1., 0., 1.]))
y_sigmoid = tf.nn.sigmoid(x_vals)

# Hyperbolic Tangent activation
print(tf.nn.tanh([-1., 0., 1.]))
y_tanh = tf.nn.tanh(x_vals)

# Softsign activation
print(tf.nn.softsign([-1., 0., 1.]))
y_softsign = tf.nn.softsign(x_vals)

# Softplus activation
print(tf.nn.softplus([-1., 0., 1.]))
y_softplus = tf.nn.softplus(x_vals)

# Exponential linear activation (ELU)
print(tf.nn.elu([-1., 0., 1.]))
y_elu = tf.nn.elu(x_vals)
```

```
# Plot the different functions
plt.plot(x_vals, y_softplus, 'r--', label='Softplus',
linewidth=2)
plt.plot(x_vals, y_relu, 'b:', label='ReLU', linewidth=2)
plt.plot(x_vals, y_relu6, 'g-.', label='ReLU6', linewidth=2)
plt.plot(x_vals, y_elu, 'k-', label='ExpLU', linewidth=0.5)
plt.ylim([-1.5,7])
plt.legend(loc='best')
plt.show()

plt.plot(x_vals, y_sigmoid, 'r--', label='Sigmoid',
linewidth=2)
plt.plot(x_vals, y_tanh, 'b:', label='Tanh', linewidth=2)
plt.plot(x_vals, y_softsign, 'g-.', label='Softsign',
linewidth=2)
plt.ylim([-2,2])
plt.legend(loc='best')
plt.show()
```

Listing 5.5 starts with some `import` statements, followed by an extensive code block that shows you how to invoke the TensorFlow activation functions that are listed in a previous section. The final section of code in Listing 5.5 plots the various TensorFlow functions.

Figure 5.2 displays the graph of the TensorFlow activation functions that are defined in the first portion of Listing 5.5.

Figure 5.3 displays the graph of the TensorFlow activation functions that are defined in the second portion of Listing 5.5.

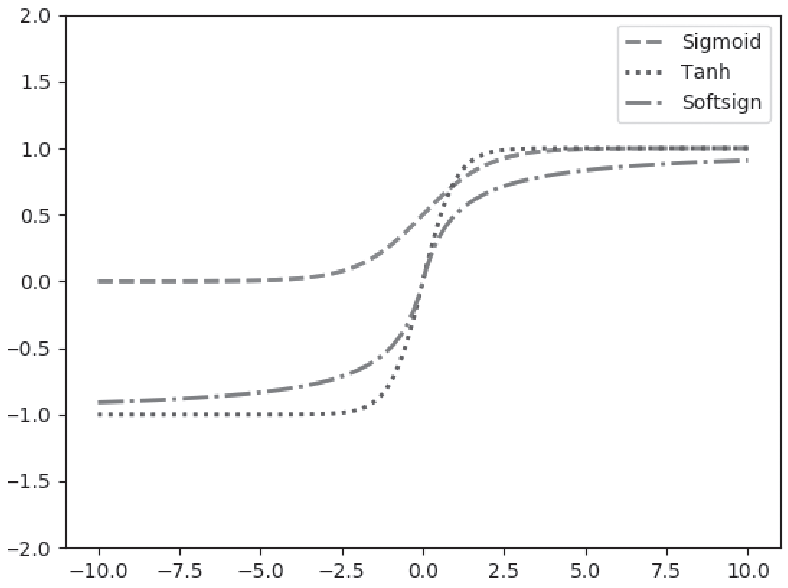

FIGURE 5.2. TensorFlow activation functions.

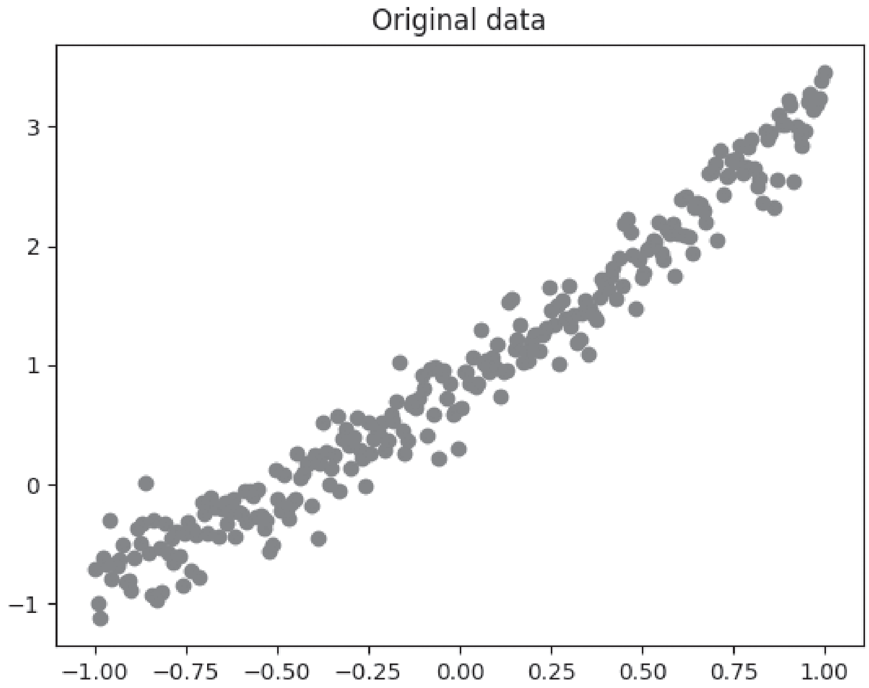

FIGURE 5.3. TensorFlow activation functions.

WHAT IS LOGISTIC REGRESSION?

Despite its name, logistic regression is a classifier as well as a model with a binary output. Logistic regression works with multiple independent variables and involves a sigmoid function for calculating probabilities. Logistic regression is essentially the result of "applying" the `sigmoid` activation function to linear regression in order to perform binary classification.

Logistic regression is useful in a variety of unrelated fields. Such fields include machine learning, various medical fields, and social sciences. Logistic regression can be used to predict the risk of developing a given disease, based on various observed characteristics of the patient. Other fields that use logistic regression include engineering, marketing, and economics.

Logistic regression can be binomial (only two outcomes for a dependent variable), multinomial (three or more outcomes for a dependent variable), or ordinal (dependent variables are ordered). For instance, suppose that a dataset consists of data that belong either to class A or class B. If you are given a new data point, logistic regression predicts whether that new data point belongs to class A or class B. By contrast, linear regression predicts a numeric value, such as the next-day value of a stock.

Setting a Threshold Value

The threshold value is a numeric value that determines which data points belong to class A and which points belong to class B. For instance, a pass/fail threshold might be 0.70. A pass/fail threshold for passing a written driver's test in California is 0.85.

As another example, suppose that p = 0.5 is the "cutoff" probability. Then we can assign class A to the data points that occur with probability > 0.5 and assign class B to data points that occur with probability <= 0.5. Since there are only two classes, we do have a classifier.

A similar (yet slightly different) scenario involves tossing a well-balanced coin. We know that there is a 50% chance of throwing heads (let's label this outcome as class A) and a 50% chance of throwing tails (let's label this outcome as class B). If we have a dataset that consists of labeled outcomes, then we have the expectation that approximately 50% of them are class A and 50% are class B.

On the other hand, we have no way to determine (in advance) what percentage of people will pass their written driver's test, or the percentage of people who will pass their course. Datasets containing outcomes for these types of scenarios need to be trained, and logistic regression is a suitable activation function for doing so.

Logistic Regression: Assumptions

Logistic regression requires the observations to be independent of each other. In addition, logistic regression requires little or no multicollinearity among the independent variables. Logistic regression handles numeric, categorical, and continuous variables, and also assumes linearity of independent variables and log odds, which is defined here:

```
odds = p/(1-p) and logit = log(odds)
```

This analysis does not require the dependent and independent variables to be related linearly; however, another requirement is that independent variables are linearly related to the log odds.

Logistic regression is used to obtain an odds ratio in the presence of more than one explanatory variable. The procedure is quite similar to multiple linear regression, with the exception that the response variable is binomial. The result is the impact of each variable on the odds ratio of the observed event of interest.

Linearly Separable Data

Linearly separable data is data that can be separated by a line (in 2D), a plane (in 3D), or a hyperplane (in higher dimensions). Linearly non-separable data is data (such as a set of clusters) that cannot be separated by a line or a hyperplane.

For example, the XOR function involves datapoints that cannot be separated by a line. If you create a truth table for an XOR function with two inputs, the points (0,0) and (1,1) belong to class 0, whereas the points (0,1) and (1,0) belong to class 1 (draw these points in a 2D plane to convince yourself). The solution involves transforming the data in a higher dimension so that it becomes linearly separable, which is the technique used in SVMs (discussed earlier in this chapter).

TENSORFLOW AND LOGISTIC REGRESSION

Listing 5.6 displays the contents of `tf2_keras_log_reg.py`, which defines a Keras-based model to perform logistic regression.

LISTING 5.6: tf2_keras_log_reg.py

```
import tensorflow as tf
import seaborn as sns
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegressionCV

# Load the Iris Dataset
iris = sns.load_dataset("iris")
X = iris.values[:, 0:4]
y = iris.values[:, 4]

# Create train and test data
train_X, test_X, train_y, test_y = train_test_split(X, y,
train_size=0.5, random_state=0)

# Make one-hot encoder
def one_hot_encode_object_array(arr):
  #One hot encode a numpy array of objects (e.g. strings)
  uniq_vals, ids = np.unique(arr, return_inverse=True)
  return tf.keras.utils.to_categorical(ids, len(uniq_vals))

train_y_hot = one_hot_encode_object_array(train_y)
test_y_hot  = one_hot_encode_object_array(test_y)

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(16, input_shape=(4,)))
model.add(tf.keras.layers.Activation('sigmoid'))
model.add(tf.keras.layers.Dense(3))
model.add(tf.keras.layers.Activation('softmax'))
model.compile(loss='categorical_crossentropy',
metrics=['accuracy'], optimizer='adam')

# train the model:
model.fit(train_X, train_y_hot, verbose=1, batch_size=1)
```

```
score, accuracy = model.evaluate(test_X, test_y_hot, batch_
size=16, verbose=0)

print("Test Score    = {:.2f}".format(score))
print("Test Accuracy = {:.2f}".format(accuracy))
```

Listing 5.6 starts with an assortment of `import` statements, and then initializes the variable `iris` with the `Iris` dataset. The variable `X` contains the first three columns (and all the rows) of the `Iris` dataset, and the variable `y` contains the fourth column (and all the rows) of the `Iris` dataset.

The next portion of Listing 5.6 initializes the training set and the test set equally: they both contain 50% of the data. The next code block is a Python function that returns a one-hot encoding of its input (one-hot encoding is described in Chapter 2). This Python function is invoked to populate `train_y_hot` and `test_y_hot` as one-hot encoded data. Next, the Keras-based model contains four hidden layers: a `Dense` layer and sigmoid activation function, followed by another `Dense` layer and a `softmax` activation function.

The next portion of Listing 5.6 compiles the model, trains the model, and then calculates the accuracy of the model via the test data. Launch the code in Listing 5.6 and you will see the following output:

```
105/105 [==============================] - 0s 2ms/sample -
loss: 1.2798 - accuracy: 0.3048
45/45 [==============================] - 0s 1ms/sample -
loss: 1.0867 - accuracy: 0.4000
Test Score    = 1.09
Test Accuracy = 0.40
```

KERAS AND EARLY STOPPING (1)

When you create a model for a neural network, you also need to decide on the number of training epochs. This number is hardly an obvious choice; in fact, a value that's too large can lead to overfitting, whereas a value that's too small can lead to underfitting.

Early stopping is a technique that allows you to specify a large value for the number of epochs, and yet the training will stop if the model performance improvement drops below a threshold value.

There are several ways that you can specify early stopping, and they involve the concept of a *callback function*. Listing 5.7 displays the contents of `tf2_keras_callback.py`, which performs early stopping via a callback mechanism.

LISTING 5.7: tf2_keras_callback.py

```
import tensorflow as tf
import numpy as np
```

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(64, activation='relu'))
model.add(tf.keras.layers.Dense(64, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))

model.compile(optimizer=tf.keras.optimizers.Adam(0.01),
              loss='mse',       # mean squared error
              metrics=['mae'])  # mean absolute error

data   = np.random.random((1000, 32))
labels = np.random.random((1000, 10))

val_data   = np.random.random((100, 32))
val_labels = np.random.random((100, 10))

callbacks = [
  # stop training if "val_loss" stops improving for over 2
epochs
  tf.keras.callbacks.EarlyStopping(patience=2,
monitor='val_loss'),
  # write TensorBoard logs to the ./logs directory
  tf.keras.callbacks.TensorBoard(log_dir='./logs')
]

model.fit(data, labels, batch_size=32, epochs=50,
callbacks=callbacks,
          validation_data=(val_data, val_labels))

model.evaluate(data, labels, batch_size=32)
```

Listing 5.7 defines a Keras-based model with three hidden layers and then compiles the model. The next portion of Listing 5.7 uses the `np.random.random` function in order to initialize the variables `data`, `labels`, `val_data`, and `val_labels`.

The interesting code involves the definition of the `callbacks` variable that specifies the `tf.keras.callbacks.EarlyStopping` class with a value of 2 for `patience`, which means that the model will stop training if there is an insufficient reduction in the value of `val_loss`.

The `callbacks` variable includes the `tf.keras.callbacks.TensorBoard` class to specify the `logs` subdirectory as the location for the TensorBoard files.

Next, the `model.fit()` method is invoked with a value of 50 for `epochs` (shown in bold), followed by the `model.evaluate()` method. Launch the code in Listing 5.7 and you will see the following output:

```
Epoch 1/50
1000/1000 [==============================] - 0s 354us/
sample - loss: 0.2452 - mae: 0.4127 - val_loss: 0.2517 -
val_mae: 0.4205
Epoch 2/50
```

```
1000/1000 [==============================] - 0s 63us/sample
- loss: 0.2447 - mae: 0.4125 - val_loss: 0.2515 - val_mae:
0.4204
Epoch 3/50
1000/1000 [==============================] - 0s 63us/sample
- loss: 0.2445 - mae: 0.4124 - val_loss: 0.2520 - val_mae:
0.4209
Epoch 4/50
1000/1000 [==============================] - 0s 68us/sample
- loss: 0.2444 - mae: 0.4123 - val_loss: 0.2519 - val_mae:
0.4205
1000/1000 [==============================] - 0s 37us/sample
- loss: 0.2437 - mae: 0.4119
(1000, 10)
```

Notice that the code stopped training after four epochs, even though fifty epochs are specified in the code.

KERAS AND EARLY STOPPING (2)

The previous section contains a code sample with minimalistic functionality with respect to the use of callback functions in Keras. However, you can also define a custom class that provides finer-grained functionality that uses a callback mechanism.

Listing 5.8 displays the contents of `tf2_keras_callback2.py`, which performs early stopping via a callback mechanism (the new code is shown in bold).

LISTING 5.8: tf2_keras_callback2.py

```
import tensorflow as tf
import numpy as np

model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(64, activation='relu'))
model.add(tf.keras.layers.Dense(64, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))

model.compile(optimizer=tf.keras.optimizers.Adam(0.01),
              loss='mse',       # mean squared error
              metrics=['mae'])  # mean absolute error

data   = np.random.random((1000, 32))
labels = np.random.random((1000, 10))

val_data   = np.random.random((100, 32))
val_labels = np.random.random((100, 10))

class MyCallback(tf.keras.callbacks.Callback):
  def on_train_begin(self, logs={}):
    print("on_train_begin")
  def on_train_end(self, logs={}):
    print("on_train_begin")
    return

  def on_epoch_begin(self, epoch, logs={}):
    print("on_train_begin")
    return

  def on_epoch_end(self, epoch, logs={}):
    print("on_epoch_end")
    return

  def on_batch_begin(self, batch, logs={}):
    print("on_batch_begin")
    return

  def on_batch_end(self, batch, logs={}):
    print("on_batch_end")
    return

callbacks = [MyCallback()]

model.fit(data, labels, batch_size=32, epochs=50,
callbacks=callbacks,
          validation_data=(val_data, val_labels))

model.evaluate(data, labels, batch_size=32)
```

The new code in Listing 5.8 that differs from Listing 5.7 is limited to the code block that is displayed in bold. This new code defines a custom Python class with several methods, each of which is invoked during the appropriate point during the Keras life cycle execution. The six methods consists of three pairs of methods for the start event and end event associated with training, epochs, and batches, as listed here:

- `on_train_begin()`
- `on_train_end()`
- `on_epoch_begin()`
- `on_epoch_end()`
- `on_batch_begin()`
- `on_batch_end()`

The preceding methods contain just a `print()` statement in Listing 5.8, and you can insert any code you wish in any of these methods. Launch the code in Listing 5.8 and you will see the following output:

```
on_train_begin
on_train_begin
Epoch 1/50
on_batch_begin
```

```
on_batch_end
  32/1000 [..............................] - ETA: 4s -
loss: 0.2489 - mae: 0.4170on_batch_begin
on_batch_end
on_batch_begin on_batch_end
// details omitted for brevity
on_batch_begin
on_batch_end
on_batch_begin
on_batch_end
992/1000 [============================>.] - ETA: 0s - loss:
0.2468 - mae: 0.4138on_batch_begin
on_batch_end
on_epoch_end
1000/1000 [==============================] - 0s 335us/
sample - loss: 0.2466 - mae: 0.4136 - val_loss: 0.2445 -
val_mae: 0.4126
on_train_begin
Epoch 2/50
on_batch_begin
on_batch_end
 32/1000 [..............................] - ETA: 0s - loss:
0.2465 - mae: 0.4133on_batch_begin
on_batch_end
on_batch_begin
on_batch_end
// details omitted for brevity
on_batch_end
on_epoch_end
1000/1000 [==============================] - 0s 51us/sample
- loss: 0.2328 - mae: 0.4084 - val_loss: 0.2579 - val_mae:
0.4241
on_train_begin
 32/1000 [..............................] - ETA: 0s - loss:
0.2295 - mae: 0.4030
1000/1000 [==============================] - 0s 22us/sample
- loss: 0.2313 - mae: 0.4077
(1000, 10)
```

KERAS AND METRICS

Many Keras-based models only specify `accuracy` as the metric for evaluating a trained model, as shown here:

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

However, there are many other built-in metrics available, each of which is encapsulated in a Keras class in the `tf.keras.metrics` namespace. A list of many such metrics is displayed in the following list:

- class Accuracy: how often predictions match labels
- class BinaryAccuracy: how often predictions match labels
- class CategoricalAccuracy: how often predictions match labels
- class FalseNegatives: the number of false negatives
- class FalsePositives: the number of false positives
- class Mean: the (weighted) mean of the given values
- class Precision: the precision of the predictions wrt the labels
- class Recall: the recall of the predictions wrt the labels
- class TrueNegatives: the number of true negatives
- class TruePositives: the number of true positives

Earlier in this chapter you learned about the "confusion matrix" that provides numeric values for TP, TN, FP, and FN; each of these values has a corresponding Keras class `TruePositive`, `TrueNegative`, `FalsePositive`, and `FalseNegative`, respectively. Perform an online search for code samples that use the metrics in the preceding list.

DISTRIBUTED TRAINING IN TF 2 (OPTIONAL)

The TF 2 API `tf.distribute.Strategy` enables you to distribute the training of a model across multiple GPUs and TPUs, as well as multiple machines with minimal code changes to existing models. This TF 2 API is easy to use, with good performance and support for multiple strategies (discussed later). Moreover, this TF 2 API works with `tf.keras` and `tf.estimator` along with minor code changes. The TF 2 API `tf.distribute.Strategy` supports various specialized strategies, as shown in the following:

- MirroredStrategy
- MultiWorkerMirroredStrategy
- TPUStrategy
- ParameterServerStrategy

The TF 2 `tf.distribute.MirrorStrategy` supports synchronous distributed training on multiple GPUs on one machine. It creates one replica per GPU device. Each variable in the model is mirrored across all the replicas. Together, these variables form a single conceptual variable called `MirroredVariable`. These variables are kept in sync with each other by applying identical updates.

The TF 2 `tf.distribute.experimental.MultiWorkerMirroredStrategy` is very similar to `MirroredStrategy`. This strategy implements synchronous distributed training across multiple workers, each with potentially multiple GPUs. In addition, this strategy creates copies of all variables in the model on each device across all workers, which is similar to `MirroredStrategy`.

The TF 2 `tf.distribute.experimental.TPUStrategy` lets users run their TensorFlow training on TPUs, which are available on Google

Colab, the TensorFlow Research Cloud, and Google Compute Engine. `TPUStrategy` has the same synchronous distributed training architecture as `MirroredStrategy`. TPUs provide their own implementation of efficient all-reduce and other collective operations across multiple TPU cores, which are used in `TPUStrategy`.

The TF 2 `tf.distribute.experimental.ParameterServerStrategy` supports parameter servers training. This strategy is suitable for multi-GPU synchronous local training or for asynchronous multi-machine training. When used to train locally on one machine, variables are not mirrored; instead, they are placed on the CPU and operations are replicated across all local GPUs. In a multi-machine setting, machines are designated as workers or as parameter servers. Each variable of the model is placed on one parameter server. Computation is replicated across all GPUs of the all the workers.

Using `tf.distribute.Strategy` with Keras

The `tf.distribute.Strategy` strategy is integrated into `tf.keras`, which means that it's seamless for Keras users to distribute their training written in Keras-based code. However, there are two code changes required: first, create an instance of the appropriate `tf.distribute.Strategy`, and second, move the creation and compiling of the Keras model inside `strategy.scope`. The following code block illustrates how to do so with a Keras model that contains one dense layer:

```
mirrored_strategy = tf.distribute.MirroredStrategy()

with mirrored_strategy.scope():
  model = tf.keras.Sequential([tf.keras.layers.Dense(1,
                                  input_shape=(1,))])
                              model.compile(loss='mse',optimi
zer='sgd')
```

For more details regarding distributed training in TF 2, navigate to this website:

https://www.tensorflow.org/guide/distribute_strategy

SUMMARY

This chapter started with an explanation of classification and classifiers, followed by a brief explanation of commonly used classifiers in machine learning. Next you saw a list of the TF 2 APIs for various activation functions, followed by a description of some of their merits.

You also learned about logistic regression that involves the sigmoid activation function, followed by a Keras-based code sample involving logistic regression. Then you saw an example of early stopping in Keras, followed by a very brief description of the classes in the `tf.keras.metrics` namespace.

Finally, you learned about the TF 2 support for distributed training, and a brief description of the available strategies.

TF 2, KERAS, AND ADVANCED TOPICS

This appendix briefly discusses an assortment of topics, such as NLP (natural language processing), `MLP`s (multilayer perceptrons), `CNN`s (convolutional neural networks), `RNN`s (recurrent neural networks), `LSTM`s (long short term memory), reinforcement learning, and deep reinforcement learning. Most of this appendix contains descriptive content, along with some Keras-based code samples that assume you have read the Keras material in the previous chapters. *This appendix is meant to be a cursory introduction to a diverse set of topics, along with suitable links to additional information.*

If you are new to deep learning, many topics in this appendix (such as `LSTM`s) will require additional study in order for you to become comfortable with them. Nevertheless, there's still value in learning about topics that are new to you: think of this appendix as a modest step toward your mastery of deep learning.

The first portion of this appendix briefly discusses deep learning, the problems it can solve, and the challenges for the future. The second part of this appendix briefly introduces the perceptron, which is essentially a "core building block" for neural networks. In fact ANNs, `MLP`s, `RNN`s, `LSTM`s, and `VAE`s are all based on multiple layers that contain multiple perceptrons.

The third part of this appendix provides an introduction to `CNN`s, followed by an example of training a Keras-based `CNN` with the `MNIST` dataset: this code sample will make more sense if you have read the section pertaining to activation functions in Chapter 5.

The fourth part of this appendix discusses the architectures of `RNNs`, `LSTM`s, `GRU`s, autoencoders, and `GAN`s. The final section of this appendix discusses reinforcement learning, the TF-Agents toolkit from Google, a short introduction to deep reinforcement learning, and also the Google Dopamine toolkit.

Again, please read the sections in Chapter 4 and Chapter 5 pertaining to the Keras material in order to derive greater benefit from the code samples in this appendix.

WHAT IS DEEP LEARNING?

Deep learning is a subset of machine learning, and it includes model architectures known as `CNNs`, `RNNs`, `LSTMs`, `GRUs`, variational autoencoders (`VAEs`), and `GANs`. A deep learning model requires at least two hidden layers in a neural network ("very deep learning" involves neural networks with at least ten hidden layers).

From a high-level viewpoint, deep learning with supervised learning involves defining a model (aka neural network) as well as:

- making an estimate for a datapoint
- calculating the loss or error of each estimate
- reducing the error via gradient descent

In Chapter 4, you learned about linear regression in the context of machine learning, which starts with initial values for `m` and `b`:

```
m = tf.Variable(0.)
b = tf.Variable(0.)
```

The training process involves finding the optimal values for `m` and `b` in the following equation:

```
y = m*x + b
```

We want to calculate the dependent variable `y` given a value for the independent variable `x`. In this case, the calculation is handled by the following Python function:

```
def predict(x):
  y = m*x + b
  return y
```

The loss is another name for the error of the current estimate, which can be calculated via the following Python function that determines the MSE value:

```
def squared_error(y_pred, y_actual):
  return tf.reduce_mean(tf.square(y_pred-y_actual))
```

We also need to initialize variables for the training data (often named `x_train` and `y_train`) and the test-related data (often named `x_test` and `x_test`), which is typically an 80/20 or 75/25 "split" between training data and test data. Then the training process invokes the preceding Python functions in the following manner:

```
loss = squared_error(predict(x_train), y_train)
print("Loss:", loss.numpy())
```

Although the Python functions in this section are simple, they can be generalized to handle complex models, such as the models that are described later in this appendix.

You can also solve linear regression via deep learning, which involves the same code that you saw earlier in this section.

What Are Hyperparameters?

Deep learning involves *hyperparameters*, which are sort of like knobs and dials whose values are initialized by you prior to the actual training process. For instance, the number of hidden layers and the number of neurons in hidden layers are examples of hyperparameters. You will encounter many hyperparameters in deep learning models, some of which are listed here:

- Number of hidden layers
- Number of neurons in hidden layers
- Weight initialization
- An activation function
- A cost function
- An optimizer
- A learning rate
- A dropout rate

The first three hyperparameters in the preceding list are required for the initial setup of a neural network. The fourth hyperparameter is required for forward propagation. The next three hyperparameters (i.e., the cost function, optimizer, and learning rate) are required in order to perform backward error propagation (aka backprop) during supervised learning tasks. This step calculates a set of numbers that is used to update the values of the weights in the neural network in order to improve the accuracy of the neural network. The final hyperparameter is useful if you need to reduce overfitting in your model. In general, the cost function is the most complex of all these hyperparameters.

During back propagation, *the vanishing gradient* problem can occur, after which some weights are no longer updated, in which case the neural network is essentially inert (and debugging this problem is generally nontrivial). Another consideration: deciding whether or not a local minima is "good enough" and preferable to expending the additional time and effort that is required to find an absolute minima.

Deep Learning Architectures

As discussed previously, deep learning supports various architectures, including `ANNs, CNNs, RNNs, and LSTMs.` Although there is overlap in terms of the types of tasks that these architectures can solve, each one has a specific reason for its creation. As you progress from `MLPs` to `LSTMs`, the architectures become more complex. Sometimes combinations of these architectures are well-suited for solving tasks. For example, capturing video and making predictions typically involves a `CNN` (for processing each input image in a video sequence) and an `LSTM` (to make predictions of the position of objects that are in the video stream).

In addition, neural networks for NLP can contain one or more `CNN`s, `RNN`s, `LSTM`s, and bi`LSTM`s (bidirectional `LSTM`s). In particular, the combination of reinforcement learning with these architectures is called deep reinforcement learning.

Although `MLP`s have been popular for a long time, they suffer from two disadvantages: they are not scalable for computer vision tasks, and they are somewhat difficult to train. On the other hand, `CNN`s do not require adjacent layers to be fully connected. Another advantage of `CNN`s is something called "translation invariance," which means that an image (such as a digit, cat, dog, and so forth) is recognized as such, regardless of where it appears in a bitmap.

Problems That Deep Learning Can Solve

As you know, back propagation involves updating the weights of the edges between consecutive layers, which is performed in a right-to-left fashion (i.e., from the output layer toward the input layer). The updates involve the chain rule (a rule for computing derivatives) and an arithmetic product of parameters and gradient values. There are two anomalous results that can occur: the product of terms approaches zero (which is called the "vanishing gradient" problem) or the product of terms becomes arbitrarily large (which is called the "exploding gradient" problem). These problems arise with the sigmoid activation function.

Deep learning can mitigate both of these problems via `LSTM`s. Keep in mind that CNN models replace the sigmoid activation function with the `ReLU` activation function. `ReLU` is a very simple continuous function that is differentiable (with a value of 1 to the right of the y-axis and a value of -1 to the left of the y-axis) everywhere except the origin. Hence, it's necessary to perform some "tweaking" to make things work nicely at the origin.

Challenges in Deep Learning

Although deep learning is powerful and has produced impressive results in many fields, there are some important ongoing challenges that are being explored, including:

- Bias in algorithms
- Susceptibility to adversarial attacks
- Limited ability to generalize
- Lack of explainability
- Lack of causality

Algorithms can contain unintentional bias, and even if the bias is removed, there can be unintentional bias in data. For example, one neural network was trained on a dataset containing pictures of Caucasian males and females. The outcome of the training process "determined" that males were physicians and that females were housewives (and did so with a high probability). The reason was simple: the dataset depicted males and females almost exclusively in those two roles. The following article contains more information regarding bias in algorithms:

https://www.technologyreview.com/s/612876/this-is-how-ai-bias-really-happensand-why-its-so-hard-to-fix

Deep learning focuses on finding patterns in datasets, and generalizing those results is a more difficult task. There are some initiatives that attempt to provide explainability for the outcomes of neural networks, but such work is still in its infancy. Deep learning finds patterns and can determine correlation, but it's incapable of determining causality.

Now that you have a bird's-eye view of deep learning, let's rewind and discuss an important cornerstone of neural networks called the perceptron, which is the topic of the next section.

WHAT ARE PERCEPTRONS?

Recall from Chapter 4 that a model for linear regression involves an output layer that contains a single neuron, whereas a multi-neuron output layer is for classifiers (discussed in Chapter 5). `DNNs` (deep neural networks) contain at least two hidden layers, and they can solve regression problems as well as classification problems. In fact, the output layer of a model for classification problems actually consists of a set of probabilities (one for each class in the dataset) whose sum equals 1.

Figure A.1 displays a perceptron with incoming edges that have numeric weights.

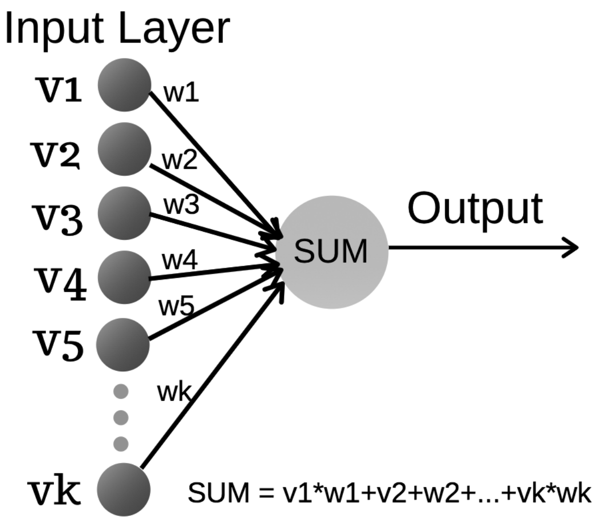

FIGURE A.1. An example of a perceptron.

Image adapted from Arunava Chakraborty, source: *https://towardsdatascience.com/the-perceptron-3af34c84838c*

The next section delves into the details of perceptrons and how they form the backbone of MLPs.

Definition of the Perceptron Function

A Perceptron involves a function f(x) where the following holds:

```
f(x) = 1 if w*x + b > 0 (otherwise f(x) = 0)
```

In the previous expression, w is a vector of weights, x is an input vector, and b is a vector of biases. The product w*x is the inner product of the vectors w and x, and activating a Perceptron is an all-or-nothing decision (e.g., a light bulb is either on or off, with no intermediate states).

Notice that the function f(x) checks the value of the linear term w*x+b, which is also specified in the sigmoid function for logistic regression. The same term appears as part of the calculation of the sigmoid value, as shown here:

```
1/[1 + e^(w*x+b)]
```

Given a value for w*x+b, the preceding expression generates a numeric value. However, in the general case, W is a weight matrix, and x and b are vectors.

The next section digresses slightly in order to describe artificial neural networks, after which we'll discuss MLPs.

A Detailed View of a Perceptron

A neuron is essentially a "building block" for neural networks. In general, each neuron receives multiple inputs (which are numeric values), each of which is from a neuron that belongs to a previous layer in a neural network. The weighted sum of the inputs is calculated and assigned to the neuron.

Specifically, suppose that a neuron N' (N "prime") receives inputs whose weights are in the set {w1, w2, w3, . . . , wn}, where these numbers specify the weights of the edges that are connected to neuron N'. Since forward propagation involves a flow of data in a left-to-right fashion, this means that the left endpoints of the edges are connected to neurons {N1, N2, . . ., Nk} in a preceding layer, and the right endpoint of all these edges is N'. The weighted sum is calculated as follows:

```
x1*w1 + x2*w2 + . . . + xn*wn
```

After the weighted sum is calculated, it's "passed" to an activation function that calculates a second value. This step is required for artificial neural networks, and it's explained later in the chapter. This process of calculating a weighted sum is repeated for every neuron in a given layer, and then the same process is repeated on the neurons in the next layer of a neural network.

The entire process is called *forward propagation*, which is "complemented" by the *backward error propagation* step (also called "backprop"). During the backward error propagation step, new weight values are calculated for the entire neural network. The combination of forward prop and backward prop is repeated for each data point (e.g., each row of data in a CSV file). The goal is to finish this training process so that the finalized neural network (also called a "model") accurately represents the data in a dataset and can also accurately predict values for the test data. Of course, the "accuracy" of a neural network depends on the dataset in question, and the accuracy can be higher than 99%.

THE ANATOMY OF AN ARTIFICIAL NEURAL NETWORK (ANN)

An ANN consists of an input layer, an output layer, and one or more hidden layers. For each pair of adjacent layers in an ANN, neurons in the left layer are connected with neurons in the right layer via an edge that has a numeric weight. If all neurons in the left layer are connected to all neurons in the right layer, it's called an `MLP` (discussed later).

Keep in mind that the perceptrons in an ANN are "stateless": they do *not* retain any information about previously processed data. Furthermore, an ANN does not contain cycles (hence ANNs are acyclic). By contrast, `RNN`s and `LSTM`s *do* retain state and they do have cycle-like behavior, as you will see later in this chapter.

Incidentally, if you have a mathematics background, you might be tempted to think of an ANN as a set of contiguous bipartite graphs in which data "flows" from the input layer (think "multiple sources") toward the output layer ("the sink"). Unfortunately, this viewpoint doesn't prove useful for understanding ANNs. A better way to understand ANNs is to think of their structure as a combination of the hyperparameters in the following list:

1. the number of hidden layers
2. the number of neurons in each hidden layer
3. the initial weights of edges connecting pairs of neurons
4. the activation function
5. a cost (aka loss) function
6. an optimizer (used with the cost function)
7. the learning rate (a small number)
8. the dropout rate (optional)

Figure A.2 is a very small example of an ANN (there are many variations: this is simply one example).

Since the output layer of the ANN in Figure A.2 contains more than one neuron, we know that it's a model for a classification task.

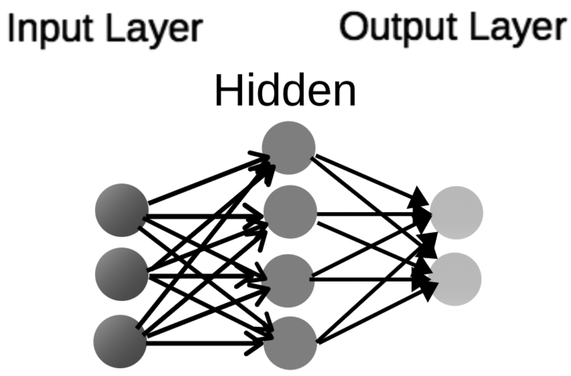

FIGURE A.2. An example of an ANN.

Image adapted from Cburnett, source: *https://commons.wikimedia.org/wiki/File:Artificial_neural_network.svg*

The Model Initialization Hyperparameters

The first three parameters in the list of bullet items in the previous section are required for initializing the neural network. The hidden layers are intermediate computational layers, each of which is composed of neurons. The number of edges between each pair of adjacent layers is flexible and determined by you. More information about network initialization is here:

http://www.deeplearning.ai/ai-notes/initialization/

The edges that connect neurons in each pair of adjacent layers (including the input layer and the output layer) have numeric weights. The initial values of these weights are often small random numbers between 0 and 1. Keep in mind that the connections between adjacent layers can affect the complexity of a model. *The purpose of the training process is to fine-tune edge weights in order to produce an accurate model.*

An ANN is not necessarily fully connected, which is to say that some edges between pairs of neurons in adjacent layers might be missing. By contrast, neural networks such as `CNNs` share edges (and their weights), which can make them more computationally feasible (but even `CNNs` can require significant training time). Note that the Keras `tf.keras.layers.Dense()` class handles the task of fully connecting two adjacent layers. As discussed later, `MLPs` are fully connected, which can greatly increase the training time for such a neural network.

The Activation Hyperparameter

The fourth parameter is the activation function that is applied to weights between each pair of consecutive layers. Neural networks with many layers typically involve different activation functions. For instance, `CNNs` use the `ReLU` activation function on feature maps (created by "applying" filters to an image), whereas the penultimate layer is "connected" to the output layer via the softmax function (which is a generalization of the sigmoid function).

The Cost Function Hyperparameter

The fifth, sixth, and seventh hyperparameters are required for backward error propagation that starts from the output layer and moves right to left toward the input layer. These hyperparameters perform the "heavy lifting" of machine learning frameworks: they compute the updates to the weights of the edges in neural networks.

The *cost function* is a function in multidimensional Euclidean space. For example, the MSE cost function is a bowl-shaped cost function that has a global minimum. In general, the goal is to minimize the MSE function in order to minimize the cost, which in turn will help us maximize the accuracy of a model (but this is not guaranteed for other cost functions). However, sometimes a local minimum might be considered "good enough" instead of finding a global minimum: you must make this decision (i.e., it's not a purely programmatic decision).

Alas, cost functions for larger datasets tend to be very complex, which is necessary in order to detect potential patterns in datasets. Another cost function is the cross-entropy function, which involves maximizing the likelihood function (contrast this with MSE). Search for online articles (such as Wikipedia) for more details about cost functions.

The Optimizer Hyperparameter

An *optimizer* is an algorithm that is chosen in conjunction with a cost function, and its purpose is to converge to the minimum value of the cost function during the training phase (see the comment in the previous section regarding a local minimum). Different optimizers make different assumptions regarding the manner in which new approximations are calculated during the training process. Some optimizers involve only the most recent approximation, whereas other optimizers use a "rolling average" that takes into account several previous approximations.

There are several well-known optimizers, including `SGD`, `RMSprop`, `Adagrad`, `Adadelta`, and `Adam`. Check online for details regarding the advantages and trade-offs of these optimizers.

The Learning Rate Hyperparameter

The *learning rate* is a small number, often between 0.001 and 0.05, which affects the magnitude of the number used to update the current weight of an edge in order to train the model with these updated weights. The learning rate has a sort of "throttling effect." If the value is too large, the new approximation might "overshoot" the optimal point; if it's too small, the training time can increase significantly. By analogy, imagine you are in a passenger jet and you're 100 miles away from an airport. The speed of the airplane decreases as you approach the airport, which corresponds to decreasing the learning rate in a neural network.

The Dropout Rate Hyperparameter

The *dropout rate* is the eighth hyperparameter, which is a decimal value between 0 and 1, typically between 0.2 and 0.5. Multiply this decimal value with 100 to determine the percentage of randomly selected neurons to ignore during each forward pass in the training process. For example, if the dropout rate is 0.2, then 20% of the neurons are selected randomly *and ignored* during the forward propagation process. A different set of neurons is randomly selected whenever a new data point is processed in the neural network. Note that the neurons are not removed from the neural network: they still exist, and ignoring them during forward propagation has the effect of "thinning" the neural network. In TF 2, the Keras `tf.keras.layers.Dropout` class performs the task of "thinning" a neural network.

There are additional hyperparameters that you can specify, but they are optional and not required in order to understand `ANN`s.

What Is Backward Error Propagation?

An ANN is typically drawn in a *left-to-right* fashion, where the leftmost layer is the input layer. The output from each layer becomes the input for the next layer. The term forward propagation refers to supplying values to the input layer and progressing through the hidden layers toward the output layer. The output layer contains the results (which are estimated numeric values) of the forward pass through the model.

Here is a key point: *backward error propagation involves the calculation of numbers that are used to update the weights of the edges in the neural network*. The update process is performed by means of a loss function (and an optimizer and a learning rate), starting from the output layer (the right-most layer) and then moving in a *right-to-left* fashion in order to update the weights of the edges between consecutive layers. This procedure trains the neural network, which involves reducing the loss between the estimated values at the output layer and the true values (in the case of supervised learning). This procedure is repeated for each data point in the training portion of the dataset. Processing the training dataset is called an epoch, and many times a neural network is trained via multiple epochs.

The previous paragraph did not explain what the loss function is or how it's chosen: that's because the loss function, the optimizer, and the learning rate are hyperparameters that are discussed in previous sections. However, two commonly used loss functions are MSE and cross entropy; a commonly used optimizer is the Adam optimizer (and `SGD` and `RMSprop` and others); and a common value for the learning rate is 0.01.

WHAT IS A MULTILAYER PERCEPTRON (MLP)?

A multilayer perceptron (`MLP`) is a feedforward artificial neural network that consists of at least three layers of nodes: an input layer, a hidden layer, and an output layer. An MLP is fully connected: given a pair of adjacent layers, every node in the left layer is connected to every node in the right layer. Apart from the nodes in the input layer, each node is a neuron, and each layer of neurons involves a nonlinear activation function. In addition, MLPs use a technique called *backward error propagation* (or simply "backprop") for training, which is also true for CNNs (convolutional neural networks).

Figure A.3 displays the contents of an MLP with two hidden layers.

One point to keep in mind: the nonlinear activation function of an MLP differentiates an MLP from a linear perceptron. In fact, an MLP can handle data that is not linearly separable. For instance, the OR function and the AND function involve linearly separable data, so they can be represented via a linear perceptron. On the other hand, the XOR function involves data that is not linearly separable, and therefore requires a neural network such as an MLP.

Activation Functions

An MLP without an activation function between any adjacent pair of layers is a linear system: at each layer, simply multiply the vector from the previous layer with the current matrix (which connects the current layer to the next layer) to produce another vector.

On the other hand, it's straightforward to multiply a set of matrices to produce a *single* matrix. Since a neural network without activation functions is a linear system, we can multiply those matrices (one matrix for each pair of adjacent layers) together to produce a single matrix: the original neural network is thereby reduced to a two-layer neural network consisting of an input layer and an output layer, which defeats the purpose of having a multilayered neural network.

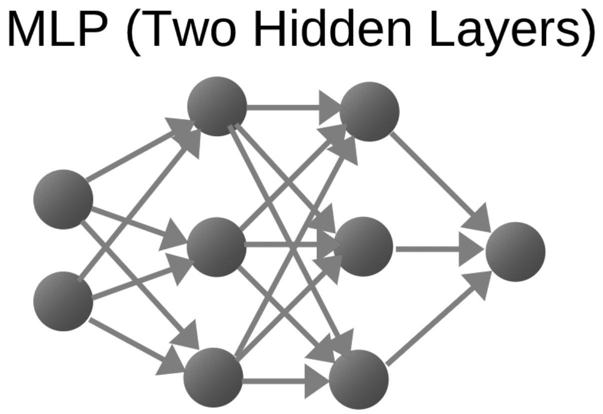

FIGURE A.3. An example of an MLP.

In order to prevent such a reduction of the layers of a neural network, an MLP must include a nonlinear activation function between adjacent layers (this is also true of any other deep neural network). The choice of nonlinear activation function is typically sigmoid, tanh (which is a hyperbolic tangent function), or ReLU (rectified linear unit).

The output of the sigmoid function ranges from 0 to 1, which has the effect of "squashing" the data values. Similarly, the output of the tanh function ranges from -1 to 1. However, the ReLU activation function (or one of its variants) is preferred for ANNs and CNNs, whereas sigmoid and tanh are used in LSTMs.

Several upcoming sections contain the details of constructing an MLP, such as how to initialize the weights of an MLP, storing weights and biases, and how to train a neural network via backward error propagation.

HOW ARE DATA POINTS CORRECTLY CLASSIFIED?

As a point of reference: a "data point" refers to a row of data in a dataset, which can be a dataset for real estate, a dataset of thumbnail images, or some other type of dataset. Suppose that we want to train an MLP for a dataset that contains four classes (aka "labels"). In this scenario, the output layer must also contain four neurons, where the neurons have index values 0, 1, 2, and 3 (a ten-neuron output layer has index values from 0 to 9 inclusive). The sum of the probabilities in the output layer always equals 1 because of the softmax activation function that is used when transitioning from the penultimate layer to the output layer.

The *index* value that has the largest probability in the output layer is compared with the *index* value one-hot encoding of the label of the current data point. If the index values are equal, then the NN has correctly classified the current data point (otherwise, it's a mismatch).

For example, the MNIST dataset contains images of hand-drawn digits from 0 through 9 inclusive, which means that an NN for the MNIST dataset has ten outputs in the final layer, one for each digit. Suppose that an image containing the digit 3 is currently being "passed" through the NN. The one-hot encoding for 3 is [0,0,0,1,0,0,0,0,0,0], and the index value with the largest value in the one-hot encoding is also 3. Now suppose that the output layer of the NN is [0.05,0.05,0.2,0.6,0.02,0.02,0.01,0.01,0.04] after processing the digit 3. As you can see, the index value with the maximum value (which is 0.6) is also 3. In this scenario, the NN has correctly identified the input image. One other point: the TF API tf.argmax() is used to calculate the total number of images that have been correctly labeled by an NN.

A *binary* classifier involves two outcomes for handling tasks such as determining spam/not-spam, fraud/not-fraud, stock increase/decrease (or temperature, or barometric pressure), and so forth. *Predicting the future value* *of a stock price is a regression task, whereas predicting whether the price will increase or decrease is a classification task.*

In machine learning, the multilayer perceptron is an NN for supervised learning of binary classifiers (and it's a type of linear classifier). However, single-layer perceptrons are only capable of learning linearly separable patterns. In fact, a famous book entitled *Perceptrons* by Marvin Minsky and Seymour Papert (written in 1969) showed that it was impossible for these classes of networks to learn an XOR function. However, an XOR function can be "learned" by a two-layer perceptron.

KERAS AND THE XOR FUNCTION

The XOR function is a well-known function that is not linearly separable in the plane. The truth table for the XOR ("exclusive OR") function is straightforward: given two binary inputs, the output is 1 if at most one input is a 1; otherwise, the output is 0. If we treat XOR as the name of a function with two binary inputs, here are the outputs:

```
XOR(0,0) = 0
XOR(1,0) = 1
XOR(0,1) = 1
XOR(1,1) = 0
```

We can treat the output values as labels that are associated with the input values. Specifically, the points (0,0) and (1,1) are in class 0 and the points (1,0) and (0,1) are in class 1. Draw these points in the plane, and you will have the four vertices of a unit square whose lower-left vertex is the origin. Moreover, each pair of diagonal elements belongs to the same class, which makes the XOR function nonlinearly separable in the plane. If you're skeptical, try to find a linear separator for the XOR function in the Euclidean plane.

Listing A.1 displays the contents of tf2_keras_xor.py, which illustrates how to create a Keras-based NN to train the XOR function.

LISTING A.1: tf2_keras_xor.py

```
import tensorflow as tf
import numpy as np

# Logical XOR operator and "truth" values:
x = np.array([[0., 0.],[0., 1.],[1., 0.],[1., 1.]])
y = np.array([[0.], [1.], [1.], [0.]])

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(2, input_dim=2,
activation='relu'))
model.add(tf.keras.layers.Dense(1))
```

```
print("compiling model...")
model.compile(loss='mean_squared_error', optimizer='adam')
print("fitting model...")
model.fit(x,y,verbose=0,epochs=1000)
pred = model.predict(x)

# Test final prediction
print("Testing XOR operator")
p1 = np.array([[0., 0.]])
p2 = np.array([[0., 1.]])
p3 = np.array([[1., 0.]])
p4 = np.array([[1., 1.]])

print(p1,":", model.predict(p1))
print(p2,":", model.predict(p2))
print(p3,":", model.predict(p3))
print(p4,":", model.predict(p4))
```

Listing A.1 initializes the NumPy array x with four pairs of numbers that are the four combinations of 0 and 1, followed by the NumPy array y that contains the logical OR of each pair of numbers in x.

The next portion of Listing A.1 defines a Keras-based model with two Dense layers. Next, the model is compiled and trained, and then the variable pred is populated with a set of predictions based on the trained model.

The next code block initializes the points p1, p2, p3, and p4 and then displays the values that are predicted for those points. The output from launching the code in Listing A.1 is here:

```
compiling model...
fitting model...
Testing XOR operator
[[0. 0.]] : [[0.36438465]]
[[0. 1.]] : [[1.0067574]]
[[1. 0.]] : [[0.36437267]]
[[1. 1.]] : [[0.15084022]]
```

Experiment with different values for epochs and see how that affects the predictions. Use the code in Listing A.1 as a "template" for other logical functions. The only modification to Listing A.1 that is required is the replacement of the variable y in Listing A.1 with the variable y that is specified as the labels for several other logic gates that are listed as follows.

The labels for the NOR function:

```
y = np.array([[1.], [0.], [0.], [1.]])
```

The labels for the OR function:

```
y = np.array([[0.], [1.], [1.], [1.]])
```

The labels for the XOR function:

```
y = np.array([[0.], [1.], [1.], [0.]])
```

The labels for the ANDR function:

```
y = np.array([[0.], [0.], [0.], [1.]])
mnist = tf.keras.datasets.mnist
```

The preceding code snippets are the only changes that you need to make to Listing A.1 in order to train a model for a different logical function. For your convenience, the companion files contains the following Keras-based code samples for the preceding functions:

```
tf2_keras-nor.py
tf2_keras-or.py
tf2_keras-xor.py
tf2_keras-and.py
```

After you have finished working with the preceding samples, try the NAND function, or create more complex combinations of these basic functions.

A HIGH-LEVEL VIEW OF CNNs

CNNs are deep NNs (with one or more convolutional layers) that are well-suited for image classification, along with other use cases, such as audio and NLP (natural language processing).

Although MLPs were successfully used for image recognition, they do not scale well because every pair of adjacent layers is fully connected, which in turn can result in massive neural networks. For large images (or other large inputs) the complexity becomes significant and adversely affects performance.

Figure A.4 displays the contents of a CNN (there are many variations: this is simply one example).

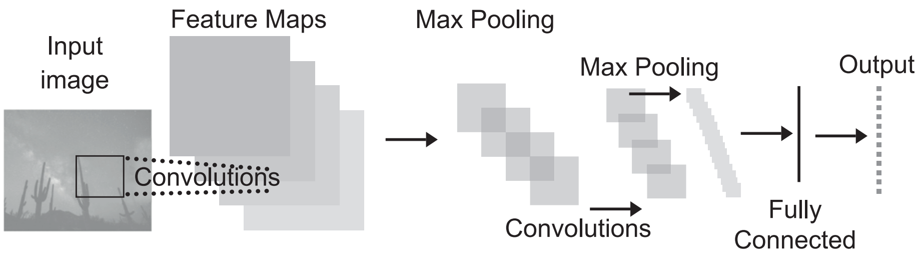

FIGURE A.4. An example of a CNN.

Adapted from source: *https://commons.wikimedia.org/w/index.php?curid=45679374*

A Minimalistic CNN

A production quality CNN can be very complex, comprising many hidden layers. However, in this section we're going to look at a minimalistic CNN (essentially a "toy" neural network), which consists of the following layers:

- `Conv2D` (a convolutional layer)
- `ReLU` (activation function)
- max pooling (reduction technique)
- fully connected (FC) layer
- `Softmax` activation function

The next subsections briefly explain the purpose of each bullet point in the preceding list of items.

The Convolutional Layer (Conv2D)

The convolutional layer is typically labeled as `Conv2D` in Python and TF code. The `Conv2D` layer involves a set of filters, which are small square matrices whose dimensions are often 3x3 but can also be 5x5, 7x7, or even 1x1. Each filter is "scanned across" an image (think of tricorders in *Star Trek* movies), and at each step, an inner product is calculated with the filter and the portion of the image that is currently "underneath" the filter. The result of this scanning process is a "feature map" that contains real numbers.

Figure A.5 displays a 7x7 grid of numbers and the inner product of a 3x3 filter with a 3x3 subregion that results in the number 4 that appears in the feature map.

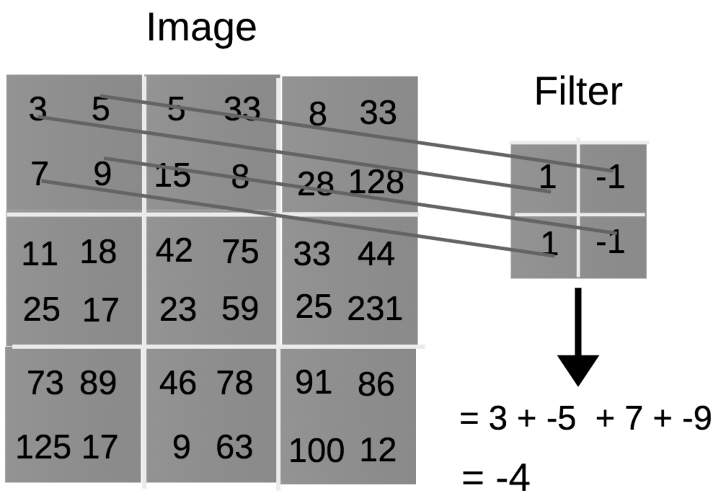

FIGURE A.5. Performing a convolution.

The `ReLU` Activation Function

After each feature map is created, it's possible that some of the values in the feature map are negative. The purpose of the `ReLU` activation function is to replace negative values (if any) with zero. Recall the definition of the `ReLU` function:

```
ReLU(x) = x if x >=0 and ReLU(x) = 0 if x < 0
```

If you draw a 2D graph of `ReLU`, it consists of two parts: the horizontal axis for x less than zero and the identity function (which is a line) in the first quadrant for x greater than or equal to 0.

The Max Pooling Layer

The third step involves "max pooling," which is actually simple to perform: after processing the feature map with the `ReLU` activation function in the previous step, partition the updated feature map into 2x2 rectangles, and select the largest value from each of those rectangles. The result is a smaller array that contains 25% of the values of the feature map (i.e., 75% of the numbers are discarded). There are several algorithms that you can use to perform this extraction: the average of the numbers in each square; the square root of the sum of the squares of the numbers in each square; or the maximum number in each square.

In the case of `CNN`s, the algorithm for max pooling selects the maximum number from each 2x2 rectangle. Figure A.6 displays the result of max pooling in a `CNN`.

As you can see, the result is a small square array whose size is only 25% of the previous feature map. This sequence is performed for each filter in the set of filters that were chosen in the `Conv2D` layer. This set can have 8, 16, 32, or more filters.

If you feel puzzled or skeptical about this technique, consider the analogy involving compression algorithms, which can be divided into two types: lossy and lossless. In case you didn't already know, `JPEG` is a lossy algorithm (i.e.,

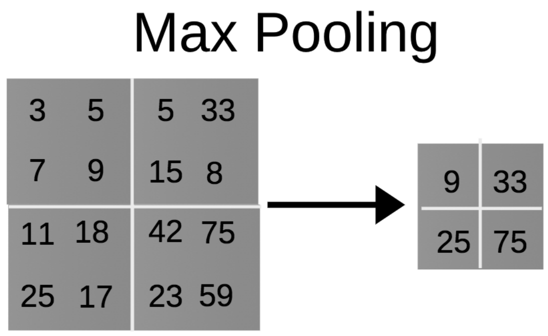

FIGURE A.6. An example of max pooling in a `CNN`.

data is lost during the compression process), and yet it works just fine for compressing images. Think of max pooling as the counterpart to lossy compression algorithms, and perhaps that will persuade you of the efficacy of this algorithm.

At the same time, your skepticism is valid. In fact, Geoffrey Hinton (often called the "godfather" of deep learning) proposed a replacement for max pooling called "capsule networks." This architecture is more complex and more difficult to train, and is also beyond the scope of this book (you can find online tutorials that discuss capsule networks in detail). However, capsule networks tend to be more "resistant" to GANs (Generative Adversarial Networks).

Repeat the previous sequence of steps (as in LeNet), and then perform a rather nonintuitive action: "flatten" all these small arrays so that they are one-dimensional vectors, and concatenate these vectors into one (very long) vector. The resulting vector is then fully connected with the output layer, where the latter consists of ten "buckets." In the case of `MNIST`, these placeholders are for the digits from 0 to 9 inclusive. Note that the Keras `tf.keras.layers.Flatten` class performs this "flattening" process.

The `softmax` activation function is "applied" to the "long vector" of numbers in order to populate the ten "buckets" of the output layer. The result: the ten buckets are populated with a set of nonzero (and nonnegative) numbers whose sum equals one. Find the *index* of the bucket containing the largest number and compare this number with the *index* of the one-hot encoded label associated with the image that was just processed. If the index values are equal, then the image was successfully identified.

More complex `CNN`s involve multiple `Conv2D` layers, multiple `FC` (fully connected) layers, different filter sizes, and techniques for combining previous layers (such as `ResNet`) to "boost" the data values' current layer. Additional information about `CNN`s is here: *https://en.wikipedia.org/wiki/Convolutional_neural_network*

CNNs WITH AUDIO SIGNALS

In addition to image classification, you can train `CNN`s with audio signals, which can be converted from analog to digital. Audio signals have various numeric parameters (such as decibel level and voltage level) that are described here:

https://en.wikipedia.org/wiki/Audio_signal

If you have a set of audio signals, the numeric values of their associated parameters become the dataset for a `CNN`. Remember that `CNN`s have no "understanding" of the numeric input values: the numeric values are processed in the same fashion, regardless of the source of the numeric values.

One use case involves a microphone outside of a building that detects and identifies various sounds. Obviously, it's important to identify the sound of a "backfire" from a vehicle versus the sound of a gunshot. In the latter case, the police would be notified about a potential crime. There are companies that use CNNs to identify different types of sounds; other companies are exploring the use of RNNs and LSTMs instead of CNNs.

CNNs AND NLPs

In the case of NLPs, it's possible to "map" words to numeric values and construct a vector of numeric values from the words in a sentence. Hence, the text in a document can be transformed into a set of numeric vectors (involving various techniques that are not discussed here) in order to create a dataset that's suitable for input to a CNN.

Another option involves the use of RNNs and LSTMs instead of CNNs for NLP-related tasks. A bidirectional architecture is being used successfully in BERT (Bidirectional Encoder Representations from Transformers). The Google AI team developed BERT (open sourced in 2018), and it's considered a breakthrough in its ability to solve NLP problems. The source code is here:

https://github.com/google-research/bert

Now that you have a high-level understanding of CNNs, let's look at a code sample that illustrates an image in the MNIST dataset (and the pixel values of that image), followed by two code samples that use Keras to train a model on the MNIST dataset.

DISPLAYING AN IMAGE IN THE MNIST DATASET

Listing A.2 displays the contents of tf2_keras-mnist_digit.py, which illustrates how to create a neural network in TensorFlow that processes the MNIST dataset.

LISTING A.2: tf2_keras-mnist_digit.py

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist

(X_train, y_train), (X_test, y_test) = mnist.load_data()

print("X_train.shape:",X_train.shape)
print("X_test.shape: ",X_test.shape)

first_img = X_train[0]

# uncomment this line to see the pixel values
#print(first_img)

import matplotlib.pyplot as plt
plt.imshow(first_img, cmap='gray')
plt.show()
```

Listing A.2 starts with some `import` statements and then populates the training data and test data from the `MNIST` dataset. The variable `first_img` is initialized as the first entry in the `X_train` array, which is the first image in the training dataset. The final block of code in Listing A.2 displays the pixel values for the first image. The output from Listing A.2 is here:

```
X_train.shape: (60000, 28, 28)
X_test.shape:  (10000, 28, 28)
```

Figure A.7 displays the contents of the first image in the `MNIST` dataset.

KERAS AND THE `MNIST` DATASET

When you read code samples that contain Keras-based models that use the `MNIST` dataset, the models use a different API in the input layer.

Specifically, a model that is not a `CNN` flattens the input images into a one-dimensional vector via the `tf.keras.layers.Flatten()` API, an example of which is here (see Listing A.3 for details):

```
tf.keras.layers.Flatten(input_shape=(28,28))
```

On the other hand, a `CNN` uses the `tf.keras.layers.Conv2D()` API, an example of which is here (see Listing A.4 for details):

```
tf.keras.layers.Conv2D(32,(3,3),activation='relu',input_
shape=(28,28,1))
```

Listing A.3 displays the contents of `tf2_simple_keras_mnist.py`, which illustrates how to create a Keras-based neural network in TensorFlow that processes the `MNIST` dataset.

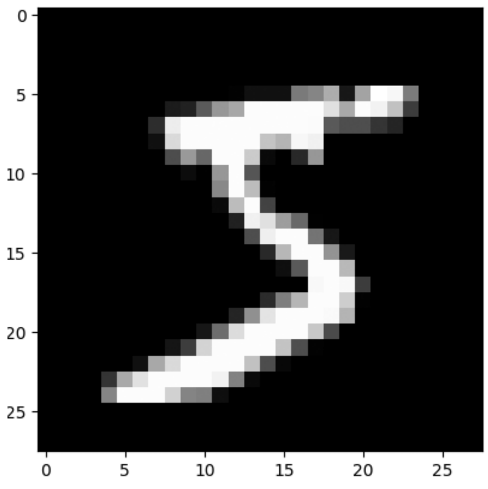

FIGURE A.7. The first image in the MNIST dataset.

LISTING A.3: tf2_simple_keras_mnist.py

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()

x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.summary()

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Listing A.3 starts with some `import` statements and then initializes the variable `mnist` as a reference to the built-in `MNIST` dataset. Next, the training-related and test-related variables are initialized with their respective portions of the `MNIST` dataset, followed by a scaling transformation for `x_train` and `x_test`.

The next portion of Listing A.3 defines a very simple Keras-based model with four layers that are created from classes in the `tf.keras.layers` package. The next code snippet displays a summary of the model definition, as shown here:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)                0
_________________________________________________________________
dense (Dense)                (None, 512)                401920
_________________________________________________________________
dropout (Dropout)            (None, 512)                0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                 5130
=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
```

The remaining portion of Listing A.3 compiles, fits, and evaluates the model, which produces the following output:

```
Epoch 1/5
60000/60000 [==============================] - 14s 225us/
step - loss: 0.2186 - acc: 0.9360
Epoch 2/5
60000/60000 [==============================] - 14s 225us/
step - loss: 0.0958 - acc: 0.9704
Epoch 3/5
60000/60000 [==============================] - 14s 232us/
step - loss: 0.0685 - acc: 0.9783
Epoch 4/5
60000/60000 [==============================] - 14s 227us/
step - loss: 0.0527 - acc: 0.9832
Epoch 5/5
60000/60000 [==============================] - 14s 225us/
step - loss: 0.0426 - acc: 0.9861
10000/10000 [==============================] - 1s 59us/step
```

As you can see, the final accuracy for this model is 98.6%, which is a respectable value.

KERAS, CNNS, AND THE MNIST DATASET

Listing A.4 displays the contents of tf2_cnn_dataset_mnist.py, which illustrates how to create a Keras-based neural network in TensorFlow that processes the MNIST dataset.

LISTING A.4: tf2_cnn_dataset_mnist.py

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

(train_images, train_labels), (test_images, test_labels) =
tf.keras.datasets.mnist.load_data()

train_images = train_images.reshape((60000, 28, 28, 1))
test_images = test_images.reshape((10000, 28, 28, 1))

# Normalize pixel values: from the range 0-255 to the range
0-1
train_images, test_images = train_images/255.0, test_
images/255.0
```

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Conv2D(32, (3, 3),
activation='relu', input_shape=(28, 28, 1)))
model.add(tf.keras.layers.MaxPooling2D((2, 2)))
model.add(tf.keras.layers.Conv2D(64, (3, 3),
activation='relu'))
model.add(tf.keras.layers.MaxPooling2D((2, 2)))
model.add(tf.keras.layers.Conv2D(64, (3, 3),
activation='relu'))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(64, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))

model.summary()

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=1)
test_loss, test_acc = model.evaluate(test_images, test_
labels)
print(test_acc)

# predict the label of one image
test_image = np.expand_dims(test_images[300],axis = 0)
plt.imshow(test_image.reshape(28,28))
plt.show()

result = model.predict(test_image)
print("result:", result)
print("result.argmax():", result.argmax())
```

Listing A.4 initializes the training data and labels, as well as the test data and labels, via the `load_data()` function. Next, the images are reshaped so that they are 28x28 images, and then the pixel values are rescaled from the range 0–255 (all integers) to the range 0–1 (decimal values).

The next portion of Listing A.4 uses the Keras `Sequential()` API to define a Keras-based model called `model`, which contains two pairs of `Conv2D` and `MaxPooling2D` layers, followed by the `Flatten` layer, and then two consecutive `Dense` layers.

Next, the model is compiled, trained, and evaluated via the `compile()`, `fit()`, and `evaluate()` methods, respectively. The final portion of Listing A.4 successfully predicts the image whose label is 4, which is then displayed via `Matplotlib`. Launch the code in Listing A.4 and you will see the following output on the command line:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 3, 3, 64)          36928
_________________________________________________________________
flatten (Flatten)            (None, 576)               0
_________________________________________________________________
dense (Dense)                (None, 64)                36928
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
_________________________________________________________________
60000/60000 [==============================] - 54s 907us/
sample - loss: 0.1452 - accuracy: 0.9563
10000/10000 [==============================] - 3s 297us/
sample - loss: 0.0408 - accuracy: 0.9868
0.9868
Using TensorFlow backend.
result: [[6.2746993e-05 1.7837329e-03 3.8957372e-04
4.6143982e-06 9.9723744e-01
  1.5522403e-06 1.9182076e-04 3.0044283e-04 2.2602901e-05
5.3929521e-06]]
result.argmax(): 4
```

Figure A.8 displays the image that is displayed when you launch the code in Listing A.4.

You might be asking yourself how non-CNN models in machine learning achieve high accuracy when every input image is flattened into a one-dimensional vector, which loses the “adjacency” information that is available in a two-dimensional image. Before CNNs became popular, one technique involved using MLPs and another technique involved SVMs as models for images. In fact, if you don’t have enough images to train a model, you can still use an SVM. Another option is to generate synthetic data using a GAN (which was its original purpose).

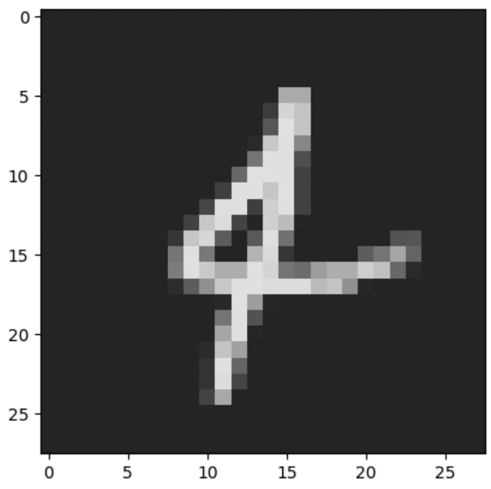

FIGURE A.8. An image in the `MNIST` dataset.

WHAT IS AN RNN?

An `RNN` is a recurrent neural network, which is a type of architecture that was developed during the 1980s. `RNN`s are suitable for datasets that contain sequential data and also for NLP tasks, such as language modeling, text generation, or auto-completion of sentences. In fact, you might be surprised to learn that you can even perform image classification (such as `MNIST`) via an `RNN`. Figure A.9 displays the contents of a simple `RNN`.

In addition to simple `RNNs` there are more powerful constructs such as `LSTMs` and `GRUs`. A basic `RNN` has the simplest type of feedback mechanism (described later), which involves a sigmoid activation function.

`RNN`s (which includes `LSTM`s and `GRU`s) differ from `ANN`s in several important ways, as listed here:

- Statefulness (all RNNs)
- Feedback mechanism (all RNNs)
- A sigmoid or tanh activation function
- Multiple gates (LSTMs and GRUs)
- BPTT (Back Propagation Through Time)
- Truncated BPTT (simple RNNs)

First, `ANN`s and `CNN`s are essentially “stateless,” whereas `RNN`s are “stateful” because they have an internal state. Hence, `RNN`s can process more complex sequences of inputs, which makes them suitable for tasks such as handwriting recognition and speech recognition.

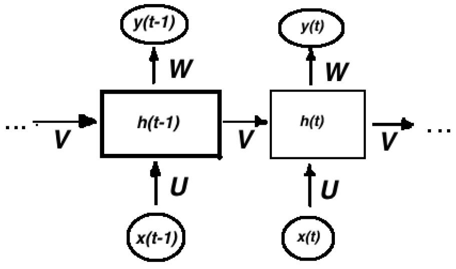

FIGURE A.9. An example of an RNN.

Image adapted from source: *https://commons.wikimedia.org/w/index.php?curid=60109157*

Anatomy of an RNN

Consider the RNN in Figure A.9. Suppose that the sequence of inputs is labeled x1, x2, x3, . . . , x(t), and also that the sequence of "hidden states" is labeled h1, h2, h3, . . . , h(t). Note that each input sequence and hidden state is a 1xn vector, where n is the number of features.

At time period t, the input is based on a combination of h(t-1) and x(t), after which an activation function is "applied" to this combination (which can also involve adding a bias vector).

Another difference is the feedback mechanism for RNNs that occurs between consecutive time periods. Specifically, the output at a *previous* time period is combined with the new input of the *current* time period in order to calculate the new internal state. Let's use the sequence {h(0), h(1), h(2), . . . h(t-1), h(t)} to represent the set of internal states of an RNN during time periods {0, 1, 2, . . . , t-1, t}, and let's also suppose that the sequence {x(0) , x(1), x(2), ... , x(t-1), x(t)} is the inputs during the same time periods.

The fundamental relationship for an RNN at time period t is here:

```
h(t) = f(W*x(t) + U*h(t-1))
```

In the preceding formula, W and U are weight matrices, and f is typically the tanh activation function.

Here is a code snippet of a TF 2 Keras-based model that involves the SimpleRNN class:

```
import tensorflow as tf
...
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.SimpleRNN(5, input_shape=(1,2),
batch_input_shape=[1,1,2], stateful=True))
...
```

Perform an online search for more information and code samples involving Keras and RNNs.

What Is BPTT?

BPTT (back propagation through time) in RNNs is the counterpart to "backprop" for CNNs. The weight matrices of RNNs are updated during BPTT in order to train the neural network.

However, there is a problem called the "exploding gradient" that can occur in RNNs, which is to say that the gradient becomes arbitrarily large (versus the gradient becoming arbitrarily small in the so-called "vanishing gradient" scenario). One way to deal with the exploding gradient problem is to use a "truncated BPTT," which means that BPTT is computed for a small number of steps instead of all time steps. Another technique is to specify a maximum value for the gradient, which involves simple conditional logic.

The good news is that there is another way to overcome both the exploding gradient and vanishing gradient problems, which involves LSTMs that are discussed later in this chapter.

WORKING WITH RNNS AND TF 2

Listing A.5 displays the contents of `tf2_rnn_model.py`, which illustrates how to create a simple Keras-based RNN model.

LISTING A.5: tf2_rnn_model.py

```
import tensorflow as tf

timesteps = 30
input_dim = 12

# number of units in RNN cell
units = 512

# number of classes to be identified
n_activities = 5
model = tf.keras.models.Sequential()

# RNN with dropout:
model.add(tf.keras.layers.SimpleRNN(units=units,
                    dropout=0.2,
                    input_shape=(timesteps, input_dim)))

# one Dense layer:
model.add(tf.keras.layers.Dense(n_activities,
activation='softmax'))

# model loss function and optimizer
model.compile(loss='categorical_crossentropy',
              optimizer=tf.keras.optimizers.Adam(),
              metrics=['accuracy'])

model.summary()
```

Launch the code in Listing A.5 and you will see the following output:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn (SimpleRNN)       (None, 512)               268800
_________________________________________________________________
dense (Dense)                (None, 5)                 2565
=================================================================
Total params: 271,365
Trainable params: 271,365
Non-trainable params: 0
```

There are many variants of RNNs, and you can read about some of them here:

https://en.wikipedia.org/wiki/Recurrent_neural_network

WHAT IS AN LSTM?

LSTMs are a special type of RNN, and they are well-suited for many use cases, including NLP, speech recognition, and handwriting recognition. LSTMs are designed for handling something called "long term dependency," which refers to the distance gap between relevant information and the location where that information is required. This situation arises when information in one section of a document needs to be "linked" to information that is in a more distant location of the document.

LSTMs were developed in 1997 and went on to exceed the accuracy performance of state-of-the-art algorithms. LSTMs also began revolutionizing speech recognition (circa 2007). Then in 2009 an LSTM won pattern recognition contests, and in 2014, Baidu used RNNs to exceed speech recognition records. Navigate to the following link in order to see an example of an LSTM: *https://commons.wikimedia.org/w/index.php?curid=60149410*

Anatomy of an LSTM

LSTMs are "stateful" and they contain three gates (forget gate, input gate, and an output gate) that involve a sigmoid function, and also a cell state that involves the `tanh` activation function. At time period `t` the input to an LSTM is based on a combination of the two vectors `h(t-1)` and `x(t)`. This pair of inputs is combined, after which a sigmoid activation function is "applied" to this combination (which can also include a bias vector) in the case of the forget gate, input gate, and the output gate.

The processing that occurs at time step t is the "short term" memory of an LSTM. The internal cell state of LSTMs maintains "long term" memory. Updating the internal cell state involves the `tanh` activation function, whereas the other gates use the sigmoid activation function, as mentioned in the previous paragraph. Here is a TF 2 code block that defines a Keras-based model for an `LSTM` (with the `LSTM` shown in bold):

```
import tensorflow as tf
. . .
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.LSTMCell(6,batch_input_
shape=(1,1,1),kernel_initializer='ones',stateful=True))
model.add(tf.keras.layers.Dense(1))
. . .
```

You can learn about the difference between an `LSTM` and an `LSTMCell` here:

https://stackoverflow.com/questions/48187283/whats-the-difference-between-lstm-and-lstmcell

In case you're interested, additional information about `LSTM`s and also how to define a custom `LSTM` cell is here:

https://en.wikipedia.org/wiki/Recurrent_neural_network
https://stackoverflow.com/questions/54231440/define-custom-lstm-cell-in-keras

Bidirectional LSTMs

In addition to one-directional LSTMs, you can also define a "bidirectional" `LSTM` that consists of two "regular" `LSTM`s: one `LSTM` for the forward direction and one `LSTM` in the backward or opposite direction. You might be surprised to discover that bidirectional `LSTM`s are well-suited for solving NLP tasks.

For instance, ELMo is a deep word representation for NLP tasks that uses bidirectional LSTMs.

An even newer architecture in the NLP world is called a "transformer,"; bidirectional transformers are used in `BERT,` which is a very well-known system (released by Google in 2018) that can solve solve complex NLP problems.

The following TF 2 code block contains a Keras-based model that involves bidirectional `LSTM`s:

```
import tensorflow as tf
. . .
model = Sequential()
model.add(Bidirectional(LSTM(10, return_sequences=True),
input_shape=(5,10)))
model.add(Bidirectional(LSTM(10)))
model.add(Dense(5))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy',
optimizer='rmsprop')
. . .
```

The previous code block contains two bidirectional `LSTM` cells, both of which are shown in bold.

LSTM Formulas

The formulas for LSTMs are more complex than the update formula for a simple RNN, but there are some patterns that can help you understand those formulas.

Navigate to the following link in order to see the formulas for an LSTM:

https://en.wikipedia.org/wiki/Long_short-term_memory#cite_note-lstm1997-1

The formulas show you how the new weights are calculated for the forget gate f, the input gate i, and the output gate o during time step t. In addition, Figure A.10 shows you how the new internal state and the hidden state (both at time step t) are calculated.

Notice the pattern for gates f, i, and o: all of them calculate the sum of two terms, each of which is a product involving x(t) and h(t), after which the sigmoid function is applied to that sum. Specifically, here's the formula for the *forget gate* at time t:

```
f(t) = sigma(W(f)*x(t) + U(f)*h(t) + b(f))
```

In the preceding formula, W(f), U(f), and b(f) are the weight matrix associated with x(t), the weight matrix associated with h(t), and the bias vector for the forget gate f, respectively.

Notice that the calculations for i(t) and o(t) have the same pattern as the calculation for f(t). The difference is that i(t) has the matrices W(i) and U(i), whereas o(t) has the matrices W(o) and U(o). Thus, f(t), i(t), and o(t) have a "parallel construction."

The calculations for c(t), i(t), and h(t) are based on the values for f(t), i(t), and o(t), as shown here:

```
c(t)  = f(t) * c(t-1) + i(t) * tanh(c'(t))
c'(t) = sigma(W(c) * x(t) + U(c) * h(t-1))
h(t)  = o(t) * tanh(c(t))
```

The final state of an LSTM is a one-dimensional vector that contains the output from all the other layers in the LSTM. If you have a model that contains multiple LSTMs, the final state vector for a given LSTM becomes the input for the next LSTM in that model.

LSTM Hyperparameter Tuning

LSTMs are also prone to overfitting, and here is a list of things to consider if you are manually optimizing hyperparameters for LSTMs:

- overfitting (use regularization such as L1 or L2)
- larger networks are more prone to overfitting
- more data tends to reduce overfitting
- train the networks over multiple epochs
- the learning rate is vitally important

- stacking layers can be helpful
- use softsign instead of softmax for LSTMs
- RMSprop, AdaGrad, or momentum are good choices
- Xavier weight initialization

Perform an online search to obtain more information about the optimizers in the preceding list.

WHAT ARE GRUs?

A `GRU` (gated recurrent unit) is an `RNN` that is a simplified type of `LSTM`. The key difference between a `GRU` and an `LSTM` is that a `GRU` has two gates (reset and update gates) whereas an `LSTM` has three gates (reset, output, and forget gates). The reset gate in a GRU performs the functionality of the input gate and the forget gate of an `LSTM`.

Keep in mind that `GRU`s and `LSTM`s both have the goal of tracking long-term dependencies effectively, and they both address the problem of vanishing gradients and exploding gradients. Navigate to the following link in order to see an example of a `GRU`:

https://commons.wikimedia.org/wiki/File:Gated_Recurrent_Unit,_base_type.svg

Navigate to the following link in order to see the formulas for a `GRU` (which are similar to the formulas for an `LSTM`):

https://en.wikipedia.org/wiki/Gated_recurrent_unit

WHAT ARE AUTOENCODERS?

An autoencoder (AE) is a neural network that is similar to an `MLP`, where the output layer is the same as the input layer. The simplest type of AE contains a single hidden layer that has fewer neurons than either the input layer or the output layer. However, there are many different types of AEs in which there are multiple hidden layers, sometimes containing more neurons than the input layer (and sometimes containing fewer neurons).

An AE uses unsupervised learning and back propagation to learn an efficient data encoding. Their purpose is dimensionality reduction: AEs set the input values equal to the inputs and then try to find the identity function. Figure A.10 displays a simple AE that involves a single hidden layer.

In essence, a basic AE compresses the input to an "intermediate" vector with fewer dimensions than the input data, and then transforms that vector into a tensor with the same shape as the input. Several use cases for AEs are listed as follows:

- document retrieval
- classification
- anomaly detection

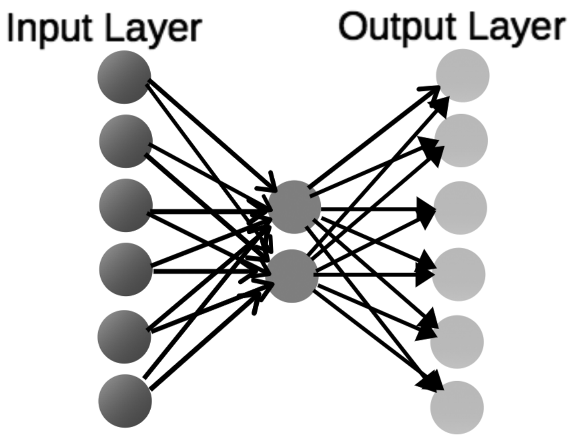

FIGURE A.10. A basic autoencoder.

Image adapted from Philippe Remy, source: *http://philipperemy.github.io/anomaly-detection/*

- adversarial autoencoders
- image denoising (generating clear images)

An example of using TensorFlow and Keras with an autoencoder in order to perform fraud detection is here:

https://www.datascience.com/blog/fraud-detection-with-tensorflow

AEs can also be used for feature extraction because they can yield better results than PCAs. Keep in mind that AEs are data-specific, which means that they only work with similar data. However, they differ from image compression (and are mediocre for data compression). For example, an autoencoder trained on faces would work poorly on pictures of trees. In summary, an AE involves:

- "squeezing" the input to a smaller layer
- learning a representation for a set of data
- typically for dimensionality reduction (PCA)
- keep only the middle "compressed" layer

As a high-level example, consider a 10x10 image (100 pixels) and an AE that has 100 neurons (10x10 pixels), a hidden layer with 50 neurons, and an output layer with 100 neurons. Hence, the AE "compresses" 100 neurons to 50 neurons.

As you saw earlier, there are numerous variations of the basic AE, some of which are listed as follows:

- LSTM autoencoders
- Denoising autoencoders
- Contractive autoencoders
- Sparse autoencoders

- Stacked autoencoders
- Deep autoencoders
- Linear autoencoders

If you're interested, the following link contains a wide assortment of autoencoders, including those that are mentioned in this section:

https://www.google.com/search?sa=X&q=Autoencoder&tbm=isch&source=univ&ved=2ahUKEwjo-8zRrIniAhUGup4KHVgvC10QiR56BAgMEBY&biw=967&bih=672

Perform an online search for code samples and more details regarding AEs and their associated use cases.

Autoencoders and PCA

The optimal solution to an autoencoder is strongly related to principal component analysis (`PCA`) if the autoencoder involves linear activations or only a single sigmoid hidden layer.

The weights of an autoencoder with a single hidden layer of size p (where p is less than the size of the input) span the same vector subspace as the one spanned by the first p principal components.

The output of the autoencoder is an orthogonal projection onto this subspace. The autoencoder weights are not equal to the principal components, and are generally not orthogonal, yet the principal components may be recovered from them using the singular value decomposition.

What Are Variational Autoencoders?

In very brief terms, a variational autoencoder is sort of an enhanced "regular" autoencoder in which the "left side" acts as an encoder, and the right side acts as a decoder. Both sides have a probability distribution associated with the encoding and decoding process.

In addition, both the encoder and the decoder are actually neural networks. The input for the encoder is a vector `x` of numeric values, and its output is a hidden representation `z` that has weights and biases. The decoder has input a (i.e., the output of the encoder), and its output is the parameters of a probability distribution of the data, which also has weights and biases. Note that the probability distributions for the encoder and the decoder are different. If you want to learn more about VAEs, navigate to the Wikipedia page that discusses VAEs in a detailed fashion.

Figure A.11 displays a high-level and simplified VAE that involves a single hidden layer.

Another interesting model architecture is a combination of a `CNN` and a `VAE`, which you can read about here:

https://towardsdatascience.com/gans-vs-autoencoders-comparison-of-deep-generative-models-985cf15936ea

In the next section, you will learn about `GAN`s, and also how to combine a `VAE` with a `GAN`.

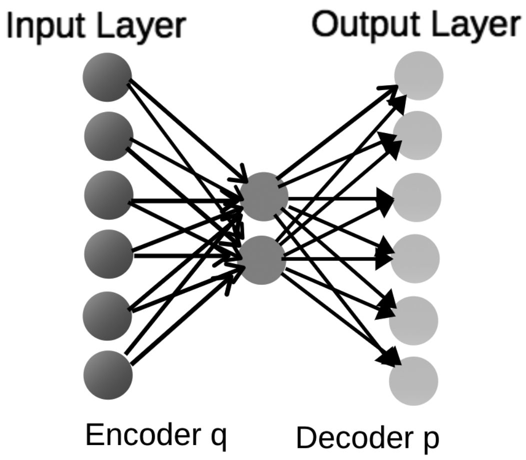

FIGURE A.11. A variational autoencoder.

WHAT ARE GANs?

A GAN is a generative adversarial network, whose original purpose was to generate synthetic data, typically for augmenting small datasets or unbalanced datasets. One use case pertains to missing persons: supply the available images of those persons to a GAN in order to generate an image of how those people might look today. There are many other use cases for GANs, some of which are listed here:

- Generating art
- Creating fashion styles
- Improving images of low quality
- Creating “artificial” faces
- Reconstructing incomplete/damaged images

Ian Goodfellow (PhD in Machine Learning from the University of Montreal) created GANs in 2014. Yann LeCun (AI research director at Facebook) called adversarial training “the most interesting idea in the last 10 years in ML.” Incidentally, Yann LeCun was one of the three recipients of the Turing Award in 2019: Yoshua Bengio, Geoffrey Hinton, and Yann LeCun.

GANs are becoming increasingly common, and people are finding creative (unexpected?) uses for them. Alas, GANs have been used for nefarious purposes, such as circumventing image-recognition systems. GANs can generate “counterfeit” images from valid images by changing the pixel values in order to deceive neural networks. Since those systems rely on pixel patterns, they can be deceived via adversarial images, which are images whose pixel values have been altered.

Navigate to the following link in order to see an example of a GAN that distorts the image of a panda: *https://arxiv.org/pdf/1412.6572.pdf*

An article that delves into details of adversarial examples (including the misclassified panda) is here:

https://openai.com/blog/adversarial-example-research/

According to an MIT paper, the modified values that trigger misclassifications exploit precise patterns that the image system associates with specific objects. The researchers noticed that datasets contain two types of correlations: patterns that are correlated with the dataset data, and non-generalizable patterns in the dataset data. GANs successfully exploit the latter correlations in order to deceive image-recognition systems. Details of the MIT paper are here: *https://gandissect.csail.mit.edu*

Various techniques are being developed to thwart adversarial attacks, but their effectiveness tends to be short-lived: new `GAN`s are created that can outwit those techniques. The following article contains more information about adversarial attacks:

https://www.technologyreview.com/s/613170/emtech-digital-dawn-song-adversarial-machine-learning

Unfortunately, there are no long-term solutions to adversarial attacks, and given their nature, it might never be possible to completely defend against them. Interestingly, `GAN`s can have problems in terms of convergence, just like other neural networks. One technique for addressing this problem is called "minibatch discrimination," details of which are here:

https://www.inference.vc/understanding-minibatch-discrimination-in-gans/

Please note that the preceding link involves Kullback Leibler Divergence and JS Divergence, which are more advanced topics. The preceding blog post also contains a link to the following Jupyter notebook:

https://gist.github.com/fhuszar/a91c7d0672036335c1783d02c3a3dfe5

If you're interested in working with `GAN`s, this GitHub link contains Python and TensorFlow code samples for "constructing attacks and defenses":

https://github.com/tensorflow/cleverhans

The VAE-GAN Model

Another interesting model is the VAE-GAN model, which is a hybrid of a VAE and a `GAN`, and details about this model are here:

https://towardsdatascience.com/gans-vs-autoencoders-comparison-of-deep-generative-models-985cf15936ea

According to the preceding link, `GAN`s are superior to `VAE`s, but they are also difficult to work with and require a lot of data and tuning. If you're interested, a `GAN` tutorial (by the same author) is available here:

https://github.com/mrdragonbear/GAN-Tutorial

WORKING WITH NLP (NATURAL LANGUAGE PROCESSING)

This section highlights some concepts in NLP, and in many cases you need to perform an online search to learn about the meaning of the concepts (try

Wikipedia). Although the concepts are treated in a very superficial manner, you will know what to pursue in order to further your study of NLP.

NLP is currently the focus of significant interest in the machine learning community. Some of the use cases for NLP are listed here:

- Chatbots
- Search (text and audio)
- Text classification
- Sentiment analysis
- Recommendation systems
- Question answering
- Speech recognition
- NLU (natural language understanding)
- NLG (natural language generation)

You encounter many of these use cases in everyday life when you visit web pages, or when you perform an online search for books or recommendations regarding movies.

NLP Techniques

The earliest approach for solving NLP tasks involved rule-based approaches, which dominated the industry for decades. Examples of techniques using rule-based approaches include regular expressions (RegExs) and context free grammars (CFGs). RegExs are sometimes used in order to remove metacharacters from text that has been "scraped" from a web page.

The second approach involved training a machine learning model with some data that was based on some user-defined features. This technique requires a considerable amount of feature engineering (a nontrivial task) and includes analyzing the text to remove undesired and superfluous content (including "stop" words), as well as transforming the words (e.g., converting uppercase to lowercase).

The most recent approach involves deep learning, whereby a neural network learns the features instead of relying on humans to perform feature engineering. One of the key ideas involves "mapping" words to numbers, which enables us to map sentences to vectors of numbers. After transforming documents to vectors, we can perform a myriad of operations on those vectors. For example, we can use the notion of vector spaces to define vector space models, where the distance between two vectors can be measured by the angle between them (this is "cosine similarity"). If two vectors are "close" to each other, then it's likelier that the corresponding sentences are similar in meaning. Their similarity is based on the *distributional hypothesis*: words in the same contexts tend to have similar meanings.

A nice article that discusses vector representations of words, along with links to code samples, is here:

https://www.tensorflow.org/tutorials/representation/word2vec

The Transformer Architecture and NLP

In 2017, Google introduced the `Transformer` neural network architecture, which is based on a self-attention mechanism that is well-suited for language understanding.

Google showed that the `Transformer` outperforms earlier benchmarks for both `RNN`s and `CNN`s involving the translation of academic English to German as well as English to French. Moreover, the `Transformer` required less computation to train, and improved the training time by as much as an order of magnitude.

The `Transformer` can process the sentence "I arrived at the bank after crossing the river" and correctly determine that the word "bank" refers to the shore of a river and not a financial institution. The `Transformer` makes this determination in a single step by making the association between "bank" and "river."

The `Transformer` computes the next representation for a given word by comparing the word to every other word in the sentence, which results in an "attention score" for the words in the sentence. The `Transformer` uses these scores to determine the extent to which other words will contribute to the next representation of a given word.

The result of these comparisons is an attention score for every other word in the sentence. As a result, "river" received a high attention score when computing a new representation for "bank."

Although `LSTM`s and bidirectional `LSTM`s are heavily utilized in `NLP` tasks, the `Transformer` has gained a lot of attention in the AI community, not only for translation between languages, but also the fact that for some tasks it can outperform both `RNN`s and `CNN`s. The `Transformer` architecture requires much less computation time in order to train a model, which explains why some people believe that the `Transformer` will supplant `RNN`s and `LSTM`s.

The following link contains a TF 2 code sample of a `Transformer` neural network that you can launch in Google Colaboratory:

https://www.tensorflow.org/alpha/tutorials/text/transformer

Another interesting and recent architecture is called "attention augmented convolutional networks," which is a combination of `CNN`s with self-attention. This combination achieves better accuracy than "pure" `CNN`s, and you can find more details in this paper: *https://arxiv.org/abs/1904.09925*

Transformer-XL Architecture

The Transformer-XL combines a Transformer architecture with two techniques called recurrence mechanism and relative positional encoding to obtain better results than a Transformer. Transformer-XL works with word-level and character-level language modeling.

The Transformer-XL and Transformer both process the first segment of tokens, and the former also keeps the outputs of the hidden layers. Consequently, each hidden layer receives two inputs from the previous hidden layer, and then concatenates them to provide additional information to the neural network.

NLP and Deep Learning

The NLP models that use deep learning can comprise CNNs, RNNs, LSTMs, and bidirectional LSTMs. For example, Google released BERT in 2018, which is an extremely powerful framework for NLP. BERT is quite sophisticated, and involves bidirectional transformers and so-called "attention" (discussed briefly later in this appendix). Deep learning for NLP often yields higher accuracy than other techniques, but keep in mind that sometimes it's not as fast as rule-based and classical machine learning methods.

In case you're interested, a code sample that uses TensorFlow and RNNs for text classification is here:

https://www.tensorflow.org/alpha/tutorials/text/text_classification_rnn

A code sample that uses TensorFlow and RNNs for text generation is here:

https://www.tensorflow.org/alpha/tutorials/text/text_generation

NLP and Reinforcement Learning

More recently reinforcement learning with NLP has become a successful area of research. One technique for NLP-related tasks involves RNN-based encoder–decoder models that have achieved good results for short input and output sequences. Another technique involves a neural network, supervised word prediction, and reinforcement learning. This particular combination avoids exposure bias, which can occur in models that use only supervised learning. More details are here: *https://arxiv.org/pdf/1705.04304.pdf*

Yet another interesting technique involves deep reinforcement learning (i.e., DL combined with RL) with NLP. In case you don't already know, DRL has achieved success in various areas, such as Atari games, defeating Lee Sedol (the world champion Go player), and robotics. In addition, DRL is also applicable to NLP-related tasks, which involves the key challenge of designing of a suitable model. Perform an online search for more information about solving NLP-related tasks with RL and DRL.

Data Preprocessing Tasks

There are some common preprocessing tasks that are performed on documents, listed as follows:

- [1] lowercasing
- [1] noise removal
- [2] normalization
- [3] text enrichment
- [3] stopword removal
- [3] stemming
- [3] lemmatization

The preceding tasks can be classified as follows:

1. [1]: mandatory tasks
2. [2]: recommended tasks
3. [3]: task dependent

In brief, preprocessing tasks involve at least the removal of redundant words ("a," "the," and so forth), removing the endings of words ("running," "runs," and "ran" are treated the same as "run"), and converting text from uppercase to lowercase.

POPULAR NLP ALGORITHMS

Some of the popular NLP algorithms are listed as follows, and in some cases they are the foundation for more sophisticated NLP toolkits:

- n-grams and skip-grams
- BoW: Bag of Words
- TF-IDF: basic algorithm in extracting keywords
- Word2Vector (Google): O/S project to describe text
- GloVe (Stanford NLP Group)
- LDA: text classification
- CF (collaborative filtering): an algorithm in news recommend system (Google News and Yahoo News)

The topics in the first half of the preceding list are discussed briefly in subsequent sections.

What Is an n-Gram?

An n-gram is a technique for creating a vocabulary that is based on adjacent words that are grouped together. This technique retains some word positions (unlike BoW). You need to specify the value of "n" that in turn specifies the size of the group.

The idea is simple: for each word in a sentence, construct a vocabulary term that contains the n words on the left side of the given word and n words that are on the right side of the given word. As a simple example, "This is a sentence" has the following 2-grams:

```
(this, is), (is, a), (a, sentence)
```

As another example, we can use the same sentence "This is a sentence" to determine its 3-grams:

```
(this, is, a), (is, a, sentence)
```

The notion of n-grams is surprisingly powerful, and it's used heavily in popular open-source toolkits such as `ELMo` and `BERT` when they pretrain their models.

What Is a Skip-Gram?

Given a word in a sentence, a skip-gram creates a vocabulary term by constructing a list that contains the n words on both sides of a given word, followed by the word itself. For example, consider the following sentence:

```
the quick brown fox jumped over the lazy dog
```

A skip-gram of size 1 yields the following vocabulary terms:

```
([the,brown], quick), ([quick,fox], brown),
([brown,jumped], fox),...
```

A skip-gram of size 2 yields the following vocabulary terms:

```
([the,quick,fox,jumped], brown),
([quick,brown,jumped,over], fox), ([brown,fox,over,the],
jumped),...
```

More details regarding skip-grams are discussed here:
https://www.tensorflow.org/tutorials/representation/word2vec#the_skip-gram_model

What Is BoW?

BoW (Bag of Words) assigns a numeric value to each word in a sentence and treats those words as a set (or bag). Hence, BoW does not keep track of adjacent words, so it's a very simple algorithm.

Listing A.6 displays the contents of the Python script `bow_to_vector.py`, which illustrates how to use the BoW algorithm.

LISTING A.6: bow_to_vector.py

```
VOCAB = ['dog', 'cheese', 'cat', 'mouse']
TEXT1 = 'the mouse ate the cheese'
TEXT2 = 'the horse ate the hay'

def to_bow(text):
  words = text.split(" ")
  return [1 if w in words else 0 for w in VOCAB]

print("VOCAB: ",VOCAB)
print("TEXT1:",TEXT1)
print("BOW1: ",to_bow(TEXT1))  # [0, 1, 0, 1]
print("")

print("TEXT2:",TEXT2)
print("BOW2: ",to_bow(TEXT2))  # [0, 0, 0, 0]
```

Listing A.6 initializes a list `VOCAB` and two text strings `TEXT1` and `TEXT2`. The next portion of Listing A.6 defines the Python function `to_bow()` that returns an array containing 0s and 1s: if a word in the current sentence appears in the vocabulary, then a 1 is returned (otherwise a 0 is returned). The last portion of Listing A.6 invokes the Python function with two different sentences. The output from launching the code in Listing A.6 is here:

```
('VOCAB: ', ['dog', 'cheese', 'cat', 'mouse'])
('TEXT1:', 'the mouse ate the cheese')
('BOW1: ', [0, 1, 0, 1])

('TEXT2:', 'the horse ate the hay')
('BOW2: ', [0, 0, 0, 0])
fitting model...
```

What Is Term Frequency?

Term frequency is the number of times that a word appears in a document, which can vary among different documents. Consider the following simple example that consists of two "documents" `Doc1` and `Doc2`:

```
Doc1 = "This is a short sentence"
Doc2 = "yet another short sentence"
```

The term frequency for the word "is" and the word "short" is given as follows:

```
tf(is) = 1/5 for doc1
tf(is) = 0 for doc2
tf(short) = 1/5 for doc1
tf(short) = 1/4 for doc2
```

The preceding values will be used in the calculation of `tf-idf` that is explained in a later section.

What Is Inverse Document Frequency (idf)?

Given a set of N documents and given a word in a document, let's define `dc` and `idf` of each word as follows:

```
dc = # of documents containing a given word
idf = log(N/dc)
```

Now let's use the same two documents `Doc1` and `Doc2` from a previous section:

```
Doc1 = "This is a short sentence"
Doc2 = "yet another short sentence"
```

The calculations of the `idf` value for the word "is" and the word "short" are shown here:

```
idf(is) = log(2/1) = log(2)
idf(short) = log(2/2) = 0
```

The following link provides more detailed information about inverse document frequency: *https://en.wikipedia.org/wiki/Tf–idf#Example_of_tf–idf*

What Is tf-idf?

The term `tf-idf` is an abbreviation for "term frequency, inverse document frequency," and it's the product of the `tf` value and the `idf` value of a word, as shown here:

```
tf-idf = tf * idf
```

A high-frequency word has a higher `tf` value but a lower `idf` value. In general, "rare" words are more relevant than "popular" ones, so they help to extract "relevance." For example, suppose you have a collection of ten documents (real documents, not the toy documents we used earlier). The word "the" occurs frequently in English sentences, but it does not provide any indication of the topics in any of the documents. On the other hand, if you determine that the word "universe" appears multiple times in a single document, this information can provide some indication of the theme of that document, and with the help of NLP techniques, assist in determining the topic (or topics) in that document.

WHAT ARE WORD EMBEDDINGS?

An *embedding* is a fixed-length vector to encode and represent an entity (document, sentence, word, or graph). Each word is represented by a real-valued vector, which can result in hundreds of dimensions. Furthermore, such an encoding can result in sparse vectors: one example is *one-hot encoding*, where one position has the value 1 and all other positions have the value 0.

Three popular word-embedding algorithms are Word2vec, GloVe, and FastText. Keep in mind that these three algorithms involve unsupervised approaches. They are also based on the distributional hypothesis, which asserts that words in the same contexts tend to have similar meanings: *https://aclweb.org/aclwiki/Distributional_Hypothesis*

A good article regarding Word2Vec in TensorFlow is here:

https://towardsdatascience.com/learn-word2vec-by-implementing-it-in-tensorflow-45641adaf2ac

This article is useful if you want to see Word2Vec with FastText in gensim:

https://towardsdatascience.com/word-embedding-with-word2vec-and-fasttext-a209c1d3e12c

Another good article, and this one pertains to the skip-gram model:

https://towardsdatascience.com/word2vec-skip-gram-model-part-1-intuition-78614e4d6e0b

A useful article that describes how FastText works "under the hood":

https://towardsdatascience.com/fasttext-under-the-hood-11efc57b2b3

Along with the preceding popular algorithms, there are also some popular embedding models, some of which are listed as follows:

- Baseline Averaged Sentence Embeddings
- Doc2Vec
- Neural-Net Language Models
- Skip-Thought Vectors
- Quick-Thought Vectors
- InferSent
- Universal Sentence Encoder

Perform an online search for more information about the preceding embedding models.

ELMo, ULMFit, OpenAI, and BERT

During 2018 there were some significant advances in NLP-related research, resulting in the following toolkits and frameworks:

```
• ELMo:    released in 02/2018
• ULMFit:  released in 05/2018
• OpenAI:  released in 06/2018
• BERT:    released in 10/2018
• MT-DNN:  released in 01/2019
```

ELMo is an acronym for “embeddings from language models,” which provides deep contextualized word representations and state-of-the-art contextual word vectors, resulting in noticeable improvements in word embeddings.

Jeremy Howard and Sebastian Ruder created ULMFit (universal language model fine-tuning), which is a transfer learning method that can be applied to any task in NLP. ULMFit significantly outperforms the state of the art on six text classification tasks, reducing the error by 18–24% on the majority of datasets.

Furthermore, with only 100 labeled examples, it matches the performance of training from scratch on 100x more data. ULMFit is downloadable here from GitHub:

https://github.com/jannenev/ulmfit-language-model

OpenAI developed GPT-2 (a successor to GPT), which is a model that was trained to predict the next word in 40GB of Internet text. OpenAI chose not to release the trained model due to concerns regarding malicious applications of their technology.

GPT-2 is a large transformer-based language model with 1.5 billion parameters, trained on a dataset of 8 million web pages (curated by humans), with an emphasis on diversity of content. GPT-2 is trained to predict the next word, given all of the previous words within some text. The diversity of the dataset causes this goal to contain naturally occurring demonstrations of many tasks across diverse domains. GPT-2 is a direct scale-up of GPT, with more than 10X the parameters and trained on more than 10X the amount of data.

BERT is an acronym for "bidirectional encoder representations from transformers." BERT can pass this simple English test (i.e., BERT can determine the correct choice among multiple choices):

```
On stage, a woman takes a seat at the piano. She:
a) sits on a bench as her sister plays with the doll.
b) smiles with someone as the music plays.
c) is in the crowd, watching the dancers.
d) nervously sets her fingers on the keys.
```

Details of BERT and this English test are here:

https://www.lyrn.ai/2018/11/07/explained-bert-state-of-the-art-language-model-for-nlp/

The BERT (TensorFlow) source code is available here on GitHub:

https://github.com/google-research/bert

https://github.com/hanxiao/bert-as-service

Another interesting development is MT-DNN from Microsoft, which asserts that MT-DNN can outperform Google BERT:

https://medium.com/syncedreview/microsofts-new-mt-dnn-outperforms-google-bert-b5fa15b1a03e

A Jupyter notebook with BERT is available, and you need the following in order to run the notebook in Google Colaboratory:

a GCP (Google Compute Engine) account
a GCS (Google Cloud Storage) bucket

Here is the link to the notebook in Google Colaboratory:

https://colab.research.google.com/github/tensorflow/tpu/blob/master/tools/colab/bert_finetuning_with_cloud_tpus.ipynb

WHAT IS TRANSLATOTRON?

Translatotron is an end-to-end speech-to-speech translation model (from Google) whose output retains the original speaker's voice; moreover, it's trained with less data.

Speech-to-speech translation systems have been developed over the past several decades with the goal of helping people who speak different languages to communicate with each other. Such systems have three parts:

- automatic speech recognition to transcribe the source speech as text
- machine translation to translate the transcribed text into the target language
- text-to-speech synthesis (TTS) to generate speech in the target language from the translated text

The preceding approach has been successful in commercial products (including Google Translate). However, Translatotron does not require separate stages, resulting in the following advantages:

- faster inference speed
- avoiding compounding errors between recognition and translation
- easier to retain the voice of the original speaker after translation
- better handling of untranslated words (names and proper nouns)

This concludes the portion of the appendix that pertains to NLP. Another area of great interest in the AI community is reinforcement learning, which is introduced in the next section.

WHAT IS REINFORCEMENT LEARNING (RL)?

Reinforcement learning is a subset of machine learning that attempts to find the maximum reward for a so-called "agent" that interacts with an "environment." RL is suitable for solving tasks that involve deferred rewards .

In fact, RL can handle tasks that involve a combination of negative, zero, and positive rewards. For example, if you decide to leave your job in order to attend school on a full-time basis, you are spending money (a negative reward) with the belief that your investment of time and money will lead to a higher paying position (a positive reward) that outweighs the cost of school and lost earnings.

One thing that might surprise you is that reinforcement learning agents are susceptible to `GAN`s. More details (along with related links) are in this article:

https://openai.com/blog/adversarial-example-research/

There are many RL applications, some of which are listed here:

- game theory
- control theory
- operations research
- information theory
- simulation-based optimization
- multi-agent systems
- swarm intelligence
- statistics and genetic algorithms
- resources management in computer clusters
- traffic light control (congestion problems)
- robotics operations
- autonomous cars/helicopters
- web system configuration/web-page indexing
- personalized recommendations
- bidding and advertising
- robot legged locomotion

- marketing strategy selection
- factory control

RL refers to goal-oriented algorithms for reaching a complex goal, such as winning games that involve multiple moves (e.g., chess or Go). RL algorithms are penalized for incorrect decisions and rewarded for correct decisions: this reward mechanism is reinforcement.

There are three main approaches in reinforcement learning. *Value-based* RL estimates the optimal value function `Q(s,a)`, which is the maximum value achievable under any policy. *Policy-based* RL searches directly for the optimal policy ϖ, which is the policy achieving maximum future reward. *Model-based* RL builds a model of the environment and plans (by lookahead) using the model.

In addition to the preceding approaches to RL (value functions, policies, and models), you will need to learn the following RL concepts:

- MDPs (Markov decision processes)
- A policy (a sequence of actions)
- The state/value function
- The action/value function
- Bellman equation (for calculating rewards)

The RL material in this appendix only addresses the following list of topics (after which you can learn the concepts in the previous list):

- NFAs (nondeterministic finite automata)
- Markov chains
- MDPs (Markov decision processes)
- Epsilon-greedy Algorithm
- Bellman equation

Another key point: *almost all RL problems can be formulated as Markov Decision Processes*, which in turn are based on Markov chains. Let's take a look at `NFA`s and Markov chains and then we can define Markov decision processes.

What Are `NFA`s?

An `NFA` is a nondeterministic finite automata, which is a generalization of a `DFA` (deterministic finite automata). Figure A.12 displays an example of an `NFA`.

An `NFA` enables you to define multiple transitions from a given state to other states. By way of analogy, consider the location of many (most?) gas stations. Usually they are located at an intersection of two streets, which means there are at least two entrances to the gas station. After you make your purchase, you can exit from the same entrance or from the second entrance. In some cases, you might even be able to exit from one location and return to the gas station from the other entrance: this would be comparable to a "loop" transition of a state in a state machine.

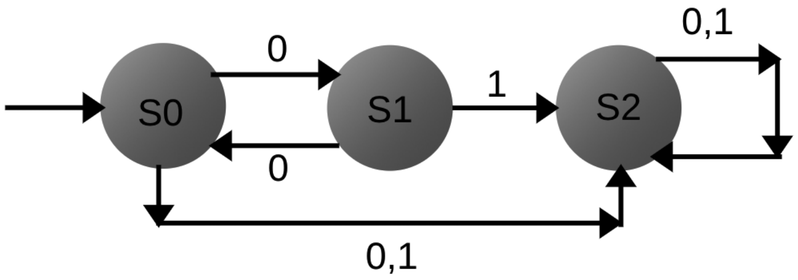

FIGURE A.12. An example of an NFA.

Image adapted from source: *https://math.stackexchange.com/questions/1240601/what-is-the-easiest-way-to-determine-the-accepted-language-of-a-deterministic-fi?rq=1*

The next step involves adding probabilities to `NFA`s in order to create a Markov chain, which is described in more detail in the next section.

What Are Markov Chains?

Markov chains are `NFA`s with an additional constraint: the sum of the probabilities of the outgoing edges of every state equals one. Figure A.13 displays a Markov chain.

As you can see in Figure A.13, a Markov chain is an `NFA` because a state can have multiple transitions. The constraint involving probabilities ensures that we can perform statistical sampling in `MDP`s that are described in the next section.

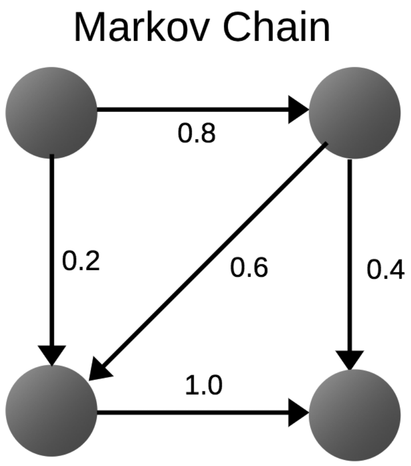

FIGURE A.13. An example of a Markov chain.

Image adapted from source: *https://en.wikipedia.org/wiki/Markov_chain*

Markov Decision Processes (MDPs)

In high-level terms, a Markov decision process is a method that samples from a complex distribution to infer its properties. More specifically, MDPs are an extension of Markov chains, which involves the addition of actions (allowing choice) and rewards (giving motivation). Conversely, if only one action exists for each state (e.g., "wait") and all rewards are the same (e.g., "zero"), an MDP reduces to a Markov chain. Figure A.14 displays an example of an MDP.

Thus, an MDP consists of a set of states and actions, and also the rules for transitioning from one state to another. One episode of this process (e.g., a single game) produces a finite sequence of states, actions, and rewards. A key property of MDPs: history does not affect future decisions. In other words, the process of selecting the next state is independent of everything that happened before reaching the current state.

MDPs are nondeterministic search problems that are solved via dynamic programming and RL, where outcomes are partly random and partly under control. As you learned earlier in this section, almost all RL problems can be formulated as MDPs; consequently, RL can solve tasks that cannot be solved by greedy algorithms. However, the epsilon-greedy algorithm is a clever algorithm that *can* solve such tasks. In addition, the Bellman equation enables us to compute rewards for states. Both are discussed in subsequent sections.

THE EPSILON-GREEDY ALGORITHM

There are three fundamental problems that arise in reinforcement learning:

- the exploration-exploitation trade-off
- the problem of delayed reward (credit assignment)
- the need to generalize

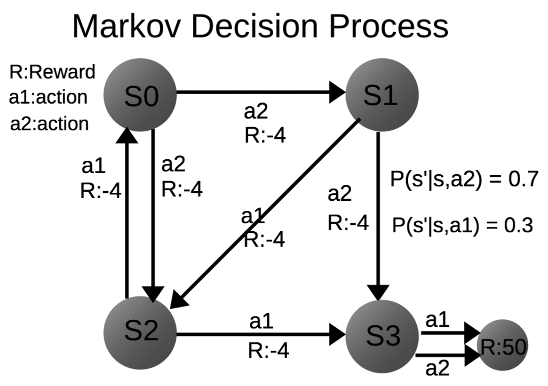

FIGURE A.14. An example of an MDP.

The term "exploration" refers to trying something new or different, whereas the term exploitation refers to leveraging existing knowledge or information. For instance, going to a favorite restaurant is an example of exploitation (you are "exploiting" your knowledge of good restaurants), whereas going to an untried restaurant is an example of exploration (you are "exploring" a new venue). When people move to a new city, they tend to explore new restaurants, whereas people who are moving away from a city tend to exploit their knowledge of good restaurants.

In general, exploration refers to making random choices, whereas exploitation refers to using a greedy algorithm. The epsilon-greedy algorithm is an example of exploration and exploitation, where the "epsilon" portion of the algorithm refers to making random selections, and "exploitation" involves a greedy algorithm.

An example of a simple task that can be solved via the epsilon-greedy algorithm is Open AI Gym's NChain environment, as shown in Figure A.15.

Each state in Figure A.15 has two actions, and each action has an associated reward. For each state, its "forward" action has reward 0, whereas its "backward" action has reward 2. Since a greedy algorithm will always select the larger reward at any state, this means that the "backward" action is always selected. Hence, we can never move toward the final state 4 that has a reward of 10. Indeed, we can never leave state 0 (the initial state) if we adhere to the greedy algorithm.

Here is the key question: how do we go from the initial state 0 to the final state, which contains a large reward? *We need a modified or hybrid algorithm in order to go through intermediate low-reward states that lead to the high reward state.*

The hybrid algorithm is simple to describe: adhere to the greedy algorithm about 90% of the time and randomly select a state for the remaining 10% of

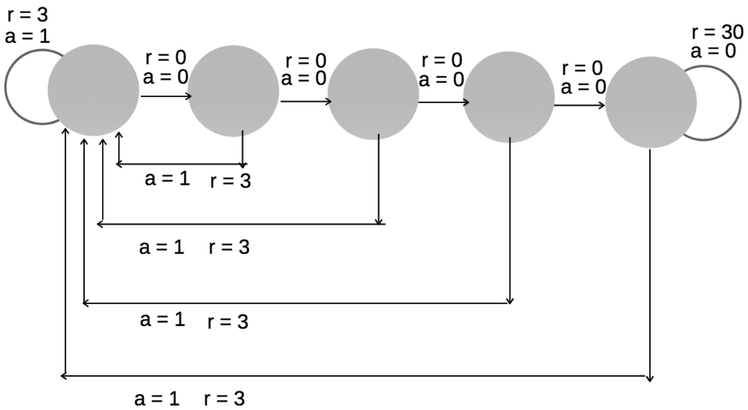

FIGURE A.15. The Open AI Gym's NChain environment.

Image adapted from *http://ceit.aut.ac.ir/~shiry/lecture/machine-learning/papers/BRL-2000.pdf* the time. This technique is simple, elegant, and effective, and it's called the *epsilon-greedy* algorithm.

Incidentally, a Python-based solution for OpenAI's NChain task is here:
https://github.com/openai/gym/blob/master/gym/envs/toy_text/nchain.py

Another central concept in reinforcement learning involves the Bellman equation, which is the topic of the next section.

THE BELLMAN EQUATION

The Bellman equations are named after Richard Bellman, who derived these equations that are ubiquitous in reinforcement learning. There are several Bellman equations, including one for the state value function and one for the action value function. Figure A.16 displays the Bellman equation for the state value function.

As you can see in Figure A.16, the value of a given state depends on the discounted value of future states. The following analogy might help you understand the purpose of the discounted value *gamma* in this equation. Suppose that you have USD 100 that you invest at a 5% annual interest rate. After one year you will have USD 105 (`=100 + 5%*100 = 100*(1+0.05)`), after two years you will have USD 110.25 (`=100*(1+0.05)*(1+0.05)`), and so forth.

Conversely, if you have a future value of USD 100 (with a 5% annual investment rate) that is two years in the future, what is its present value? The answer involves dividing 100 by powers of `(1+0.05)`. Specifically, the present value of USD 100 from two years in the future equals `100/[(1+0.05)*(1+0.05)]`.

In analogous fashion, the Bellman equation enables us to calculate the current value of a state by calculating the discounted reward of subsequent states. The discount factor is called *gamma*, and it's often a value between 0.9 and 0.99. In the preceding example involving USD 100, the value of gamma is 0.9523.

Other Important Concepts in RL

After you have studied the basic concepts in RL, you can delve into the following topics:

- Policy gradient (rules for "best" actions)
- Q-value
- Monte Carlo

$$V^{\pi}(s) = E_{\pi}[\sum_{k=0}^{\infty} \gamma^{k} r_{t+k+1} | s_t = s]$$

FIGURE A.16. The Bellman equation.

- dynamic programming
- Temporal difference (TD)
- Q-learning
- Deep Q network

The preceding topics are explained in online articles (suggestion: use Wikipedia as a starting point for RL concepts), and they will be much more relevant after you grasp the introductory concepts in RL that are discussed in earlier sections. Be prepared to spend some time learning these topics, because some of them are quite challenging in nature.

RL TOOLKITS AND FRAMEWORKS

There are many toolkits and libraries for reinforcement learning, typically based on Python, Keras, Torch, or Java. Some of them are listed here:

- OpenAI gym: A toolkit for developing and comparing reinforcement learning algorithms
- OpenAI universe: A software platform for measuring and training an AI's general intelligence across the world's supply of games, websites, and other applications
- DeepMind Lab: A customizable 3D platform for agent-based AI research
- rllab: A framework for developing and evaluating reinforcement learning algorithms, fully compatible with OpenAI Gym
- TensorForce: Practical deep reinforcement learning on TensorFlow with Gitter support and OpenAI Gym/Universe/DeepMind Lab integration
- tf-TRFL: A library built on top of TensorFlow that exposes several useful building blocks for implementing RL agents
- OpenAI lab: An experimentation system for RL using OpenAI Gym, Tensorflow, and Keras
- MAgent: A platform for many-agent reinforcement learning
- Intel Coach: Coach is a Python reinforcement learning research framework containing implementation of many state-of-the-art algorithms

As you can see from the preceding list, there is a considerable variety of available RL toolkits, and you can visit their home pages to determine which ones have the features that meet your specific requirements.

TF-Agents

Google created the TF-Agents library for RL in TensorFlow. Google TF-Agents is open source and downloadable from Github:

https://github.com/tensorflow/agents

The core elements of RL algorithms are implemented as agents. An agent encompasses two main responsibilities: defining a policy to interact with the environment, and how to learn/train that policy from collected experience. TF-Agents implements the following algorithms:

- DQN: Human-Level Control through Deep Reinforcement Learning, Mnih et al., 2015
- DDQN: Deep Reinforcement Learning with Double Q-Learning, Hasselt et al., 2015
- DDPG: Continuous Control with Deep Reinforcement Learning, Lillicrap et al., 2015
- TD3: Addressing Function Approximation Error in Actor-Critic Methods, Fujimoto et al., 2018
- REINFORCE: Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning, Williams, 1992
- PPO: Proximal Policy Optimization Algorithms, Schulman et al., 2017
- SAC: Soft Actor Critic, Haarnoja et al., 2018

Before you can use TF-Agents, first install the nightly build version of TF-Agents with this command (`pip` or `pip3`):

```
# the --upgrade flag ensures you'll get the latest version
pip install --user --upgrade tf-nightly
pip install --user --upgrade tf-agents-nightly # requires
tf-nightly
```

There are "end-to-end" examples training agents under each agent directory, an example of which is here for DQN:

```
tf_agents/agents/dqn/examples/v1/train_eval_gym.py
```

Keep in mind that TF-Agents is in prerelease status and therefore under active development, which means that interfaces may change at any time.

WHAT IS DEEP REINFORCEMENT LEARNING (DRL)?

Deep reinforcement learning is a surprisingly effective combination of deep learning and RL that has shown remarkable results in a variety of tasks. For example, DRL has won game competitions such as Go (Alpha Go versus world champion Lee Sedol) and even prevailed in the complexity of StarCraft (AlphaStar of DeepMind).

With the release of ELMo and `BERT` in 2018 (discussed earlier in this appendix), DRL made significant advances in `NLP` with these toolkits, surpassing previous benchmarks in `NLP`.

Google released the Dopamine toolkit for DRL, which is downloadable here from GitHub: *https://github.com/google/dopamine*

The `keras-rl` toolkit supports state-of-the-art Deep RL algorithms in Keras, which are also designed for compatibility with OpenAI (discussed earlier in this appendix). This toolkit includes the following:

- Deep Q learning (DQN)
- Double DQN
- Deep deterministic policy gradient (DDPG)
- Continuous DQN (CDQN or NAF)
- Cross-entropy method (CEM)
- Dueling network DQN (Dueling DQN)
- Deep SARSA
- Asynchronous advantage actor-critic (A3C)
- Proximal policy optimization algorithms (PPO)

Please keep in mind that the details of the algorithms in the preceding list require a decent understanding of reinforcement learning. The `keras-rl` toolkit is downloadable here from GitHub: *https://github.com/keras-rl/keras-rl*

MISCELLANEOUS TOPICS

This section contains a very brief description of other areas of TensorFlow that might be of interest to you:

- TFX (TensorFlow Extended)
- TensorFlow Probability
- TensorFlow Graphics
- TF Privacy

The following subsections provide a very brief description of these topics, along with links where you can find additional information.

TFX (TensorFlow Extended)

TFX is a TensorFlow-based ML platform that provides a configuration framework and shared libraries to integrate common components needed to define, launch, and monitor your ML system. TFX involves pipelines that define a data flow through several components (based on TFX libraries), in order to perform a given ML task.

TFX pipeline components enable you to perform a variety of ML tasks, including modeling, training, and serving inference. You can also manage deployments to online, native mobile, and JavaScript targets. A TFX pipeline often includes the following components:

- ExampleGen is the initial input component of a pipeline that ingests and optionally splits the input dataset
- StatisticsGen calculates statistics for the dataset
- SchemaGen examines the statistics and creates a data schema
- ExampleValidator looks for anomalies and missing values in the dataset
- Transform performs feature engineering on the dataset
- Trainer trains the model
- Evaluator performs deep analysis of the training results

- ModelValidator helps you validate your exported models, ensuring that they are "good enough" to be pushed to production
- Pusher deploys the model on a serving infrastructure

TFX is downloadable here on GitHub: *https://github.com/tensorflow/tfx*

TensorFlow Probability

TensorFlow Probability (TFP) is a Python library built on TensorFlow that combines probabilistic models and deep learning on modern hardware. TFP is suitable for data scientists (among others) who want to encode domain knowledge to understand data and make predictions. TFP includes:

- multiple probability distributions and bijectors
- tools to build deep probabilistic models
- variational inference and Markov chain Monte Carlo
- optimizers such as Nelder-Mead, BFGS, and SGLD

Since TFP is based on TensorFlow, TFP enables you to manage models in one language in a start-to-finish manner. More details regarding TFP are here:
https://www.tensorflow.org/probability

TensorFlow Graphics

TensorFlow Graphics is intended to help you train ML systems that contain complex 3D vision tasks. As such, TensorFlow Graphics provides a set of differentiable graphics and geometry layers (cameras, spatial transformations, mesh convolutions, and so forth) and 3D viewer functionalities (such as 3D TensorBoard) to train and debug ML models. More details regarding TensorFlow Graphics are here:
https://github.com/tensorflow/graphics

TF Privacy

TensorFlow Privacy is a Python library that includes implementations of TensorFlow optimizers for training machine learning models with differential privacy. The library comes with tutorials and analysis tools for computing the privacy guarantees provided. More information is here: *https://github.com/tensorflow/privacy*

SUMMARY

This appendix started with an overview of aspects of deep learning, along with some of the issues (such as the vanishing gradient and exploding gradient) that deep learning has solved. You learned about the challenges that exist in deep learning, which include bias in algorithms, susceptibility to adversarial attacks, limited ability to generalize, lack of explainability in neural networks, and the lack of causality.

Then you learned about perceptrons and how they are used as neural networks. Then you saw a TF 2 code sample that shows you how to define a hidden layer in a neural network.

You also learned about the architecture of an `ANN`, along with commonly used hyperparameters. Next, you saw TF code samples for an `XOR` function and an `OR` function. In addition, you saw a Keras-based model for a `CNN` and the `MNIST` dataset.

Then you learned about the architecture of an `RNN`, followed by a Keras-based code sample. Next you saw the architecture of an `LSTM`, as well as a basic code sample. You also got an introduction to variational autoencoders and some of their use cases. In addition, you were introduced to `GAN`s and how you can use them.

In addition, you learned about some basic concepts in `NLP`, such as n-grams, BoW, tf-idf, and word embeddings.

Finally, you learned about reinforcement learning, including the epsilon-greedy algorithm and the Bellman equation, followed by some aspects of deep reinforcement learning, which combines deep learning with reinforcement learning.

Congratulations! You have reached the end of this book, which has covered many TF 2 concepts and has introduced you to Keras, as well as linear regression, logistic regression, and deep learning. You can delve further into machine learning algorithms or proceed with deep learning, and good luck in your journey!

INDEX