TensorFlow 2.x in the Colaboratory Cloud

An Introduction to Deep Learning on Google's Cloud Service

David Paper

Apress®

TensorFlow 2.x in the Colaboratory Cloud

An Introduction to Deep Learning on Google's Cloud Service

David Paper

Apress®

TensorFlow 2.x in the Colaboratory Cloud: An Introduction to Deep Learning on Google's Cloud Service

David Paper
Logan, UT, USA

ISBN-13 (pbk): 978-1-4842-6648-9 ISBN-13 (electronic): 978-1-4842-6649-6
https://doi.org/10.1007/978-1-4842-6649-6

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Jonathan Gennick
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media LLC, 1 New York Plaza, Suite 4600, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

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

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484266489. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

I dedicate this book to my Mother. God bless her!

Table of Contents

About the Author

Dr. David Paper is a full professor at Utah State University (USU) in the Management Information Systems department. He has over 30 years of higher education teaching experience. At USU, he has over 26 years of teaching experience in both the classroom and distance education over satellite. Dr. Paper has taught a variety of classes at the undergraduate, graduate, and doctorate levels, but he specializes in technology education. He has competency in several programming languages, but his focus is currently on deep learning (Python) and database programming (PyMongo). Dr. Paper has published three technical books for industry professionals: *Web Programming for Business: PHP Object-Oriented Programming with Oracle*, *Data Science Fundamentals for Python and MongoDB* (Apress), and *Hands-on Scikit-Learn for Machine Learning Applications: Data Science Fundamentals with Python* (Apress). He has authored more than 100 academic publications. Besides growing up in family businesses, Dr. Paper has worked for Texas Instruments; DLS, Inc.; and the Phoenix Small Business Administration. He has performed information system consulting work for IBM, AT&T, Octel, the Utah Department of Transportation, and the Space Dynamics Laboratory.

About the Technical Reviewer

Mark Mucchetti is an industry technology leader in healthcare and ecommerce. He has been working with computers and writing software for over 30 years, starting with BASIC and Turbo C on an Intel 8088 and now using Node.js in the cloud. For much of that time, he has been building and growing engineering groups, combining his deep love of technical topics with his management skills to create world-class platforms. Mark has also worked in databases, release engineering, front- and back-end coding, and project management. He works as a technology executive in the Los Angeles area, coaching and empowering people to achieve their peak potential as individuals of bold, forward-momentum, and efficient technology teams.

Introduction

We apply the TensorFlow 2.x end-to-end open source platform within the Google Colaboratory cloud service to demonstrate deep learning exercises with Python code to help readers solve deep learning problems. The book is designed for those with intermediate to advanced programming skills and some experience with machine learning algorithms. We focus on application of the algorithms rather than theory. So readers should read about the theory online or from other sources if appropriate. The reader should also be willing to spend a lot of time working through the code examples because they are pretty deep. But the effort will pay off because the exercises are intended to help the reader tackle complex problems.

The book is organized into ten chapters. Chapter 1 introduces the topic of deep learning, neural networks, TensorFlow 2.x, Google Colaboratory, tensors, and input pipelines. Chapter 2 works through a complete deep learning example. Chapter 3 introduces TensorFlow datasets and works through a complete example. Chapter 4 demonstrates how to work with other types of data. Chapter 5 introduces classification and works through a complete deep learning example. Chapter 6 introduces regression and works through deep learning examples. Chapter 7 introduces convolutional neural networks (CNNs) and their architecture and works through a complete example. Chapter 8 introduces recurrent neural networks and works through an automated text generation example. Chapter 9 uses recurrent neural networks for sentiment analysis. Chapter 10 uses recurrent neural networks for time series forecasting.

Note Download this book's example data by clicking the "Download source code" button found on the book's catalog page at www.apress.com/us/book/9781484266489. It can also be downloaded directly from www.github.com/Apress/tensorflow-2.x-in-the-colab-cloud.

CHAPTER 1

Introduction to Deep Learning

We introduce the basic concepts of deep learning. We use TensorFlow 2.x, the Google cloud service, and Google Drive Interactive to make the concepts come alive with Python coding examples.

Notebooks for chapters are located at the following URL: *https://github.com/paperd/tensorflow*.

So what is deep learning? **Deep learning** is a machine learning technique that provides insights from data through automated learning algorithms with the purpose of informing decision making. Deep learning algorithms use successive layers to progressively extract higher-level features from raw input. Whew, that's a mouthful. Let's break it down a bit. Deep learning emphasizes learning successive layers of increasingly meaningful representations from the data. Each layer of a deep learning model learns from the data. So each layer passes down what it learns to the next layer. In image processing, lower layers may identify edges, while higher layers may identify concepts relevant to a human such as digits, letters, or faces. Don't worry if this is confusing because we have yet to define the basics, which we are about to do now.

Neural Networks

Successive layers are almost always learned by models called neural networks. **Neural networks** are a set of algorithms modeled loosely after the human brain that are designed to recognize patterns. These networks interpret sensory data through a kind of machine perception based on labeling or clustering raw input.

A layer is the core building block in deep learning. A **layer** is a container of artificial neurons that usually receives weighted input, transforms it with a set of mostly nonlinear functions, and then passes these values as output to the next layer. **Artificial neurons** are

D. Paper, *TensorFlow 2.x in the Colaboratory Cloud*, https://doi.org/10.1007/978-1-4842-6649-6_1

elementary units in a neural network that receive one or more inputs and sum them to produce an output. Every layer in a neural network is composed of artificial neurons.

A layer can be thought of as a data processing module that acts as a filter for data. Data goes into a layer, and it comes out in a more useful form. That is, layers extract representations out of the data fed into them. Of course, we hope that the representations are meaningful to help us solve the problem at hand. It takes a lot of *practice* and *experimentation* to reap benefits that are meaningful. So we demonstrate and explain numerous code examples to help you gain insights. But we recommend working through the examples many times because deep learning is a very complex and intricate subject.

A very common deep learning problem is the identification of digits 0–9. We can solve this problem by creating a neural network composed of successive layers to help us *automatically* predict a digit from its image data.

For example, suppose we have an image of the digit 8 in our dataset. If our neural network is robust, it should be able to correctly predict that the digit is 8 from the image data without human intervention! That is, the network model is able to *predict* with a high degree of accuracy images of digits. Of course, humans can easily distinguish digits between 0 and 9, but the ability of a computer model to do this is amazing and at the heart of what deep learning is all about.

A neural network is a collection of *neurons* with *synapses* connecting them. It is organized into three main parts:

- Input layer
- Hidden layer
- Output layer

When training a neural network, data is initially passed to the input layer. So the **input layer** brings the initial data into the system for further processing by subsequent layers of artificial neurons. It then passes the data through an activation function before passing it on to the first hidden layer. An **activation function** is an algorithm that defines the output of a neuron given an input or set of inputs.

A **hidden layer** is in between input layers and output layers where artificial neurons take in a set of weighted inputs and produce an output through an activation function. A network can have multiple hidden layers. The **output layer** produces the result for given inputs. It is the place where all the computation is done. Neurons tend to be remarkably

simple with nothing but a floating-point value, an input, and an output. That floating-point value is what we refer to as the *weight* of a neuron.

So neurons take inputs from their previous layer, transform them to keep values within a manageable range with an activation function, and send the transformed inputs along with their weights to neurons in the next layer. Since values at the input layer are generally centered at zero and have already been appropriately scaled, they don't need transformation with an activation function.

Learning Representation from Data

Machine learning algorithms discover rules to execute a data processing task. So, to conduct machine learning predictions, we need three things:

1. Input data points
2. Examples of expected output
3. A way to measure algorithm performance

Input data points are data of some kind. An example of input data could be pictures. Image recognition in deep learning requires pictures. Of course, deep learning models require numeric data. So how do the models interpret images? The pictures must be transformed in some way! Don't worry about how this is done because we cover it in the next chapter.

Note All data, not just image data, must be fit to a numerical representation before it can be consumed by a deep learning model.

To make predictions from data in deep learning, we need *examples of expected output*. So the data must contain a representation of each data example and what each data example represents. We can better understand with an example. When predicting digits, the data must contain representations of each digit and what the digit represents. If an example from the data is the digit 9, we must have the representation of the digit 9 and a target value of 9. We cover how to represent a digit and its target value in the next chapter.

Finally, we need to *gauge algorithmic performance*. We do this by determining the distance between the *algorithm's current output and its expected output*. This distance is often called loss or error. The *loss* is used as feedback to adjust the way the algorithm

works. Such an adjustment is called *learning*. For example, if our neural network model predicts that a digit is 3 but it is really 8, our model has at least some loss. That is, there is some distance between what the model predicts and its expected output (the actual target values).

TensorFlow 2.x

TensorFlow is a *Python* open source library for numerical computation created to facilitate machine learning and deep learning problem solving. TensorFlow bundles together machine learning modules, deep learning modules, and associated algorithms into a common programming environment. TensorFlow 2.x is the most current version of the software. We use the 2.x designation because the software is changing so rapidly.

Google Colab

Google Colab (short for Google Colaboratory) is a cloud service that offers a data science workspace for *Python* very similar to the *Jupyter Notebook*. Actually, any Jupyter Notebook can be directly loaded into the Colab cloud service.

Colab notebooks are stored in Google Drive and can be shared as you would with Google Docs or Sheets. Simply click the Share button at the top right of any Colaboratory notebook or follow the Google Drive sharing instructions.

Peruse `https://colab.research.google.com/notebooks/welcome.ipynb` to get started. The site offers a nice tutorial, but you can browse YouTube videos or other tutorials to deepen your Colab skills. Of course, we walk you through the basics.

Google Drive

Google Drive is a cloud-based file storage and synchronization service that you are most likely already familiar with, so we won't spend much time on it. But we need to show how to connect Google Colab with Google Drive. It just takes a few simple steps:

1. Sign into your Google email account.
2. Open a new browser tab and browse to *Google Colab*.
3. Click the *Google Colab* link.
4. Click *Google Drive* in the top menu from the pop-up window.

All notebooks on your Google Drive account appear in the window. You should see no notebooks appear unless you've worked with Colab in the past. Notebooks are saved in Google Drive My Drive inside the *Colab Notebooks* directory. This directory is automatically created when Colab is connected to Google Drive.

If you want to create a new notebook, click *NEW NOTEBOOK*. Or click *CANCEL*, which takes you to the *Welcome to Colaboratory* screen. This screen offers the main menu for Google Colab as well as the table of contents that helps you get started.

Note The connection we just established between Google Colab and Google Drive is *persistent*. That is, we only need to establish this connection *once* unless browser history is cleared.

Create a New Notebook

Within the Colab environment, it is easy to create a new notebook. Open Google Colab in a browser (if not already open). From the pop-up window, click *NEW NOTEBOOK*. If already in the Colab environment, click *File* in the top-left menu under *Welcome to Colaboratory*. Click *New notebook* from the drop-down menu. A code cell is now ready for executing Python code! Add code or text cells by clicking the + *Code* or + *Text* button. For more options, click *Insert* from the menu.

To create your first piece of code, add the following in the code cell:

```
string = 'Peter picked a pail of pickled peppers'
string
```

To execute code, click the *little arrow to the left*. The output from the code cell shows the contents of the *string* variable.

Tip We recommend copying and pasting code from the website.

GPU Hardware Accelerator

To vastly speed up processing, we use the GPU available from the Google Colab cloud service. Colab provides a free Tesla K80 GPU of about 12 GB. It's very easy to enable the GPU in a Colab notebook:

1. Click *Runtime* in the top-left menu.
2. Click *Change runtime type* from the drop-down menu.
3. Choose *GPU* from the *Hardware accelerator* drop-down menu.
4. Click *SAVE*.

Note The GPU must be enabled in *each* notebook. But it only has to be enabled once.

Test if GPU is active:

```
import tensorflow as tf

# display tf version and test if GPU is active
tf.__version__, tf.test.gpu_device_name()
```

Import the *tensorflow* library and display the version of TensorFlow as well as the status of the GPU. If '/device:GPU:0' is displayed, the GPU is active. If '..' is displayed, the regular CPU is active.

Download a File from a URL

Let's get to work! We can directly download a file from a URL with the *tf.keras.utils.get_file* utility. We need the tensorflow library, but we already imported it. We recommend creating a new code cell by clicking + *Code*.

The following code cell downloads a CSV file from a URL:

```
# import keras module
from tensorflow import keras

ds = 'auto-mpg.data'
```

```
url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/
auto-mpg.data'
dataset_path = tf.keras.utils.get_file(ds, url)
dataset_path
```

Import the keras module from the tensorflow library. Use *tf.keras.utils.get_ file* to download a dataset from the *UCI Machine Learning Repository*. This repository is a collection of databases, domain theories, and data generators for the empirical analysis of machine learning algorithms.

Tip We highly recommend testing small pieces of code in their own code cells to reduce debugging time and effort.

Prepare the Dataset

As is, the dataset needs some preprocessing. For example, it is without feature headings. We recommend creating a new code cell for each example. Do so now by clicking + *Code*.

The following code cell creates a pandas dataframe by accessing the CSV file from the path we created in the previous code cell:

```
# import the pandas library
import pandas as pd

cols = ['MPG','Cylinders','Displacement','Horsepower','Weight',
        'Acceleration', 'Model Year', 'Origin']

raw_dataset = pd.read_csv(dataset_path, names=cols,
                      na_values = "?", comment='\t',
                      sep=" ", skipinitialspace=True)
```

Import pandas and create a list to hold feature names. Use the *read_csv()* function to place the CSV data into a pandas dataframe. A **pandas dataframe** is a two-dimensional (or 2D) data structure with data aligned in a tabular row and column fashion. Let's look at the last five records:

```
raw_dataset.tail()
```

The Pandas *tail* method returns the last n rows. By default, it returns the last five rows. It is useful for quickly verifying data.

We can save the original data by creating a copy:

```
# create a copy
data = raw_dataset.copy()

# verify contents by displaying some data
data.tail()
```

Colab Abends

An AbEnd (also abnormal end or abend) is an abnormal termination of software or a program crash. When we run Colab for a long time (several hours), read a large dataset into memory, and process said data or create a really large notebook, it may crash. When this happens, we have two choices:

1. Restart runtime.
2. Close the notebook and restart from scratch.

To restart runtime, click *Runtime* in the top menu, click *Restart runtime* from the drop-down menu, and click *YES* when prompted. And rerun your notebook from the beginning. Colab recommends this option. To restart from scratch, clear browser history and open Colab.

Colab Strange Results

Sometimes unexpected errors or other strange results arise when working with Colab. If this happens, restart runtime or open Colab from scratch as described in the "Colab Abends" section for the notebook you are working on.

Tensors

A **tensor** is a container for *numeric* data. Tensors can contain an arbitrary number of dimensions. A dimension is often called an *axis*.

In deep learning, tensors are considered a generalization of matrices represented by *n-dimensional* arrays. The dimensionality of a tensor is often described by its number of axes. So tensors are defined by how many axes they have in total. The **rank** is the number of axes represented by a tensor.

The best way to understand tensors is through examples. So let's start with the simplest type.

Scalars (0D Tensors)

A **scalar** is a tensor of only one number. So a scalar is considered a zero-dimensional (or 0D) tensor. Examples include a numpy float32 or float64 number.

Let's look at an example:

```
import numpy as np

# create numpy scalar 9
scalar = np.array(9)
scalar
```

Import the numpy module. Assign a numpy array containing 9 to a variable and display the result.

Signal the rank of the scalar tensor as so:

```
# signal its rank
print (str(scalar.ndim) + 'D')
```

The *ndim* attribute signals the number of axes (or dimensionality) of a numpy tensor. Since our variable is a scalar, we see 0D displayed.

Vectors (1D Tensors)

A **vector** is an array of numbers. So a vector is a one-dimensional (or 1D) tensor. A 1D tensor has exactly one axis.

Create a numpy vector containing six elements:

```
# create numpy vector [0, 1, 0, 0, 0, 0]
vector = np.array([0, 1, 0, 0, 0, 0])
vector
```

We see [0, 1, 0, 0, 0, 0] displayed.

Signal the rank of the vector:

```
# signal its rank
print (str(vector.ndim) + 'D')
```

We see 1D displayed.

Matrices (2D Tensors)

A **matrix** is an array of vectors. The simplest matrix is a two-dimensional (or 2D) tensor. A 2D tensor has two axes. Its axes are generally referred to as *rows* and *columns*.

Create a numpy matrix:

```
# create a numpy matrix
matrix = np.array([[0, 1, 0, 0, 0, 0],
                   [0, 0, 1, 0, 0, 0],
                   [0, 0, 0, 1, 0, 0],
                   [0, 0, 0, 0, 1, 0],
                   [0, 0, 0, 0, 0, 1]])
matrix
```

Rows are entries from the first axis, and columns are entries from the second axis. So the first row from our example is [0, 1, 0, 0, 0, 0], and the first column is [0, 0, 0, 0, 0, 0]. Since the matrix has five rows and six columns, it is considered a 5 × 6 matrix.

Signal the rank of the matrix:

```
# signal its rank
print (str(matrix.ndim) + 'D')
```

We see 2D displayed.

3D Matrices (3D Tensors)

We can create a 3D tensor by packing 2D tensors into a new array. A 3D tensor can be visually interpreted as a cube of numbers.

Let's look at an example:

```
# create a 3D tensor
D3 = np.array([[[0, 1, 2]],
               [[3, 4, 5]],
               [[6, 7, 8]]])

# signal its rank
print (str(D3.ndim) + 'D')
```

By packing 3D tensors into an array, we can create 4D tensors and so on. In deep learning, we generally manipulate tensors that are 0D–4D. With video processing, we can go up to 5D.

Key Attributes of Tensors

1. Rank
2. Shape
3. Datatype

As discussed earlier, **rank** is the number of axes, and it is displayed with the *ndim* attribute. A 0D tensor has zero axes, a 1D tensor has one axis, a 2D tensor has two axes, and a 3D tensor has three axes.

Shape is a tuple of integers that describe the number of dimensions along each axis. So our 3D matrix has shape (3, 1, 3), 2D matrix has shape (5, 6), vector has shape (6,), and scalar has an empty shape (). We can display shape with the *shape* attribute.

Let's see the shape of our 3D matrix:

```
# 3 instances of 1 x 3 matrices
D3.shape
```

Display the shape of our 2D matrix:

```
# 5 rows and 6 columns (or 5 x 6 matrix)
matrix.shape
```

Display the shape of our vector:

```
# 6 element vector
vector.shape
```

Display the shape of our scalar:

```
# just a scalar number
scalar.shape
```

Datatype is the description of data contained in a tensor. Use the *dtype* attribute to display datatype.

Display datatypes of our tensors:

```
# dtype of tensors

print (scalar.dtype)
print (vector.dtype)
print (matrix.dtype)
print (D3.dtype)
```

We see that all of our tensors contain int64 values.

Input Pipelines

An **input pipeline** is a sequence of data processing components that manipulate and apply data transformations. Pipelines are very common in machine learning and deep learning systems because these systems are *data rich*. That is, they demand large volumes of data to perform well. Input pipelines are the best way to transform large datasets because they break data down into manageable components.

Each component of an input pipeline pulls in a large amount of data, processes it in some manner, and spits out the result. The next component pulls in the resultant data, processes it in another manner, and spits out its own output. The pipeline continues until all of its components have finished their work.

The tf.data API

TensorFlow application revolves around the concept of a dataset encapsulated in the tf.data API. The *tf.data API* enables you to build fast, flexible, and easy-to-use input pipelines from simple, reusable pieces. It is the recommended API for building input pipelines in TensorFlow.

Let's create a simple TensorFlow tensor:

```
import tensorflow as tf

X = tf.range(5)
X
```

Import the tensorflow library if necessary. Create a TensorFlow tensor with shape (5,) and datatype int32. The vector contains values [0, 1, 2, 3, 4], which correspond to shape (5,).

Tip Compare your output with website code output to verify results.

Use the *numpy()* attribute to display values from a TensorFlow tensor:

```
X.numpy()
```

We can also access individual elements from a tensor:

```
# first element from tensor
X[0].numpy()
```

Or access multiple elements:

```
# 2nd, 3rd, and 4th elements from tensor
X[1:4].numpy()
```

The code slices the second, third, and fourth elements from the tensor.

Function from_tensor_slices

Function *from_tensor_slices* creates a tf.data.Dataset from all slices of a tensor. In our case, it creates a dataset whose elements are all the slices of *X* (along the first

dimension). A **tf.data.Dataset** is the standard TensorFlow API to build input pipelines. It represents a potentially large set of elements.

Let's demonstrate with an example:

```
dataset = tf.data.Dataset.from_tensor_slices(X)
dataset
```

We just created a TensorSliceDataset object sliced from X with shape () and datatype tf.int32.

Iterate a tf.data.Dataset

Iterate over a *tf.data.Dataset* with a simple loop:

```
for item in dataset:
  print (item)
```

We see each value along with its shape and datatype.

Tensors and numpy

TensorFlow tensors play nice with numpy. We can create a TensorFlow tensor from a numpy array and vice versa. We can even apply TensorFlow operations to numpy arrays and numpy operations to TensorFlow tensors.

Create a numpy array from the TensorFlow dataset we just created as shown in Listing 1-1.

Listing 1-1. Create a numpy array from a TensorFlow dataset

```
# create a variable to hold a line break
br = '\n' # this is just a convenient way to include a line break

# import numpy
import numpy as np

# technique 1
```

```
ls = []
for item in dataset:
  e = item.numpy()
  ls.append(e)

np_arr = np.asarray(ls, dtype=np.float32)
print (type(np_arr))
print (np_arr, br)

# technique 2

ls = [item.numpy() for item in dataset]
np_arr = np.asarray(ls, dtype=np.float32)

print (type(np_arr))
print (np_arr)
```

We show two techniques to create a numpy array from a TensorFlow dataset. Technique 1 initializes a list, iterates the tensor, converts each value to numpy, and appends it to a list. The list is then converted to a numpy array. Technique 2 performs the same logic, but uses list comprehension for more compact code.

We can convert the numpy array back to a TensorFlow dataset with the *constant* method:

```
tf_arr = tf.constant(np_arr)
tf_arr
```

However, constants are immutable. That is, their values cannot be modified. So we can use the *variable* method if we need to modify values.

Use the variable method:

```
tf_arr = tf.Variable(np_arr)
tf_arr
```

Chaining Transformations

We can apply transformations to a tf.data.Dataset by calling its transformation methods. Each method returns a *new* dataset, which allows us to chain transformations. Let's start with a single transformation.

Create a tf.data.Dataset and show its values:

```
dataset = tf.data.Dataset.range(5)

for item in dataset:
  print (item)
```

The dataset contains values [0, 1, 2, 3, 4].

Use the *repeat()* transformation method to repeat elements in a tensor:

```
data_rep = dataset.repeat(3)

for item in data_rep:
  print (item)
```

The new dataset contains three sets of the original. We repeat data to enlarge a dataset for better model performance without getting new data.

Now, let's chain transformations:

```
data_batch = dataset.repeat(3).batch(7)

for item in data_batch:
  print (item)
```

What happened? The first transformation, *repeat(3)*, creates three copies of the original dataset. We chain the first transformation into the second with *batch(7)*, which creates batches of seven elements each.

So the new dataset consists of three tensors. The first tensor contains [0, 1, 2, 3, 4, 0, 1], the second tensor contains [2, 3, 4, 0, 1, 2, 3], and the third tensor contains [4]. By the time we get to the third batch, we run out of data.

We can drop the final batch:

```
data_drop = dataset.repeat(3).batch(7, drop_remainder=True)

for item in data_drop:
  print (item)
```

The new dataset consists of two tensors. The first tensor contains [0, 1, 2, 3, 4, 0, 1], and the second tensor contains [2, 3, 4, 0, 1, 2, 3]. Transformation methods don't modify datasets. They create new ones so we can keep track of each dataset by naming them differently.

Create equal batches:

```
data_equal = dataset.repeat(3).batch(5)

for item in data_equal:
  print (item)
```

The new dataset consists of three tensors. Each tensor contains [0, 1, 2, 3, 4].

Mapping Tensors

Use the *map* method to transform elements in a tensor. The **map** function returns a map object of the results after applying the given function to each element of a given tensor. The returned object is an iterator. An **iterator** is a Python object capable of returning its members one at a time. Lists, tuples, and strings are common iterators in Python.

Whew! Let's look at the example in Listing 1-2 to help us understand how the map function works.

Listing 1-2. A simple map function example

```
# create a dataset
dataset = tf.data.Dataset.range(7)

# repeat and batch it
data_batch = dataset.repeat(3).batch(7)

# display the batched dataset
for row in data_batch:
  print (row)

# map() a function on it
data_map = data_batch.map(lambda x: x ** 2)

# display the first batch
print ()
for item in data_map.take(1):
  print (item)
```

We create a new dataset with values [0, 1, 2, 3, 4, 5, 6]. We chain the repeat transformation to the batch transformation to create a new dataset with three tensors. Each tensor contains [0, 1, 2, 3, 4, 5, 6]. We square each element by mapping with a lambda function. A **lambda function** is a single-line function declared with no name that can have any number of arguments, but can only have one expression. Instead of iterating the entire dataset, we can take one or more samples with the *take* method. In our case, we just take the first sample (or first tensor) from the dataset.

So the mapped dataset contains three tensors. Each tensor contains [0, 1, 4, 9, 16, 25, 36] because the lambda function squares a value and the map function maps the lambda function expression onto each element in a tensor.

Filter a tf.data.Dataset

What if we want to filter a dataset? The **filter** method filters a given sequence with the help of a function that tests whether each element in the sequence is true or not.

Let's look at the example in Listing 1-3.

Listing 1-3. A simple filter() function example

```
# create a dataset
dataset = tf.data.Dataset.range(7)

# display the dataset
for row in dataset:
  print (row)

# apply a filter
data_filter = dataset.filter(lambda x: x < 6 and x > 3)

print ()
for item in data_filter:
  print (item)
```

We create a new dataset with values [0, 1, 2, 3, 4, 5, 6]. We filter the dataset to extract values between 3 and 6. Since we use less than and greater than in the lambda function, we don't include either the 3 or 6 value. So the filtered dataset contains [4, 5] because the lambda function extracts values between 3 and 6 non-inclusive.

Shuffling a Dataset

Deep learning algorithms work best when instances in the training set are independent and identically distributed. A simple way to ensure this is to shuffle instances with the *shuffle* method.

Create a tf.data.Dataset to shuffle:

```
# create a dataset
dataset = tf.data.Dataset.range(10).repeat(3)
print ('dataset has', len(list(dataset)), 'elements')
```

Shuffle the dataset:

```
# shuffle data into batches of 7
ds = dataset.shuffle(buffer_size=5).batch(7)

for item in ds:
  print (item)
```

We get tensors of seven elements. Notice that the last tensor has only two elements. We have four tensors of size seven, which equals 28. Since the dataset has 30 elements, we have two elements left over.

We set buffer size to 5. So TensorFlow keeps a buffer of the next five samples (or tensors) and randomly selects one of those five samples. It then adds the next element to the buffer. Each sample contains a batch of data. So each sample in our example contains seven elements because we set batch size to 7. Performance can be improved by experimenting with different batch and buffer sizes, but getting it right takes time and energy.

Once a dataset is shuffled, each dataset iteration creates a new shuffle:

```
# rerun to get a different shuffle
for item in ds:
  print (item)
```

TensorFlow Math

TensorFlow provides several operations for math computations with the *tf.math* module. Peruse *www.tensorflow.org/api_docs/python/tf/math* for all possible math operations. To perform math operations, tensors must have the same shape.

Vector Tensors

Let's create some data:

```
# create data

v1 = np.array([0, 1, 4, 8, 16])
v2 = np.array([0, 3, 9, 27, 81])
```

Convert numpy arrays to tensor constants and add:

```
conv1 = tf.constant(v1)
conv2 = tf.constant(v2)

result = tf.add(conv1, conv2)
result
```

The result is [0, 4, 13, 35, 97].

Convert numpy arrays to tensor variables and add:

```
varv1 = tf.Variable(v1)
varv2 = tf.Variable(v2)

result = tf.add(varv1, varv2)
result
```

We get the same result. The only difference between constants and variables is that constant values cannot be changed. That is, they are immutable.

Subtract tensor variables:

```
result = tf.subtract(varv2, varv1)
result
```

Subtracting var1 from var2 is [0, 2, 5, 19, 65].

Mix constants and variables:

```
result = tf.add(conv1, varv2)
result
```

The result is [0, 4, 13, 35, 97].

Test for equivalency:

```
result = tf.equal(varv1, varv2)
result
```

The result is [True, False, False, False, False].

Multiply tensor constants:

```
result = tf.multiply(conv1, conv2)
result
```

The result is [0, 3, 36, 216, 1296].

Divide a tensor by a value:

```
result = tf.divide(conv2, 3)
result
```

The result is [0, 1, 3, 9, 27].

Matrix Tensors

Math operations work on n-dimensional tensors. So let's perform math operations on matrix tensors.

Create some data:

```
# create data
m1 = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
m2 = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])

m1.shape, m2.shape
```

We just created a pair of 3 × 3 matrices. That is, both have three rows and three columns.

Convert numpy matrices to tensors and add:

```
conm1 = tf.constant(m1)
conm2 = tf.constant(m2)

result = tf.add(conm1, conm2)
result
```

The result is [[2, 0, 0], [0, 2, 0], [0, 0, 2]].

Test for equivalency:

```
result = tf.equal(conm1, conm2)
result
```

The result is [[True, True, True], [True, True, True], [True, True, True]] because the tensors are equivalent.

tf.data.Dataset Tensors

We can perform math operations on tf.data.Dataset tensors.

Create a dataset:

```
# create a dataset
m = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
m
```

We just created a 3 × 4 matrix. So the matrix consists of three rows and four columns.

Convert the dataset to a tf.data.Dataset:

```
dataset = tf.data.Dataset.from_tensor_slices(m)
dataset
```

The tf.data.Dataset is a *TensorSliceDataset* because we transformed the numpy dataset with the *from_tensor_slices* method. The dataset contains three tensors. The shape of each tensor is (4,), which means that each has four elements. Elements are datatype int64.

Display the tensors:

```
for t in dataset:
  print (t)
```

The tf.data.Dataset contains three tensors with values [1, 2, 3, 4], [5, 6, 7, 8], and [9, 10, 11, 12].

Transform tensor values:

```
squared_data = dataset.map(lambda x: x ** 2)

# display tensors
for item in squared_data:
  print (item)
```

We map the lambda function to tensor elements. So each element is squared.

Save a Notebook

Although *Autosave* is implemented in Google Colab, there is a delay between the moment you execute a cell and when the save occurs. So we recommend periodically saving.

Manually saving a notebook just takes two steps:

1. Click *File* in the top-left menu.
2. Click *Save* from the drop-down menu.

The notebook is saved in Google Drive My Drive in the *Colab Notebooks* directory.

Download a Notebook to a Local Drive

Google Drive is an excellent place to store Colab notebooks. But we also like to save notebooks to a local drive.

Download a notebook to a local drive:

1. Be sure to save the notebook.
2. Click *File* in the top-left menu.
3. Click *Download .ipynb* from the drop-down menu.

The notebook is now located in the *Downloads* directory.

Load a Notebook from a Local Drive

We use Google Drive as backup because it only provides 15 GB of free space. If you work for a company, they may provide extra storage. Given this case, you may want to use Google Drive for primary storage.

To load a notebook from a local drive

1. Open *Google Colab*.
2. From the pop-up menu, click *Upload*.
3. Click *Choose File*.
4. Locate the notebook on your local drive and open it.

CHAPTER 2

Build Your First Neural Network with Google Colab

We work through a complete deep learning example with Python's TensorFlow 2.x library in the Google Colab cloud service. We also demonstrate how to link your Google Drive with the Colab cloud service.

Notebooks for chapters are located at the following URL: *https://github.com/paperd/tensorflow*.

Building a competent neural network with TensorFlow 2.x is relatively easy. Data science professionals follow a few steps:

1. Get raw data.
2. Explore and preprocess raw data.
3. Split raw data into train-test sets.
4. Create a tf.data.Dataset.
5. Prepare the input pipeline.
6. Create and train a neural network model.

Data scientists want raw data because it is untouched. They want to clean, munge, and sculpt raw data to chisel out meaning for their specific problem at hand. If data is already processed, much of its meaning may already be lost. It is always a good idea to explore data to get a sense of what it looks like before attempting to preprocess. Once data is cleansed and wrangled, it is split into train-test sets.

D. Paper, *TensorFlow 2.x in the Colaboratory Cloud*, https://doi.org/10.1007/978-1-4842-6649-6_2

We create a tf.data.Dataset for TensorFlow consumption. Once in the proper form, we prepare the input pipeline. During preparation, further data cleansing and wrangling may be necessary. We then create and train a neural network model with input pipeline data.

Although reality requires all of these steps, we focus on data modeling with TensorFlow 2.x as it possesses a steep learning curve for the novice. So we begin with a simple, preprocessed dataset to free learners from the burden of data preprocessing and other associated tasks.

GPU Hardware Accelerator

To vastly speed up processing, we can use the Google *Colab GPU*. However, we must enable the GPU in each newly created notebook:

1. Click *Runtime* in the top-left menu.
2. Click *Change runtime type* from the drop-down menu.
3. Choose *GPU* from the *Hardware accelerator* drop-down menu.
4. Click *SAVE*.

The free GPU has usage limits. Consult the following URL to find out more: https://research.google.com/colaboratory/faq.html#resource-limits.

Test if GPU is active:

```
import tensorflow as tf

# display tf version and test if GPU is active
tf.__version__, tf.test.gpu_device_name()
```

Import the *tensorflow* library. If '/device:GPU:0' is displayed, the GPU is active. If '..' is displayed, the regular CPU is active.

The load_digits Dataset

The *load_digits* dataset is part of the scikit-learn library, which is a free software machine learning library for the *Python* programming language. The library features various classification, regression, and clustering algorithms that are easy to use and manipulate.

The load_digits dataset is heavily preprocessed, so we don't have to worry about cleaning or wrangling. It consists of 1,797 8 × 8–pixel images. Each image is a 64-pixel matrix that represents a handwritten digit from 0 to 9. We get 64 pixels by multiplying 8 by 8. A **pixel** is an integer with a value between 0 and 255 used to represent image data. The load_digits dataset is commonly used for training machine learning systems to recognize images of handwritten digits through algorithmic *classification*.

The *images* container in load_digits holds image data of handwritten digits. The *data* container holds flattened feature vectors. A **feature vector** is an n-dimensional vector of numerical features that represent some object. In our case, a feature vector of 64 pixels represents an image of a handwritten digit. The flattening step is needed to prepare image data for input into fully connected neural network layers. Feature vectors have a length of 64 because each 8 × 8 image is flattened by multiplying 8 rows and 8 columns of pixels. The *target* container holds target values, and the *target_names* container holds target names. The *DESCR* container holds descriptions of the dataset.

Explore the Dataset

Load the dataset:

```
import tensorflow as tf
from sklearn.datasets import load_digits

# get data
digits = load_digits()

# get available containers (or keys) from dataset
print (digits.keys())
```

If you haven't already done so, import the *tensorflow* library. Also, import the dataset from the *sklearn* library. Get the data and display its keys.

Create variables to hold data, images, targets, and target names:

```
# create variables

data = digits.data
images = digits.images
targets = digits.target
target_names = digits.target_names
```

Listing 2-1 displays basic information about the dataset.

Listing 2-1. Information about the dataset

```
br = '\n' # create a newline variable

# display tensor information

print ('data container:')
print (str(data.ndim) + 'D tensor')
print (data.shape)
print (data.dtype, br)

print ('image container:')
print (str(images.ndim) + 'D tensor')
print (images.shape)
print (images.dtype, br)

print ('targets container:')
print (str(targets.ndim) + 'D tensor')
print (targets.shape)
print (targets.dtype, br)

print ('target names container:')
print (target_names)
```

The data container is a 2D tensor with 1,797 flattened vectors of 64-pixel elements. The images container is a 3D tensor with 1,797 8 × 8 matrices. The images container holds the original un-flattened images. The targets container is a 1D tensor with 1,797 target values between 0 and 9. The target_names container holds classification labels 0–9. So, classification is based on 10 classes represented by digits between 0 and 9.

Let's visualize the first data element:

```
# first image begins at index 0

image = images[0]
import matplotlib.pyplot as plt
plt.imshow(image, cmap='binary')
plt.show()
```

We see that the image is zero, but computers can't *see* as we do. However, we can train them to understand that the digit is zero.

Display the target value (or label) and the feature image (8 × 8 matrix) of the first image:

```
# first digit begins at index 0
target = targets[0]
print ('digit is:', target, br)

# image matrix of first image
print ('image matrix:', br)
print (image)
```

The computer recognizes the first label as 0. And it associates the first image matrix with that label. With images and their associated labels, we can train a computer to distinguish between digit images.

Image Matrix

A closer inspection of the image matrix is key to training:

1. Numbers in the matrix represent grayscale intensity.
2. Every 0 represents *white space*.
3. Higher numbers represent darker shades of *gray* to *black*.

A *grayscale* image is one where the value of each pixel is a single sample representing only an amount of light. That is, it only carries intensity information. Grayscale images are composed exclusively of shades of gray. The contrast of an image ranges from black at the weakest intensity to white at the strongest.

The lower the number, the lighter the gray (with zero being white). The higher the number, the darker shade of gray (with high numbers approaching black). So a computer is able to understand the shape of a digit with grayscale intensity matrix mapping.

Split Data into Train-Test Sets

We split a dataset to find a good fitting model with training data and generalize the model to new data with test data. With the load_digits dataset, we can use image data directly or reshape the flattened data. Let's reshape flattened data. This is a good exercise because you may have to reshape a dataset in the future.

Create variables to hold input dimensions:

```
# Input image dimensions
img_rows, img_cols = 8, 8
```

Reshape feature data:

```
# Reshape
X = data.reshape(data.shape[0], img_rows, img_cols)
print ('X reshaped:', X.shape)
print ('number of dimensions:', X.ndim)
```

The new feature dataset is (1797, 8, 8), which is what we want.

Establish the target dataset:

```
y = targets
y.shape
```

The target dataset is (1797,).

Now that we have the feature dataset and associated targets, we are ready to split:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, random_state=0)
```

Import the *train_test_split* method. Split data into train and test sets. Two-thirds of the dataset is for training, and one-third is for testing. Random state is set for reproducibility of results. That is, results are consistent. We use this split for the modeling example.

Alternatively, we can use images directly as shown in Listing 2-2.

Listing 2-2. Alternative method to split image data

```
X_alt = images
y_alt = targets

print ('X:', X_alt.shape)
print ('number of dimensions:', X_alt.ndim, br)

X_tra, X_tes, y_tra, y_tes = train_test_split(
    X_alt, y_alt, test_size=0.33, random_state=0)
```

```
print ('X_train:', X_tra.shape)
print ('number of dimensions:', X_tra.ndim)
train_percent = X_tra.shape[0] / X_alt.shape[0]
print ('train data percent of X data:', train_percent, br)

print ('X_test:', X_tes.shape)
print ('number of dimensions:', X_tes.ndim)
test_percent = X_tes.shape[0] / X_alt.shape[0]
print ('test data percent of X data:', test_percent, br)

num_images = X_tra.shape[0] + X_tes.shape[0]
print ('total number of images:', num_images)
```

Build the Input Pipeline

A TensorFlow input pipeline expects feature data as float32 or float64 and label data as int32 or int64:

```
X_train.dtype, y_train.dtype
```

Feature data is float64 and label data is int64.

Feature data with a large spread of values may incite error in a neural network, which makes the learning process unstable. Scaling mitigates this problem. Scaling may also speed up calculations in an algorithm. *Feature scaling* is a technique to scale the range of feature data.

Scale train and test feature data:

```
# scale by dividing by the number of pixels in an image
s_train = X_train / 255.0
s_test = X_test / 255.0
```

We scale feature images by dividing them by 255. Image pixels are stored as integer numbers in the range 0–255, which is the range that a single 8-bit byte can hold. The division ensures that input pixels are scaled between 0.0 and 1.0.

Create data objects for TensorFlow consumption:

```
train_dataset = tf.data.Dataset.from_tensor_slices((s_train, y_train))
test_dataset = tf.data.Dataset.from_tensor_slices((s_test, y_test))
```

Let's see what our tensors look like:

```
print ('train:', train_dataset)
print ('test: ', test_dataset)
```

We see that both train and test tensors consist of 8 × 8 float64 images and int64 scalar target values.

Explore TensorFlow Data

Let's explore what is actually inside the TensorFlow dataset we just created.

Display a slice from the *first* feature image and its target from the train set:

```
for feature, label in train_dataset.take(1):
  print (feature[0], br)
  print (label)
```

We display the first slice from the first image for brevity. We also display the first target. The *take* method grabs samples from a TensorFlow dataset. In this case, we grabbed only the first sample, but we can take *n* elements.

Let's grab the first two labels from the test set:

```
for _, label in test_dataset.take(2):
  print (label)
```

Now, we build the input pipeline.

Shuffle Data

Before we discuss data shuffling, we need to understand a few concepts. An **epoch** is one cycle through the full training set. Training a neural network typically requires more than a few epochs. A **batch** of data is a group of training samples. Deep learning models don't process an entire dataset at once. They break data into small batches.

Shuffling data after each epoch ensures that we won't be *stuck* with too many bad batches. How does this work? Shuffling reduces model variance, which produces more generalized results and reduces overfitting. **Overfitting** is when a model trains data too well. Overfitting happens when a model learns detail and noise in the training data so well that it negatively impacts performance of the model with new data.

Shuffling ensures that each data point creates an *independent* change on the model without being biased by preceding data points. That is, it ensures that training data fed to the model contains all flavors of the data.

A great metaphor is *shuffling a deck of cards*. We shuffle a deck of cards before playing a card game because we want to make sure that each player has the same chance of getting a specific card as another player. Just as shuffling a deck of cards removes bias, so does shuffling data before each epoch when training a neural network.

Continue Pipeline Construction

We need to shuffle and batch our TensorFlow consumable train data. We only batch test data. It doesn't need to be shuffled because it is new data. Once we set batch and buffer sizes, we are ready to shuffle and batch.

Set batch and buffer sizes:

```
BATCH_SIZE = 64
SHUFFLE_BUFFER_SIZE = 100
```

Tweaking batch and buffer sizes can increase performance. We *arbitrarily* set batch size to 64 and shuffle buffer size to 100. You can try different values to see how results are impacted. We suggest making buffer size relatively large, or shuffling won't be very effective.

It is a good idea to give the new datasets their own names:

```
train_ds = train_dataset\
.shuffle(SHUFFLE_BUFFER_SIZE)\
.batch(BATCH_SIZE)

test_ds = test_dataset.batch(BATCH_SIZE)
```

Notice that we only shuffle training data. The reason is because test data is supposed to represent data that our model has not seen. That is, test data is supposed to represent new data. Once we run the shuffle method on training data, the model *automatically* shuffles the data before each epoch!

Let's explore our new datasets:

```
train_ds, test_ds
```

Shapes are (None, 8, 8). We get *None* as an extra dimension. What happened? This extra dimension is added because TensorFlow models can accommodate any batch size!

Feedforward Neural Networks

A feedforward model is the simplest type of network. A **feedforward** neural network is one where information only travels forward in the network. Data moves from the input nodes through the hidden nodes (if any) and to the output nodes. There are no cycles or loops in the network.

The layers are *fully connected*, which means that each layer is fully connected to the next one. Fully connected layers connect every node (or neuron) in the previous layer with every node in the successive layer. That is, each neuron in a *layer* receives an input from all neurons present in the *previous* layer. Fully connected layers are typically referred to as *densely connected*.

Number of Layers

Typically, *more layers increase model performance*. But more layers require more computing resources. When we don't have much training data, a simple network with few layers tends to perform as good as or better than a complex network with many layers.

It is a good idea to start with a simple network when exploring a new dataset. We also suggest drawing a small random sample on the dataset and training it with the simple network to get an idea of its potential performance. Following this suggestion saves time and computing resources upfront. And we get acquainted with the dataset. If the simple model doesn't train well on the small sample, we can get more data and/or try to find out why we got poor performance before moving to a more complex model.

Our Model

The first model we build contains an input layer, a hidden layer, and an output layer. The first layer in a neural network is *always* an input layer to inform the model of the shape of incoming data.

The first layer of our model is a Flatten layer. **Flattening** is the process of converting data into 1D arrays. When training a fully connected network, input layer data should always be either 1D vectors or 2D matrices.

The *Flatten* layer reshapes each tensor into a shape that is equal to the number of elements contained in the tensor not including the batch dimension. This layer has no parameters. Since this is the *input* layer, we specify *input_shape* as *(8, 8)* to inform the model that each feature image is represented by an 8 × 8 matrix.

The second layer is a densely connected *hidden* layer. The *Dense* layer adds the fully connected layer to the neural network. It contains 256 neurons (or nodes) and uses the *relu* activation function.

The final layer is the *output* layer. It is also a densely connected layer that contains ten neurons and uses the *softmax* activation function. The output layer must always contain the same number of neurons as the number of classes. Since we are classifying digits 0–9, we have ten classes.

An **activation function** defines the output of a node given an input or set of inputs. It is an algorithm that activates each node in a neural net.

ReLu (or rectified linear unit) is a piecewise linear function that outputs input values directly if positive or zero if negative. It is the default activation function for many networks because it facilitates easier training and often achieves better performance than other activation functions.

Softmax output is large if the score is large or small if the score is small. It is frequently used in classification problems where classes are mutually exclusive. In this case, each image can be one and only one digit.

Let's begin by defining the input shape:

```
for item in train_ds.take(1):
  s = item[0].shape

in_shape = s[1:]
in_shape
```

Take the first sample from the train dataset and retrieve its shape. Since the Flatten layer needs the shape of each image, we slice this portion.

Import libraries to build layers:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
```

Build the model:

```
model = Sequential([
    Flatten(input_shape=in_shape),
    Dense(256, activation='relu'),
    Dense(10, activation='softmax')
])
```

Model Summary

The **summary** method displays the characteristics of a model. It shows layers, output shapes, and parameters.

Let's try the method:

```
model.summary()
```

The layers, output shapes, and parameters are displayed. The model begins with a Flatten layer with output shape (None, 64) and 0 parameters. Batch size can be any number, so *None* is shown. The first layer accepts the input shape of the data, but doesn't act on the data in any way. So, each input tensor is a 64-element vector and there are 0 trainable parameters. Remember that each layer receives output from the previous layer.

The first Dense hidden layer receives tensors from the previous layer into 256 neurons at this layer. The output shape is (None, 256) because this layer has 256 neurons. Trainable parameters are determined by multiplying neurons at this layer by neurons from the previous layer and adding neurons at this layer to the result. So, this layer has 16,640 parameters because it inputs 64 neurons from the previous layer that are passed to 256 neurons. Multiply 64 by 256 to get 16, 384. But we still have to account for the 256 neurons in this layer. So add 16,384 to 256 to get 16,640 parameters!

The output layer receives tensors from the hidden layer into ten neurons at this layer. The output shape is (None, 10) because this layer has 10 neurons. The previous hidden layer has 256 neurons. Multiply 10 by 256 to get 2,560 parameters. But we have ten neurons at this layer. So add 2,560 to 10 to get 2,570 parameters.

Compile the Model

The **compile** method configures the model for training. We set the optimizer, loss function, and metrics. The **loss function** (or objective function) is the quantity minimized during training. It represents a measure of success. The **optimizer** finds parameters that minimize the given loss function.

We use the Adam optimizer. *Adam* is an adaptive learning rate method that computes learning rates for different parameters. Adam is great because it *automatically* adapts learning rate for optimum training performance.

We use the *sparse_categorical_crossentropy* loss function because our targets are integers that are mutually exclusive. That is, a digit can only be one of the ten digits.

Compile the model:

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

We use the *Adam* optimizer because it performs well. We tried other optimizer options, but this one worked pretty well for this case, and it automatically adapts learning rate.

Peruse the following URL to see the available optimizers:

www.tensorflow.org/api_docs/python/tf/keras/optimizers

Train the Model

Since train_ds and test_ds are composed of both images and labels, we include them as parameters to the *fit* method for training. We run the model for 60 epochs, which means that we pass the data to the model 60 times. We train with train_ds and validate with test_ds.

Train the model:

```
history = model.fit(train_ds, epochs=60, validation_data=(test_ds))
```

Our training accuracy is close to 97%. And test accuracy is close to 95%. Our model is overfitting a bit because test accuracy is lower than train accuracy. Due to randomization effects, your results may differ slightly.

It is always a good idea to evaluate the model for generalization purposes:

```
model.evaluate(test_ds)
```

So our model *generalizes* at about 95% with new data.

Model History

The fit method automatically keeps a record of the loss and metric values during training. This is why we assigned training to variable *history*. The *history.history* object is a dictionary that contains the training record.

Assign the training record to a variable:

```
history_dict = history.history
```

Display a list of the keys in the dictionary:

```
keys = history_dict.keys()
print ('keys:', keys, br)
```

We use the loss, accuracy, val_loss, and val_accuracy keys to refer to the training metrics.

Get the length of the dictionary so we can reference the final metric values:

```
length = len(history_dict['loss']) - 1
```

Subtract 1 from the length because Python list indexing starts at 0.

Listing 2-3 shows the final metric values.

Listing 2-3. Display the final training metrics

```
final_loss = history_dict['loss'][length]
final_loss_val = history_dict['val_loss'][length]
final_acc = history_dict['accuracy'][length]
final_acc_val = history_dict['val_accuracy'][length]
print ('final loss (train/test):')
print (final_loss, final_loss_val, br)
print ('final accuracy (train/test):')
print (final_acc, final_acc_val)
```

Since we have the training metrics, we can plot the training and validation loss as well as the training and validation accuracy as shown in Listing 2-4. Validation loss and accuracy are based on test data because it is new to the model.

Listing 2-4. Visualize train and test loss and accuracy

```
import matplotlib.pyplot as plt

acc = history_dict['accuracy']
val_acc = history_dict['val_accuracy']
loss = history_dict['loss']
val_loss = history_dict['val_loss']

epochs = range(1, len(acc) + 1)

plt.figure(figsize=(12,9))
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# clear previous figure
plt.clf()

plt.figure(figsize=(12,9))
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.ylim((0.5,1))
plt.show()
```

Import matplotlib.pyplot to enable plotting. Accuracy and loss metrics are saved in variables. The remainder of the code uses plotting methods to display the results. From the visualization, we see the training process. We recommend plotting training loss and accuracy with every neural network training exercise.

The visualization verifies that our model is overfitting because training accuracy is higher than validation (or test) accuracy. Of course, the overfitting is not drastic. The visualization also shows where training and validation metrics converge or diverge. To generalize so that a model works with new data, training and test accuracy should be as closely aligned as possible.

Predictions

Deep learning leverages algorithms to automatically model and find patterns in data with the goal of predicting target outputs or responses. If we can predict from new data, we can gain insights to help decision making.

One way to make predictions is with the *predict* method on the test data:

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

The *predictions* variable holds all of the predictions based on *test_ds*. Each prediction is represented by an array of values that provide a set of probabilities. The number of values in the array is based on the number of target classes. So each array holds ten values. The value in an array with the highest probability is the predicted digit.

Let's look at the first prediction (at index 0):

```
predictions[0]
```

It's difficult to identify the highest probability from float numbers, so let's make it easier to see:

```
predictions[0].round(2)
```

The array value with the highest probability is the prediction. Each array position represents a digit between 0 and 9. So position one is for digit 0, position two is for digit 1, and so on.

Use the following algorithm to elicit confidence in the first prediction:

```
import numpy as np

confidence = 100*np.max(predictions[0])
print (str(np.round(confidence, 2)) + '%')
```

So we are very confident that our prediction ws correct.

With this algorithm, we predict the digit based on the first image in the test dataset:

```
first_pred = np.argmax(predictions[0])
print ('predicted:', first_pred)
```

Displayed is the predicted digit based on the first image in the test dataset. Was the prediction correct?

Display the first label from the test dataset:

```
print ('actual:', y_test[0])
```

If the predicted digit is the same as the actual label, the prediction is correct! Since results may vary, we cannot be sure what digit was predicted.

Let's make predictions based on the first five images in the test dataset.

Listing 2-5 displays the first five predictions from test data and then displays the confidence we have in each prediction.

Listing 2-5. Five predictions and their confidences

```
# first five predictions based on test data:

print ('first five predictions:', end=' ')

p5 = []
for i in range(5):
  p = predictions[i]
  v = np.max(p)
  p5.append(p.tolist().index(v))

print (p5)
```

```
# confidence in first five predictions:

print ()
print ('Confidence in our predictions:', br)

c = []
for i in range(5):
  conf = str(round(100*np.max(predictions[i]), 2))
  c.append(conf)
  print (conf + '% for prediction:', p5[i])
```

We can also compare our first five predictions against the actual target values to see how well our model performed.

Listing 2-6 displays the predicted digit, actual digit, confidence in the prediction, and actual image for the first five data elements from the test dataset.

Listing 2-6. First five predictions, actual labels, confidences, and images

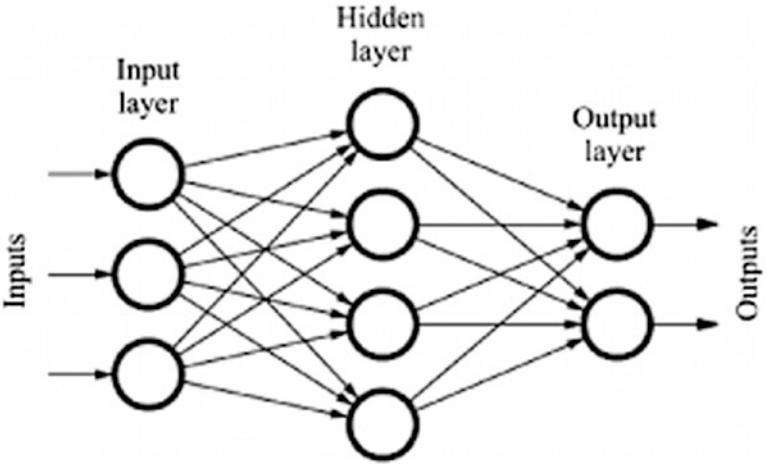
```
# first five predictions from test data

prediction_5 = [np.argmax(predictions[i])\
                for i, row in enumerate(p5)]

# display predicted digits, actual digits, confidences and images

for i, row in enumerate(prediction_5):
  print ('predicted:', target_names[row])
  print ('actual:', target_names[y_test[i]])
  print (str(c[i]) + '%')
  fig, ax = plt.subplots()
  image = ax.imshow(X_test[i], cmap='bone')
  plt.title(target_names[y_test[i]])
  plt.show()
  print (br)
```

Get an Image

To get an image for this book, just follow these simple steps:

1. Go to the GitHub URL for this book: *https://github.com/paperd/tensorflow*.
2. Locate the image you want to download and click it.
3. Click the *Download* button.
4. Right-click anywhere inside the image.
5. Click *Save image as…*.
6. Save the image on your computer.
7. Drag and drop the image to your Google Drive *Colab Notebooks* folder.

For this lesson, go to the book URL, click *chapter2*, click *figures*, click *Figure0201.png*, click the *Download* button, right-click inside the image, click *Save image as…*, and save the image on your computer. Drag and drop the image to your Google Drive *Colab Notebooks* folder.



Mount Google Drive to Display an Image

We know that Colab notebooks are saved to Google Drive. But we can also load images and other data from Google Drive into a Colab notebook. If you don't have a Google email account, create one. Just follow three simple steps:

1. Install the *Pillow* module (if necessary).
2. Mount Google Drive.
3. Point to the image and show it.

The Pillow library should already be installed, but you can install it as so:

```
!pip install Pillow
```

The *!* symbol enables us to invoke shell commands like installing a Python module in the notebook.

Run the code cell to begin the mounting process:

```
from google.colab import drive
drive.mount('/content/gdrive')
```

To continue mounting, click the URL link, choose the Google account you wish to use, click the *Allow* button, copy the authorization code, paste it in the *Enter your authorization code:* textbox, and press the *Enter* key on your keyboard. It sounds like a lot of work, but it is actually really easy. The drive is mounted to *gdrive/My Drive/Colab Notebooks*.

Now, be sure you have the image on *Google Drive*. The image is included on our GitHub site for the book. You just have to save it to your computer and drag it into a Google Drive directory. Of course, you can use any image that you wish.

We saved the image into the *Colab Notebooks* directory. You can save it anywhere you wish, but be sure to point to it correctly.

Listing 2-7 creates the path to the image and displays the image.

Listing 2-7. Display an image from Google Drive

```
# Be sure to copy the image to this directory on Google Drive
img_path = 'gdrive/My Drive/Colab Notebooks/Figure0201.png'

from PIL import Image
import matplotlib.pyplot as plt
```

```
img  = Image.open(img_path)
plt.imshow(img)
```

The image name is *Figure0201.png*. We import *Image* from the *PIL* library. Python expects PIL when accessing the Pillow module. We also import *matplotlib.pyplot*, which is a module in the Python plotting library *matplotlib*. We open the image and display it in the Colab notebook.

CHAPTER 3

Working with TensorFlow Data

We introduce TensorFlow datasets (TFDS). We discuss many facets of a TFDS with code examples. We continue with a complete TFDS modeling example.

Notebooks for chapters are located at the following URL: *https://github.com/paperd/tensorflow*.

TFDS is a collection of datasets ready to use with TensorFlow. Like all TensorFlow consumable datasets, TFDS are exposed as tf.data.Datasets, which allows us to create easy-to-use, high-performance input pipelines.

Enable the GPU (if not already enabled):

1. Click *Runtime* in the top-left menu.
2. Click *Change runtime type* from the drop-down menu.
3. Choose *GPU* from the *Hardware accelerator* drop-down menu.
4. Click *SAVE*.

Test if GPU is active:

```
import tensorflow as tf

# display tf version and test if GPU is active
tf.__version__, tf.test.gpu_device_name()
```

Import the *tensorflow* library. If '/device:GPU:0' is displayed, the GPU is active. If '.' is displayed, the regular CPU is active.

D. Paper, *TensorFlow 2.x in the Colaboratory Cloud*, https://doi.org/10.1007/978-1-4842-6649-6_3

TensorFlow Datasets (TFDS)

The following URLs provide more information about TFDS:

- *www.tensorflow.org/datasets*
- *www.tensorflow.org/datasets/overview*
- *www.tensorflow.org/datasets/catalog/overview*

Start with the first URL as it introduces TFDS. The second URL demonstrates how to display a list of all TFDS and additional technical information. The third URL shows how TFDS are categorized.

Colab Abends

When we run Google Colab for a long time (several hours) without pause or load large datasets into memory and process said data, it may crash (or abend). When this happens, you have two choices that we know of:

1. Restart all runtimes.
2. Close the program and restart it from scratch.

Available TFDS

Let's begin by displaying a list of available TFDS:

```
import tensorflow_datasets as tfds

# See available datasets
tfds.list_builders()
```

Begin by importing the *tfds* module. Use the *list_builders()* method to display. Find out how many TFDS are in the tensorflow_datasets container:

```
print (str(len(tfds.list_builders())) + ' datasets')
```

Wow! There are 244 datasets (as of this writing) we can use to practice TensorFlow.

Peruse the following URL for a really nice tutorial on TFDS:

https://colab.research.google.com/github/tensorflow/datasets/blob/master/docs/overview.ipynb

Load a TFDS

We can load a TFDS with one line of code! In the tutorial on the TFDS website, it states that we must install *tensorflow-datasets*. But, in Google Colab, we don't have to do this.

Tip If you are working in an environment other than Google Colab, you can install the tfds module by running the following code snippet: `!pip install tensorflow-datasets`

Import the tfds module:

```
import tensorflow_datasets as tfds
```

Load training data directly with tfds.load:

```
# load train set
train, info = tfds.load('mnist', split='train', with_info=True)
info
```

The **tfds.load** function loads the named dataset into a tf.data.Dataset. We add the *info* element to enable display of useful metadata about the dataset. **Metadata** is a set of data that describes and gives information about other data.

Load test data directly:

```
# load test data
test, info = tfds.load('mnist', split='test', with_info=True)
info
```

We loaded the MNIST train and test dataset from the TFDS container. The **MNIST** database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is *commonly* used for training various image processing systems. The database is also *widely* used for training and testing in the field

of machine learning. The database consists of a set of 60,000 training examples and 10,000 test examples. We included the *info* element, which provides detailed information about the dataset.

Note In machine learning vernacular, a data element is typically described as either an example or sample. The word example is used interchangeably with sample.

Although feedforward neural nets don't tend to perform well on images, MNIST is an exception because it is heavily preprocessed and the images are small. That is, MNIST images are roughly of the same small size, centered in the middle of the image space, and vertically oriented.

Extract Useful Information

The info element includes methods that allow us to extract specific information about a dataset.

Listing 3-1 extracts information about the classes.

Listing 3-1. Extracting meaningful information from a dataset

```
# create a variable to hold a return symbol
br = '\n'

# display number of classes
num_classes = info.features['label'].num_classes
class_labels = info.features['label'].names

# display class labels
print ('number of classes:', num_classes)
print ('class labels:', class_labels)
```

We just extracted the number of classes and class labels with the *features* method.

Inspect the TFDS

We have two ways to inspect an element:

1. Print the element.
2. Print the element with the *element_spec* method.

Print elements:

```
# display training and test set
print (train)
print (test)
```

Print elements with element_spec:

```
# display with element_spec method
print (train.element_spec)
print (test.element_spec)
```

Either way, the output is very similar. Tensor shapes are (28, 28, 1). So training and test data consists of 28 × 28 images. The 1 value means that images are displayed in grayscale. Image data (the feature set) is composed of tf.uint8 data, and label (or target) data is composed of tf.int64 data.

A *grayscale* image is one where the value of each pixel is a single sample representing only an amount of light. That is, it carries only intensity information. Grayscale images are composed exclusively of shades of gray. The contrast of an image ranges from black at the weakest intensity to white at the strongest.

We can also display training examples from a TFDS with one line of code:

```
# Show train feature image examples

fig = tfds.show_examples(train, info)
```

The *show_examples* method displays sample images from a tf.data.Dataset.

Feature Dictionaries

All TFDS contain *feature dictionaries* that map feature names to tensor values. By default, tfds.load returns a *dictionary* of tf.Tensors. A **tf.Tensor** represents a rectangular array of data in TensorFlow.

A typical dataset, like MNIST, has two keys: *image* and *label*. Let's inspect *one* sample with take(1). The number we feed into the *take* function renders the number of samples we receive from the dataset.

Take one sample from the train dataset and display its keys:

```
for sample in train.take(1):
  print (list(sample.keys()))
```

We see the two keys as expected. The image key references the images in the dataset. The label key references the labels in the dataset. The formal dictionary structure is represented as *{'image': tf.Tensor, 'label': tf.Tensor}*.

Now that we know the keys, we can easily display the feature shape and target value from the first train sample:

```
for sample in train.take(1):
  print ('feature shape:', sample['image'].shape)
  print ('target value: ', sample['label'].numpy())
```

The shape of the first feature sample is (28, 28, 1), and the value of the first label is 4. We used the *numpy()* method to convert the target tensor to a scalar value. Since any dataset consumed by a machine learning algorithm *must* have the **same shape**, we get the shape of a dataset from a single sample!

Let's get nine examples from the train set:

```
n, ls = 9, []
for sample in train.take(n):
  ls.append(sample['label'].numpy())
ls
```

We see [4, 1, 0, 7, 8, 1, 2, 7, 1], which matches the labels from *show_examples* in the previous section.

By using the *as_supervised=True* parameter with tfds.load, we get a tuple of (feature, label) instead of a dictionary:

```
ds = tfds.load('mnist', split='train', as_supervised=True)
ds = ds.take(1)

for image, label in ds:
  print (image.shape, br, label)
```

The sample is in the form (image, label).

We can also get a *numpy* tuple of (feature, label):

```
ds = tfds.load('mnist', split='train', as_supervised=True)
ds = ds.take(1)

for image, label in tfds.as_numpy(ds):
  print (type(image), type(label), label)
```

We use *tfds.as_numpy* to convert tf.Tensor to np.array and tf.data.Dataset to Generator[np.array].

Finally, we can get a *batched* tf.Tensor:

```
image, label = tfds.as_numpy(tfds.load(
    'mnist', split='train', batch_size=-1, as_supervised=True,
))

type(image), image.shape
```

By using *batch_size=-1*, we can load the full dataset in a single batch. We see numpy tensor (60000, 28, 28, 1), which means training data contains 60,000 28 × 28 grayscale images.

In summary, *tfds.load* returns a dictionary by default, a tuple with *as_supervised=True* of tf.Tensor, or a np.array with *tfds.as_numpy*. Be careful that your dataset can fit in memory and that all examples have the same shape.

Build the Input Pipeline

Scale, shuffle, batch, and prefetch train data:

```
train_sc = train.map(lambda items:\ (tf.cast(items['image'],\
           tf.float32) / 255.,\ items['label']))

train_ds = train_sc.shuffle(10000).batch(32).prefetch(1)
Use the train dataset we loaded earlier in the chapter. Although this looks
complicated, we use a lambda function to divide each image by 255. We then
shuffle, batch, and prefetch.
```

Prefetching improves model performance because it adds efficiency to the batching process. While our training algorithm is working on one batch, TensorFlow is working on the dataset in parallel to get the next batch ready.

Scale, batch, and prefetch test data:

```
test_sc = test.map(lambda items:\ (tf.cast(items['image'],\
          tf.float32) / 255.,\ items['label']))

test_ds = test_sc.batch(32).prefetch(1)
```

Use the test set we loaded earlier in the chapter. We don't shuffle test data because it is considered new data.

Inspect the tensors:

```
train_ds, test_ds
```

As expected, feature shapes are (None, 28, 28, 1). The first dimension is *None*, which indicates that batch size can be any value.

Build the Model

Import libraries:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Dropout
```

Create the model:

```
# clear previous model
tf.keras.backend.clear_session()

model = Sequential([
  Flatten(input_shape=[28, 28, 1]),
  Dense(512, activation='relu'),
  Dense(10, activation='softmax')
])
```

The model has an input layer, a dense hidden layer, and a dense output layer. The input layer flattens images for processing in the next layer. The hidden layer accepts data into 512 neurons. The output layer accepts data from the hidden layer into ten neurons that represent the ten digit classes.

Model Summary

Display a summary of the model:

```
model.summary()
```

The first layer accepts 28 × 28 grayscale images. So we get output shape (None, 784). *None* is the first parameter because TensorFlow models can accept any batch size. We get the second parameter *784* by multiplying 28 by 28 by 1. So each image has 784 pixels. The number of parameters is 0 because the first layer doesn't act on the data.

The second layer output shape is (None, 512) since we have 512 neurons. The number of trainable parameters is *401920*. We get 401408 by multiplying 784 neurons from the first layer by 512 neurons in this layer. We then add 512 neurons from this layer to get 401920.

The third layer output shape is (None, 10) since we have ten classes. We get *5130* by multiplying 10 by 512 neurons from the second layer and adding 10 neurons from this layer.

Compile and Train the Model

Compile the model with optimizer, loss, and metrics parameters:

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Train the model:

```
epochs = 3
history = model.fit(train_ds, epochs=epochs, verbose=1,
                    validation_data=test_ds)
```

We train the model for three epochs. That is, we pass data three times through the model. We validate the model with test data. We get pretty good results with not much overfitting with just three epochs!

Generalize on Test Data

It's always a good idea to evaluate based on test data:

```
model.evaluate(test_ds)
```

Visualize Performance

Get the training record into a variable:

```
# get training record into a variable
history_dict = history.history
```

Listing 3-2 plots accuracy and loss for the model.

Listing 3-2. Visualize training performance

```
import matplotlib.pyplot as plt

acc = history_dict['accuracy']
val_acc = history_dict['val_accuracy']
loss = history_dict['loss']
```

```
val_loss = history_dict['val_loss']

epochs = range(1, len(acc) + 1)

plt.figure(figsize=(12,9))
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.ylim((0.5,1))
plt.show()

# clear previous figure

plt.clf()

plt.figure(figsize=(12,9))
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

The visualization shows that our model fits data pretty well because train and test accuracy are very well aligned!

DatasetBuilder (tfds.builder)

Since tfds.load is really just a thin convenience wrapper around DatasetBuilder, we can build an input pipeline with the MNIST dataset directly with tfds.builder. **DatasetBuilder** is an abstract base class for all datasets. That is, every TensorFlow dataset is exposed as a DatasetBuilder.

DatasetBuilder performs several duties for TensorFlow datasets:

- Where to download data from and how to extract it and write it to a standard format with *DatasetBuilder.download_and_prepare*
- How to load data from disk with *DatasetBuilder.as_dataset*
- All information about data including names, types, feature shapes, number of records in each train and test split, and source URLs with *DatasetBuilder.info*
- Ability to directly instantiate any DatasetBuilder with *tfds.builder*

Unlike tfds.load, however, we have to manually fetch the DatasetBuilder by name, call *download_and_prepare()*, and *call as_dataset()*. The advantage of *tfds.builder* is that it allows more control over the loading process should we need it.

Listing 3-3 shows how to load MNIST with tfds.builder.

Listing 3-3. Load MNIST with tfds.builder

```
mnist_builder = tfds.builder('mnist')
mnist_info = mnist_builder.info
mnist_builder.download_and_prepare()
datasets = mnist_builder.as_dataset()
```

Begin with *tfds.builder* to create the dataset. Include information with the *info* method. Process data with the *download_and_prepare* method. Place the processed dataset into a variable.

Build the train and test sets:

```
mnist_train, mnist_test = datasets['train'], datasets['test']
```

Use the feature dictionary to get critical information:

```
for sample in mnist_train.take(1):
  print ('feature shape:', sample['image'].shape)
  print ('target value: ', sample['label'].numpy())
```

We see that the first feature has a shape of (28, 28, 1) and target value of 4.

MNIST Metadata

Like tfds.load, tfds.builder can access metadata about MNIST:

```
mnist_info
```

We see a lot of useful information about MNIST.
Access feature information:

```
mnist_info.features
```

We see useful information about images and labels.
Listing 3-4 displays the number of classes and class labels.

Listing 3-4. Number of classes and class labels

```
# display number of classes
num_classes = mnist_info.features['label'].num_classes
class_labels = mnist_info.features['label'].names

# display class labels
print ('number of classes:', num_classes)
print ('class labels:', class_labels)
```

Access shapes and datatypes:

```
print (mnist_info.features.shape)
print (mnist_info.features.dtype)
```

Access image information:

```
print (mnist_info.features['image'].shape)
print (mnist_info.features['image'].dtype)
```

Access label information:

```
print (mnist_info.features['label'].shape)
print (mnist_info.features['label'].dtype)
```

Train and test splits:

```
print (mnist_info.splits)
```

Available split keys:

```
print (list(mnist_info.splits.keys()))
```

Number of train and test examples:

```
print (mnist_info.splits['train'].num_examples)
print (mnist_info.splits['test'].num_examples)
```

Note tfds.load has access to the same metadata as tfds.builder.

Show Examples

As demonstrated earlier in the chapter, *tfds.show_examples* allows us to conveniently visualize images (and labels) from an image classification dataset.

Let's show examples from the test set:

```
fig = tfds.show_examples(mnist_test, info)
```

Prepare DatasetBuilder Data

Prepare the input pipeline for DatasetBuilder train and test data.

Scale, shuffle, batch, and prefetch train data:

```
train_sc = mnist_train.map(lambda items:\
                           (tf.cast(items['image'], \ tf.float32)
                           / 255.,\ items['label']))

train_build = train_sc.shuffle(1024).batch(128).prefetch(1)
```

Scale, batch, and prefetch test data:

```
test_sc = mnist_test.map(lambda items:\
                         (tf.cast(items['image'],\ tf.float32) / 255.,\
                          items['label']))
```

```
test_build = test_sc.batch(128).prefetch(1)
```

Inspect tensors:

```
train_build, test_build
```

As expected, feature shapes are (None, 28, 28, 1).

Build the Model

Create the model:

```
tf.keras.backend.clear_session()
```

```
model = Sequential([
  Flatten(input_shape=[28, 28, 1]),
  Dense(512, activation='relu'),
  Dense(10, activation='softmax')
])
```

Compile the Model

Compile:

```
model.compile(loss='sparse_categorical_crossentropy',
              optimizer='adam', metrics=['accuracy'])
```

Train the Model

Train:

```
model.fit(train_build, epochs=3, validation_data=test_build)
```

As expected, results are very similar to training with tfds.load.

Generalize on Test Data

Evaluate based on test data:

```
model.evaluate(test_build)
```

Load CIFAR-10

Let's use the DatasetBuilder method tfds.builder to manipulate another dataset and show some examples. The **CIFAR-10** dataset consists of 60,000 32 × 32 color images in ten classes, with 6000 images per class. There are 50,000 training images and 10,000 test images.

The ten classes are

```
[airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck]
```

The classes are completely mutually exclusive. There is no overlap between automobiles and trucks. Class *automobile* includes sedans, SUVs, and other items of that sort. Class *truck* includes only big trucks. Neither includes pickup trucks.

The dataset is divided into five training batches and one test batch, each with 10,000 images. The test batch contains exactly 1,000 randomly selected images from each class. Training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, training batches contain exactly 5,000 images from each class. Since CIFAR-10 is preprocessed with batches, we can use this information to better batch data for the training model.

Listing 3-5 loads and processes the dataset for TensorFlow consumption.

Listing 3-5. Prepare the CIFAR-10 dataset for TensorFlow consumption

```
cifar10_builder = tfds.builder('cifar10')
cifar10_info = cifar10_builder.info
cifar10_builder.download_and_prepare()
cifar10_train = cifar10_builder.as_dataset(split='train')
cifar10_test = cifar10_builder.as_dataset(split='test')
```

Inspect the train set:

```
cifar10_train
```

We see that image tensors have shape (32, 32, 3). And label tensors have shape (). So each image is represented by 32 × 32 pixels. The *3* value means that images are in color. Image tensors are datatype tf.uint8. Each label is a scalar value. Label tensors are datatype tf.int64.

TensorFlow leverages the RGB color model to produce color images. The **RGB color model** is an additive color model where red, green, and blue lights are added together in various ways to reproduce a broad array of colors. The name of the model comes from the initials of the three additive primary colors, namely, red, green, and blue.

Inspect the Dataset

Get information about the dataset:

```
cifar10_info
```

Access feature information:

```
cifar10_info.features
```

Get class names:

```
cifar10_info.features['label'].names
```

Get available split keys:

```
print (list(cifar10_info.splits.keys()))
```

Show train examples:

```
fig = tfds.show_examples(cifar10_train, info)
```

Use the feature dictionary to display train labels:

```
[sample['label'].numpy() for sample in cifar10_train.take(9)]
```

To simplify coding, we used list comprehension.

For completeness, show test examples:

```
fig = tfds.show_examples(cifar10_test, info)
```

Prepare the Input Pipeline

Scale, shuffle, batch, and prefetch train data:

```
train_sc = cifar10_train.map(lambda items:\
                             (tf.cast(items['image'],\ tf.float32) / 255.,\
                              items['label']))

train_cd = train_sc.shuffle(1024).batch(128).prefetch(1)
```

Scale, batch, and prefetch test data:

```
test_sc = cifar10_test.map(lambda items:\
                           (tf.cast(items['image'],\ tf.float32) / 255.,\
                            items['label']))

test_cd = test_sc.batch(128).prefetch(1)
```

Inspect tensors:

```
train_cd, test_cd
```

Model the Data

Create the model:

```
tf.keras.backend.clear_session()

model = Sequential([
  Flatten(input_shape=[32, 32, 3]),
  Dense(512, activation='relu'),
  Dense(10, activation='softmax')
])
```

We *must* get the input shape correct. For CIFAR-10, input shape is (32, 32, 3)!

Inspect the model:

```
model.summary()
```

The first layer accepts and flattens 32 × 32 color images. So we get output shape (None, 3072). *None* is the first parameter because TensorFlow models can accept any batch size. We get the second parameter *3072* by multiplying 32 by 32 by 3. Each image has 1,024 pixels, which is the result of multiplying 32 by 32. Since each image is in color, we multiply 1,024 by 3 to get 3,072 neurons. The number of parameters is 0 because this is the first layer.

The second layer output shape is (None, 512) since we have 512 neurons. The number of parameters is *1573376*. We get 1572864 by multiplying 3,072 neurons from the first layer by 512 neurons in this layer. We then add 512 neurons from this layer to get 1,573,376.

The third layer output shape is (None, 10) since we have ten classes. We get *5130* by multiplying 10 by 512 neurons from the second layer and adding 10 neurons from this layer.

Compile the model:

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Train the model:

```
epochs = 3
history = model.fit(train_cd, epochs=epochs, verbose=1,
                    validation_data=test_cd)
```

Accuracy below 50% is not good. Our model performed very poorly because feedforward neural networks are not designed to work well with image data.

Our goal was to show you how to load and model a TFDS. We introduce models that work well with image data in a later chapter.

CHAPTER 4

Working with Other Data

In the previous chapter, we showed you how to work with a TFDS. But what if we want to work with another type of dataset? In this chapter, we show you how to work with other types of data with TensorFlow.

Notebooks for chapters are located at the following URL: *https://github.com/paperd/tensorflow*.

Enable the GPU (if not already enabled):

1. Click *Runtime* in the top-left menu.
2. Click *Change runtime type* from the drop-down menu.
3. Choose *GPU* from the *Hardware accelerator* drop-down menu.
4. Click *SAVE*.

Test if GPU is active:

```
import tensorflow as tf

# display tf version and test if GPU is active
tf.__version__, tf.test.gpu_device_name()
```

Import the *tensorflow* library. If '/device:GPU:0' is displayed, the GPU is active. If '.' is displayed, the regular CPU is active.

Basic Mechanics

To create an input pipeline, we start with a data source. To construct a dataset from data in memory that TensorFlow can work with, we can use either tf.data.Dataset.from_tensor_slices() or tf.data.Dataset.from_tensors(). The *from_tensor_slices* method creates a dataset

D. Paper, *TensorFlow 2.x in the Colaboratory Cloud*, https://doi.org/10.1007/978-1-4842-6649-6_4

with a separate element for each piece of the input tensor. The *from_tensors* method combines the input and returns a dataset with a single element. We work *exclusively* with the from_tensor_slices method because it enables us to work with each data element.

Create a simple 1D tensor and make it consumable for TensorFlow:

```
# create a 1D tensor
ds = tf.data.Dataset.from_tensor_slices([8, 3, 0, 8, 2, 1])
ds.element_spec
```

We create a TensorSpec from the six-element list and make it TensorFlow consumable with the from_tensor_slices method. The shape is () because the from_tensor_slices creates an object from the slices of an array.

Listing 4-1 demonstrates how to iterate the dataset.

Listing 4-1. Iterate and display the from_tensor_slices dataset

```
# iterate and display tensor values
for elem in ds:
  print(elem.numpy())

print ()

# iterate without numpy method
for elem in ds:
  print(elem)
```

The dataset has six tensors. The first loop displays each element in the tensor as numpy values. The second loop displays the raw tensors.

We can also create a Python iterator using *iter* and consume its elements with the *next* method:

```
it = iter(ds)

# display the first element
next(it).numpy()
```

We see the first element from the tensor displayed.

To view the remaining elements, just keep running the following:

```
next(it).numpy()
```

Now, create a tensor with tf.data.Dataset_from_tensors():

```
# create a 1D tensor
ds = tf.data.Dataset.from_tensors([8, 3, 0, 8, 2, 1])
ds.element_spec
```

Notice that the shape is (6,), which means that the single tensor is composed of six elements.

Let's iterate the single tensor as shown in Listing 4-2.

Listing 4-2. Iterate and display the from_tensors dataset

```
# iterate and display tensor values
for elem in ds:
  print(elem.numpy())

print ()

# iterate without numpy method
for elem in ds:
  print(elem)
```

We have a dataset with a *single* tensor that contains six elements.

TensorFlow Dataset Structure

A dataset contains elements that each have the same nested structure, and the individual components of the structure can have any type representable by tf.TypeSpec, including tf.Tensor, tf.sparse.SparseTensor, tf.RaggedTensor, tf.TensorArray, or tf.data.Dataset. The *Dataset.element_spec* property allows us to inspect the type of each element component. The property returns a nested structure of *tf.TypeSpec* objects that match the structure of the element. The element may be a single component, a tuple of components, or a nested tuple of components.

We can better understand the structure with an example as shown in Listing 4-3.

Listing 4-3. Data structure example

```
br = '\n'  # enter a line break in Colab

# create random uniform numbers
scope = tf.random.uniform([4, 10])

print ('shape:', scope.shape, br)

ds = tf.data.Dataset.from_tensor_slices(scope)

print (ds.element_spec, br)
# Let's look at the first element:

it = iter(ds)

# print first element
print ('first element with an iterator:', br)
print (next(it).numpy(), br)

print ('all four elements:', br)
for i, row in enumerate(ds):
  print ('element ' + str(i+1))  # add 1 as index starts at 0
  print (row.numpy(), br)
```

The shape of *scope* is (4, 10), which means that we have a tensor with four elements, each of which contains ten randomly generated uniformly distributed numbers between 0 and 1. We generate a TensorFlow consumable dataset from scope with the from_tensor_slices method and display the TensorSpec with the element_spec method. We continue by displaying each element from the TensorSpec.

Note Along with example and sample, the term *element* is also used to describe a tensor in a dataset.

Simply, we create a dataset containing four elements. Each element contains ten random uniform numbers between 0 and 1. We convert the dataset to one that TensorFlow can consume (or work with). We display each element by iterating the TensorFlow dataset.

Reading Input Data

If all of your input data fits in memory, the simplest way to create a dataset for TensorFlow consumption is to convert it to a tf.Tensor object with Dataset.from_tensor_slices().

Colab Abends

As previously noted, when running Google Colab for a long time (e.g., several hours) without pause or loading large datasets into memory and processing said data, it may crash (or abend). When this happens, you have two choices that we know of:

1. Restart all runtimes.
2. Close the program and restart it from scratch.

To restart all runtimes, click *Runtime* in the top menu, click *Restart runtime* from the drop-down menu, and click *YES* when prompted. Google Colab recommends this option. If you restart from scratch, clear browser history first and then start Google Colab from scratch.

Batch Size

Batch size is the number of training examples processed by the neural network model in one pass. Don't get batch size confused with epochs! An **epoch** is a complete pass through the training dataset. So the number of epochs is the number of complete passes through the training dataset. Epochs have nothing to do with the processing of training examples! They just represent the number of passes through the network. Simply, during each pass (or epoch) through the network, batches of the training dataset are processed.

The size of a batch must be more than or equal to one and less than or equal to the number of samples in the training dataset. This makes sense because you can't have a batch that is bigger than the total number of training examples.

TensorFlow is *optimized* to run batches of training data much greater than one. So a batch size of one is very inefficient! Since a batch size of one represents the entire training dataset, we are not really batching data. So unless you don't want to batch data, don't use a batch size of one!

Keras Data

The *tf.keras.datasets* module provides seven preprocessed datasets for practicing TensorFlow. Peruse the following URL to get more information about the datasets: *https://keras.io/api/datasets/*.

Let's grab the Keras MNIST dataset:

```
train, test = tf.keras.datasets.mnist.load_data(path='mnist.npz')
```

Both train and test data contain MNIST images and labels in a tuple:

```
type(train), type(test)
```

Explore the shape of train data:

```
print ('train data:', br)
print (train[0].shape)
print (train[1].shape)
```

Explore the shape of test data:

```
print ('test data:', br)
print (test[0].shape)
print (test[1].shape)
```

Train data consists of 60,000 28 × 28 feature images and 60,000 labels. Test data consists of 10,000 28 × 28 feature images and 10,000 labels.

Build the Input Pipeline

Split train and test data into their respective images and labels. Scale image data. Create TensorFlow consumable data with the from_tensor_slices method.

Begin with train data:

```
train_images, train_labels = train
train_sc = train_images / 255  # divide by 255 to scale

train_k = tf.data.Dataset.from_tensor_slices(
    (train_sc, train_labels))
train_k.element_spec
```

Continue with test data:

```
test_images, test_labels = test
test_sc = test_images / 255  # divide by 255 to scale

test_k = tf.data.Dataset.from_tensor_slices(
    (test_sc, test_labels))
test_k
```

Shuffle train data, batch, and prefetch train and test data:

```
BATCH_SIZE = 128
SHUFFLE_BUFFER_SIZE = 1000

train_kd = train_k.shuffle(
    SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE).prefetch(1)
test_kd = test_k.batch(BATCH_SIZE).prefetch(1)
```

Inspect tensors:

```
train_kd, test_kd
```

Create the Model

Import libraries:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Dropout
```

Clear previous models from memory:

```
# clear any previous models
tf.keras.backend.clear_session()
```

Build the model:

```
model = Sequential([
  Flatten(input_shape=[28, 28]),
  Dense(512, activation='relu'),
  Dropout(0.5),
  Dense(10, activation='softmax')
])
```

The model is a feedforward neural network with densely connected layers. That is, all neurons see the data. The first layer accepts 28 × 28 images and flattens each image into a 1D array consisting of 784 pixels. The second layer accepts data into 512 neurons and uses *relu* activation to minimize loss. The third layer uses dropout to mitigate overfitting. The fourth layer is the output layer. It accepts data into ten neurons because our data has ten class labels. It uses *softmax* activation to reduce loss.

Dropout is a regularization technique (patented by Google) for reducing overfitting in neural networks. The technique works by dropping out units in a neural network.

Model Summary

Display a summary of the model:

```
model.summary()
```

Output shape of the first layer is (None, 784). *None* is the first parameter because TensorFlow models accept any batch size. We get the second parameter of 784 by multiplying 28 by 28 image pixels because we want flattened images. We have 0 parameters because this layer accepts input shape but doesn't act on the data.

Output shape of the second layer is (None, 512). The second parameter is 512 because this is the number of neurons set for this layer. The number of parameters is 401,920 derived by multiplying 512 (neurons at this layer) by 784 (neurons in the previous layer) and adding 512 to account for neurons at this layer.

Output shape of the third layer is (None, 512). Using dropout doesn't impact the neuron count. So this layer inherits (None, 512) from the previous layer and has no parameters.

Output shape of the fourth layer is (None, 10). The second parameter is 10 to reflect the number of classes. The number of parameters is 5,130 derived by multiplying 10 (neurons at this layer) by 512 (neurons in the previous layer) and adding 10 to account for neurons at this layer.

Compile the Model

Compile:

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Train the Model

Train:

```
epochs = 10
history = model.fit(train_kd, epochs=epochs, verbose=1,
                    validation_data=test_kd)
```

Since epochs is set at 10, our model processes the dataset *ten times*. Since train and test accuracy are closely aligned, we don't have much overfitting.

Scikit-Learn Data

We can also read data from the scikit-learn library. *Scikit-learn* is a free software machine learning library for the Python programming language.

Grab a dataset from this library:

```
from sklearn.datasets import fetch_lfw_people

faces = fetch_lfw_people(min_faces_per_person=70, resize=0.4)
```

The *fetch_lfw_people* dataset is a collection of JPEG pictures of famous people collected over the Internet. All details are available on the official website: *http://vis-www.cs.umass.edu/lfw/*. Each picture is centered on a single face. The typical machine learning task is face verification. So, given a pair of pictures, we predict whether the two images are of the same person.

An alternative machine learning task is face recognition (or face identification). So, given the picture of the face of an unknown person, we identify the name of the person by referring to a gallery of previously seen pictures of identified persons.

Explore the Data

Display the keys:

```
# get available keys from dataset
faces.keys()
```

The dataset contains feature images (in the images container) and targets (in the target container). It also contains target names and a description of the dataset. The data container consists of flattened vectors of each image.

Listing 4-4 displays shapes, target names, and class labels.

Listing 4-4. Information about the dataset

```
image, target = faces.images, faces.target
data = faces.data
names = faces.target_names

print ('feature image tensor:', br)
print (image.shape, br)
print ('target tensor:', br)
print (target.shape, br)
print ('flattened image tensor:', br)
print (data.shape, br)
print ('target names:', br)
print (names, br)
print ('class labels:', len(names))
```

The feature image tensor consists of 1,288 50 × 37 face images. The target tensor consists of 1,288 targets. The data shape consists of 1,288 flattened vectors each with 1,850 elements. We get 1,850 by multiplying 50 by 37.

Listing 4-5 explores the first example.

Listing 4-5. Explore the first example

```
# display the first data example

i = 0

print ('first image example:', br)
print (image[i], br)

print ('first target example:', br)
print (target[i], br)

print ('name of first target:', br)
print (names[target[i]], br)

print ('first data example (flattened image):', br)
print (data[i], br)
```

```
print ('first image:', br)

import matplotlib.pyplot as plt

# display the first image in the dataset
plt.imshow(image[i], cmap='bone')
plt.title(names[target[i]])
```

Each image is represented by a 2D matrix. The target value for the first image is 5, which just happens to be an image of Hugo Chavez.

Build the Input Pipeline

Create train and test sets. Scale feature images as shown in Listing 4-6.

Listing 4-6. Create train and test sets and scale feature images

```
from sklearn.model_selection import train_test_split

# create train and test data
X_train, X_test, y_train, y_test = train_test_split(
    image, target, test_size=0.33, random_state=0)

# scale feature image data and create TensorFlow tensors
x_train = X_train / 255.0
x_test = X_test / 255.0

# get shapes
print ('x_train shape:', end=' ')
print (x_train.shape)
print ('x_test shape:', end=' ')
print (x_test.shape)

# get sample entries
print (x_train[0])
print (X_train[0][0][0])
print (x_train[0][0][0])
```

The *train_test_split* module provides an easy way to split a dataset into train and test sets. We also have the flexibility to set the train and test size to our needs. The *random_state* parameter provides a way to reproduce our results.

Our train set contains 67% of the data. The remaining 33% is placed in the test set. The training feature data shape is (862, 50, 37), and the test feature data shape is (426, 50, 37). So training feature data consists of 862 50 × 37-pixel images, and test feature data consists of 426 50 × 37-pixel images.

Divide images by 255 to scale them. Scaling makes vector operations simpler.

Display the first pixel image from the train set. Display the first pixel from the first pixel image and display its scaled value.

Continue by slicing data into TensorFlow consumable pieces:

```
faces_train = tf.data.Dataset.from_tensor_slices(
    (X_train, y_train))
faces_test = tf.data.Dataset.from_tensor_slices(
    (X_test, y_test))
```

Set batch and buffer sizes:

```
BATCH_SIZE = 16
SHUFFLE_BUFFER_SIZE = 100
```

Shuffle train data, batch, and prefetch train and test data:

```
faces_train_ds = (faces_train
                  .shuffle(SHUFFLE_BUFFER_SIZE)
                  .batch(BATCH_SIZE).prefetch(1))
faces_test_ds = (faces_test
                 .batch(BATCH_SIZE).prefetch(1))
```

Inspect tensors:

```
faces_train_ds, faces_test_ds
```

Build the Model

Build a simple model and train the data as shown in Listing 4-7.

Listing 4-7. Model for faces data

```
import numpy as np

class_labels = len(names)

# clear previous model and generate a seed
tf.keras.backend.clear_session()
np.random.seed(0)
tf.random.set_seed(0)

model = Sequential([
  Flatten(input_shape=[50, 37]),
  Dense(16, activation='relu'),
  Dense(class_labels, activation='softmax')
])
```

Import the numpy library so we can generate a random seed. A **random seed** specifies the *starting point* when a computer generates a random number sequence. We set a random seed to ensure the same starting point so we can obtain reproducible results. That is, each time we train the model, we get the same results. We set the seed to 0. Of course, you can set the seed to any number, but be sure to use the same number each time.

The input shape reflects the image size of feature data. That is, each image is represented by 50 × 37 pixels.

Model Summary

Run a model summary:

```
model.summary()
```

Output shape of the first layer is (None, 1850). We get 1,850 by multiplying 50 by 37. The number of parameters is 0 because this is the first layer.

Output shape of the second layer is (None, 16). We have 16 neurons acting on the data at this layer. The number of parameters is 29,616. We get 29,600 by multiplying 16 by 1,850. We get 29,616 by adding 16 to 29,600 to account for the number of neurons at this layer.

Output shape of the third layer is (None, 7). We have seven neurons at this layer to account for the seven class labels. The number of parameters is 119. We get 112 by multiplying 16 by 7. We get 119 by adding 7 to 112 to account for the number of neurons at this layer. The None parameter is present because TensorFlow accepts any batch size.

Compile the Model

Compile:

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Train the Model

Train:

```
history = model.fit(faces_train_ds, epochs=10,
                    validation_data=faces_test_ds)
```

Performance is not good because feedforward nets don't work well with image data. However, we create this simple model to show you how to train the dataset.

Numpy Data

We can load numpy data directly and scale it as shown in Listing 4-8.

Listing 4-8. Load numpy data from a URL

```
DATA_URL = 'https://storage.googleapis.com/\
tensorflow/tf-keras-datasets/mnist.npz'

path = tf.keras.utils.get_file('mnist.npz', DATA_URL)
with np.load(path) as data:
  train_examples = data['x_train']
  train_labels = data['y_train']
  test_examples = data['x_test']
  test_labels = data['y_test']

train_scaled = train_examples / 255.
test_scaled = test_examples / 255.
```

Identify a URL path to a numpy file. Use tf.keras.utils.get_file to access the path. Load numpy data with the *np.load* function. Split data into train and test sets. Scale feature image data.

Load Numpy Arrays with tf.data.Dataset

Convert train and test images and labels to a TensorFlow consumable form:

```
train_dataset = tf.data.Dataset.from_tensor_slices(
    (train_scaled, train_labels))
test_dataset = tf.data.Dataset.from_tensor_slices(
    (test_scaled, test_labels))
```

Prepare Data for Training

Shuffle train data, batch, and prefetch train and test data:

```
BATCH_SIZE = 128
SHUFFLE_BUFFER_SIZE = 1000

train_np = train_dataset.shuffle(
    SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE).prefetch(1)
test_np = test_dataset.batch(BATCH_SIZE).prefetch(1)
```

Inspect tensors:

```
train_np, test_np
```

Create the Model

Clear previous sessions and set random seeds:

```
tf.keras.backend.clear_session()
np.random.seed(0)
tf.random.set_seed(0)
```

Use the same model we used with Keras MNIST:

```
model = Sequential([
  Flatten(input_shape=[28, 28]),
  Dense(512, activation='relu'),
  Dropout(0.5),
  Dense(10, activation='softmax')
])
```

Model Summary

Since we used the same model, the summary is the same as before:

```
model.summary()
```

Compile the Model

Compile:

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Train the Model

Train:

```
epochs = 10
history = model.fit(train_np, epochs=epochs, verbose=1,
                    validation_data=test_np)
```

Get Wine Data from GitHub

You can access any freestanding dataset for this book directly from GitHub with a few simple steps:

1. Visit the book URL: *https://github.com/paperd/tensorflow*.
2. Locate the dataset and click it.

3. Click the *Raw* button.
4. Copy the URL to Colab and assign it to a variable.
5. Read the dataset with the Pandas *read_csv* method.

For our purposes, a *freestanding dataset* is one that is not stored in a software environment such as a TFDS. Let's read a dataset from GitHub.

In our case, visit the book URL, click *chapter4*, click *data*, click *winequality-red.csv*, click the *Raw* button, copy the URL, paste it into a code cell in Colab, and assign it to a variable.

We've already located the URL and assigned it to a variable:

```
url = 'https://raw.githubusercontent.com/paperd/tensorflow/\
master/chapter4/data/winequality-red.csv'
```

Read the dataset into a pandas dataframe:

```
import pandas as pd

wine = pd.read_csv(url, sep = ';')
```

Verify that data was read properly:

```
wine.head()
```

CSV Data

A *CSV* dataset is a comma-separated values file that allows data to be saved in a tabular format. When opened in a program like Microsoft Excel, it looks like a conventional spreadsheet but with a *.csv* extension. CSV files can be used with any spreadsheet program that we know of including Microsoft Excel or Google Sheets.

A great place to get CSV datasets for machine learning is the UCI Machine Learning Repository. The main site for the repository is located at the following URL: *https://archive.ics.uci.edu/ml/datasets.php*.

A CSV dataset commonly cited in machine learning literature is *winequality-red.csv*. This dataset contains red variants of the Portuguese *Vinho Verde* wine. For detailed information, peruse the following article:

P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis. Modeling wine preferences by data mining from physicochemical properties. In Decision Support Systems, Elsevier, 2009, 47(4):547–553. ISSN: 0167-9236.

For general information about the dataset, peruse the following URL:

https://archive.ics.uci.edu/ml/datasets/wine+quality

Dataset Characteristics

The dataset consists of eleven independent feature variables and one target variable. Feature variables include

- Fixed acidity
- Volatile acidity
- Citric acid
- Residual sugar
- Chlorides
- Free sulfur dioxide
- Total sulfur dioxide
- Density
- pH
- Sulfates
- Alcohol

The target variable is

- Quality

The target variable (quality) can take on a score between 0 and 10. A score of 0 means that the quality is very low. A score of 10 means that the quality is very high. The dataset contains 1,599 examples.

Get Data

It is a good idea to know how to extract a dataset directly from the UCI Machine Learning Repository. Let's do this now.

Identify the dataset URL:

```
url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/
wine-quality/winequality-red.csv'
```

Establish the path to the dataset:

```
path = tf.keras.utils.get_file('winequality-red.csv', url)
path
```

Create a pandas dataframe from the CSV file and place it into a Python variable:

```
import pandas as pd
```

```
data = pd.read_csv(path, sep = ';')
```

View records from the beginning of the dataframe:

```
data.head()
```

View records from the end of the dataframe:

```
data.tail()
```

Identify class labels that appear in the dataset:

```
data.quality.unique()
```

Since quality is the target, we use the *unique()* function to pull distinct labels from the dataset. The dataset contains scores 3, 4, 5, 6, 7, and 8. However, quality can take on scores between 0 and 10, so there are *eleven* possible class labels.

Display datatypes:

```
data.dtypes
```

Features are datatype float64, and the target is datatype int64.

Display the number of examples in the dataset:

```
len(data)
```

We have 1,599 examples.

Split Data into Train and Test Sets

Create the target set:

```
# create a copy of the dataframe
df = data.copy()

# create the target
target = df.pop('quality')
```

Create a copy of the dataframe to preserve the original data. Pop the target column into a variable to create the target dataset. The *pop* method permanently removes the column from the copy.

Show what happened to the dataframe copy:

```
df.head()
```

Notice that the quality column is no longer in the dataframe.

Convert the dataframe to numpy values:

```
features = df.values
labels = target.values
```

Split into train and test sets, and scale feature data as shown in Listing 4-9.

Listing 4-9. Split data into train and test sets and scale feature data

```
X_train, X_test, y_train, y_test = train_test_split(
    features, labels, test_size=0.33, random_state=0)

# scale feature image data and create TensorFlow tensors
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_std = scaler.fit_transform(X_train)
X_test_std = scaler.fit_transform(X_test)
```

Feature data are not images. They are scalar values. So we use the *StandardScaler* module to transform continuous feature data to *mean of 0* and *standard deviation of 1* before applying machine learning techniques.

Prepare Data for TensorFlow Consumption

Slice train and test sets into tf.Data.Dataset data:

```
train_wine = tf.data.Dataset.from_tensor_slices(
    (X_train_std, y_train))
test_wine = tf.data.Dataset.from_tensor_slices(
    (X_test_std, y_test))
```

Define the variable to hold line break:

```
br = '\n'
```

Create a function to view tensors:

```
def see_samples(data, num):
  for feat, targ in data.take(num):
    print ('Features: {}'.format(feat))
    print ('Target: {}'.format(targ), br)
```

View the first three tensors:

```
n = 3
see_samples(train_wine, n)
```

Define batch and buffer sizes:

```
BATCH_SIZE = 16
SHUFFLE_BUFFER_SIZE = 100
```

Shuffle train data, batch, and prefetch train and test data:

```
train_wine_ds = (train_wine.shuffle(
    SHUFFLE_BUFFER_SIZE).
    batch(BATCH_SIZE).
    prefetch(1))
test_wine_ds = (test_wine.batch(
    BATCH_SIZE).
    prefetch(1))
```

Inspect tensors:

```
train_wine_ds, test_wine_ds
```

Build the Model

Clear sessions and generate seeds:

```
tf.keras.backend.clear_session()
np.random.seed(0)
tf.random.set_seed(0)
```

Create the model:

```
model = Sequential([
  Dense(30, activation='relu', input_shape=[11,]),
  Dense(11, activation='softmax')
])
```

The input shape is [11,] because the dataset has eleven features. The output shape is eleven because the dataset has eleven class labels.

Model Summary

Inspect the model:

```
model.summary()
```

Output shape of the first layer is (None, 30) because we have 30 neurons at this layer. The number of parameters is 360. We get 330 by multiplying 30 neurons by 11 features. We get 360 by adding 30 neurons at this layer to 330. Output shape of the second layer is (None, 11) because we have 11 neurons at this layer to account for the number of possible labels. The number of parameters is 341. We get 330 by multiplying 30 neurons from the previous layer by 11 neurons at this layer. We get 341 by adding 11 neurons at this layer to 330.

Compile the Model

Compile:

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Train the Model

```
history = model.fit(train_wine_ds, epochs=10,
                    validation_data=test_wine_ds)
```

We don't get very good performance, but we just wanted to show you how to train a CSV dataset.

Get Abalone Data from GitHub

In the next section, we work with the abalone dataset. We show you how to load it from GitHub as an alternative.

We've already located the appropriate URL and assigned it to a variable:

```
url = 'https://raw.githubusercontent.com/paperd/tensorflow/\
master/chapter4/data/abalone.data'
```

Read the dataset into a pandas dataframe:

```
# add column headings
cols = ['Sex', 'Length', 'Diameter', 'Height', 'Whole',
        'Shucked', 'Viscera', 'Shell', 'Rings']

abd = pd.read_csv(url, names=cols)
```

Verify data:

```
abd.head()
```

Data Datasets

The only difference between data with the *.data* extension and CSV data is the file extension. But we can work with it in the same manner. Instead of loading it from GitHub, we can download the dataset from the UCI repository, copy it to Google Drive, and access it from Colab.

Abalone Dataset

The abalone dataset is type DATA. The general site for the data is

https://archive.ics.uci.edu/ml/datasets/Abalone

To download abalone data, go to the URL, click *Data Folder*, and click *abalone.data*. The dataset is automatically downloaded to your *Downloads* directory. Since we are working with the Colab cloud service, copy the file to the *Colab Notebooks* directory on your *Google Drive*. The easiest way to copy is to drag and drop the file from your Downloads directory into Google Drive.

The other file of interest is abalone.names, which provides an in-depth description of the dataset. Like the data file, it can be accessed from the Data Folder by clicking *abalone.names*.

Dataset Characteristics

The dataset is used for predicting age of abalone shells from physical measurements. The age of an abalone shell is partially determined by cutting through the cone, staining it, and counting the number of rings through a microscope. Other measurements supplement age prediction.

Feature variables include

- Sex
- Length
- Diameter
- Height
- Whole
- Shucked
- Viscera
- Shell

The target variable is

- Rings

The target variable rings can take on a score between 1 and 29. Such scores represent the number of rings for an abalone shell. So the *number of rings* is the value to predict. An interesting notion about this dataset is that we can use it either as a continuous value experiment or as a classification problem.

Mount Google Drive to Colab

If we want to load data files directly from Google Drive, we must mount Colab to Google Drive:

```
from google.colab import drive
drive.mount('/content/gdrive')
```

After you execute the code snippet, click the URL, choose a Google account, click the *Allow* button, copy the authorization code and paste it into the textbox *Enter your authorization code:*, and press the *Enter* key on your keyboard.

Note Be sure that you have the file in the Colab Notebooks directory on your Google Drive!

Establish the path in Colab:

```
# establish path (be sure to copy file to Google Drive)

path = 'gdrive/My Drive/Colab Notebooks/'
abalone = path + 'abalone.data'
abalone
```

Read Data

Since the dataset doesn't contain column headings, we need to define them before reading the dataset:

```
cols = ['Sex', 'Length', 'Diameter', 'Height', 'Whole',
        'Shucked', 'Viscera', 'Shell', 'Rings']

ab_data = pd.read_csv(abalone, names=cols)
```

Now we have the same dataset that we loaded directly from GitHub.

Explore Data

Display records from the beginning of the dataset:

```
ab_data.head(3)
```

Display records from the end of the dataset:

```
ab_data.tail(3)
```

Return the number of records:

```
len(ab_data)
```

We have 4,177 records.

Display the output classes used in the dataset:

```
# classes used
print ('classes:', br)
print (np.sort(ab_data['Rings'].unique()))
```

Display the number of output classes:

```
# number of classes
print ('number of classes:', len(ab_data['Rings'].unique()))
```

We have 28 classes used in the dataset.

Display class distribution:

```
instance = ab_data['Rings'].value_counts()
instance.to_dict()
```

Classes range from 1 to 27, and 29. Each class represents the age of an abalone shell in years. The distribution is very uneven. For example, we have 689 instances of shells that are nine years old, but only 1 instance of a shell that is a one-year-old. Typically, machine learning algorithms don't work well with imbalanced data because prediction is biased toward the classes with the most instances. **Imbalanced data** is the problem of classification when there is an unequal distribution of classes in the training set. However, problems like fraud detection are routinely handled by machine learning where classes are expected to be imbalanced.

Display datatypes:

```
ab_data.dtypes
```

Features, with the exception of sex, are float64. The target is int64.

Display information about all columns:

```
ab_data.info(verbose=True)
```

There is no missing data.

Display shape:

```
ab_data.shape
```

We have 4,177 examples with nine attributes each.

Create Train and Test Sets

Split data:

```
train, test = train_test_split(ab_data)

print(len(train), 'train examples')
print(len(test), 'test examples')
```

We have 3,132 train examples and 1,045 test examples. If not specified, the default test size is 25%.

Create Feature and Target Sets

It's a good idea to create copies of train and test sets to preserve original data. Otherwise, the pop method can wreak havoc as it removes data permanently.

Create targets:

```
train_copy, test_copy = train.copy(), test.copy()

# create targets
train_target, test_target = train_copy.pop('Rings'),\
test_copy.pop('Rings')
```

Verify targets:

```
len(train_target), len(test_target)
```

Verify train feature data:

```
train_copy.head(3)
```

When wrangling data, it is a good idea to verify contents.

Convert feature data to numpy:

```
train_features, test_features = train_copy.values,\
test_copy.values
```

Scale Features

We can only scale continuous values. Since the sex feature is not continuous, it cannot be scaled. So we slice off the continuous values to scale. We then recreate train and test sets with the sex feature and scaled continuous values.

Display a sample to verify slicing:

```
train_features[0], test_features[0]
```

Slicing is complex. So it is a good idea to display a sample to ensure that slicing worked as expected.

Create two train sets (one with sex and the other with continuous values):

```
train_sex = [row[0] for row in train_features]
train_f = [row[1:] for row in train_features]
train_sex[0], train_f[0]
```

So far, so good!

Create two test sets (one with sex and the other with continuous values):

```
test_sex = [row[0] for row in test_features]
test_f = [row[1:] for row in test_features]
test_sex[0], test_f[0]
```

Again, our slices match the original data.

Scale continuous values:

```
train_sc = scaler.fit_transform(train_f)
test_sc = scaler.fit_transform(test_f)
```

Create Train and Test Sets with Sex and Scaled Values

Now that we've scaled continuous values, we need to recombine them with the sex feature as shown in Listing 4-10.

Listing 4-10. Recombine continuous features with the sex feature

```
train_ds = [np.append(train_sex[i], row)
for i, row in enumerate(train_sc)]

test_ds = [np.append(test_sex[i], row)
for i, row in enumerate(test_sc)]

train_ds[0], test_ds[0]
```

Recombine train and test features and display. Notice that the continuous features are scaled.

Convert Numpy Feature Sets into Pandas DataFrames

To properly build a TensorFlow consumable dataset with noncontinuous data, we need the feature data in pandas dataframe form:

```
col = ['Sex', 'Length', 'Diameter', 'Height', 'Whole',
       'Shucked', 'Viscera', 'Shell']

train_ab = pd.DataFrame(train_ds, columns=col)
test_ab = pd.DataFrame(test_ds, columns=col)
```

We need the original column names to add to the dataframe.

Verify train features:

```
train_ab.tail(3)
```

Verify test features:

```
test_ab.tail(3)
```

Build the Input Pipeline

Prepare train and test data for TensorFlow consumption:

```
train_ds = tf.data.Dataset.from_tensor_slices(
    (dict(train_ab), train_target))
test_ds = tf.data.Dataset.from_tensor_slices(
    (dict(test_ab), test_target))
```

Notice that we convert train and test feature data to Python dictionaries. We do this to enable construction of categorical feature columns.

Shuffle train data, batch, and prefetch train and test data:

```
BATCH_SIZE = 32
SHUFFLE_BUFFER_SIZE = 100

train_ads = train_ds.shuffle(
    SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE).prefetch(1)
test_ads = test_ds.batch(BATCH_SIZE).prefetch(1)
train_ads, test_ads
```

Notice that the shapes include each feature column name.

Explore a Batch

Since we converted feature data to dictionaries, we can display interesting information about the data as shown in Listing 4-11.

Listing 4-11. Display information about a sample batch

```
def see_format(data, num, feature, indx):
  for feature_batch, label_batch in data.take(num):
    print('Every feature:', list(feature_batch.keys()))
    print('One example from a batch of ' + feature + ':',
          feature_batch[feature][indx])
```

```
    print('One example from a batch of targets:',
          label_batch[indx])

print ('train sample:')
see_format(train_ads, 1, 'Height', 0)
print ()
print ('test sample:')
see_format(test_ads, 1, 'Sex', 0)
```

Categorical Columns

TensorFlow consumption is *limited to numeric data*. So we must convert any categorical data. The only culprit in this scenario is the 'Sex' feature because it is represented by string values of either 'M', 'F', or 'I'. So sex for the abalone shell is either male, female, or infant.

Since we cannot feed strings directly to a model, we must first map them to numeric values. The categorical vocabulary columns feature provides a way to represent strings as a *one-hot vector*. This process is called **one-hot encoding**, which is a technique that converts categorical variables to numerical in an interpretable format.

The strings are converted in the following manner:

1. 'M' => 1 0 0
2. 'F' => 0 1 0
3. 'I' => 0 0 1

Listing 4-12 one-hot encodes the sex feature.

Listing 4-12. One-hot encode the sex feature

```
from tensorflow import feature_column

sex_one_hot =\
feature_column.categorical_column_with_vocabulary_list(
    'Sex', ['M', 'F', 'I'])

print (type(sex_one_hot))
```

```
feature_columns =\
[tf.feature_column.indicator_column(sex_one_hot)]

feature_layer = tf.keras.layers.DenseFeatures(feature_columns)
```

Import the *feature_column* module. One-hot encode 'Sex'. Create the feature_columns list. We create a list so we can have multiple categorical features in a training dataset. Finally, create the *feature_layer* for the model.

For a comprehensive example, peruse the following URL:

www.tensorflow.org/tutorials/structured_data/feature_columns

Build the Model

Clear session and generate a seed:

```
tf.keras.backend.clear_session()
np.random.seed(0)
tf.random.set_seed(0)
```

Create the model:

```
model = tf.keras.Sequential([
  feature_layer,
  Dense(128, activation='relu'),
  Dense(128, activation='relu'),
  Dense(29, activation='sigmoid')
])
```

Notice that the first layer is *feature_layer*, which informs the model about one-hot encoded features.

Compile the Model

Compile:

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Train the Model

Train:

```
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
model.fit(train_ads, validation_data=test_ads, epochs=1)
```

We only trained for one epoch because we knew that performance would be horrible. How did we know this? Check out the next section to find out.

Imbalanced and Irregular Data

The abalone dataset is *not* a good dataset to make predictions for two reasons:

1. The dataset is imbalanced.
2. The dataset is irregular.

An **imbalanced dataset** is one where the classes are not represented equally. That is, classes don't have the same number of examples. This dataset is especially imbalanced because some classes have one example while others have hundreds of examples. Training with an imbalanced dataset won't produce good results. So we won't learn much. The reason is that predictions are biased toward classes with more instances!

An **irregular dataset** is one with too many target (or label) classes, but not enough data. We should always check the number of samples (or examples) per label in our dataset. A class label with not enough samples is harder to learn from.

Dealing with Imbalanced Data

We can deal with imbalanced data in a variety of ways. We can change the algorithm. Some algorithms may work better than others. So try a variety of them. Oversample by adding more instances to the minority class or classes. We can also under-sample by removing observations from the majority class or classes. Finally, we can augment the data.

For a more detailed explanation, consult the following URL:

https://towardsdatascience.com/methods-for-dealing-with-imbalanced-data-5b761be45a18

CHAPTER 5

Classification

Classification is a supervised learning method for predicting a class label for a given example of input data. Although we introduced classification with MNIST, we work through the famous Fashion-MNIST dataset to delve deeper into the topic.

Fashion-MNIST is intended as a direct replacement for MNIST to better benchmark machine learning algorithms. It shares the same image size and structure of training and test splits, but is a more challenging classification problem.

MNIST benchmarking has several associated problems. It's far too easy for standard machine learning algorithms to achieve over 97% accuracy. It's even easier for deep learning models to achieve over 99% accuracy. The dataset is overused. Finally, MNIST cannot represent modern computer vision tasks.

Notebooks for chapters are located at the following URL: *https://github.com/paperd/tensorflow*.

Enable the GPU (if not already enabled):

1. Click *Runtime* in the top-left menu.
2. Click *Change runtime type* from the drop-down menu.
3. Choose *GPU* from the *Hardware accelerator* drop-down menu.
4. Click *SAVE*.

Test if GPU is active:

```
import tensorflow as tf

# display tf version and test if GPU is active
tf.__version__, tf.test.gpu_device_name()
```

Import the *tensorflow* library. If '/device:GPU:0' is displayed, the GPU is active. If '.' is displayed, the regular CPU is active.

D. Paper, *TensorFlow 2.x in the Colaboratory Cloud*, https://doi.org/10.1007/978-1-4842-6649-6_5

Fashion-MNIST Dataset

Fashion-MNIST is a dataset of clothing article images created by Zalando Research consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28 × 28 grayscale image associated with a label from ten classes.

Zalando Research is an organization that utilizes an agile design process that combines invaluable human experience with the power of machine learning. Zalando concentrates on exploring novel ways to use generative models in fashion design for rapid visualizations and prototyping.

Load Fashion-MNIST as a TFDS

Since Fashion-MNIST is a tfds.data.Dataset, we can easily load it with tfds.load. To get a list of all TFDS, just run the tfds.list_builders method as demonstrated in Chapter 3.

Load train and test examples as a tf.data.Dataset:

```
import tensorflow_datasets as tfds

train, info = tfds.load('fashion_mnist', split='train',
                        with_info=True, shuffle_files=True)
test = tfds.load('fashion_mnist', split='test')
```

Since we already have *info* from the train data, we don't need to load it again for the test data.

Verify train and test data:

```
train.element_spec, test.element_spec
```

Each image consists of 28 × 28 pixels. The *1* dimension indicates that images are grayscale. Each label is a scalar.

Explore the Dataset

Display information about the dataset:

```
info
```

We see the name, description, and homepage. We also see the shape and datatype of feature images and labels. We see that we have 70,000 examples with train and test splits of 60,000 and 10,000, respectively. A lot of other information is also included.

Extract the number of classes and class labels:

```
br = '\n'

num_classes = info.features['label'].num_classes
class_labels = info.features['label'].names
print ('number of classes:', num_classes, br)
print ('class labels:', class_labels)
```

We have ten classes representing ten clothing articles.

Let's see some training examples:

```
fig = tfds.show_examples(train, info)
```

The show_examples method displays sample images and labels. The label name and associated class number is displayed under each image. For example, the class number for *Pullover* is *2* because it is the third label in the class labels list.

Listing 5-1 is a custom function that displays samples from the train dataset.

Listing 5-1. Custom function for displaying sample data

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

def display_samples(data, num, cmap):
  for example in data.take(num):
    image, label = example['image'], example['label']
    print ('Label:', class_labels[label.numpy()], end=', ')
    print ('Index:', label.numpy())
    plt.imshow(image.numpy()[:, :, 0].astype(np.float32),
               cmap=plt.get_cmap(cmap))
    plt.show()
```

Import a couple of libraries. The function accepts a dataset, number of samples to display, and a colormap. A **colormap** is an array of colors used to map pixel data to the actual color values. The matplotlib library provides a variety of built-in colormaps.

We assign an example image and label to variables. We then display the label name and its associated class number. We end by displaying the image with the *imshow()* function. We use *[:, :, 0]* to grab all pixels from each image.

Invoke the function to display a couple of training images:

```
# choose colormap by changing 'indx'

indx = 5
cmap = ['coolwarm', 'viridis', 'plasma',
        'seismic', 'copper', 'twilight']
samples = 2
display_samples(train, samples, cmap[indx])
```

Now, let's build a custom function to display a grid of samples.

Before we create the function, take 30 samples from the train set as shown in Listing 5-2.

Listing 5-2. Take samples from the train set

```
num = 30
images, labels = [], []
for example in train.take(num):
  image, label = example['image'], example['label']
  images.append(tf.squeeze(image.numpy()))
  labels.append(tf.squeeze(label.numpy()))
```

Create the function as shown in Listing 5-3.

Listing 5-3. Function that displays a grid of examples

```
def display_grid(feature, target, n_rows, n_cols, cl):
  plt.figure(figsize=(n_cols * 1.5, n_rows * 1.5))
  for row in range(n_rows):
    for col in range(n_cols):
      index = n_cols * row + col
      plt.subplot(n_rows, n_cols, index + 1)
      plt.imshow(feature[index], cmap='binary',
                 interpolation='nearest')
      plt.axis('off')
      plt.title(cl[target[index]], fontsize=12)
  plt.subplots_adjust(wspace=0.2, hspace=0.5)
```

Invoke the function:

```
rows = 5
cols = 6
display_grid(images, labels, rows, cols, class_labels)
```

We can also pinpoint metadata with DatasetInfo:

```
print ('Number of training examples:', end=' ')
print (info.splits['train'].num_examples)

print ('Number of test examples:', end=' ')
print (info.splits['test'].num_examples)
```

Build the Input Pipeline

Build the input pipeline for train and test data as shown in Listing 5-4.

Listing 5-4. Build the input pipeline

```
BATCH_SIZE = 128
SHUFFLE_SIZE = 5000

train_f1 = train.shuffle(SHUFFLE_SIZE).batch(BATCH_SIZE)
train_f2 = train_f1.map(lambda items: (
    tf.cast(items['image'], tf.float32) / 255., items['label']))
train_fs = train_f2.cache().prefetch(1)

test_f1 = test.batch(BATCH_SIZE)
test_f2 = test_f1.map(lambda items: (
    tf.cast(items['image'], tf.float32) / 255., items['label']))
test_fs = test_f2.cache().prefetch(1)
```

Shuffle and batch train data. Scale train images. Cache and prefetch train images. Batch test data. Scale test images. Cache and prefetch test images. Use batch size of 128 and shuffle buffer size of 5,000 for this experiment.

Note Do not shuffle test data because it is considered new to the neural network model.

Caching a TFDS can significantly improve performance. The *cache* method of a tf.data.Dataset can cache a dataset either in memory or on local storage, which saves operations like file opening and data reading from being executed during each epoch.

Adding prefetch is a good idea because it adds efficiency to the batching process. While our training algorithm is working on one batch, TensorFlow is working on the dataset in parallel to get the next batch ready. So prefetch can dramatically improve training performance.

For more information on TFDS performance improvement, peruse

- *www.tensorflow.org/datasets/performances*
- *www.tensorflow.org/guide/data_performance*

Verify train and test tensors:

```
train_fs.element_spec, test_fs.element_spec
```

Both train and test images are 28 × 28 × 1. The 1 value means that images are grayscale. That is, images are in black and white.

Build the Model

Let's build a simple feedforward neural net as shown in Listing 5-5.

Listing 5-5. Simple feedforward neural network

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Dropout

# clear previous model and generate a seed
tf.keras.backend.clear_session()
np.random.seed(0)
tf.random.set_seed(0)

model = Sequential([
  Flatten(input_shape=[28, 28, 1]),
  Dense(512, activation='relu'),
  Dropout(0.4),
  Dense(10, activation='softmax')
])
```

Import requisite libraries. Clear any previous model sessions and generate a seed to facilitate reproducibility of results. The first layer flattens images. The second layer uses *relu* activation on 512 neurons to process the data. The third layer uses dropout to reduce overfitting. The fourth layer uses softmax activation on ten neurons to account for class labels.

Model Summary

Display a summary of the model:

```
model.summary()
```

Output shape of the first layer is (None, 784). We get 784 by multiplying 28 by 28. We have no parameters at this layer because it is only used to bring data into the model.

Output shape of the second layer is (None, 512) because we have 512 neurons at this layer. We get parameters of 401,920 by multiplying 512 neurons at this layer by 784 neurons from the previous layer and adding 512 at this layer.

Output shape of the third layer is (None, 512), and parameters are 0 because dropout doesn't impact neurons or parameters. Output shape of the fourth layer is (None, 10) because we have ten neurons at this layer to deal with ten output classes. We get parameters of 5,130 by multiplying 10 neurons at this layer by 512 from the previous layer and adding 10 neurons at this layer.

Note *None* is used because TensorFlow models can accept any batch size.

Compile the Model

Define a gradient descent optimizer that tweaks model parameters to minimize the loss function:

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Train the Model

Train the model with ten epochs:

```
epochs = 10
history = model.fit(train_fs, epochs=epochs,
                    verbose=1, validation_data=test_fs)
```

We get pretty good accuracy with not much overfitting.

Generalize on Test Data

Although training provides accuracy and loss metrics, it is always a good idea to explicitly evaluate the model on test data to see how well the model generalizes on new data:

```
print('Test accuracy:', end=' ')
test_loss, test_acc = model.evaluate(test_fs, verbose=2)
```

Visualize Performance

The *fit* method automatically records the history of the training process as a dictionary. So we can assign training information to a variable. In this case, we assign it to history. The history attribute of the variable contains the dictionary information.

Display the dictionary keys to inform us on how to plot the results:

```
hist_dict = history.history
print (hist_dict, '\n')
print (hist_dict.keys())
```

The dictionary history.history contains loss, accuracy, val_loss, and val_accuracy metrics that the model measures at the end of each epoch on the training set and validation (or test) set.

The *params* variable provides all parameters involved with training:

```
history.params
```

Plot training performance as shown in Listing 5-6.

Listing 5-6. Plot training performance

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')
plt.show()
```

```
plt.plot(history.history['loss'], label='loss')
plt.plot(history.history['val_loss'], label = 'val_loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.ylim([0.05, .7])
plt.legend(loc='lower right')
plt.show()
```

We don't have much overfitting, and our model accuracy is pretty good for such a simple neural network.

We can also use pandas to plot training performance:

```
import pandas as pd
```

```
pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
```

Predict Labels for Test Images

Now that we have a trained model, we can make predictions based on test images. We predict from test images because the model sees these images as *new* data.

Begin by predicting the label for each image in the test set:

```
tf.random.set_seed(0)
```

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

We use the *predict* method on the processed test set *test_fs*.

Since Fashion-MNIST has ten class labels, each prediction consists of an array of ten numbers that represent the model's *confidence* in how well the image corresponds to each of the ten different articles of clothing.

Let's take a look at the *first* prediction:

```
predictions[0]
```

The first prediction is at index 0 because Python indexes range from 0 to 9999 for test set size of 10,000. It's hard to tell which of the values has the highest number by looking at the array of float number values.

Round the numbers in the prediction array to make it easier to see the position in the array with the highest confidence:

```
np.round(predictions[0], 2)
```

Now, we can clearly see the value with the highest number. The position with the highest confidence corresponds to the position in the class labels array.

We can directly derive the confidence in the prediction with the following code:

```
c = 100*np.max(predictions[0])
c
```

Display the confidence:

```
'{:.2%}'.format(np.max(predictions[0]))
```

Display the prediction for the *first* image in the test set:

```
np.argmax(predictions[0])
```

The prediction must be between 0 and 9 because our targets are between 0 and 9.

Use the *class_labels* array we created earlier to see the actual fashion article:

```
class_labels[np.argmax(predictions[0])]
```

So we have the predicted clothing article for the first image in the test set.

Display the *first* actual test image:

```
# take the first batch of images

for image, label in test_fs.take(1):
  label

class_labels[label[0].numpy()]
```

We take the first image from a batch of 128 images because we set batch size to 128. If the prediction matches the test image, it was a correct prediction.

Build a Prediction Plot

Now that we have a trained model, we can build a prediction plot.

Since we want to see how well the model performs on new data, take 30 samples from the test set as shown in Listing 5-7.

Listing 5-7. Take samples from the test set

```
num = 30
images, labels = [], []
for example in test.take(num):
  image, label = example['image'], example['label']
  images.append(tf.squeeze(image.numpy()))
  labels.append(tf.squeeze(label.numpy()))
```

Squeeze each image to remove the 1 dimension for plotting purposes.

Build a plotting function as shown in Listing 5-8.

Listing 5-8. Function to build a prediction plot

```
def display_test(feature, target, num_images,
                 n_rows, n_cols, cl, p):
  for i in range(num_images):
    plt.subplot(n_rows, 2*n_cols, 2*i+1)
    if cl[target[i]] != cl[np.argmax(p[i])]:
      plt.imshow(feature[i], cmap='Reds')
    else:
      plt.imshow(feature[i], cmap='Blues')
    val = 100*np.max(p[i])
    rounded = str(np.round(val, 2)) + '%'
    plt.title(cl[target[i]] + ' (' +\ cl[np.argmax(p[i])] + ') ' +\
              rounded )
  plt.tight_layout()
plt.show()
```

Invoke the function as shown in Listing 5-9.

Listing 5-9. Invoke the prediction plot function

```
num_rows, num_cols = 6, 5
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
display_test(images, labels, num_images, num_rows,
             num_cols, class_labels, predictions)
```

Any clothing article in *red* means that the prediction was incorrect. Above each article of clothing is the actual label, prediction in parentheses, and confidence in the prediction.

Load Fashion-MNIST as a Keras Dataset

Although TFDS are recommended for TensorFlow 2.x application, we show you how to work with a Keras dataset because of its popularity in industry. The newness of TensorFlow 2.x means that it doesn't yet have the industry penetration of Keras.

Load the Keras dataset:

```
train, test = tf.keras.datasets.fashion_mnist.load_data()
```

Explore the Data

Let's look at the data shapes:

```
print ('train data:', br)
print (train[0].shape)
print (train[1].shape, br)
print ('test data:', br)
print (test[0].shape)
print (test[1].shape)
```

The Keras dataset contains the same data as the Fashion-MNIST TFDS. Train data consists of 60,000 28 × 28 feature images and 60,000 labels. Test data consists of 10,000 28 × 28 feature images and 10,000 labels. Train images are contained in the train tuple. So train[0] represents images and train[1] represents labels.

Let's see what the first image represents:

```
class_labels[train[1][0]]
```

Since we access training labels with train[1], we grab the first label with train[1][0].

Visualize the First Image

Plot the first image with Matplotlib's imshow function:

```
plt.imshow(train[0][0], cmap='binary')
plt.axis('off')
```

Assign images and labels to variables for convenience:

```
train_images = train[0]
train_labels = train[1]
```

Plot the first image based on the train_images variable:

```
plt.imshow(train_images[0], cmap='binary')
plt.axis('off')
```

Access the label name of the first image:

```
class_labels[train_labels[0]]
```

Labels are the class IDs ranging from 0 to 9:

```
train_labels
```

Visualize Sample Images

Since we can't use the TFDS show_examples method, we create code as shown in Listing 5-10.

Listing 5-10. Code to visualize examples

```
n_rows = 5
n_cols = 6
plt.figure(figsize=(n_cols * 1.5, n_rows * 1.5))
for row in range(n_rows):
```

```
  for col in range(n_cols):
    index = n_cols * row + col
    plt.subplot(n_rows, n_cols, index + 1)
    plt.imshow(train_images[index], cmap='binary',
               interpolation='nearest')
    plt.axis('off')
    plt.title(class_labels[train_labels[index]], fontsize=12)
plt.subplots_adjust(wspace=0.2, hspace=0.5)
```

Prepare Data for Training

To prepare training data for TensorFlow consumption, we need to grab images and labels from train and test data. We scale images to ensure that each input parameter (a pixel, in our case) has a similar data distribution. The distribution of such data resembles a Gaussian curve centered at zero. Scaling data makes convergence faster while training the network.

As we know, pixel data is represented by a range from 0 to 255. So we divide each feature image by 255 to scale it. Once images are scaled, we convert images and labels to a tf.Tensor object with from_tensor_slices as shown in Listing 5-11.

Listing 5-11. Prepare the input pipeline

```
# add test images and labels to the mix

test_images, test_labels = test

train_pictures = train_images / 255.  # divide by 255 to scale
train_targets = train_labels.astype(np.int32)

test_pictures = test_images / 255.  # divide by 255 to scale
test_targets = test_labels.astype(np.int32)

print ('train images:', len(train_pictures))
print ('train labels:', len(train_targets), br)
print ('test images', len(test_pictures))
print ('test labels', len(test_targets))
```

```
train_ds = tf.data.Dataset.from_tensor_slices(
    (train_pictures, train_targets))
test_ds = tf.data.Dataset.from_tensor_slices(
    (test_pictures, test_targets))
```

Display the tensors:

```
train_ds.element_spec, test_ds.element_spec
```

Finish the input pipeline by shuffling, batching, and prefetching train data and batching and prefetching test data:

```
BATCH_SIZE = 128
SHUFFLE_BUFFER_SIZE = 5000

train_ks = train_ds.shuffle(
    SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE).prefetch(1)
test_ks = test_ds.batch(BATCH_SIZE).prefetch(1)
```

Set BATCH_SIZE to 128 so the model will run faster than with a smaller batch size. Experiment with this number and see what happens. We also set SHUFFLE_BUFFER_SIZE to 5000 so the shuffle method works well. Again, experiment with this number to see what happens.

Display the finalized input pipeline tensors:

```
train_ks.element_spec, test_ks.element_spec
```

Build the Model

Create the same model that we did for the TFDS as shown in Listing 5-12.

Listing 5-12. Keras model

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Dropout

# clear previous model and generate a seed
tf.keras.backend.clear_session()
np.random.seed(0)
tf.random.set_seed(0)
```

```
model = Sequential([
  Flatten(input_shape=[28, 28]),
  Dense(512, activation='relu'),
  Dropout(0.4),
  Dense(10, activation='softmax')
])
```

As we already know, it is always a good idea to clear any previous modeling sessions. Remember that we already created a model and trained it earlier in the chapter. Also, generating a seed ensures that the results are consistent. In machine learning, such consistency is called *reproducibility*. That is, the seed provides a starting point for the random generator that allows us to reproduce results in a consistent manner. You can use any integer value for the random seed number. We use *0*.

The first layer, Flatten(), flattens the 2D matrix of 28 × 28 images to a 1D array of 784 pixels. The second layer is the first true layer. It accepts input images into 128 neurons and performs *relu* activation. The final layer is the output layer. It accepts output from the input layer with ten neurons that represent the ten classes of clothing articles and performs softmax activation.

Model Summary

Inspect the model:

```
model.summary()
```

Compile the Model

Compile:

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Train the Model

Fit the model. Passing a dataset of (feature, label) pairs is all that's needed for Model.fit and Model.evaluate:

```
history = model.fit(train_ks, epochs=10, validation_data=test_ks)
```

Generalize on Test Data

Although model fit information provides validation loss and accuracy during training, it is always a good idea to explicitly evaluate the model on test data because accuracy and loss values may differ:

```
print('Test accuracy:', end=' ')
test_loss, test_acc = model.evaluate(test_ks, verbose=2)
```

Visualize Training

The fit method automatically records the history of the training process as a dictionary. So we can assign training history to a variable. In this case, we assigned it to *history*. The history attribute of the variable contains the dictionary information.

Display the dictionary keys to inform us on how to plot results:

```
hist_dict = history.history
print (hist_dict, '\n')
print (hist_dict.keys())
```

Dictionary *history.history* contains loss and other metrics the model measures at the end of each epoch on the training set and validation set.

The *params* variable provides all parameters involved with the model:

```
history.params
```

Listing 5-13 plots training history.

Listing 5-13. Training history plots

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')
plt.show()
```

```
plt.plot(history.history['loss'], label='loss')
plt.plot(history.history['val_loss'], label = 'val_loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.ylim([0.05, .7])
plt.legend(loc='lower right')
plt.show()
```

Overfitting is minimal because training accuracy is closely aligned with test accuracy. We did employ dropout in the model to reduce overfitting. **Dropout** is a regularization method that approximates training a large number of neural networks with different architectures in parallel. The definition doesn't make it easier to understand for anyone new to deep learning. So let's explain how it works.

During training, some number of layer outputs are randomly ignored or *dropped out*. By randomly dropping out layer outputs, effectively the layer looks like and is treated like a layer with a different number of nodes and connectivity to the prior layer. So each update to a layer during training is performed with a different *view* of the configured layer. Dropout is a simple but very effective technique to reduce overfitting.

Setting dropout at *0.4* means that we randomly drop out 40% of the layer outputs. You can easily experiment with dropout by changing this value. However, you should keep dropout rate at or less than *0.5* because otherwise you are removing too much data!

Tip Experiment with dropout levels, but keep it at or less than 0.5 to avoid removing too much data.

If the model is overfitting (train accuracy is greater than test accuracy) with the dropout value you set, you can increase it. If the model is underfitting (train accuracy is less than test accuracy), you can decrease it.

Use pandas to plot:

```
import pandas as pd

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
```

Predict Labels for Test Images

We can make predictions on test images since we have a trained model.

Predict the label for each image from test set *test_ks*:

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

As with the Fashion-MNIST TFDS, a prediction is an array of ten numbers that represent the model's confidence that the image corresponds to each of the ten articles of clothing.

Predict the First Image

Let's look at the *first* prediction:

```
predictions[0]
```

Round the array values for convenience:

```
np.round(predictions[0], 2)
```

Directly access the confidence in the prediction:

```
100*np.max(predictions[0])
```

Convert the value to a percentage:

```
str(np.round(100*np.max(predictions[0]), 2)) + '%'
```

Display the prediction for the first image:

```
np.argmax(predictions[0])
```

The prediction can only be between 0 and 9 because our targets are between 0 and 9.

Use the class_labels array created earlier to see the actual fashion article:

```
class_labels[np.argmax(predictions[0])]
```

Display the first actual test label for comparison:

```
class_labels[test_labels[0]]
```

If the prediction and actual images match, the prediction is correct.

Predict Four Images

Make four predictions from the test set:

```
pred_4 = predictions[:4]
```

Show the predictions:

```
ls = [np.argmax(row) for row in pred_4]
ls
```

Get class labels:

```
np.array(class_labels)[ls]
```

Compare predictions to the test dataset:

```
actual_4 = test_labels[:4]
actual_4
```

Show actuals as class labels:

```
np.array(class_labels)[actual_4]
```

Visualize as shown in Listing 5-14.

Listing 5-14. Visualize the first four actual test images

```
# slice off the first four images from the test data
img_4 = test_images[:4]

# plot images

plt.figure(figsize=(7.2, 2.4))
for index, image in enumerate(img_4):
  plt.subplot(1, 4, index + 1)
  plt.imshow(image, cmap='twilight', interpolation='nearest')
  plt.axis('off')
  plt.title(class_labels[actual_4[index]], fontsize=12)
plt.subplots_adjust(wspace=0.2, hspace=0.5)
```

Explore Misclassifications

Let's explore predictions and actual test image labels to find some misclassifications.

The first step is to identify prediction labels:

```
rng = (len(predictions))
y_pred = [np.argmax(row) for row in predictions]
```

The *y_pred* variable holds a list of prediction labels.

Next, grab some predictions and actual targets:

```
# find first n predictions and actual targets

n = 20
y_n = [y_pred[i] for i, row in enumerate(range(n))]
y_actual = [test_labels[i] for i, row in enumerate(range(n))]

y_n, y_actual
```

Find misclassifications:

```
# compare predictions against actual targets

miss_indx_list = [index for index, (x, y) in
                  enumerate(zip(y_n, y_actual))
                  if x != y]

miss_indx_list
```

If you don't get any misclassifications, increase the size of *n* and rerun the last two code snippets.

We now have an array of misclassifications by their index.

Listing 5-15 displays the confidence for misclassifications.

Listing 5-15. Confidence in misclassifications

```
# display confidence for each misclassification:

for row in miss_indx_list:
  val = 100*np.max(predictions[row])
  rounded = str(round(val, 2)) + '%'
```

```
  print ('index:', row, 'confidence:', rounded,
         'pred:', class_labels[np.argmax(predictions[row])],
         'actual:', class_labels[test_labels[row]])
```

Visualize Misclassifications

Create a function to visualize misclassifications as shown in Listing 5-16.

Listing 5-16. Function to visualize misclassifications

```
def see_misses(indx):
  plt.imshow(test_images[indx], cmap='nipy_spectral')
  plt.show()
  print ('actual:', class_labels[test_labels[indx]])
  print ('predicted:', class_labels[np.argmax(predictions[indx])])
  print ('confidence', rounded)
```

Invoke the function:

```
for row in miss_indx_list:
  val = 100*np.max(predictions[row])
  rounded = str(round(val, 2)) + '%'
  see_misses(row)
```

The confidence may still be pretty high even though the prediction was incorrect. Although neural networks tend to perform well for prediction, their estimated predicted probabilities by the output of a softmax layer tend to be too high. That is, they are too confident in their predictions. Remember that the probability with the highest value is the predicted class from the prediction array. And this probability is the confidence. However, the prediction is not necessarily correct unless the model has 100% accuracy.

Create a more sophisticated visualization as shown in Listing 5-17.

Listing 5-17. Sophisticated visualization of misclassifications

```
# Plot the first X test images, their true labels,
# their predicted labels, and prediction confidence.

num_rows = 8
num_cols = 8
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  if class_labels[test_labels[i]] != class_labels[y_pred[i]]:
    plt.imshow(test_images[i], cmap='Reds')
  else:
    plt.imshow(test_images[i], cmap='Blues')
  val = 100*np.max(predictions[i])
  rounded = str(np.round(val, 2)) + '%'
  plt.title(class_labels[test_labels[i]] + ' (' +\
            class_labels[y_pred[i]] + ') ' + rounded )
plt.tight_layout()
plt.show()
```

Misclassifications, if any, are in red. Above each article of clothing is the actual label, prediction in parentheses, and confidence in the prediction.

Predict from a Single Image

We can make a prediction on a single image. We choose a number between 0 and 9,999 because we want an image from the test set. Alternatively, we can generate a random number between 0 and 9,999.

Randomly choose an image from the test set:

```
beg, end = 0, len(test_images) - 1
rng = np.random.default_rng()
indx = int(rng.uniform(beg, end, size=1))
indx
```

Displayed is the index of the randomly chosen image.

Grab the image from the test set:

```
# Grab the image from the test dataset

img = test_images[indx]
label = class_labels[test_labels[indx]]
img.shape, label
```

TensorFlow models are optimized to make predictions on a batch or collection of examples at once. So add the single image to a batch where it is the only member.

Create a batch with a single image:

```
img_batch = (np.expand_dims(img, 0))
img_batch.shape
```

The *expand_dims* method inserts a new axis that appears at the axis position in the expanded array shape. So the new shape is (1, 28, 28).

Now, we can predict:

```
pred_single = model.predict(img_batch)
```

Display prediction:

```
np.argmax(pred_single)
```

The index position in the prediction array is displayed.

To make it easier, display prediction by label name:

```
class_labels[np.argmax(pred_single)]
```

Display the actual label:

```
class_labels[test_labels[indx]]
```

Visualize Single Image Prediction

Visualize the prediction as shown in Listing 5-18.

Listing 5-18. Visualization of single image prediction

```
pred = class_labels[np.argmax(pred_single)]
actual = class_labels[test_labels[indx]]
```

```
# get confidence from the predictions object
val = 100*np.max(predictions[indx])
rounded = str(np.round(val, 2)) + '%'

# display actual image
plt.imshow(test_images[indx], cmap=plt.cm.binary)
plt.show()

print ('actual:', actual)
print ('predicted:', pred)
print ('confidence', rounded)
```

Get prediction by label name and actual label name for the image. Get prediction confidence from the predictions object that we created earlier. The predictions object holds all predictions based on the test set. The *indx* value provides the position of the confidence value in the predictions object.

Confusion Matrix

Create a confusion matrix to visually demonstrate how well the model classified articles of clothing from Fashion-MNIST:

```
tf.math.confusion_matrix(test_labels, y_pred)
```

To understand the TensorFlow confusion matrix, imagine that above the first row of the matrix are classes 0–9 from left to right. These represent predictions. Imagine that to the left of the first column of the matrix are classes 0–9 from top to bottom. These represent actual labels. Correct classifications are along the diagonal. Misclassifications are not on the diagonal.

The first set of correct classifications is in position row one, column one and represents class 0. The second set is in position row two, column two and represents class 1. And so on…

An example misclassification set is row one, column two. This represents the number of predictions for class 1 that were misclassified as class 0. For a technical explanation of the TensorFlow confusion matrix, peruse the following URL:

https://stackoverflow.com/questions/46616837/understanding-a-tensorflow-confusion-matrix-for-binary-classification

For a general explanation of a confusion matrix, peruse the following URL:

www.dataschool.io/simple-guide-to-confusion-matrix-terminology/

Number of Hidden Layers

For many problems, we can begin with a single hidden layer and get reasonable results as we did with our Fashion-MNIST model in this chapter. For more complex problems, we should add layers until we start overfitting the training set.

Number of Neurons in Hidden Layers

The number of neurons in the input and output layers is based on the type of input and output your task requires. For example, the Fashion-MNIST task requires $28 \times 28 = 784$ input neurons and ten output neurons. For hidden layers, it is very difficult to determine. You can try increasing the number of neurons gradually until the network starts overfitting. In practice, we typically pick a model with more layers and neurons than we need and then use early stopping and other regularization techniques to reduce overfitting. Since this is an introduction, we won't delve any deeper into tuning networks.

Generally, we get a better model by increasing the number of layers rather than the number of neurons per layer. Of course, the number of layers and neurons we include is limited by our available computational resources.

CHAPTER 6

Regression

Regression is a supervised learning method for predicting a continuous output of an event based on the relationship between variables (or features) obtained from a dataset. A **continuous** outcome is a real value such as an integer or floating-point value often quantified as amounts and sizes. Regression is a widely popular type of deep learning modeling.

Since regression predicts a quantity, performance is measured error in those predictions. Regression performance can be measured in many ways. But the most common are mean squared error (MSE), mean absolute error (MAE), and root mean squared error (RMSE).

MSE is one of the most commonly used metrics. However, it is least useful when a single bad prediction would ruin the entire model's predicting abilities. That is, when the dataset contains a lot of noise. It is most useful when the dataset contains outliers or unexpected values. Unexpected values are those that are too high or too low.

MAE is not very sensitive to outliers in comparison to MSE since it doesn't punish huge errors. It is typically used when performance is measured on continuous variable data. It provides a linear value that averages the weighted individual differences equally.

RMSE errors are squared before they are averaged. As such, RMSE assigns a higher weight to larger errors. So RMSE is much more useful when large errors are present and they drastically affect the model's performance. A benefit of RMSE is that units of error score are the same as the predicted value.

We thoroughly work through the famous *Boston Housing* dataset. We demonstrate how to load the data, build the input pipeline, and model the data. We also show you how to use the model to make predictions. We end by modeling a different dataset. The *Cars* dataset might not be as popular, but we want to give you experience with another set of data.

Notebooks for chapters are located at the following URL: *https://github.com/paperd/tensorflow*.

D. Paper, *TensorFlow 2.x in the Colaboratory Cloud*, https://doi.org/10.1007/978-1-4842-6649-6_6

Enable the GPU (if not already enabled):

1. Click *Runtime* in the top-left menu.
2. Click *Change runtime type* from the drop-down menu.
3. Choose *GPU* from the *Hardware accelerator* drop-down menu.
4. Click *SAVE*.

Test if GPU is active:

```
import tensorflow as tf

# display tf version and test if GPU is active
tf.__version__, tf.test.gpu_device_name()
```

Import the *tensorflow* library. If '/device:GPU:0' is displayed, the GPU is active. If '.' is displayed, the regular CPU is active.

Boston Housing Dataset

Boston Housing is a dataset derived from information collected by the US Census Service concerning housing in the area of Boston, Massachusetts. It was obtained from the StatLib archive (*http://lib.stat.cmu.edu/datasets/boston*) and has been used extensively throughout the machine learning literature to benchmark algorithms. The dataset is small in size with only 506 cases.

The name of this dataset is *boston*. It contains 12 features and 1 outcome (or target). The features are as follows:

1. CRIM – Per capita crime rate by town
2. ZN – Proportion of residential land zoned for lots over 25,000 sq. ft.
3. INDUS – Proportion of non-retail business acres per town
4. CHAS – Charles River dummy variable (1 if tract bounds river and 0 otherwise)
5. NOX – Nitric oxide concentration (parts per 10 million)
6. RM – Average number of rooms per dwelling
7. AGE – Proportion of owner-occupied units built prior to 1940

8. DIS – Weighted distances to five Boston employment centers
9. RAD – Index of accessibility to radial highways
10. TAX – Full-value property tax rate per $10,000
11. PTRATIO – Pupil-teacher ratio by town
12. LSTAT – % lower status of the population

The target is

- MEDV – Median value of owner-occupied homes in $1000s

Data was collected in the 1970s, so don't be shocked by the low median value of homes.

Boston Data

You can access any dataset for this book directly from GitHub with a few simple steps:

1. Visit the book URL: *https://github.com/paperd/tensorflow*.
2. Locate the dataset and click it.
3. Click the *Raw* button.
4. Copy the URL to Colab and assign it to a variable.
5. Read the dataset with the Pandas read_csv method.

For convenience, we've already located the appropriate URL and assigned it to a variable:

```
url = 'https://raw.githubusercontent.com/paperd/tensorflow/\
master/chapter6/data/boston.csv'
```

Read the dataset into a pandas dataframe:

```
import pandas as pd

data = pd.read_csv(url)
```

Verify that the dataset was read properly:

```
data.head()
```

Explore the Dataset

Get datatypes:

```
data.dtypes
```

All features are datatype float64 or int64. The label *MEDV* is datatype float64.

Get general information:

```
data.info()
```

The dataset contains 506 records.

Create a dataframe that holds basic statistics with the *describe* method and transpose it for easier viewing:

```
data_t = data.describe()
desc = data_t.T
desc
```

Target specific statistics from the transposed dataframe:

```
desc[['mean', 'std']]
```

Describe a specific feature from the original dataframe:

```
data.describe().LSTAT
```

Get columns:

```
cols = list(data)
cols
```

Create Feature and Target Sets

Create the target:

```
# create a copy of the DataFrame
df = data.copy()

# create the target
target = df.pop('MEDV')
print (target.head())
```

Since we popped MEDV from the copy, it should only contain the features:

```
df.head()
```

Get Feature Names from the Features DataFrame

Since the target is no longer part of the dataframe, it's easy to get the features:

```
feature_cols = list(df)
feature_cols
```

Get the number of features:

```
len(feature_cols)
```

Convert Features and Labels

Convert pandas dataframe values to float with the *values* method:

```
features = df.values
labels = target.values
```

```
type(features), type(labels)
```

Split Dataset into Train and Test Sets

Listing 6-1 creates train and test sets.

Listing 6-1. Train and test sets

```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(
    features, labels, test_size=0.33, random_state=0)
```

```
br = '\n'
```

```
print ('X_train shape:', end=' ')
print (X_train.shape, br)
```

```
print ('X_test shape:', end=' ')
print (X_test.shape)
```

Scale Data and Create TensorFlow Tensors

With image data, we scale by dividing each element by 255.0 to ensure that each input parameter (a pixel, in our case) has a similar data distribution. However, features represented by continuous values are scaled differently. We scale continuous data to have a mean (μ) of 0 and standard deviation (σ) of 1. Symbol μ is pronounced mu, and symbol σ is pronounced sigma. A sigma of 1 is called unit variance.

Listing 6-2 scales train and test data.

Listing 6-2. Scale train and test sets

```
# scale feature data and create TensorFlow tensors

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_std = scaler.fit_transform(X_train)
X_test_std = scaler.fit_transform(X_test)

train = tf.data.Dataset.from_tensor_slices(
    (X_train_std, y_train))
test = tf.data.Dataset.from_tensor_slices(
    (X_test_std, y_test))
```

Let's view the first tensor as shown in Listing 6-3.

Listing 6-3. Display the first tensor

```
def see_samples(data, num):
  for feat, targ in data.take(num):
    print ('Features: {}'.format(feat), br)
    print ('Target: {}'.format(targ))

n = 1
see_samples(train, n)
```

The first sample looks as we expect.

Prepare Tensors for Training

Shuffle train data, batch, and prefetch train and test data:

```
BATCH_SIZE, SHUFFLE_BUFFER_SIZE = 16, 100

train_bs = train.shuffle(
    SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE).prefetch(1)
test_bs = test.batch(BATCH_SIZE).prefetch(1)
```

Inspect tensors:

```
train_bs, test_bs
```

Create a Model

If we don't have a lot of training data, one technique to avoid overfitting is to create a small network with few hidden layers. We do just that!

The 64-neuron input layer accommodates our 12 input features. We have one hidden layer with 64 neurons. The output layer has one neuron because we are using regression with one target.

Create a model as shown in Listing 6-4.

Listing 6-4. Create a model

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
import numpy as np

# clear any previous model
tf.keras.backend.clear_session()

# generate a seed for replication purposes
np.random.seed(0)
tf.random.set_seed(0)

# notice input shape accommodates 12 features!
model = Sequential([
  Dense(64, activation='relu', input_shape=[12,]),
```

```
  Dense(64, activation='relu'),
  Dense(1)
])
```

Model Summary

Inspect the model:

```
model.summary()
```

Output shape of the first layer is (None, 64) because we have 64 neurons at this layer. We get parameters of 832 by multiplying 64 neurons at this layer by 12 features and adding 64 neurons at this layer.

Output shape of the second layer is (None, 64) because we have 64 neurons at this layer. We get parameters of 4,160 by multiplying 64 neurons at this layer by 64 neurons from the previous layer and adding 64 at this layer.

Output shape of the third layer is (None, 1) because we have one target. We get parameters of 65 by adding 64 neurons from the previous layer to 1 neuron at this layer.

Compile the Model

Compile:

```
rmse = tf.keras.metrics.RootMeanSquaredError()

model.compile(loss='mse', optimizer='RMSProp',
              metrics=[rmse, 'mae', 'mse'])
```

Mean squared error (MSE) is a common loss function used for regression problems. Mean absolute error (MAE) and RMSE are also commonly used metrics. With some experimentation, we found that optimizer *RMSProp* performed pretty well with this dataset. RMSProp is an algorithm used for full-batch optimization. It tries to solve the problem that gradients may vary widely in magnitude by only using the sign of the gradient, which guarantees that all weight updates are of the same size.

Train the Model

Train the model for 50 epochs:

```
history = model.fit(train_bs, epochs=50, validation_data=test_bs)
```

Visualize Training

Create variable *hist* to hold the model's history as a pandas dataframe. Create another variable *hist['epoch']* to hold epoch history. Display the last five rows to get an idea about performance.

Here's the code:

```
hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
hist.tail()
```

Build the plots as shown in Listing 6-5.

Listing 6-5. Visualize training performance

```
import matplotlib.pyplot as plt

def plot_history(history, limit1, limit2):
  hist = pd.DataFrame(history.history)
  hist['epoch'] = history.epoch

  plt.figure()
  plt.xlabel('epoch')
  plt.ylabel('MAE [MPG]')
  plt.plot(hist['epoch'], hist['mae'], label='Train Error')
  plt.plot(hist['epoch'], hist['val_mae'], label = 'Val Error')
  plt.ylim([0, limit1])
  plt.legend()
  plt.title('MAE by Epoch')
  plt.show()

  plt.clf()

  plt.figure()
  plt.xlabel('Epoch')
  plt.ylabel('MSE [MPG]')
  plt.plot(hist['epoch'], hist['mse'], label='Train Error')
  plt.plot(hist['epoch'], hist['val_mse'], label = 'Val Error')
```

```
  plt.ylim([0, limit2])
  plt.legend()
  plt.title('MSE by Epoch')
  plt.show()

  plt.clf()

  plt.figure()
  plt.xlabel('Epoch')
  plt.ylabel('RMSE [MPG]')
  plt.plot(hist['epoch'], hist['root_mean_squared_error'],
           label='Train Error')
  plt.plot(hist['epoch'], hist['val_root_mean_squared_error'],
           label = 'Val Error')
  plt.ylim([0, limit2])
  plt.legend()
  plt.title('RMSE by Epoch')
  plt.show()

# set limits to make plot readable

mae_limit, mse_limit = 10, 100
plot_history(history, mae_limit, mse_limit)
```

Since the validation error is worse than the train error, the model is overfitting. What can we do? The first step is to estimate when performance begins to degrade. From the visualizations, can you see when this happens?

Early Stopping

With classification, our goal is to maximize accuracy. Of course, we also want to minimize loss. With regression, our goal is to minimize MSE or one of the other common error metrics. From the visualizations, we see that our model is overfitting because validation error is higher than training error. We also see that once training error and validation error cross, performance begins to degrade.

There is one simple tuning experiment we can run to make this model more useful. We can stop the model when training and validation errors are very close to each other. This technique is called early stopping. **Early stopping** is a widely used approach that stops training at the point when performance on a validation dataset starts to degrade.

Let's modify our training experiment to automatically stop training when the validation score doesn't improve. We use an *EarlyStopping* callback that tests a training condition for every epoch. When performance starts to degrade, training is automatically stopped.

All we need to do is update the fit method and retrain as shown in Listing 6-6.

Listing 6-6. Early stopping

```
# clear the previous model
tf.keras.backend.clear_session()

# generate a seed for replication purposes
np.random.seed(0)
tf.random.set_seed(0)

# monitor 'val_loss' for early stopping
early_stop = tf.keras.callbacks.EarlyStopping(monitor='val_loss')

history = model.fit(train_bs, epochs=50, validation_data=test_bs,
                    callbacks=[early_stop])
```

Instead of allowing the algorithm to automatically early stop, we can add some control. Just include a parameter that forces the model to continue to a point that gives us the best performance. The *patience* parameter can be set to a given number of epochs after which training will be stopped if there is no improvement.

Let's try this and see what happens as shown in Listing 6-7.

Listing 6-7. Early stopping with patience

```
# clear the previous model
tf.keras.backend.clear_session()

# generate a seed for replication purposes
np.random.seed(0)
tf.random.set_seed(0)
```

```
# set number of patience epochs
n = 4

early_stop = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', patience=n)

history = model.fit(train_bs, epochs=50, validation_data=test_bs,
                    callbacks=[early_stop])
```

Experiment with the patience parameter to find better results.

Let's visualize:

```
hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch

train_limit, test_limit = 10, 100
plot_history(history, train_limit, test_limit)
```

Remove Bad Data

The Boston dataset has some inherent bad data. What is wrong with the data? Prices of homes are capped at $50,000 because the Census Service censored the data. They decided to set the maximum value of the price variable to 50,000 USD, so no price can go beyond that value.

What do we do? While maybe not ideal, we can remove data with prices at or above 50,000 USD. This is not ideal because we may be removing perfectly good data, but there is no way to know this. Another reason is because the dataset is so small to begin with. Neural nets are meant to perform at their best with larger datasets.

To explore this topic further, we recommend this URL:

https://towardsdatascience.com/things-you-didnt-know-about-the-boston-housing-dataset-2e87a6f960e8

Get Data

We don't want to attempt to clean a dataset processed for TensorFlow consumption. So just reload the raw dataset:

```
# get the raw data
```

```
url = 'https://raw.githubusercontent.com/paperd/tensorflow/\
master/chapter6/data/boston.csv'

boston = pd.read_csv(url)
```

Verify data:

```
boston.head()
```

Remove Noise

Remove the bad data, which hopefully reduces inherent noise:

```
print ('data set before removing noise:', boston.shape)

# remove noise
noise = boston.loc[boston['MEDV'] >= 50]
data = boston.drop(noise.index)

print ('data set without noise:', data.shape)
```

Noise is the irrelevant information or randomness in a dataset. We removed several records from the dataset.

Create Feature and Target Data

Create feature and target sets:

```
# create a copy of the dataframe
df = data.copy()

# create feature and target sets
target, features = df.pop('MEDV'), df.values
labels = target.values
```

Build the Input Pipeline

Create the input pipeline by splitting data into train and test sets, scaling feature data, and slicing data into TensorFlow consumable pieces. Finish the pipeline by shuffling train data, batching, and prefetching train and test data.

Listing 6-8 includes the code to build the input pipeline.

Listing 6-8. Build the input pipeline

```
X_train, X_test, y_train, y_test = train_test_split(
    features, labels, test_size=0.33, random_state=0)

# standardize feature data and create TensorFlow tensors
X_train_std = scaler.fit_transform(X_train)
X_test_std = scaler.fit_transform(X_test)

# slice data for TensorFlow consumption
train = tf.data.Dataset.from_tensor_slices(
    (X_train_std, y_train))
test = tf.data.Dataset.from_tensor_slices(
    (X_test_std, y_test))

# shuffle, batch, prefetch

BATCH_SIZE = 16
SHUFFLE_BUFFER_SIZE = 100

train_n = train.shuffle(
    SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE).prefetch(1)
test_n = test.batch(BATCH_SIZE).prefetch(1)
```

Inspect tensors:

```
train_n, test_n
```

Compile and Train

Listing 6-9 includes the code for compiling and training the model.

Listing 6-9. Compile and train the model

```
rmse = tf.keras.metrics.RootMeanSquaredError()

model.compile(loss='mse', optimizer='RMSProp',
              metrics=[rmse, 'mae', 'mse'])

tf.keras.backend.clear_session()
```

```
# generate a seed for replication purposes
np.random.seed(0)
tf.random.set_seed(0)

n = 4

early_stop = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', patience=n)

history = model.fit(train_n, epochs=50, validation_data=test_n,
                    callbacks=[early_stop])
```

Visualize

Plot results:

```
hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch

train_limit, test_limit = 10, 100

plot_history(history, train_limit, test_limit)
```

Our model is not perfect, but we did improve performance.

Generalize on Test Data

Evaluate:

```
loss, rmse, mae, mse = model.evaluate(test_n, verbose=2)

print ()

print('"Testing set Mean Abs Error: {:5.2f} thousand dollars'.
      format(mae))
```

MAE provides a good idea of performance for linear continuous data in an easy-to-understand manner. With this dataset, we can expect model predictions to be off by the MAE value in thousands of dollars on average.

Make Predictions

Use the *predict* method to make predictions from processed test data *test_n*:

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

Display the *first* prediction:

```
# predicted housing price
first = predictions[0]
print ('predicted price:', first[0], 'thousand')

# actual housing price
print ('actual price:', y_test[0], 'thousand')
```

Compare predicted and actual prices to gauge model performance.

Display the first five predictions and compare against actual target values:

```
five = predictions[:5]
actuals = y_test[:5]
print ('pred', 'actual')
for i, p in enumerate(range(5)):
  print (np.round(five[i][0],1), actuals[i])
```

Visualize Predictions

Listing 6-10 displays predictions against actual housing prices.

Listing 6-10. Predictions against actual prices plot

```
fig, ax = plt.subplots()
ax.scatter(y_test, predictions)
ax.plot([y_test.min(), y_test.max()],\
        [y_test.min(), y_test.max()], 'k--', lw=4)
ax.set_xlabel('Measured')
ax.set_ylabel('Predicted')
plt.show()
```

The diagonal line is a plot of the actual housing prices. The further away a prediction is from the diagonal, the more erroneous it is.

Load Boston Data from Scikit-Learn

Since Boston data is included in *sklearn.datasets,* we can load it from this environment:

```
from sklearn import datasets

dataset = datasets.load_boston()
data, target = dataset.data, dataset.target
```

Access keys:

```
dataset.keys()
```

The list of keys informs us about accessing feature names:

```
feature_names = dataset.feature_names
feature_names
```

Notice that the sklearn dataset has an extra feature *B*. This column might be considered by some to be controversial because it singles out Black (or African American) people in a township.

We want to remove noise from the entire dataset, so build a dataframe with feature data and add target data:

```
df_sklearn = pd.DataFrame(dataset.data, columns=feature_names)

df_sklearn['MEDV'] = dataset.target
df_sklearn.head()
```

Check information:

```
df_sklearn.info()
```

Remove Noise

Remove noisy data:

```
# remove noisy data

print ('data set before removing noise:', df_sklearn.shape)

noise = df_sklearn.loc[df_sklearn['MEDV'] >= 50]
df_clean = df_sklearn.drop(noise.index)

print ('data set without noise:', df_clean.shape)
```

Build the Input Pipeline

Build the pipeline as shown in Listing 6-11.

Listing 6-11. Build the input pipeline

```
# create a copy of the dataframe
df = df_clean.copy()

# create the target
target = df.pop('MEDV')

# convert features and target data
features = df.values
labels = target.values

# create train and test sets

X_train, X_test, y_train, y_test = train_test_split(
    features, labels, test_size=0.33, random_state=0)

X_train_std = scaler.fit_transform(X_train)
X_test_std = scaler.fit_transform(X_test)

# slice data into a TensorFlow consumable form

train = tf.data.Dataset.from_tensor_slices(
    (X_train_std, y_train))
test = tf.data.Dataset.from_tensor_slices(
    (X_test_std, y_test))

# finalize the pipeline

BATCH_SIZE = 16
SHUFFLE_BUFFER_SIZE = 100

train_sk = train.shuffle(
    SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE).prefetch(1)
test_sk = test.batch(BATCH_SIZE).prefetch(1)
```

Model Data

Model data as shown in Listing 6-12.

Listing 6-12. Model data

```
# clear any previous model
tf.keras.backend.clear_session()

# generate a seed for replication purposes
np.random.seed(0)
tf.random.set_seed(0)

# new model with 13 input features

model = Sequential([
  Dense(64, activation='relu', input_shape=[13,]),
  Dense(64, activation='relu'),
  Dense(1)
])

# compile the new model

rmse = tf.keras.metrics.RootMeanSquaredError()

model.compile(loss='mse', optimizer='RMSProp',
              metrics=[rmse, 'mae', 'mse'])

# train

n = 4

early_stop = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
                                              patience=n)

history = model.fit(train_sk, epochs=50, validation_data=test_sk,
                    callbacks=[early_stop])
```

Model Cars Data

Let's practice with another dataset.

Get Cars Data from GitHub

We've already located the URL and assigned it to a variable:

```
cars_url = 'https://raw.githubusercontent.com/paperd/tensorflow/\
master/chapter6/data/cars.csv'
```

Read data into a pandas dataframe:

```
cars = pd.read_csv(cars_url)
```

Verify data:

```
cars.head()
```

Get information about the dataset:

```
cars.info()
```

Convert Categorical Column to Numeric

Machine learning algorithms can only train numeric data. So we must convert any non-numeric feature. The *Origin* column is categorical, not numeric. To remedy, one solution is to encode the data as one-hot. **One-hot encoding** is a process that converts categorical data into a numeric form for machine learning algorithm consumption.

We start by slicing off the Origin feature column from the original dataframe into its own dataframe. We then use this dataframe as a template to build a new feature column in the original dataframe for each category from the original Origin feature.

Create a copy of the dataframe:

```
# create a copy of dataframe
df = cars.copy()

origin = df.pop('Origin')
```

Define one-hot encoded feature columns for US, Europe, and Japanese cars:

```
df['US'] = (origin == 'US') * 1.0
df['Europe'] = (origin == 'Europe') * 1.0
df['Japan'] = (origin == 'Japan') * 1.0
df.tail(8)
```

For *US* origin, we assign *1.0 0.0 0.0*. For *Europe* origin, we assign *0.0 1.0 0.0*. For *Japan* origin, we assign *0.0 0.0 1.0*. So we replace the Origin feature with three one-hot encoded features.

Alternatively, we can one-hot encode with pandas. Start by creating a copy of the dataframe:

```
# create a copy of df
alt = cars.copy()
```

One-hot encode:

```
# get one hot encoding of columns 'Origin'
one_hot = pd.get_dummies(alt['Origin'])
```

Drop the original column:

```
# drop column as it is now encoded
alt = alt.drop('Origin', axis=1)
```

Join the encoded column to the dataframe:

```
# join the encoded df
alt = alt.join(one_hot)
alt.tail(8)
```

Slice Extraneous Data

Since the name of each car has no impact on any predictions we might want to make, we can tuck it away into its own dataframe in case we want to revisit it in the future:

```
try:
  name = df.pop('Car')
except:
  print("An exception occurred")
```

If an exception occurs, the *Car* column has already been removed. You can run this piece of code several times with no impact to results.

Verify:

```
df.tail(8)
```

Create Features and Labels

Our goal is to predict miles per gallon for cars in this dataset. So the target is *MPG*, and the features are the remaining feature columns.

Create feature and target sets as shown in Listing 6-13.

Listing 6-13. Create feature and target sets

```
# create data sets
features = df.copy()
target = features.pop('MPG')

# get feature names
feature_cols = list(features)
print (feature_cols)

# get number of features
num_features = len(feature_cols)
print (num_features)

# convert feature and target data to float
features = features.values
labels = target.values
(type(features), type(labels))
```

Build the Input Pipeline

Build the input pipeline as shown in Listing 6-14.

Listing 6-14. Build the input pipeline

```
# split
X_train, X_test, y_train, y_test = train_test_split(
    features, labels, test_size=0.33, random_state=0)
```

```
print ('X_train shape:', end=' ')
print (X_train.shape, br)

print ('X_test shape:', end=' ')
print (X_test.shape)

# scale
X_train_std = scaler.fit_transform(X_train)
X_test_std = scaler.fit_transform(X_test)

# slice

train = tf.data.Dataset.from_tensor_slices(
    (X_train_std, y_train))
test = tf.data.Dataset.from_tensor_slices(
    (X_test_std, y_test))

# shuffle, batch, prefetch

BATCH_SIZE = 16
SHUFFLE_BUFFER_SIZE = 100

train_cars = train.shuffle(
    SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE).prefetch(1)
test_cars = test.batch(BATCH_SIZE).prefetch(1)
```

Inspect tensors:

```
train_cars, test_cars
```

Model Data

Model data as shown in Listing 6-15.

Listing 6-15. Model data

```
# clear any previous model
tf.keras.backend.clear_session()

# create the model
```

```
model = Sequential([
  Dense(64, activation='relu', input_shape=[num_features]),
  Dense(64, activation='relu'),
  Dense(1)
])

# compile
rmse = tf.keras.metrics.RootMeanSquaredError()
optimizer = tf.keras.optimizers.RMSprop(0.001)

model.compile(loss='mse', optimizer=optimizer,
              metrics=[rmse, 'mae', 'mse'])

# train
tf.keras.backend.clear_session()

n = 4

early_stop = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', patience=n)

car_history = model.fit(train_cars, epochs=100, validation_data=test_cars,
                        callbacks=[early_stop])
```

Inspect the Model

Summary:

```
model.summary()
```

Output shape of the first layer is (None, 64) because we have 64 neurons at this layer. We get parameters of 640 by multiplying 64 neurons at this layer by 9 features and adding 64 neurons at this layer.

Output shape of the second layer is (None, 64) because we have 64 neurons at this layer. We get parameters of 4,160 by multiplying 64 neurons at this layer by 64 neurons from the previous layer and adding 64 at this layer.

Output shape of the third layer is (None, 1) because we have one target. We get parameters of 65 by adding 64 neurons from the previous layer to 1 neuron at this layer.

Visualize Training

Visualize:

```
hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
```

```
train_limit, test_limit = 10, 100
```

```
plot_history(history, train_limit, test_limit)
```

Generalize on Test Data

Generalize:

```
loss, rmse, mae, mse = model.evaluate(test_cars, verbose=2)
```

```
print ()
print('Testing set Mean Abs Error: {:5.2f} MPG'.format(mae))
```

Make Predictions

Predictions:

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

Visualize Predictions

Visualize predictions as shown in Listing 6-16.

Listing 6-16. Visualize predictions for cars data

```
fig, ax = plt.subplots()
ax.scatter(y_test, predictions)
ax.plot([y_test.min(), y_test.max()],\
        [y_test.min(), y_test.max()], 'k--', lw=4)
ax.set_xlabel('Measured')
ax.set_ylabel('Predicted')
plt.show()
```

The further the prediction is away from the diagonal true values line, the more erroneous it is.

CHAPTER 7

Convolutional Neural Networks

With feedforward neural networks, we achieved good training performance with MNIST and Fashion-MNIST datasets. But images in these datasets are simple and centered within the input space that contains them. That is, they are centered within the pixel matrix that holds them. **Input space** is all the possible inputs to a model.

Feedforward neural networks are very good at identifying patterns. So, if images occupy the same position within their input space, feedforward nets can quickly and effectively identify image patterns. And, if images are simple in terms of number of image pixels, patterns emerge more easily. But, if images don't occupy the same positions in their input spaces, feedforward nets have great difficulty identifying patterns and thereby perform horribly! So we need a different model to work with these types of images.

We can train complex and off-center images with convolutional neural networks and get good results. A **convolutional neural network** (CNN or ConvNet) is a class of deep neural networks most commonly applied to visual imagery analysis. CNNs are inspired by their resemblance to biological processes in that the connectivity between neurons resembles the organization of the visual cortex in humans.

A CNN works differently than a feedforward network because it treats data as spatial. Instead of neurons being connected to every neuron in the previous layer, they are instead *only* connected to neurons close to them, and all have the same weight. The simplification in the connections means that the network upholds the spatial aspect of the dataset.

Suppose the image is a profile of a child's face. A CNN doesn't think the child's eye is repeated all over the image. It can efficiently locate the child's eye in the image because of the filtering process it undertakes.

D. Paper, *TensorFlow 2.x in the Colaboratory Cloud*, https://doi.org/10.1007/978-1-4842-6649-6_7

A CNN processes an image dataset by assigning importance to various elements in each image, which enables it to differentiate between images. Importance is calibrated with learnable weights and biases. Preprocessing is much lower when compared to other classification algorithms because a CNN has the ability to learn how to adjust its filters during training.

The core building block of a CNN is the convolutional layer. A **convolutional layer** contains a series of filters that transform an input image by extracting features into feature maps for consumption by the next layer in the network. The transformation convolves the image with a set of learnable filters (or convolutional kernels) that have a small receptive field.

Convolution preserves the relationship between pixels by learning image features using small squares of input data. A **convolutional kernel** is a filter used on a *subset* of the pixel values of an input image. So a convolutional kernel is one of the small squares of input data that learns image features. A **receptive field** is the part of an image where a convolutional kernel operates at a given point in time. *Feature maps* of a CNN capture the result of applying convolutional kernels to an input image. So individual neurons respond to stimuli *only* in a restricted region of the receptive field (or visual field).

Whew! Simply, a convolutional kernel is a small matrix with its height and width smaller than the image to be convolved. During training, the kernel slides across the entire height and width of the input image, and the dot product of the kernel and the image is computed at every spatial position of the image. These computations create feature maps as output. So the entire image is convolved by a convolutional kernel! Such convolution is the key to the efficiency of a CNN because the filtering process allows it to adjust filter parameters during training.

We begin by discussing the CNN architecture. We start with some sample images to help you understand the type of data we are working with. We continue by building a complete CNN experiment. We work with the famous *cifar10* dataset. This dataset contains 60,000 images created to allow deep learning aficionados to create and test deep learning models. We demonstrate how to load the data, build the input pipeline, and model the data. We also show you how to make predictions.

Notebooks for chapters are located at the following URL: *https://github.com/paperd/tensorflow*.

Enable the GPU (if not already enabled):

1. Click *Runtime* in the top-left menu.
2. Click *Change runtime type* from the drop-down menu.
3. Choose *GPU* from the *Hardware accelerator* drop-down menu.
4. Click *SAVE*.

Test if GPU is active:

```
import tensorflow as tf

# display tf version and test if GPU is active
tf.__version__, tf.test.gpu_device_name()
```

Import the *tensorflow* library. If '/device:GPU:0' is displayed, the GPU is active. If '.' is displayed, the regular CPU is active.

CNN Architecture

Like a feedforward neural network, a CNN consists of multiple layers. However, the convolutional layer and pooling layer make it unique. Like other neural networks, it also has a ReLU (rectified linear unit) layer and a fully connected layer. The ReLU layer in *any* neural net acts as an activation function ensuring nonlinearity as the data moves through each layer of the network. Without ReLU activation, the data being fed into each layer would lose the dimensionality that we want it to maintain. That is, we would lose the integrity of the original data as it moves through the network. The fully connected layer allows a CNN to perform classification on the data.

As noted earlier, the most important building block of a CNN is the convolutional layer. Neurons in the first convolutional layer are connected to every single pixel in the input image, but only to pixels in their receptive fields, that is, only to pixels close to them. A convolutional layer works by placing a filter (or convolutional kernel) over an array of image pixels. The filtering process creates a *convolved feature map*, which is the output of a convolutional layer.

A *feature map* is created by projecting input features from our data to hidden units to form new features to feed to the next layer. A *hidden unit* corresponds to the output of a single filter at a single particular x/y offset in the input volume. Simply, a hidden unit is the value at a particular x,y,z coordinate in the output volume.

Once we have a convolved feature map, we move to the pooling layer. The *pooling layer* subsamples a particular feature map. Subsampling shrinks the size of the input image to reduce computational load, memory usage, and the number of parameters. Reducing the number of parameters that the network needs to process also limits the risk of overfitting. The output of the pooling layer is a *pooled feature map*.

We can pool feature maps in two ways. *Max pooling* takes the maximum input of a particular convolved feature map. *Average pooling* takes the average input of a particular convolved feature map.

The process of creating pooled feature maps results in feature extraction that enables the network to build up a picture of the image data. With a picture of the image data, the network moves into the fully connected layer to perform classification. As we did with feedforward nets, we flatten the data for consumption by the fully connected layer because it can only process linear data.

Conceptually, a CNN is pretty complex as you can tell from our discussion. But implementing a CNN in TensorFlow is pretty straightforward. Each input image is typically represented as a 3D tensor of shape *height*, *width*, and *channels*. When classifying a 3D color image, we feed CNN image data in three channels, namely, *red*, *green*, and *blue*. Color images are typically referred to as *RGB* images. A batch (e.g., mini-batch) is represented as a 4D tensor of shape *batch size*, *height*, *width*, and *channels*.

Load Sample Images

The scikit-learn *load_sample_image* method allows us to practice with two color images – *china.jpg* and *flower.jpg*. The method loads the numpy array of a single sample image and returns it as a 3D numpy array consisting of height by width by color.

Load the images:

```
from sklearn.datasets import load_sample_image

china, flower = load_sample_image('china.jpg'),\
load_sample_image('flower.jpg')
china.shape, flower.shape
```

Both china and flower images are represented as 427 × 640-pixel matrices with three channels to account for RGB color.

Display Images

Listing 7-1 displays the images.

Listing 7-1. Display china and flower images

```
import matplotlib.pyplot as plt

# function to plot RGB images
def plot_color_image(image):
    plt.imshow(image, interpolation="nearest")
    plt.axis("off")

plot_color_image(china)
plt.show()
plot_color_image(flower)
```

Scale Images

Scaling images improves training performance. Since each image pixel is represented by a byte from 0 to 255, we divide each image by 255 to scale it.

Listing 7-2 scales images.

Listing 7-2. Scale images

```
import numpy as np

# slice off a few pixels prior to scaling

br = '\n'

print ('pixels as loaded:', br)

print ('china pixels:', end = '  ')
print (np.around(china[0][0]))

print ('flower pixels:', end = ' ')
print (np.around(flower[0][0]), br)

# scale images
china_sc, flower_sc = china / 255., flower / 255.
```

```
# slice off some pixels to verify that scaling worked

print ('pixels scaled:', br)

print ('china pixels:', end = '  ')
print (np.around(china_sc[0][0], decimals=3))

print ('flower pixels:', end = ' ')
print (np.around(flower_sc[0][0], decimals=3))
```

Scaling worked because pixel intensity is between 0 and 1.

Display Scaled Images

Plot scaled images:

```
plot_color_image(china_sc)
plt.show()
plot_color_image(flower_sc)
```

Scaling doesn't impact the images, which makes sense because scaling modifies pixel intensity proportionally. That is, each pixel number is converted proportionally to a number between 0 and 1.

Get More Images

Let's get a couple more images. To get images for this book, just follow these simple steps:

1. Go to the GitHub URL for this book: *https://github.com/paperd/tensorflow*.
2. Locate the image you want to download and click it.
3. Click the *Download* button.

4. Right-click anywhere inside the image.
5. Click *Save image as…*.
6. Save the image on your computer.
7. Drag and drop the image to your Google Drive *Colab Notebooks* folder.
8. Repeat steps 1–7 as necessary for multiple images.

For this lesson, go to the book URL, click *chapter7*, click *images*, click *fish.jpg*, click the *Download* button, right-click inside the image, and click *Save image as…* to save it on your computer. Drag and drop the image to your Google Drive *Colab Notebooks* folder. Repeat the same process for the *happy_moon.jpg* image.

Mount Google Drive

Mount Colab to Google Drive:

```
from google.colab import drive
drive.mount('/content/drive')
```

Click the URL, choose a Google account, click *Allow*, copy the authorization code and paste it into Colab in the textbox *Enter your authorization code:*, and press the *Enter* key on your keyboard.

Copy Images to Google Drive

Before executing the code in this section, be sure that you have the *fish.jpg* and *happy_moon.jpg* images in the *Colab Notebooks* directory on your Google Drive!

Check your Google Drive account to verify the proper path. We saved the images to the Colab Notebooks directory, which is recommended. If you save them somewhere else, you must change the paths accordingly.

Listing 7-3 loads the images to the Colab environment, scales the images, and displays them.

Listing 7-3. Load, scale, and display images

```
# be sure to copy images to the directory on Google Drive

from PIL import Image
import numpy as np

# create paths to images
fish_path = 'drive/My Drive/Colab Notebooks/fish.jpg'
moon_path = 'drive/My Drive/Colab Notebooks/happy_moon.jpg'

# create images
fish, moon  = Image.open(fish_path), Image.open(moon_path)

# convert images to numpy arrays and scale
fish_np, moon_np = np.array(fish), np.array(moon)
fish_sc, moon_sc = fish_np / 255., moon_np / 255.

# display images
plot_color_image(fish_sc)
plt.show()
plot_color_image(moon_sc)
```

Verify that new images were scaled properly:

```
# slice off some pixels and display

print ('fish pixels:', end = ' ')
print (np.around(fish_sc[0][0], decimals=3), br)

print ('moon pixels:', end = ' ')
print (np.around(moon_sc[0][0], decimals=3))
```

All is well so far.

Check Image Shapes

For machine learning applications, images must be of the *same* shape.

Let's explore image shapes:

```
print ('original shapes:')
display (china_sc.shape, flower_sc.shape)
print (), print ('new shapes:')
display (fish_sc.shape, moon_sc.shape)
```

Uh-oh! Shapes are not the same! What do we do?

Resize Images

Let's resize the fish and moon images to equalize shapes:

```
fish_rs = np.array(tf.image.resize(
    fish_sc, [427, 640]))
```

```
moon_rs = np.array(tf.image.resize(
    moon_sc, [427, 640]))
```

```
fish_rs.shape, moon_rs.shape
```

Now, all four images have size of (427, 640, 3).

Plot the resized images:

```
plot_color_image(fish_rs)
plt.show()
plot_color_image(moon_rs)
```

Success! We resized the new images to correspond to the original ones.

Create a Batch of Images

Create a batch with all four images:

```
new_images = np.array([china_sc, flower_sc,
                       fish_rs, moon_rs])
```

```
new_images.shape
```

Now, we have a batch of *four* 427 × 640 color images. RGB color is indicated by the 3 dimension.

Create Filters

Let's create two simple 7 × 7 filters. We want our first filter to have a vertical white line in the middle and our second to have a horizontal white line in the middle. *Filters* are used to extract features from images during the process of convolution. Typically, filters are referred to as *convolutional kernels*.

Create the filters:

```
# assign some variables
batch_size, height, width, channels = new_images.shape

# create 2 filters
ck = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
ck.shape
```

The *zeros* method returns a given shape and type filled with zeros. Since variable *ck* is filled with zeros, all of its pixels are black. Remember that pixel image values are integers that range from 0 (black) to 255 (white).

So *ck* is a 4D tensor that contains *two* 7 × 7 convolutional kernels with three channels. The filters must have three channels to match the color images in the batch of images we created.

Add a vertical white line and horizontal white line:

```
ck[:, 3, :, 0] = 1  # add vertical line
ck[3, :, :, 1] = 1  # add horizontal line
```

The code changes the intensity of select pixels to get a vertical white line and a horizontal white line.

Plot Convolutional Kernels

Listing 7-4 plots the two convolutional kernels we just created.

Listing 7-4. Convolutional kernel plots

```
# function to plot filters

def plot_image(image):
    plt.imshow(image, cmap="gray", interpolation="nearest")
    plt.axis("off")

print ('vertical convolutional kernels:')

plot_image(ck[:, :, 0, 0])
plt.show()

print ('horizontal convolutional kernels:')
plot_image(ck[:, :, 0, 1])
```

We see that the vertical and horizontal white lines (or convolutional kernels) are in position. So we have successfully created two simple convolutional kernels.

Apply a 2D Convolutional Layer

Apply a 2D convolutional layer to the batch of images:

```
# apply a 2D convolutional layer
outputs = tf.nn.conv2d(new_images, ck, strides=1, padding='SAME')
```

The *tf.nn.conv2d* method computes a 2D convolution given 4D input and convolutional kernel tensors. We set strides equal to *1*. A **stride** is the number of pixels we shift the convolutional kernels over the input matrix during training. With strides of 1, we move the convolutional kernels one pixel at a time. We set padding to *SAME*. **Padding** is the number of pixels added to an image when it is being processed by the CNN. For example, if padding is set to zero, every pixel value that is added will be of value zero. Padding set to SAME means that we use zero padding.

After applying the convolutional layer, the variable *outputs* contains the feature maps based on our images. Since each convolutional kernel creates a feature map (and we have two convolutional kernels), each image has two feature maps.

Visualize Feature Maps

Visualize the feature maps we just created as shown in Listing 7-5.

Listing 7-5. Feature maps plot

```
rows = 4  # one row for each image
columns = 2  # two feature maps for each image
cnt = 1
fig = plt.figure(figsize=(8, 8))
for i, img in enumerate(outputs):
  for j in (0, 1):
    fig.add_subplot(rows, columns, cnt)
    plt.imshow(outputs[i, :, :, j], cmap='gray')
    plt.axis('off')
    cnt += 1
plt.show()
```

Since we have two convolutional kernels and four images, the convolutional layer produces eight feature maps. Just multiply 2 by 4! So we have two feature maps for each image. Wow! With two simple convolutional kernels, we were able to extract excellent facsimiles of our batch of images by applying a single convolutional layer.

CNN with Trainable Filters

We just *manually* defined two convolutional kernels. But, in a real CNN, we typically define convolutional kernels as trainable variables so the neural net can learn the convolutional kernels that work best.

Create a simple model that lets the network decide the best convolutional kernels:

```
conv = tf.keras.layers.Conv2D(filters=32, kernel_size=3,
                              strides=1, padding='SAME',
                              activation='relu')
```

We create a *Conv2D* layer with 32 convolutional kernels. Each convolutional kernel is a 3 × 3 tensor indicated by *kernel_size*. We use strides of *1* both horizontally and vertically. Padding is *SAME*. Finally, we apply *relu* activation to the output.

Convolutional layers have quite a few hyperparameters including number of filters (or convolutional kernels), height and width of convolutional kernels, strides, padding type, and activation type. To get the best performance, we can tune the hyperparameters. But tuning is an advanced topic that we believe is not appropriate for an introductory book. Instead, we provide fundamental examples that you can practice to develop practical skills.

Building a CNN

Although a CNN is a sequential neural net, it does differ from a feedforward sequential neural net in two important ways. First, it has a convolutional base that is not fully connected. Second, it has a pooling layer to reduce the sample size of feature maps created by each convolutional layer. We still use a fully connected layer for classification.

We begin this experiment by loading a dataset of color images. We continue by preparing the data for TensorFlow consumption. We then build and test a CNN model. The dataset we use is *cifar10*. We previously modeled this dataset with a feedforward model, but our results were horrible. So we want to show you how much better a CNN works with complex color images.

Load Data

The recommended way to load cifar10 is as a TFDS:

```
# import TFDS library
import tensorflow_datasets as tfds
```

Load train and test sets:

```
train, info = tfds.load('cifar10', split='train',
                        with_info=True, shuffle_files=True)
test = tfds.load('cifar10', split='test')
```

Since we already have *info* from the train set, we don't need it again for the test set.

Verify tensors:

```
train.element_spec, test.element_spec
```

Train and test shapes are 32 × 32 × 3. So each image is a 32 × 32 three-channel image. The *3* dimension informs the model that images are RGB color.

Display Information About the Dataset

The *info* object displays information about the data:

```
info
```

We see the name, description, homepage, and shapes and datatypes of feature images and labels. We also see that the dataset has 60,000 images with train and test splits of 50,000 and 10,000, respectively.

Extract Class Labels

Extract some useful information from the *info* object:

```
br = '\n'

num_classes = info.features['label'].num_classes
class_labels = info.features['label'].names
print ('number of classes:', num_classes, br)
print ('class labels:', class_labels)
```

Display Samples

The *show_examples* method shows a few examples:

```
fig = tfds.show_examples(train, info)
```

Build a Custom Function to Display Samples

Listing 7-6 is a function that displays samples.

Listing 7-6. Function to display samples

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

def display_samples(data, num, cmap):
  for example in data.take(num):
    image, label = example['image'], example['label']
    print ('Label:', class_labels[label.numpy()], end=', ')
```

```
    print ('Index:', label.numpy())
    plt.imshow(image.numpy()[:, :, 0].astype(np.float32),
               cmap=plt.get_cmap(cmap))
    plt.show()
```

The function retrieves images and label names. It then displays the image with its label name and index. The index is the class label as a number.

Invoke the function:

```
# choose colormap by changing 'indx'

cmap = ['coolwarm', 'viridis', 'plasma',
        'seismic', 'twilight', 'Spectral']
indx, samples = 5, 3
display_samples(train, samples, cmap[indx])
```

Change the color by adjusting *indx* between 0 and 5. Change the number of samples displayed by adjusting *samples*. Peruse the following URL to learn more about colormaps: *https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html*.

Build a Custom Function to Display a Grid of Examples

Begin by taking 30 examples from the train set as shown in Listing 7-7.

Listing 7-7. Processed examples from the train set

```
num = 30
images, labels = [], []
for example in train.take(num):
  image, label = example['image'], example['label']
  images.append(tf.squeeze(image.numpy()))
  labels.append(label.numpy())
```

To enable image plotting, we remove (or squeeze) the 3 dimension from the image matrix.

Build the function as shown in Listing 7-8.

Listing 7-8. Function to display a grid of examples

```
def display_grid(feature, target, n_rows, n_cols, cl):
  plt.figure(figsize=(n_cols * 1.5, n_rows * 1.5))
  for row in range(n_rows):
    for col in range(n_cols):
      index = n_cols * row + col
      plt.subplot(n_rows, n_cols, index + 1)
      plt.imshow(feature[index], cmap='binary',
                 interpolation='nearest')
      plt.axis('off')
      plt.title(cl[target[index]], fontsize=12)
  plt.subplots_adjust(wspace=0.2, hspace=0.5)
```

Invoke the function:

```
rows, cols = 5, 6
display_grid(images, labels, rows, cols, class_labels)
```

Voilà!

Pinpoint Metadata

Leverage the info object to pinpoint metadata:

```
print ('Number of training examples:', end=' ')
print (info.splits['train'].num_examples)

print ('Number of test examples:', end=' ')
print (info.splits['test'].num_examples)
```

Build the Input Pipeline

Build the input pipeline as shown in Listing 7-9.

Listing 7-9. Build the input pipeline

```
BATCH_SIZE = 128
SHUFFLE_SIZE = 5000

train_1 = train.shuffle(SHUFFLE_SIZE).batch(BATCH_SIZE)
train_2 = train_1.map(lambda items: (
    tf.cast(items['image'], tf.float32) / 255., items['label']))
train_cf = train_2.cache().prefetch(1)

test_1 = test.batch(BATCH_SIZE)
test_2 = test_1.map(lambda items: (
    tf.cast(items['image'], tf.float32) / 255., items['label']))
test_cf = test_2.cache().prefetch(1)
```

We build the input pipeline by shuffling train data, batching, scaling, caching, and prefetching. We scale images by mapping with a lambda function. Adding the *cache* method increases performance on a TFDS because data is read and written only once rather than during each epoch. Adding the *prefetch* method is a good idea because it adds efficiency to the batching process. That is, while our training algorithm is working on one batch, TensorFlow is working on the dataset in parallel to get the next batch ready. So prefetch can dramatically improve training performance.

Verify that train and test sets are created correctly:

```
train_cf.element_spec, test_cf.element_spec
```

Create the Model

Begin with a relatively robust CNN model because it is the only way to get decent performance from complex color images. Don't be daunted by the number of layers! Remember that a CNN has a convolutional base and fully connected network. So we can think of a CNN in two parts. First, we build the convolutional base that includes one or more convolutional layers and pooling layers. Pooling layers are included to subsample the feature maps outputted from convolutional layers to reduce computational expense. Next, we build a fully connected layer for classification.

Create the model by following these steps:

1. Import libraries.
2. Clear previous models.
3. Create the model.

Import libraries:

```
# import libraries

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D,\
Dense, Flatten, Dropout
```

Clear previous models and plant a seed:

```
# clear previous models and generate a seed
tf.keras.backend.clear_session()
np.random.seed(0)
tf.random.set_seed(0)
```

Create the model as shown in Listing 7-10.

Listing 7-10. Build the model

```
# build the model

model = Sequential([
  Conv2D(32, (3, 3), activation = 'relu', padding='same',
         input_shape=[32, 32, 3], strides=1),
  MaxPooling2D(2),
  Conv2D(64, (3, 3), activation='relu', padding='same'),
  MaxPooling2D(2),
  Conv2D(64, (3, 3), activation='relu', padding='same'),
  Flatten(),
  Dense(64, activation='relu'),
  Dropout(0.5),
  Dense(10, activation='softmax')
])
```

The first layer is the convolutional base, which uses 32 convolutional kernels and kernel size of 3 × 3. We use *relu* activation, *same* padding, and strides of *1*. We also set shape at 32 × 32 × 3 to match the 32 × 32-pixel images. Since images are in color, we include the 3 value at the end. Next, we include a max pooling layer of size 2 (so it divides each spatial dimension by a factor of 2) to subsample the feature maps from the first convolutional layer. We then repeat the same structure twice, but increase the number of convolutional kernels to 64. It is common practice to double the number of convolutional kernels after each pooling layer.

We continue with the fully connected network, which flattens its inputs because a dense network expects a 1D array of features for each instance. We need to add the fully connected layer to enable classification of our ten labels. We continue with a dense layer of 64 neurons. We add dropout to reduce overfitting. The final dense layer accepts ten inputs to match the number of labels. It uses *softmax* activation.

Model Summary

Inspect the model:

```
model.summary()
```

Parameters are the number of learnable weights during training. The convolutional layer is where the CNN begins to learn. But calculating parameters for a CNN is more complex than a feedforward network.

The first layer is a convolutional layer with 32 neurons acting on the data. Filter size is 3 × 3. So we have a 3 × 3 × 32 filter since our input has 32 dimensions (or neurons) for a total of 288. Multiply 288 by 3 to account for the 3D RGB color images for a total of 864. Add 32 neurons at this layer to get a total of 896 parameters.

There are no parameters to learn at the pooling layers. So we have 0 parameters.

The second convolutional layer has 64 neurons acting on the data. Filter size is 3 × 3. So we have a 3 × 3 × 64 filter since we have 64 dimensions for a total of 576. Multiply 32 neurons from the previous convolutional layer by 576 to get a total of 18,432. Add 64 neurons at this layer to get a total of 18,496 parameters.

The third convolutional layer has 64 neurons acting on the data. Filter size is 3 × 3. So we have a 3 × 3 × 64 filter since we have 64 dimensions for a total of 576. Multiply 64 neurons from the previous convolutional layer by 576 to get a total of 36,864. Add 64 neurons at this layer to get a total of 36,928 parameters.

The fully connected dense layer is calculated as before. We get 262,144 by multiplying 4,096 neurons at this layer by 64 neurons from the previous layer. Add 64 neurons at this layer to get a total of 262,208 parameters.

The output layer has 650 parameters by multiplying 64 neurons from the previous layer by 10 at this layer and adding 10 neurons at this layer. Whew!

Model Layers

Inspect model layers:

```
model.layers
```

Compile the Model

Through experimentation, we found that the *Nadam* optimizer performed the best:

```
model.compile(optimizer='nadam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Train the Model

Train the model for ten epochs:

```
epochs = 10
history = model.fit(train_cf, epochs=epochs,
                    verbose=1, validation_data=test_cf)
```

Although our model is not state of the art, we do much better than we did with a feedforward net.

Generalize on Test Data

Generalize:

```
print('Test accuracy:', end=' ')
test_loss, test_acc = model.evaluate(test_cf, verbose=2)
```

Visualize Training Performance

Listing 7-11 visualizes training.

Listing 7-11. Visualization of training performance

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')
plt.show()
```

```
plt.plot(history.history['loss'], label='loss')
plt.plot(history.history['val_loss'], label = 'val_loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.ylim([0.5, 1.0])
plt.legend(loc='lower right')
plt.show()
```

Predict Labels for Test Images

Predict on *test_cf* data:

```
predictions = np.argmax(model.predict(test_cf), axis=-1)
```

Wrap the *predict* method with the *argmax* method to get predicted labels directly rather than generating probability arrays.

Get predictions by class number:

```
# predictions by class number
predictions
```

Get predictions by label:

```
# predictions by class label
np.array(class_labels)[predictions]
```

Get the first five predictions:

```
# 5 predictions
pred_5 = predictions[:5]
pred_5
```

Convert label numbers to label names:

```
pred_labels = np.array(class_labels)[pred_5]
pred_labels
```

Get the first five actual labels as shown in Listing 7-12.

Listing 7-12. First five actual labels

```
# take the first batch of images
ls = []
for _, label in test_cf.take(1):
  ls.append(label.numpy())

# slice first five from batch
actuals = ls[0][0:5]

# convert to labels
actuals = [class_labels[row] for row in actuals]
actuals
```

Get the first batch of images. Since we set batch size at 128, we get the first 128 images. Slice the first five images from the batch. Convert to label names.

Compare *pred_labels* to *actual_labels* to get an idea of prediction performance.

Build a Prediction Plot

Begin by taking 20 samples from the test set as shown in Listing 7-13.

Listing 7-13. Take samples from the test set

```
num = 20
images, labels = [], []
for example in test.take(num):
  image, label = example['image'], example['label']
  images.append(tf.squeeze(image.numpy()))
  labels.append(label.numpy())
```

Build a Custom Function

Build a function to display results as shown in Listing 7-14.

Listing 7-14. Function to display results

```
def display_test(feature, target, num_images,
                 n_rows, n_cols, cl, p):
  for i in range(num_images):
    plt.subplot(n_rows, 2*n_cols, 2*i+1)
    plt.imshow(feature[i])
    title_obj = plt.title(cl[target[i]] + ' (' +\ cl[p[i]] + ') ')
    if cl[target[i]] == cl[p[i]]:
      title_obj
    else:
      plt.getp(title_obj, 'text')
      plt.setp(title_obj, color='r')
  plt.tight_layout()
plt.show()
```

Invoke the function:

```
num_rows, num_cols = 5, 4
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
display_test(images, labels, num_images, num_rows,
             num_cols, class_labels, predictions)
```

Titles in red indicate misclassifications.

Build a CNN with Keras Data

Although loading data as a TFDS is recommended, Keras is very popular in industry. So let's build a model from *keras.datasets*.

Load train and test data:

```
train_k, test_k = tf.keras.datasets.cifar10.load_data()
```

Verify data shapes:

```
print ('train data:', br)
print (train_k[0].shape)
print (train_k[1].shape, br)
print ('test data:', br)
print (test_k[0].shape)
print (test_k[1].shape)
```

Create class label names:

```
class_labels = ['airplane', 'automobile', 'bird', 'cat', 'deer',
                'dog', 'frog', 'horse', 'ship', 'truck']
```

Create Variables to Hold Train and Test Samples

Create variables to hold train and test data as shown in Listing 7-15.

Listing 7-15. Create variables to hold train and test data

```
# create simple variables from train tuple
train_images = train_k[0]
train_labels = train_k[1]

# display first train label
print ('1st train label:', class_labels[train_labels[0][0]])

# create simple variables from test tuple
test_images = test_k[0]
test_labels = test_k[1]

# display first test label
print ('1st test label: ', class_labels[test_labels[0][0]])
```

Display Sample Images

It's always a good idea to display some images. In this case, we display 30 images from the training dataset. Visualization allows us to verify that images and labels correspond. That is, a frog image is labeled as a frog and so on.

Listing 7-16 displays sample images from the training set.

Listing 7-16. Display sample images

```
n_rows = 5
n_cols = 6
plt.figure(figsize=(n_cols * 1.5, n_rows * 1.5))
for row in range(n_rows):
  for col in range(n_cols):
    index = n_cols * row + col
    plt.subplot(n_rows, n_cols, index + 1)
    plt.imshow(train_images[index], cmap='binary',
               interpolation='nearest')
    plt.axis('off')
    plt.title(class_labels[int(train_labels[index])], fontsize=12)
plt.subplots_adjust(wspace=0.2, hspace=0.5)
```

Create the Input Pipeline

Build the input pipeline by scaling images and slicing them into TensorFlow consumable pieces. Continue by shuffling (where appropriate), batching, and prefetching.

Listing 7-17 creates the input pipeline.

Listing 7-17. Build the input pipeline

```
# scale images

train_img_sc = train_images / 255.  # divide by 255 to scale
train_lbls = train_labels.astype(np.int32)

test_img_sc = test_images/255.  # divide by 255 to scale
test_lbls = test_labels.astype(np.int32)
```

```
# slice data

train_ks = tf.data.Dataset.from_tensor_slices(
    (train_img_sc, train_lbls))
test_ks = tf.data.Dataset.from_tensor_slices(
    (test_img_sc, test_lbls))

# shuffle, batch, and prefetch

BATCH_SIZE = 128
SHUFFLE_BUFFER_SIZE = 5000

train_ds = train_ks.shuffle(
    SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE).prefetch(1)
test_ds = test_ks.batch(BATCH_SIZE).prefetch(1)
```

Inspect tensors:

```
train_ds, test_ds
```

Create the Model

Listing 7-18 creates the model.

Listing 7-18. Create the model

```
# import libraries
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D,\
Dense, Flatten, Dropout

# clear previous models and generate a seed
tf.keras.backend.clear_session()
np.random.seed(0)
tf.random.set_seed(0)

# build the model
model = Sequential([
  Conv2D(32, 3, activation = 'relu', padding='same',
         input_shape=[32, 32, 3]),
  MaxPooling2D(2),
```

```
  Conv2D(64, 3, activation='relu', padding='same'),
  MaxPooling2D(2),
  Conv2D(64, 3, activation='relu', padding='same'),
  Flatten(),
  Dense(64, activation='relu'),
  Dropout(0.5),
  Dense(10, activation='softmax')
])
```

Compile and Train

Listing 7-19 compiles and trains the model.

Listing 7-19. Compile and train the model

```
# compile
model.compile(optimizer='nadam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# train
epochs = 10
history = model.fit(train_ds, epochs=epochs,
                    verbose=1, validation_data=test_ds)
```

Predict

Predict from the model:

```
pred_ks = np.argmax(model.predict(test_images), axis=-1)
```

Visualize Results

Listing 7-20 visualizes training performance results.

Listing 7-20. Visualize training performance results

```
# plot the first X (num_rows * num_cols) test images
# (true and predicted labels)

num_rows = 5
num_cols = 4
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
  ax = plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plt.imshow(test_images[i])
  title = class_labels[int(test_labels[i])] +
  ' (' +\class_labels[pred_ks[i]] + ') '
  plt.title(title)
  if class_labels[int(test_labels[i])] !=\
     class_labels[pred_ks[i]]:
    ax.set_title(title, style='italic', color='red')
  plt.axis('off')
plt.tight_layout()
```

Epilogue

Many improvements to the fundamental CNN architecture have been developed over the past few years that vastly improve prediction performance. Although we don't cover these advances in this lesson, we believe that we provided the basic foundation with CNNs to help you comfortably work with these recent advances and even the many advances to come in the future.

CHAPTER 8

Automated Text Generation

Feedforward neural nets are generally great for classification and regression problems. CNNs are great for complex image classification. But activations for feedforward nets and CNNs flow only in one direction, from the input layers to the output layer. Since signals flow in only one direction, feedforward and convolutional nets are not ideal if patterns in data change over time. So we need a different network architecture to work with data impacted by time.

A **recurrent neural network** (RNN) looks a lot like a feedforward neural network, but it also has connections pointing backward. That is, output from one layer can act as input back into another layer earlier in the network. The capability of one layer to inform another layer earlier in the network means that a RNN has a built-in feedback loop mechanism that allows it to act as a forecasting engine. So RNNs are great as forecasters because they naturally work well as data changes with time.

A RNN remembers the past, and its decisions are influenced by what it has learned from the past. Feedforward and convolutional networks only remember what they learn during training. For example, a feedforward image classifier learns what an image looks like during training and then uses that knowledge to classify other images in production. While RNNs learn similarly during training, they also remember what they learned so they can make good decisions as data changes.

Notebooks for chapters are located at the following URL: *https://github.com/paperd/tensorflow*.

D. Paper, *TensorFlow 2.x in the Colaboratory Cloud*, https://doi.org/10.1007/978-1-4842-6649-6_8

Natural Language Processing

A fascinating advancement in machine learning is the ability to teach a machine how to understand human communication. The area of machine learning that concentrates on understanding how humans communicate is natural language processing. **Natural language processing** (NLP) is a field in machine learning concentrating on the ability of a computer to understand, analyze, manipulate, and potentially generate human language. RNNs are a very important variant of neural networks used heavily for NLP.

RNNs are great for NLP because their standard input is a word instead of the entire sample taken as standard input by sequential nets like feedforward and convolutional networks. So RNNs have the flexibility to work with varying lengths of sentences, which cannot be achieved by sequential neural networks because of their fixed structure. RNNs can also share features learned across different positions of text because of their flexible structure.

The feedback looping ability of a RNN allows it to parse each word of a sentence and run an activation on it. The activation value from the word can then be fed back to the layer that is parsing the sentence. So the activation value informs the sentence of what was learned from each word! And the cycle continues for each word until the network understands the sentence. In machine learning speak, a RNN treats each word of a sentence as a separate input occurring at a particular time 't' and uses the activation value for this input 't-1' as feedback to the original sentence.

Enable the GPU (if not already enabled):

1. Click *Runtime* in the top-left menu.
2. Click *Change runtime type* from the drop-down menu.
3. Choose *GPU* from the *Hardware accelerator* drop-down menu.
4. Click *SAVE*.

Test if GPU is active:

```
import tensorflow as tf

# display tf version and test if GPU is active
tf.__version__, tf.test.gpu_device_name()
```

Import the *tensorflow* library. If '/device:GPU:0' is displayed, the GPU is active. If '.' is displayed, the regular CPU is active.

Generating Text with a RNN

As noted, RNNs are commonly used for natural language tasks. Typically, we can model natural language tasks by character or word. We begin by building a *character-level* model that generates text. In the next chapter, we build a word-level model that predicts sentiment.

The Text File

We are going to work with the book *A Tale of Two Cities* by Charles Dickens. For convenience, we've already downloaded the plain text UTF-8 and processed it so you don't have to. To get the *processed* text file, just follow these simple steps:

1. Go to the GitHub URL for this book: *https://github.com/paperd/tensorflow*.
2. Locate the file: click *chapter8*, click *data*, and click *two_cities.txt.*
3. Click the *Raw* button.
4. Copy the text (*Ctrl*+*A*+*C*).
5. Paste it into *Notepad* or another basic text editor (*Ctrl*+*V*).
6. Save it on your computer as *two_cities.txt*.
7. Drag and drop the file to your Google Drive *Colab Notebooks* folder.

Note Please get the processed version because our code examples are based on this version. And we want to focus learning on text generation rather than text processing.

If you want to get the raw text file and process it yourself, go to URL *www.gutenberg.org/ebooks/98* and follow these steps:

1. Click *Plain Text UTF-8* for the format choice.
2. Copy the text (Ctrl+A+C).
3. Paste it into *Notepad* or another basic text editor.

4. Click *Save As…*.
5. Change Save as type to *All Files*.
6. Save it on your computer as *two_cities.txt*.
7. Process the text file.

An excellent guide to processing raw text from Project Gutenberg is at `https://jss367.github.io/Getting-text-from-Project-Gutenberg.html`.

Note If you process the raw text file yourself, *don't* use it with our code examples in this chapter because your processing may not coincide with our code. We just want to make it available if you want to practice text processing.

Mount Google Drive

We must mount Google Drive to enable access to the text file:

```
from google.colab import drive
drive.mount('/content/drive')
```

After you execute the code cell, click the URL, choose a Google account, click the *Allow* button, copy the authorization code and paste it into the textbox *Enter your authorization code:*, and press the *Enter* key on your keyboard.

Note Be sure that you have the file in the *Colab Notebooks* directory on your Google Drive!

Read the Corpus into Memory

In NLP, a text document is often referred to as a corpus. A **corpus** is a collection of written texts, especially the entire works of a particular author or a body of writing on a particular subject.

Note All code is based on the *processed* version of the text file.

Read corpus into memory:

```
two_cities = 'drive/My Drive/Colab Notebooks/two_cities.txt'

with open(two_cities) as f:
  corpus = f.read()
```

Verify Corpus

Display text from the beginning of the corpus:

```
print (corpus[:74])
```

Since we start from the beginning, it is pretty easy to verify. But verifying the end takes a bit more work.

Get the length of the corpus:

```
len(corpus)
```

Now, we know where the corpus ends.

With a bit of trial and error, we can display the famous quote from the end:

```
print (corpus[757116:])
```

If you want to explore other online books for NLP, a great place to start is *Project Gutenberg* at the following URL: *www.gutenberg.org/*.

Create Vocabulary

Since our goal is to generate text with a character-level model, we train the model to predict the next character in a sequence. We can then repeatedly call the model to generate longer sequences of text.

Create a vocabulary of unique characters contained in the corpus:

```
# unique characters in the corpus
vocab = sorted(set(corpus))
print ('{} unique characters'.format(len(vocab)))
```

We have a vocabulary of 74 unique characters stored in *vocab*.

Vectorize the Text

Algorithms process numbers, not text. So we must devise a numerical representation of the corpus. An easy solution is to vectorize the text. **Text vectorization** is the process of converting text into numerical representation.

Let's start by creating dictionary *int_map* to hold integer mappings of unique characters. Next, create numpy array *char_map* to hold character mappings of each integer representation. The numpy array allows us to translate encoded integer mappings back into their character representations. Once we vectorize the corpus, we can build the input pipeline for TensorFlow consumption.

Create Integer Mappings

Create the dictionary:

```
# create a dictionary with integer representations of characters
int_map = {key : value for value, key in enumerate(vocab)}
int_map['a']
```

We use dictionary comprehension to create *int_map*, which holds integer mappings for the corpus. **Dictionary comprehension** is a method for creating dictionaries using simple expressions. A dictionary comprehension takes the form *{key: value for (key, value) in iterable}*. In our case, the key is a unique character in the corpus, and the value is the integer mapping of the unique character. So integer *45* represents the letter *a*.

Create Character Mappings

Create the numpy array that holds character mappings:

```
# create numpy array to hold character mappings of integers

import numpy as np

char_map = np.array(vocab)
char_map[45]
```

It looks like our mappings work.

Map a Sequence

Map a sequence as shown in Listing 8-1.

Listing 8-1. Map a text sequence

```
# create variable to hold line break
br = '\n'

# simple sequence
sequence = 'hello world'
print ('original sequence:', sequence, br)

# map to integer representations
maps = np.array([int_map[c] for c in sequence])
print ('integer mappings:', maps, br)

# map integer representations back into characters
s = [char_map[i] for i in maps]

# create string from list of characters
s = ''.join(s)
print ('translation:', s)
```

Sequence hello world is vectorized with *int_map* and translated back into the original sequence with *char_map*.

Vectorize the Corpus

Now that we have our mapping algorithms, we are ready to vectorize the corpus:

```
# vectorize the corpus
encoded = np.array([int_map[c] for c in corpus])
encoded[:20], char_map[encoded[:20]]
```

Numpy array *encoded* holds the vectorized corpus. We display 20 integer mappings and their character equivalents to verify that all is well.

Predict the Next Character

In a RNN, there are multiple activations on each neuron because of its built-in feedback loops. A **time step** is a single activation on a neuron. At each time step during training, our goal is to predict the next probable character given a character or sequence of characters. So input to the model must be a sequence of characters. But the sequence of characters must be constructed to work with our model.

Create Training Input Sequences

To create training instances, we divide the encoded corpus into input sequences. Each input sequence contains *seq_length* characters from the corpus. The *seq_length* is the maximum-length sentence we want for a single input sequence in characters. We break the corpus into equal-length sequences for better performance.

For each input sequence, the sample contains the input sequence, and the corresponding target contains the input sequence shifted one character to the right. Targets are created this way because we are trying to predict the next character in a sequence. So we break the text into chunks of *seq_length + 1*.

Begin by converting the encoded corpus into TensorFlow tensors:

```
# initialize maximum length sequence for a single input
seq_length = 100

# create training dataset
ds = tf.data.Dataset.from_tensor_slices(encoded)
ds
```

Display Samples

Display some samples from the training dataset:

```
for i in ds.take(6):
  print (i.numpy(), ':', char_map[i])
```

We convert the integer representations of each character back to their character state with the *char_map* array created earlier.

Batch Sequences

Create batched sequences:

```
sequences = ds.batch(seq_length + 1, drop_remainder=True)
```

The batch method lets us easily convert individual characters to sequences of the desired size. Use the parameter *drop_remainder=True* to ensure that all batches are the same size.

Display the first batch:

```
for i in sequences.take(1):
  print (char_map[i], br)
  print ('batch size:', len(i))
```

Batch size is *101* to account for the target set.

Note Each target is represented by shifting the input sequence one character to the right.

Create Samples and Targets

Build a function that creates sample and target sets:

```
def create_sample_target(piece):
  sample = piece[:-1]
  target = piece[1:]
  return sample, target
```

The function shifts an input sequence by *1* to form the sample and target texts for each batch. So each sample contains the first 100 characters, and each target contains the second character up to the 101st character.

Create the dataset:

```
dataset = sequences.map(create_sample_target)
```

Map the function onto *sequences* to create a dataset composed of the desired sample and target sets.

Display the sample and target from the first input sequence:

```
for sample, target in  dataset.take(1):
  print ('sample:', char_map[sample], br)
  print ('target:', char_map[target])
```

As desired, the target is one character ahead of the sample so the algorithm can learn from the target how to predict the next character.

Time Step Prediction

Let's look at the first five time steps from the sample and target sets as shown in Listing 8-2.

Listing 8-2. Time step prediction

```
for i, (input_idx, target_idx) in enumerate(
    zip(sample[:5], target[:5])):
  print('Step:', i)
  print(' input:', input_idx.numpy(),
        char_map[input_idx])
  print(' expected output:', target_idx.numpy(),
        char_map[target_idx])
  if i < 4: print()
```

Each index of the sample and target vectors is processed as a single time step. That is, each character processed in a sample and target is a time step. So, for the input at time step 0, the model receives the index (input_idx) for *A* and tries to predict the index for *a space* as the next character. At the next time step, the model repeats the same process. But the RNN model considers the context of the previous step in addition to the current input character. The output verifies that the sample and target sets were created properly.

Create Training Batches

The dataset is already split into manageable text sequences. But we need to shuffle the data and pack it into batches before feeding it to the model.

Set batch and buffer sizes:

```
BATCH_SIZE = 64
BUFFER_SIZE = 10000
```

Shuffle, batch, cache, and prefetch:

```
corpus_ds = (dataset
  .shuffle(BUFFER_SIZE)
  .batch(BATCH_SIZE, drop_remainder=True)
  .cache().prefetch(1))

corpus_ds
```

TensorFlow data is designed to work with infinite sequences. So it doesn't attempt to shuffle the entire sequence in memory. Instead, it maintains a buffer where it shuffles elements. We set *BUFFER_SIZE=10000* to give TensorFlow a fairly large buffer size, but not too big that we cause memory issues. We see that our dataset contains training samples and targets with batch sizes of 64 and sequence lengths of 100.

Build the Model

Begin by initializing some important variables. We set *vocab_size* to the number of unique characters in the corpus. We set *embedding dimension* to 256. **Word embedding** is a learning technique in NLP where words or phrases from the vocabulary are mapped to vectors of real numbers.

In practice, we use word embedding vectors with dimensions between 50 and 500. We use 256 because we believe it is a nice compromise between processing time and performance. Increasing the number of word embeddings provides better performance. But higher embedding dimensions are computationally expensive. We set *rnn_units* to 1024, which represents the number of neurons output from a layer.

Initialize variables:

```
# length of the vocabulary in chars
vocab_size = len(vocab)

# the embedding dimension
embedding_dim = 256

# number of RNN units
rnn_units = 1024
```

Create the model as shown in Listing 8-3.

Listing 8-3. Create the model

```
# generate seed for reproducibility
tf.random.set_seed(0)
np.random.seed(0)

# clear any previous models
tf.keras.backend.clear_session()

# import libraries
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import GRU, Dense,\
Embedding
from tensorflow.keras import losses

# create the model
model = Sequential([
  Embedding(vocab_size, embedding_dim,
            batch_input_shape=[BATCH_SIZE, None]),
  GRU(rnn_units, return_sequences=True,
      stateful=True, recurrent_initializer='glorot_uniform'),
  Dense(vocab_size)
])
```

The first layer is an embedding layer with vocabulary size, embedding dimensions, and input shape of the batch as inputs. The output from the embedding layer feeds into the second layer, which is a GRU layer with 1024 neurons (identified by the *rnn_units* parameter). To retain what was learned at this layer, we set *return_sequences=True* and *stateful=True*. We also tell the GRU layer to draw samples from a uniform distribution by setting *recurrent_initializer='glorot_uniform'*. The output from the GRU layer feeds into the final dense layer with vocabulary size as input.

Gated recurrent units (GRUs) are a gating mechanism in RNNs that are similar to a long short-term memory (LSTM) with a forget gate, but have fewer parameters than a LSTM and lack an output gate.

For a deep discussion of GRUs, consult the following URL: *https://arxiv.org/ftp/arxiv/papers/1701/1701.05923.pdf*.

Display Model Summary

Inspect the model:

```
model.summary()
```

The first layer is an embedding. So calculate the number of learnable parameters by multiplying vocabulary size of 74 by embedding dimension of 256 for a total of 18,944.

The second layer is a GRU. The number of learnable parameters is thereby based on the formula $3 \times (n^2 \times mn + 2n)$ where m is the input dimension and n is the output dimension. Multiply by *3* because there are three sets of operations for a GRU that require weight matrices of these sizes. Multiply n by *2* because of the feedback loops of a RNN. So we get 3,938,304 learnable parameters. Here's how we break down the result:

* $3 \times (1024^2 + 1024 \times 256 + 2 \times 1024)$

* $3 \times (1048576 + 262144 + 2048)$

* 3×1312768

* 3,938,304

As we can see, calculating learnable parameters for the second layer is pretty complex. So let's break it down logically. A GRU layer is a feedforward layer with feedback loops. Learnable parameters for a feedforward network are calculated by multiplying output from the previous layer (256 neurons) with neurons at the current layer (1024 neurons). With a feedforward network, we also have to account for the 1024 neurons at this layer. But we multiply the 1024 neurons at this layer by 2 because of the feedback mechanism of a RNN. Finally, the current layer's 1024 neurons are fed back resulting in 1024^2 learnable parameters. A GRU uses three sets of operations (hidden state, reset gate, and update gate) requiring weight matrices, so we multiply the learnable parameters by 3.

The third layer is dense. So calculate the number of learnable parameters by multiplying output dimension of 74 by input dimension of 1,024 and adding 74 to account for the number of neurons at this layer for a total of 75,850.

Check Output Shape

Display the shape of the first batch in the dataset:

```
for sample, target in corpus_ds.take(1):
  example_batch_predictions = model(sample)

example_batch_predictions.shape
```

The first batch has *batch_size* of *64*, *sequence_length* of *100*, and *vocab_size* of *74* as expected. Notice that the model summary output shape from the dense layer is (64, None, 74). The sequence length is not included because the model can be run on inputs of any length.

Calculate Loss

We sample from the output distribution to predict character indices. The output distribution is defined by the logits over our character vocabulary. A **logit** is a probability value between 0 and 1 and negative infinity and infinity derived from a logit function. Simply, a logit is a prediction. The logit function is an inverse to the sigmoid function as it limits values between 0 and 1 across the Y-axis rather than the X-axis. Since our model returns logits, we need to set the *from_logits* flag to calculate loss.

Build a function to calculate loss:

```
def loss(labels, logits):
  return losses.sparse_categorical_crossentropy(
      labels, logits, from_logits=True)
```

Invoke the function:

```
pre_trained_loss = loss(target, example_batch_predictions)
```

The model expects a 3D tensor consisting of batch size, sequence length, and vocabulary size.

Let's see if the shape is correct:

```
print('pred shape: ', example_batch_predictions.shape)
print('scalar_loss: ', pre_trained_loss.numpy().mean())
```

Great! The prediction shape is as expected with batch size of 64, sequence length of 100, and vocabulary size of 74. We also display the average loss from the pretrained model.

Compile the Model

Compile the model:

```
model.compile(loss=loss,
              optimizer='adam')
```

Configure Checkpoints

With a RNN, we want to save what the model learns at each time step. One way to do this is to save the checkpoints that hold this information with a callback method. **Checkpoints** capture the exact value of all TensorFlow parameters (or tf.Variable.objects) used by a model.

Listing 8-4 saves checkpoints so we can recall what the RNN has learned.

Listing 8-4. Configure and save checkpoints

```
import os

# directory where the checkpoints will be saved
checkpoint_dir = './training_checkpoints'

# name of the checkpoint files
checkpoint_files = os.path.join(checkpoint_dir, 'ckpt_{epoch}')

# callback method
checkpoint_callback=tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_files, save_weights_only=True)
```

Train the Model

Train the model for ten epochs:

```
EPOCHS = 10

history = model.fit(corpus_ds, epochs=EPOCHS,
                    callbacks=[checkpoint_callback])
```

We can add more epochs to improve performance. We tell the model to save checkpoints with *checkpoint_callback*.

Rebuild the Model for Text Creation

Now that we have a trained RNN, we rebuild the model to predict one character at a time. We rebuild in three steps:

1. Restore weights from checkpoints.
2. Rebuild with batch size of 1.
3. Load weights and reshape the model to ensure tensors with batch size of 1.

Restore Weights from Checkpoints

Restore weights from checkpoints established during training. We restore checkpoints to obtain what the RNN learned at each time step.

Restore:

```
tf.train.latest_checkpoint(checkpoint_dir)
```

Rebuild with Batch Size of 1

Since we constructed our sequences to look ahead one character, we predict one character at a time. Because of the way the RNN state is passed from time step to time step, the model only accepts a fixed batch size once it is built.

Rebuild the model with batch size of 1 (instead of 64) as shown in Listing 8-5.

Listing 8-5. Rebuild the model

```
# generate seed for reproducibility
tf.random.set_seed(0)
np.random.seed(0)

# clear any previous models
tf.keras.backend.clear_session()

# set batch size to 1
BATCH_SIZE = 1

# Rebuild model
model = Sequential([
  Embedding(vocab_size, embedding_dim,
            batch_input_shape=[BATCH_SIZE, None]),
  GRU(rnn_units, return_sequences=True, stateful=True,
      recurrent_initializer='glorot_uniform'),
  Dense(vocab_size)
])
```

Load Weights and Reshape

Load the weights saved from the trained RNN and reshape the model to ensure that tensors have a batch size of 1:

```
model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))
model.build(tf.TensorShape([1, None]))
```

Model Summary

Inspect the model:

```
model.summary()
```

Notice that the first parameter for output shapes is 1 indicating that batch size was changed as expected.

Create Components to Generate Text

To create new text, create a function and initialize a set of variables to feed to the function. To prepare the starting string for TensorFlow consumption, vectorize and reshape it before passing it to the function.

Create the Function

Create the text generation function as shown in Listing 8-6.

Listing 8-6. Create the text function

```
def create_text(model, input_eval, temperature, start_string):

  # Empty string to store our results
  new_text = []

  # Here batch size == 1
  model.reset_states()

  for i in range(n):
    # model encoded input
    predictions = model(input_eval)

    # remove batch dimension so we can manipulate predictions
    predictions = tf.squeeze(predictions, 0)

    # divide predictions by temperature
    predictions = predictions / temperature

    # use a categorical distribution to predict character
    # returned by model
    predicted_id = tf.random.categorical(
        predictions, num_samples=1)[-1,0].numpy()

    # pass predicted character as next input to model
    # with previous hidden state
    input_eval = tf.expand_dims([predicted_id], 0)
```

```
        # append generated characters to text
        new_text.append(char_map[predicted_id])

    return (start_string + ''.join(new_text))
```

The function accepts the model, vectorized starting string, temperature, and original starting string. It begins by initializing a list to hold the new text created and resetting the states of the model. The function continues by iterating *n* times (the number of characters we wish to create).

Temperature is a hyperparameter of neural networks used to control the randomness of predictions by scaling the logits before applying softmax activation. A bit more simply, temperature represents how much to divide the logits by before computing softmax activation.

During iteration, the function models the encoded starting string and places the result in *predictions*. It then removes the extra *1* dimension so it can divide the contents of *predictions* by the *temperature*. The next task of the function is to use a categorical distribution to predict the next character returned by the model. The function needs to add the *1* dimension back so that it can pass the predicted character as the next input to the model along with the previous hidden state. Newly generated characters are appended to the *new_text* array. The process repeats until the loop is extinguished.

Initialize Variables

Initialize the variables:

```
n = 500
temp = 0.3
start_string = 'Tale'
```

Set *n* to the number of characters we wish to create. Continue by setting the temperature and the starting string. Low temperatures result in more predictable text, while higher temperatures result in more surprising text. You can experiment to find the best setting. You can also experiment with the starting string. We chose *Tale* because we know that the corpus contains the name.

Vectorize and Reshape the Starting String

Vectorize the starting string because the model only recognizes numbers. Reshape the vectorized starting string for TensorFlow consumption. That is, add the 1 dimension to the starting string so that the model can process it. Display shapes to verify that all is well.

Vectorize the starting string and reshape it for TensorFlow consumption as shown in Listing 8-7.

Listing 8-7. Vectorize and reshape the starting string

```
# vectorize starting string
input_vectorized = [int_map[s] for s in start_string]
print ('original shape:', end=' ')
print (str(np.array(input_vectorized).ndim) + 'D', br)

# reshape string for TensorFlow model consumption
input_vectorized = tf.expand_dims(input_vectorized, 0)
print ('new shape:', input_vectorized.shape)
```

The new shape is (1, 4). The *1* dimension indicates batch size of 1. The *4* dimension indicates the length of start_string. Experiment with the starting string to see generation of different texts.

Create New Text

Plant random seeds and invoke the function:

```
tf.random.set_seed(0)
np.random.seed(0)

print (create_text(model, input_vectorized, temp, start_string))
```

Wow! Although the sentences are nonsensical, the model creates actual sentences. If you think about it, what we just did was amazing. The model ingested a corpus and was able to learn from it.

CHAPTER 9

Sentiment Analysis

We've already demonstrated how to train a character-level RNN to create original text. Now, we create a *word-level* RNN to analyze sentiment.

Sentiment analysis is the interpretation and classification of polarity, emotions, and intentions within text data using NLP text analysis tools. Polarity can be positive, negative, or neutral. Emotions can vary across a wide range of feelings such as anger, happiness, frustration, and sadness, to name a few. Intentions can also vary across a wide range of motives such as interested or not interested. A common application of sentiment analysis is to identify customer sentiment toward products, brands, or services through online feedback. General application includes social media monitoring, brand monitoring, customer service, customer feedback, and market research.

For an excellent discussion of sentiment analysis, consult the following URL:

https://monkeylearn.com/sentiment-analysis/#:~:text=Sentiment%20analysis%20is%20the%20interpretation,or%20services%20in%20online%20feedback

Sentiment analysis is a very common NLP task. Technically, it computationally identifies and categorizes opinions expressed in a text corpus to determine attitude or sentiment. Typically, sentiment analysis is used to determine a positive, negative, or neutral opinion toward a particular topic or product.

Notebooks for chapters are located at the following URL: *https://github.com/paperd/tensorflow*.

IMDb Dataset

A popular dataset used to practice NLP is the IMDb reviews dataset. **IMDb** is a benchmark dataset for binary sentiment classification. The dataset contains 50,000 movie reviews labeled as either positive (1) or negative (0). Reviews are preprocessed with each encoded as a sequence of word indexes in the form of integers. Words within the reviews are indexed by their overall frequency within the dataset. The 50,000 reviews

D. Paper, *TensorFlow 2.x in the Colaboratory Cloud*, https://doi.org/10.1007/978-1-4842-6649-6_9

are split into 25,000 for training and 25,000 for testing. So we can predict the number of positive and negative reviews using either classification or other deep learning algorithms.

IMDb is popular because it is simple to use, relatively easy to process, and challenging enough for machine learning aficionados. We enjoy working with IMDb because it's just plain fun to work with movie data.

Enable the GPU (if not already enabled):

1. Click *Runtime* in the top-left menu.
2. Click *Change runtime type* from the drop-down menu.
3. Choose *GPU* from the *Hardware accelerator* drop-down menu.
4. Click *SAVE*.

Test if GPU is active:

```
import tensorflow as tf

# display tf version and test if GPU is active
tf.__version__, tf.test.gpu_device_name()
```

Import the *tensorflow* library. If '/device:GPU:0' is displayed, the GPU is active. If '.' is displayed, the regular CPU is active.

Load IMDb as a TFDS

The recommended way to load IMDb is as a TFDS:

```
import tensorflow_datasets as tfds

imdb, info = tfds.load(
    'imdb_reviews/subwords8k', with_info=True,
    as_supervised=True, shuffle_files=True)
```

We use the *imdb_reviews/subwords8k* TFDS so we train the model on a smaller vocabulary. The *subwords8k* subset has a vocabulary size of 8,000, which means that we are training the model on the 8,000 most commonly used words in the reviews. It also means that we don't have to build our own vocabulary dictionary! We can get good

performance with this subset and substantially reduce training time. Loading the TFDS also gives us access to the *tfds.features.text.SubwordTextEncoder*, which is the TFDS text encoder.

We set *with_info=True* to enable access to information about the dataset and the encoder. We set *as_supervised=True* so that the returned TFDS has a two-tuple structure (input, label) in accordance with *builder.info.supervised_keys*. If set to *False* (the default), the returned TFDS will have a dictionary with all features included. We set *shuffle_files=True* because shuffling typically improves performance.

Display the Keys

By loading the dataset as a TFDS, we have access to its keys:

```
imdb.keys()
```

We see that the dataset is split into test, train, and unsupervised samples.

Split into Train and Test Sets

Since we are building a supervised model, we only have interest in the train and test samples:

```
train, test = imdb['train'], imdb['test']
```

Display the First Sample

It's always a good idea to explore a sample from the dataset:

```
br ='\n'

for sample, target in train.take(1):
  print ('encoded review:')
  print (sample, br)
  print ('target:', target.numpy())
```

The first training example contains an encoded review and a label. The review is already encoded as a tensor of integers with datatype *int64*. The label is a scalar value of either 0 (negative) or 1 (positive) with datatype *int64*.

The shape of the review tensor indicates the number of words it contains. For readability, we convert the target tensor to values with the *numpy* method.

Display Information About the TFDS

The *info* object gives us access to metadata:

```
info
```

Peruse Metadata

See the number of examples in train and test splits:

```
train_size = info.splits['train'].num_examples
test_size = info.splits['test'].num_examples
```

```
train_size, test_size
```

See supervised keys:

```
info.supervised_keys
```

See feature information:

```
info.features
```

See the TFDS name and a slice of its description:

```
info.name, info.description[0:25]
```

We can even slice the citation string to get the title:

```
info.citation[184:242]
```

Create the Encoder

An encoder is built into the TFDS *SubwordTextEncoder*. With the encoder, we can easily decode (integer to text) and encode (text to integer). We access the encoder from the dataset's *info* object.

Create an encoder based on the IMDb dataset we loaded into memory:

```
encoder = info.features['text'].encoder
```

Now that the encoder is built, we can use it to vectorize strings and decode vectorized strings back into text strings.

Test the encoder:

```
sample_string = 'What a Beautiful Day!'

encoded_string = encoder.encode(sample_string)
print ('Encoded string:', encoded_string)

original_string = encoder.decode(encoded_string)
print ('Original string:', original_string)
```

Use the Encoder on Samples

Create a function that returns the label rating in readable form:

```
def rev(d):
  if tf.math.equal(d, 0): return 'negative review'
  elif tf.math.equal(d, 1): return 'positive review'
```

Display the first review as shown in Listing 9-1.

Listing 9-1. Display the first review

```
for sample, target in train.take(1):
  print ('review:', end=' ')
  text = encoder.decode(sample)
  print (text[0:100])
  print ('opinion:', end=' ')
  print ('\'' + rev(target) + '\'')
```

Display multiple reviews as shown in Listing 9-2.

Listing 9-2. Display multiple reviews

```
n = 6

for i, sample in enumerate(train.take(n)):
  if i > 0:
    print ('review', str(i+1) +':', end=' ')
    text = encoder.decode(sample[0])
    print (text[0:100])
    print ('opinion:', end=' ')
    print ('\'' + rev(sample[1]) + '\'')
    if i < n-1:
      print ()
```

We skip the first review because we've already seen it.

Display vocabulary size:

```
print('Vocabulary size: {}'.format(encoder.vocab_size))
```

Finish the Input Pipeline

Create batches of the encoded strings (or reviews) to greatly enhance performance. Since machine learning algorithms expect batches of the same size, use the *padded_batch* method to zero-pad the sequences so that each review is the same length as the longest string in the batch.

Initialize variables:

```
BUFFER_SIZE = 10000
BATCH_SIZE = 64
```

Shuffle (where appropriate), batch, cache, and prefetch train and test sets as shown in Listing 9-3.

Listing 9-3. Finish the input pipeline

```
train_ds = (train
            .shuffle(BUFFER_SIZE)
            .padded_batch(BATCH_SIZE)
            .cache().prefetch(1))
```

```
test_ds = (test
           .padded_batch(BATCH_SIZE)
           .cache().prefetch(1))
```

Inspect tensors:

```
train_ds, test_ds
```

Consult the following URL for updates on padding character tensors: *www.tensorflow.org/tutorials/text/text_classification_rnn*

Create the Model

Plant seeds, import libraries, clear previous models, and create the model as shown in Listing 9-4.

Listing 9-4. Create the model

```
import numpy as np

# generate seed for reproducibility
tf.random.set_seed(0)
np.random.seed(0)

# import libraries
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import GRU, Dense,\
Embedding

# clear any previous models
tf.keras.backend.clear_session()

# build the model
embed_size = 128
model = Sequential([
  Embedding(encoder.vocab_size, embed_size, mask_zero=True,
            input_shape=[None]),
  GRU(128, return_sequences=True),
  GRU(128),
  Dense(1, activation="sigmoid")
])
```

The first layer is an embedding layer. The *embedding* layer is used to create word vectors for incoming words. During training, representations of word categories (or word vectors) are learned in a way where similar categories are closer to one another. So word vectors can store relationships between words like *good* and *great*. Word vectors are dense because our model learns word relationships. As a result, word vectors aren't padded with a huge number of zeros like what we do with one-hot encodings.

The embedding layer accepts vocabulary size, embedding size, and input shape. We set *mask_zero=True* to inform the model to ignore padding tokens by all downstream layers. Ignoring padding tokens improves performance.

The next two layers are GRU layers, and the final layer is a single-neuron dense output layer. The output layer uses sigmoid activation to output the estimated probability that the review expresses a positive or negative sentiment regarding the movie.

Model Summary

Inspect the model:

```
model.summary()
```

The first layer is an embedding. So calculate the number of learnable parameters by multiplying vocabulary size of 8185 by embedding dimension (embed_size) of 128 for a total of 1,047,680.

The second layer is a GRU. The number of learnable parameters is thereby based on the formula $3 \times (n^2 \times mn + 2n)$ where m is the input dimension and n is the output dimension. Multiply by *3* because there are three sets of operations for a GRU that requires weight matrices of these sizes. Multiply n by *2* because of the feedback loops of a RNN. So we get 99,072 learnable parameters.

Here's how we break down the result:

* $3 \times (128^2 + 128 \times 128 + 2 \times 128)$

* $3 \times (16384 + 16384 + 256)$

* 3×33024

* 99,072

As we can see, calculating learnable parameters for the second layer is pretty complex. So let's break it down logically. A GRU layer is a feedforward layer with feedback loops. Learnable parameters for a feedforward network are calculated by multiplying output from the previous layer (128 neurons) with neurons at the current layer (128 neurons). With a feedforward network, we also have to account for the 128 neurons at this layer. But we multiply the 128 neurons at this layer by 2 because of the feedback mechanism of a RNN. Finally, the current layer's 128 neurons are fed back resulting in 128^2 learnable parameters. A GRU uses three sets of operations (hidden state, reset gate, and update gate) requiring weight matrices, so we multiply the learnable parameters by 3.

The third layer is a GRU. We get 99,072 learnable parameters because *n* and *m* are exactly the same as the second layer. So the calculations are the same.

The final layer is dense. So calculate the number of learnable parameters by multiplying output dimension of 1 by input dimension of 128 and adding 1 to account for the number of neurons at this layer for a total of 129.

Compile the Model

Compile:

```
model.compile(loss='binary_crossentropy', optimizer='adam',
              metrics=['accuracy'])
```

Train the Model

We found that two epochs provide pretty good accuracy without tuning. However, your results may differ. So you can experiment to your heart's content. But keep in mind that training text models requires a lot of training time!

```
# to suppress unimportant error messages
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)

history = model.fit(train_ds, epochs=2, validation_data=test_ds)
```

Generalize on Test Data

Although model fit information provides validation loss and accuracy values during training, it is always a good idea to explicitly evaluate a model on test data because accuracy and loss values may differ:

```
test_loss, test_acc = model.evaluate(test_ds)
```

Visualize Training Performance

Visualize as shown in Listing 9-5.

Listing 9-5. Visualize training performance

```
import matplotlib.pyplot as plt

# history.history contains the training record
history_dict = history.history

acc = history_dict['accuracy']
val_acc = history_dict['val_accuracy']
loss = history_dict['loss']
val_loss = history_dict['val_loss']

epochs = range(1, len(acc) + 1)

plt.figure(figsize=(12,9))
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.ylim((0.5,1))
plt.show()

# clear previous figure
plt.clf()
```

```
plt.figure(figsize=(12,9))
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Make Predictions from Fabricated Reviews

Let's make predictions from reviews that we fabricate. Begin by creating a function that returns the predictions. Since we create our own reviews, the function must convert the text review for TensorFlow consumption.

Create the function:

```
def predict_review(text):
  encoded_text = encoder.encode(text)
  encoded_text = tf.cast(encoded_text, tf.float32)
  prediction = model.predict(tf.expand_dims(encoded_text, 0))
  return prediction
```

The function accepts a text review. It begins by encoding the review. It then converts the encoded review to float32. The function ends by making the prediction and returning it to the calling environment. We add the *1* dimension to the encoded text so it can be consumed by the TensorFlow model.

Test the function with a fabricated review:

```
review = ('Just loved it. My kids thought the movie was cool. '
         'Even my wife liked it.')

pred = predict_review(review)
pred, pred.shape
```

We have a prediction. Predictions greater than 0.5 mean that the review is positive. Otherwise, the review is negative.

Let's make the prediction more palatable by creating another function:

```
def palatable(pred):
  score = tf.squeeze(pred, 0).numpy()
  return score[0]
```

The function converts the prediction to a numpy scalar.

Invoke the function:

```
score = palatable(pred)
score, score.shape
```

The function removes the *1* dimension from the prediction.

Let's go a step further by creating a function that returns the review in plain English:

```
def impression(score):
  if score >= 0.5:
    return 'positive impression'
  else:
    return 'negative impression'
```

Invoke the function:

```
impression(score)
```

As expected, the review is positive.

Let's try another one:

```
review = ('The movie absolutely sucked. '
          'No character development. '
          'Dialogue just blows.')

pred = predict_review(review)
score = palatable(pred)
print (impression(score))
```

As expected, the review is negative.

Make Predictions on a Test Data Batch

We can also predict from the test set. Let's make predictions on the *first* test batch with the *predict* method. Since test data is already in tensor form, we don't need to encode.

Predict from the test set batch and display the first review as shown in Listing 9-6.

Listing 9-6. Make predictions based on a batch from the test set

```
# get predictions from 1st test batch
for sample, target in test_ds.take(1):
  y_pred_64 = model.predict(sample)

# display first review from this batch
print ('review:', end=' ')
print (encoder.decode(sample[0])[177:307])
# display first label from this batch
print ('label:', end=' ')
print (target[0].numpy(), br)

# display number of examples in the batch
print ('samples and target in first batch:', end=' ')
len(sample), len(target)
```

We take the first batch from *test_ds*. We make predictions with the *predict* method and place them in y_pred_64. Variable *y_pred_64* holds 64 predictions because batch size is 64. We then display the first review and its associated label from this batch. Remember that label *1* means the review is positive and label *0* means it is negative. We end by displaying the size of the sample and target to verify that we have 64 examples in our first batch.

Get the first prediction:

```
print (y_pred_64[0])
```

Make it palatable:

```
impression(y_pred_64[0])
```

Compare the prediction to the actual label:

```
impression(y_pred_64[0]), impression(target[0].numpy())
```

If the prediction matches the actual label, it was correct.

Listing 9-7 displays prediction efficacy for the first five predictions.

Listing 9-7. Prediction efficacy for five predictions

```
for i in range(5):
  p = impression(y_pred_64[i])
  t = impression(target[i].numpy())
  print (i, end=': ')
  if p == t: print ('correct')
  else: print ('incorrect')
```

Prediction Accuracy for the First Batch

Create a function to convert an impression back to a label of either 1 or 0:

```
def convert_label(feeling):
  if feeling == 'positive impression':
    return 1
  else: return 0
```

Return prediction accuracy for the entire first batch as shown in Listing 9-8.

Listing 9-8. Prediction accuracy for the first batch

```
ls = []
n = len(target)
for i, _ in enumerate(range(n)):
  t = target[i].numpy() # labels
  p = convert_label(impression(y_pred_64[i])) # predictions
  if t == p: ls.append(True)
correct = ls.count(True)
acc = correct / n
batch_accuracy = str(int(np.round(acc, 2) * 100)) + '%'
print ('accuracy for the first batch:', batch_accuracy)
```

We begin by traversing the first batch and comparing labels to predictions. If a prediction is correct, we add this information to a list. We continue by counting the number of correct predictions. We end by dividing correct predictions by the batch size to get overall prediction accuracy.

Leverage Pretrained Embeddings

Amazingly, we can reuse modules from pretrained models on the IMDb dataset. The *TensorFlow Hub project* is a library with hundreds of reusable machine learning modules. A **module** is a self-contained piece of a TensorFlow graph, along with its weights and assets, that can be reused across different tasks in a process known as transfer learning. **Transfer learning** is a machine learning method where a model developed for a task is reused as the starting point for a model on a different task.

You can browse the library by perusing the following URL:

http://tfhub.dev

Once you locate a module, copy the URL into your model. The module is automatically downloaded along with its pretrained weights. A huge advantage of using pretrained models is that we don't have to create and train our own models from scratch!

Load the IMDb Dataset

Since we are using a pretrained model, we can load the IMDb dataset with the full vocabulary:

```
data, info = tfds.load('imdb_reviews', as_supervised=True,
                       with_info=True, shuffle_files=True)
```

We use the full vocabulary because we don't have to worry about training with it!

Display metadata:

```
info
```

Build the Input Pipeline

Create train and test sets:

```
train, test = data['train'], data['test']
```

Batch and prefetch:

```
batch_size = 32
train_set = train.repeat().batch(batch_size).prefetch(1)
test_set = test.batch(batch_size).prefetch(1)
```

Inspect tensors:

```
train_set, test_set
```

Create the Pretrained Model

Import the TF Hub library and create a skeleton model to house the pretrained module as shown in Listing 9-9.

Listing 9-9. Create the model

```
import tensorflow_hub as hub

# clear any previous models
tf.keras.backend.clear_session()

model = tf.keras.Sequential([
  hub.KerasLayer(
      'https://tfhub.dev/google/tf2-preview/nnlm-en-dim50/1',
      dtype=tf.string, input_shape=[], output_shape=[50]),
  Dense(128, activation='relu'),
  Dense(1, activation='sigmoid')
])
```

The *hub.KerasLayer* downloads the sentence encoder module. Each string input into this layer is automatically encoded as a 50D vector. So each vector represents 50 words. Each word is embedded based on an embedding matrix pretrained on the 7 billion–word Google News corpus. The next two dense layers are added to provide a basic model for sentiment analysis. Using TF Hub is convenient and efficient because we can use what was already learned from the pretrained model.

Compile the Model

Compile the model:

```
model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
```

Train the Model

Train the model:

```
# to suppress unimportant error messages
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)

history = model.fit(train_set,
                    steps_per_epoch=train_size // batch_size,
                    epochs=5, validation_data=test_set)
```

Training time is substantially reduced!

Make Predictions

Make predictions based on the first *test_set* batch:

```
for sample, target in test_set.take(1):
  y_pred_32 = model.predict(sample)
```

Since batch size is 32, we have 32 predictions for each batch.

Display misclassifications in test_set by index as shown in Listing 9-10.

Listing 9-10. Misclassifications in the first batch by index

```
for i in range(batch_size):
  p = convert_label(impression(y_pred_32[i]))
  l = target[i].numpy()
  if p != l:
    print ('pred:', p, 'actual:', l, 'indx:', i)
```

Calculate Prediction Accuracy for the First Batch

Listing 9-11 shows the code to calculate prediction accuracy for the first batch.

Listing 9-11. Calculate prediction accuracy for the first batch

```
ls = []
n = len(target)
for i, _ in enumerate(range(n)):
  t = target[i].numpy() # labels
  p = convert_label(impression(y_pred_32[i])) # predictions
  if t == p: ls.append(True)
correct = ls.count(True)
acc = correct / n
batch_accuracy = str(int(np.round(acc, 2) * 100)) + '%'
print ('accuracy for the first batch:', batch_accuracy)
```

Instead of finding misclassifications, the code finds correct predictions. The code begins by comparing an actual label to a predicted one. If the actual label is predicted correctly, a Boolean *True* is appended to a list. Once the first batch is traversed, the number of elements in the list is counted. This count is divided by batch size of 32 to determine accuracy, which is displayed as a percentage.

Explore IMDb with Keras

Since Keras is very popular in industry, we demonstrate how to train IMDb with keras.datasets. We use the *keras.datasets.imdb.load_data* function to load the dataset in a format-ready fashion for use in neural network and deep learning models.

Loading the Keras IMDb has some advantages. First, words have already been encoded with integers. Second, encoded words are arranged by their absolute popularity in the dataset. So sentences in each review are comprised of a sequence of integers. Third, calling *imdb.load_data* the first time downloads IMDb to your computer and stores it in your home directory under *~/.keras/datasets/imdb.pkl* as a 32-megabyte file. The imdb.load_data function also provides additional arguments including number of top words to load (where words with a lower integer are marked as zero in the returned data), number of top words to skip (to avoid words like *the*), and the maximum length of reviews to support.

Load the Keras IMDb:

```
train, test = tf.keras.datasets.imdb.load_data()
```

The function loads data into train and test tuples. So *train[0]* contains training reviews and *train[1]* contains training labels. And *test[0]* contains test reviews and *test[1]* contains test labels. Each review is represented as a numpy array of integers with each integer representing a word. The labels contain lists of integer labels (0 is a negative review and 1 is positive).

For readability, create variables to represent reviews and labels:

```
train_reviews, train_labels = train[0], train[1]
test_reviews, test_labels = test[0], test[1]
```

Display the shape of train and test review samples:

```
train_reviews.shape, test_reviews.shape
```

As expected, we have 25,000 train and 25,000 test reviews.

Display the shape of train and test labels:

```
train_labels.shape, test_labels.shape
```

As expected, we have 25,000 train and 25,000 test labels.

Explore the Train Sample

Display label categories and number of unique words:

```
print ('categories:', np.unique(train_labels))
print ('number of unique words:',
       len(np.unique(np.hstack(train_reviews))))
```

The dataset is labeled by two categories that represent sentiment of each review. And the train sample contains 88,585 unique words.

Let's see how many words are in the longest training review:

```
longest = np.amax([len(i) for i in train_reviews])

print ('longest review:', longest)
```

We create a list containing the number of words in each review and then find the length of the review with the maximum number of words.

Get the index of the longest review:

```
mid_result = np.where([len(i) for i in train_reviews] == longest)
longest_index = mid_result[0][0]
longest_index
```

We use the *np.where* function to find the index. We used double indexing because the function returns a tuple containing a list that holds the index we desire.

Create a Decoding Function

Create a function that decodes a review into English readable form as shown in Listing 9-12.

Listing 9-12. Function that decodes a review

```
def readable(review):
  index = tf.keras.datasets.imdb.get_word_index()
  reverse_index = dict([(value, key)\
                        for (key, value) in index.items()])
  return ' '.join( [reverse_index.get(i - 3, '?')\
                    for i in review])
```

The function uses the *tf.keras.datasets.imdb.get_word_index* utility to obtain a dictionary of words and their uniquely assigned integers. The function then creates another dictionary containing key-value groupings as value and key groupings from the first dictionary. Finally, it returns the words based on their IDs (or keys). The indices are offset by 3 because 0, 1, and 2 are reserved indices for *padding, start of sequence,* and *unknown*.

Invoke the Decoding Function

Let's see what the longest review looks like as shown in Listing 9-13.

Listing 9-13. Decode the longest review

```
review = readable(train_reviews[longest_index])
print ()
print ('review:', end=' ')

# just display a slice of the full review
print (review[:50] + ' ...', br)

label = train_labels[longest_index]
idea = impression(label)
print (idea, br)

# verify length of review
print (len(train_reviews[longest_index]))
```

Since we already know the index of the longest review, we can easily retrieve it from *train_reviews*. Display a slice of it since the review is pretty long. We can also easily retrieve the label from *train_labels*. Make the label readable and display it. Finally, display the length of the longest review.

Let's see what the shortest review looks like. But we can't do this directly.

We must first find the minimum number of words:

```
shortest = np.amin([len(i) for i in train_reviews])
print ('shortest review:', shortest)
```

Since we don't know which review is the shortest, we use the *amin* method to return the minimum.

We can now get the index of the shortest review in the train sample:

```
result = np.where([len(i) for i in train_reviews] == shortest)
shortest_index = result[0][0]
shortest_index
```

We use the *where* method to return the index we seek. Since the method returns all reviews that meet the criterion, we grab the first one and display its index.

Listing 9-14 displays the review, its label in readable form, and its length.

Listing 9-14. Display the shortest review

```
review = readable(train_reviews[shortest_index])
print (review[2:], br)

label = train_labels[shortest_index]
idea = impression(label)
print (idea, br)

# verify length of review
print (len(train_reviews[shortest_index]))
```

Continue Exploring the Training Sample

Return the average review length:

```
length = [len(i) for i in train_reviews]

print ('average review length:', np.mean(length))
```

Display the first label and its review encoded as integers as shown in Listing 9-15.

Listing 9-15. First label and its review

```
first_label = train_labels[0]

print('label:', first_label, end=' ')

idea = impression(first_label)
print ('(' + idea + ')', br)

# display slice of first review
print (train_reviews[0][:20])

# display readable slice of first review
print (readable(train_reviews[0][:20]))
```

Display the first review in readable form:

```
review = readable(train_reviews[0])
print (review[2:105] + ' ...')
```

Train Keras IMDb Data

Limit vocabulary size to improve performance:

```
# limit vocabulary to 8000 most commonly used words in reviews
vocab_size = 8000
```

Cut text size to 80 words to improve performance:

```
maxlen = 80
```

Load Data

Load data with the limited vocabulary:

```
(x_train, y_train), (x_test, y_test) =\
tf.keras.datasets.imdb.load_data(num_words=vocab_size)
```

Display information about train and test data:

```
print ('train and test features:')
print (len(x_train), 'train sequences')
print (len(x_test), 'test sequences', br)
print ('sequence shape before padding:')
print ('x_train shape:', x_train.shape)
print ('x_test shape:', x_test.shape)
```

Pad Samples

Convert train and test sets to numpy:

```
x_train = np.asarray(x_train)
x_test = np.asarray(x_test)
```

Import the appropriate library:

```
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

Pad samples to ensure that all sequences are of the same length:

```
print('padded sequences (samples, maxlen):')
x_train = pad_sequences(x_train, maxlen=maxlen)
x_test = pad_sequences(x_test, maxlen=maxlen)
print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)
```

Build the Input Pipeline

Initialize pipeline variables:

```
buffer_size = 10000
batch_size = 512
```

Prepare train data for TensorFlow consumption:

```
train_k = tf.data.Dataset.from_tensor_slices(
    (x_train, y_train))
train_ks = train_k.shuffle(
    buffer_size).batch(batch_size).prefetch(1)
```

Prepare test data for TensorFlow consumption:

```
test_k = tf.data.Dataset.from_tensor_slices(
    (x_test, y_test))
test_ks = test_k.batch(batch_size).prefetch(1)
```

Build the Model

Clear any previous models:

```
tf.keras.backend.clear_session()
```

Create the model:

```
embed_size = 128
model = Sequential([
  Embedding(vocab_size, embed_size, mask_zero=True,
            input_shape=[None]),
  GRU(128, return_sequences=True),
  GRU(128),
  Dense(1, activation="sigmoid")
])
```

Compile the Model

Compile:

```
model.compile(loss='binary_crossentropy', optimizer='adam',
              metrics=['accuracy'])
```

Train the Model

Suppress error messages:

```
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
```

Train:

```
epochs = 2

model.fit(train_ks, batch_size=BATCH_SIZE,
          epochs=epochs, validation_data=(test_ks))
```

Predict

Get predictions:

```
k_pred = model.predict(test_ks)
```

Display the first prediction:

```
impression(k_pred[0][0])
```

Display a slice of the review:

```
pred_first = readable(x_test[0])
pred_first[26:53]
```

Display the impression:

```
impression(y_test[0])
```

CHAPTER 10

Time Series Forecasting with RNNs

We've already leveraged RNNs for NLP. In this chapter, we create experiments to forecast with time series data. We use the famous *Weather* dataset to demonstrate both a univariate and a multivariate example.

A RNN is well suited for time series forecasting because it remembers the past and its decisions are influenced by what it has learned from the past. So it makes good decisions as data changes. **Time series forecasting** is deploying a model to predict future values based on previously observed values.

Time series data is different than what we've worked with so far because it is a sequence of observations taken sequentially in time. Time series data includes a time dimension, which is an explicit order dependence between observations.

Weather Forecasting

Forecasting the weather is a difficult and complex endeavor. But leading-edge companies like Google, IBM, Monsanto, and Facebook are leveraging AI technology to realize accurate and timely weather forecasts. Given the introductory nature of our lessons, we cannot hope to demonstrate such complex AI experiments. But we show you how to build simple time series forecasting models with weather data.

Notebooks for chapters are located at the following URL: *https://github.com/paperd/tensorflow*.

D. Paper, *TensorFlow 2.x in the Colaboratory Cloud*, https://doi.org/10.1007/978-1-4842-6649-6_10

The Weather Dataset

We introduce time series forecasting with RNNs for a univariate problem. We then forecast a multivariate time series. We use weather time series data to train our models. The data we use is recorded by the Max Planck Institute for Biogeochemistry.

Find out more about the institute by perusing the following URL:

www.bgc-jena.mpg.de/index.php/Main/HomePage

Enable the GPU (if not already enabled):

1. Click *Runtime* in the top-left menu.
2. Click *Change runtime type* from the drop-down menu.
3. Choose *GPU* from the *Hardware accelerator* drop-down menu.
4. Click *SAVE*.

Test if GPU is active:

```
import tensorflow as tf

# display tf version and test if GPU is active
tf.__version__, tf.test.gpu_device_name()
```

Import the *tensorflow* library. If '/device:GPU:0' is displayed, the GPU is active. If '..' is displayed, the regular CPU is active.

Get the dataset as shown in Listing 10-1.

Listing 10-1. Get weather data

```
import os

p1 = 'https://storage.googleapis.com/tensorflow/'
p2 = 'tf-keras-datasets/jena_climate_2009_2016.csv.zip'
url = p1 + p2

zip_path = tf.keras.utils.get_file(
    origin = url,
    fname='jena_climate_2009_2016.csv.zip',
    extract=True)
csv_path, _ = os.path.splitext(zip_path)
```

Use the *splitext* method to extract the CSV file from the URL. Create the appropriate path for easy loading into pandas.

Load the CSV file into pandas:

```
import pandas as pd

df = pd.read_csv(csv_path)
```

Explore the Data

Display the features of the dataframe:

```
list(df)
```

Display the first few records:

```
df.head(3)
```

Display the last few records:

```
df.tail(3)
```

We see that data collection began on January 1, 2009, and ended on December 31, 2016. The last data recorded was on January 1, 2017, but it's irrelevant for our purposes since it is the only recorded piece of data for that year. We also see that data is recorded every 10 minutes. So the time step for this experiment is 10 minutes. A **time step** is a single occurrence of an event.

The first timestamp begins on January 1, 2009 (01.01.2009), with data recorded from 00:00:00 to 00:10:00. The second timestamp begins immediately after 00:10:00 to 00:20:00. This pattern continues throughout the day with the last time step at 23:50:00. The second day (and all subsequent days), January 2, 2009 (02.01.2009), follows the same pattern. Generally, time series forecasting predicts the observation at the next time step.

Display a concise summary of the dataframe:

```
df.info()
```

The dataset contains 15 columns:

- Date Time – Date-time reference
- p (mbar) – Atmospheric pressure in millibars

- T (degC) – Temperature in Celsius
- Tpot (K) – Temperature in Kelvin
- Tdew (degC) – Temperature in Celsius relative to humidity
- rh (%) – Relative humidity
- VPmax (mbar) – Saturation vapor pressure in millibars
- VPact (mbar) – Vapor pressure in millibars
- VPdef (mbar) – Vapor pressure deficit in millibars
- sh (g/kg) – Specific humidity in grams per kilogram
- H2OC (mmol/mol) – Water vapor concentration in millimoles per mole
- rho (g/m**3) – Air density in grams per meter cubed
- wv (m/s) – Wind speed in meters per second
- max. wv (m/s) – Maximum wind speed in meters per second
- wd (deg) – Wind direction in degrees

We have 14 features because *Date Time* is a reference column. There is no missing data. All data is float64 except the Date Time reference object. And the dataset contains 420,551 rows of data with indexes ranging from 0 to 420550.

Display statistical data for the 14 features:

```
stats = df.describe()
stats.transpose()
```

The *describe* method generates descriptive statistics. The *transpose* method transposes indexes and columns.

We can also display statistics for one or more features:

```
stats = df.describe()
stats[['p (mbar)', 'T (degC)']].transpose()
```

Display the shape of the dataframe:

```
df.shape
```

The dataframe contains 420,551 rows with 15 columns included in each row.

Plot Relative Humidity Over Time

Since data is in a pandas dataframe, it is easy to plot any of the 14 features against *Date Time* time steps.

Plot yearly periodicity of relative humidity:

```
import matplotlib.pyplot as plt

# create new dataframe with just relative humidity column
rh = df['rh (%)']

# plot it!
rh.plot()
```

Since we have an observation every 10 minutes, each hour has six observations. And each day has 144 (6 observations × 24 hours) observations.

Plot the first 10 days:

```
rh10 = df['rh (%)'][0:1439]
rh10.plot()
```

Since we have 144 observations per day, we plot a total of 1,440 (10 days × 144 observations/day) observations. Notice that indexing begins a *0*.

With a narrow view of the data (the first 10 days), we can see daily periodicity. We also see that fluctuation is pretty chaotic, which means that prediction is more difficult.

Explore humidity by time step at a granular level:

```
df[['Date Time','rh (%)']].head()
```

Forecast a Univariate Time Series

Scale Data

Convert the dataframe into a numpy array:

```
rh_np = rh.to_numpy()
```

Scale numpy data for efficient training as shown in Listing 10-2.

Listing 10-2. Scale data

```
br ='\n'

# original data
print ('first five unscaled observations:', rh_np, br)

# scale relative humidity data
rh_sc = tf.keras.utils.normalize(rh_np)
print ('shape after tf function:', rh_sc.shape)

# squeeze out '1' dimension
rh_sq = tf.squeeze(rh_sc)
print ('shape after squeeze:', rh_sq.shape, br)

# convert to numpy
rh_scaled = rh_sq.numpy()
print ('first five scaled observations:', rh_scaled[:5])
```

Scale relative humidity data. Squeeze out the extra *1* dimension added by the TensorFlow function so we can convert the TensorFlow tensor into a numpy array for easier processing. Display the first five scaled observations to verify that scaling works as expected.

Establish Training Split

Calculate train split size:

```
import numpy as np

# train split with 75% of data
train_split = int(np.round(df.shape[0] * .75))
train_split
```

Calculate test split size:

```
# test split with 25% of data
test_split = df.shape[0] - train_split
test_split
```

Calculate number of days of train and test data:

```
# calculate number of days of data
print (np.round(train_split / 144, 2))
print (np.round(test_split / 144, 2))
```

For this experiment, use the first 315,413 rows of data for training and the remaining 105,138 (420,551 – 315,413) rows for the test set. Training data accounts for about 2,190 (315, 413/144) days of data. And test data accounts for about 730 (105, 138/144) days of data.

Create Features and Labels

Create a function that splits a dataset into features and labels as shown in Listing 10-3.

Listing 10-3. Function that creates features and labels

```
def create_datasets(data, origin, end, window, target_size):

  # list to hold feature set of windows
  features = []

  # list to hold labels
  labels = []

  # establish starting point that reflects window size
  origin = origin + window

  # enable split for test data
  if end is None:
    end = len(data) - target_size

  # create feature set of 'window-sized' elements
  for i in range(origin, end):

    # create index set to identify each window
    indices = range(i-window, i)

    # reshape data from (window,) to (window, 1)
    features.append(np.reshape(data[indices], (window, 1)))
```

```
    # create labels
    labels.append(data[i+target_size])

  return np.array(features), np.array(labels)
```

The function accepts a dataset, an index where we want to start the split, an ending index, the size of each window, and the target size. Parameter *window* is the size of the past window of information. The *target_size* is how far in the future we want our model to learn how to predict.

The function creates lists to hold features and labels. It then establishes the starting point that reflects the window size. To create a test set, the function checks the *end* value. If *None*, it uses the length of the entire dataset less the target size as the ending value so the test set can start where the training set left off.

Once training and test starting points are established, the function creates the feature windows and labels. The *indices* for each feature window are established as the next window during iteration. That is, each subsequent set of *indices* begins where the last set ended. Feature windows are reshaped for TensorFlow consumption and added to the *features* list. Labels are created as the last observation in the next window and then added to the *labels* list. Both features and labels are returned as numpy arrays.

The function may seem confusing, but all it really does is create a feature set that holds windows of relative humidity observations (for our experiment) and another that holds targets. Targets are based on the last relative humidity observation from the next window of data. This makes sense because the last relative humidity from the next window is a pretty good indication of future relative humidity. So the feature set becomes a set of windows that contain time step observations. And the label set contains predictions for each window.

Create Train and Test Sets

For the train set, start at index 0 from the dataset and continue up to 315,412. For the test set, take the remainder. Set the window size to 20 and target to 0.

Invoke the function as shown in Listing 10-4.

Listing 10-4. Create train and test sets

```
# create train and test sets

import numpy as np

window = 20
target = 0

x_train, y_train = create_datasets(rh_scaled, 0, train_split,
                                   window, target)

x_test, y_test = create_datasets(rh_scaled, train_split, None,
                                 window, target)
```

Inspect train and test data:

```
print ('train:', end=' ')
print (x_train.shape, y_train.shape)

print ('test:', end=' ')
print (x_test.shape, y_test.shape)
```

As expected, shapes reflect size of each dataset, window size, and the 1 dimension. The *1* dimension indicates that we are making one prediction into the future. The train set contains 315,393 records composed of windows holding 20 relative humidity readings. The test set contains 105,118 records composed of windows holding 20 relative humidity readings.

So why do we have 315,393 training observations instead of the original 315,413? The reason is that we need the first window to act as history. So just subtract the first window of 20 from 315,413. For test data, subtract the first window of 20 from 105,138 to get 105,118.

We can create bigger windows, but this dramatically increases the amount of data we must process. With only 20 observations per window, we already have 6,307,860 (315,393 × 20) data points for training and 2,102,360 (105,118 × 20) data points for testing!

View Windows of Past History

Inspect the first window from the train set:

```
print ('length of window:', len(x_train[0]), br)
print ('first window of past history:')
print (x_train[0], br)
print ('target relative humidity to predict:')
print (y_train[0])
```

As expected, the window contains 20 relative humidity readings. So how did we get the target?

Take the last entry from the next window:

```
print ('target from the 1st window:', end='   ')
print (np.round(y_train[0], 8))
print ('last obs from the 2nd window:', end=' ')
print (np.round(x_train[1][19][0], 8))
```

Verify by inspecting the second window:

```
print ('second window of past history:')
print (x_train[1], br)
print ('target relative humidity to predict:')
print (y_train[1])
```

Let's see if the pattern holds for the next few windows as shown in Listing 10-5.

Listing 10-5. Inspect the patterns for the next few windows

```
print ('target from the 2nd window:', end='   ')
print (np.round(y_train[1], 8))
print ('last obs from the 3rd window:', end=' ')
print (np.round(x_train[2][19][0], 8), br)

print ('target from the 3rd window:', end='   ')
print (np.round(y_train[2], 8))
print ('last obs from the 4th window:', end=' ')
print (np.round(x_train[3][19][0], 8), br)
```

```
print ('target from the 4th window:', end='   ')
print (np.round(y_train[3], 8))
print ('last obs from the 5th window:', end=' ')
print (np.round(x_train[4][19][0], 8), br)

print ('target from the 5th window:', end='   ')
print (np.round(y_train[4], 8))
print ('last obs from the 6th window:', end=' ')
print (np.round(x_train[5][19][0], 8))
```

Plot a Single Example

Create a function that returns a list of time steps from *-length* to *0*:

```
def create_time_steps(length):
  return list(range(-length, 0))
```

Start at -length to 0 to use the previous window as history.

Create another function that accepts a single data window and its target, delta, and a title as shown in Listing 10-6.

Listing 10-6. Function that plots an example

```
def plot(plot_data, delta=0, title='Data Window'):
  labels = ['history', 'actual future', 'model prediction']
  marker = ['r.-', 'b*', 'g>']
  time_steps = create_time_steps(plot_data[0].shape[0])
  if delta: future = delta
  else: future = 0
  plt.title(title)
  for i, obs in enumerate(plot_data):
    if i:
      plt.plot(future, obs, marker[i], markersize=10,
               label=labels[i])
    else:
      plt.plot(time_steps, obs.flatten(), marker[i],
               label=labels[i])
```

```
  plt.legend()
  plt.xlim([time_steps[0], (future+5)*2])
  plt.xlabel('time step')
  return plt
```

Parameter *delta* indicates a change in a variable. The default is no change. The function plots each element in the data window with its associated time step.

Invoke the function based on the first data window and target from the train set:

```
plot([x_train[0], y_train[0]])
```

The *actual future* for each window is its label, which is the last entry of the next window.

Create a Visual Performance Baseline

Before training, it is a good idea to create a simple visual performance baseline to compare against model performance. Of course, there are many ways to do this, but a very simple way is to use the average of the last 20 observations.

Create a function that returns the average of a window of observations:

```
def baseline(history):
  return np.mean(history)
```

Plot the first data window with a baseline prediction:

```
plot([x_train[0], y_train[0], baseline(x_train[0])], 0,
     'baseline prediction')
```

Create a Baseline Metric

It's also a good idea to create a baseline metric to compare against our model. The simplest approach is to predict the last value for each window. We can then find the average mean squared error of the predictions and use this value as our metric.

Create a baseline metric as shown in Listing 10-7.

Listing 10-7. Create a baseline metric

```
# display shape of test set
print ('TensorFlow shape:', x_test[0].shape, br)

# remove '1' dimension for easier processing
x_test_np = tf.squeeze(x_test)
print ('numpy shape:', x_test_np[0].shape, br)

# predict last value for each window
y_pred = x_test_np[:, -1]

# compute average MSE
MSE = np.mean(tf.keras.losses.mean_squared_error(
    y_test, y_pred))
print ('MSE:', MSE)
```

The MSE is very small, which means that our baseline metric might be hard to beat. Why? In general, machine learning has a pretty significant limitation. Unless the learning algorithm is hardcoded to look for a specific kind of simple model, parameter learning can sometimes fail to find a simple solution to a simple problem. Our time series problem is a very simple problem.

Finish the Input Pipeline

Shuffle train data, batch, and cache as shown in Listing 10-8.

Listing 10-8. Finish the input pipeline

```
BATCH_SIZE = 256
BUFFER_SIZE = 10000

# prepare the train set
train = tf.data.Dataset.from_tensor_slices((x_train, y_train))
train_one = (train.cache()
.shuffle(BUFFER_SIZE)
.batch(BATCH_SIZE)
.repeat())
```

```
# prepare the test set
test = tf.data.Dataset.from_tensor_slices((x_test, y_test))
test_one = test.batch(BATCH_SIZE).repeat()
```

Inspect tensors:

```
train_one, test_one
```

As expected, windows have 20 observations and 1 prediction.

Explore a Data Window

Verify that features and labels are batched in 256-element windows:

```
for feature, label in train_one.take(1):
  print (len(feature), len(label))
```

Display the first window from the batch:

```
for feature, label in train_one.take(1):
  print ('feature:')
  print (feature[0].numpy(), br)
  print ('label:', label[0].numpy())
```

As expected, the window contains a feature set with 20 observations and a label with one prediction.

Create the Model

Establish the input shape:

```
input_shape = x_train.shape[-2:]
input_shape
```

Input shape indicates window size of 20 with 1 prediction.

Import requisite libraries, clear previous model sessions, generate a seed for reproducibility, and create the model as shown in Listing 10-9.

Listing 10-9. Create the model

```
# clear any previous models
tf.keras.backend.clear_session()

#import libraries
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import GRU, Dense

# generate seed to ensure reproducibility
tf.random.set_seed(0)

neurons = 32  # number of neurons in GRU layer

model = Sequential([
  GRU(neurons, input_shape=input_shape),
  Dense(1)
])
```

A RNN is well suited to time series data because its layers can provide feedback to earlier layers. Specifically, it processes time series data time step by time step while remembering information it sees during training. Our model uses a GRU layer, which is a specialized RNN layer capable of remembering information over long periods of time. So it is especially well suited for time series modeling.

Model Summary

Inspect the model:

```
model.summary()
```

We use the formula $3 \times (n^2 \times mn + 2n)$ to calculate the number of learnable parameters for a GRU, where m is the input dimension and n is the output dimension. For any neural net, multiply output from the previous layer by neurons at the current layer ($m \times n$). In addition, account for neurons at the current layer. But count them twice because of feedback ($2n$). A GRU layer has feedback, so square the output dimension (n^2). Multiply by 3 because there are three sets of operations for a GRU (hidden state, reset gate, and update gate) that require weight matrices.

Here is the breakdown to calculate parameters for the GRU layer:

- $3 \times (32^2 + 32 + 2 \times 32)$
- $3 \times (1024 + 32 + 64)$
- 3×1120
- *3,360*

The output dimension is 32. The input dimension doesn't exist since there is no previous layer.

The dense layer has 33 learnable parameters calculated by multiplying neurons from the previous layer (32) by neurons at the current layer (1) and adding the number of neurons at the current layer (1).

Verify Model Output

Make an *untrained* prediction from the model:

```
for x, y in test_one.take(1):
  print(model.predict(x).shape)
```

The prediction shows batch size of 256 with 1 prediction. So the model is working as expected.

Compile the Model

Compile:

```
model.compile(optimizer='adam', loss='mse')
```

Train the Model

Train:

```
num_train_steps = 400
epochs = 10
```

```
history = model.fit(train_one, epochs=epochs,
                    steps_per_epoch=num_train_steps,
                    validation_data=test_one,
                    validation_steps=50)
```

Generalize on Test Data

Generalize:

```
test_loss = model.evaluate(test_one, steps=num_train_steps)
```

Make Predictions

Make three predictions as shown in Listing 10-10.

Listing 10-10. Make some predictions

```
n = 3
title = 'GRU prediction'

for i, (x, y) in enumerate(test_one.take(n)):
  p = model.predict(x)[0]
  plot([x[0].numpy(), y[0].numpy(), p], 0,
       title + ' window ' + str(i))
  plt.show()
```

Although visual inspection is pretty, it doesn't efficiently gauge overall performance.

Plot Model Performance

Listing 10-11 plots model performance:

Listing 10-11. Plot model performance

```
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1)
```

```
plt.figure()
plt.plot(epochs, loss, 'bo', label='training loss')
plt.plot(epochs, val_loss, 'b', label='validation loss')
plt.title('training and validation loss')
plt.legend()
plt.show()
```

Voilà! Our model performs pretty well!

Forecast a Multivariate Time Series

We just demonstrated how to make a single prediction based on a single feature. Now, let's make multiple predictions on multiple variables. We can choose any of the 14 features (we don't want to predict from the date-time reference).

The following are the available 14 features:

- p (mbar) – Atmospheric pressure in millibars
- T (degC) – Temperature in Celsius
- Tpot (K) – Temperature in Kelvin
- Tdew (degC) – Temperature in Celsius relative to humidity
- rh (%) – Relative humidity
- VPmax (mbar) – Saturation vapor pressure in millibars
- VPact (mbar) – Vapor pressure in millibars
- VPdef (mbar) – Vapor pressure deficit in millibars
- sh (g/kg) – Specific humidity in grams per kilogram
- H2OC (mmol/mol) – Water vapor concentration in millimoles per mole
- rho (g/m**3) – Air density in grams per meter cubed
- wv (m/s) – Wind speed in meters per second
- max. wv (m/s) – Maximum wind speed in meters per second
- wd (deg) – Wind direction in degrees

Create a variable to hold the features we wish to consider for our experiment:

```
mv_features = ['Tdew (degC)', 'sh (g/kg)', 'H2OC (mmol/mol)', 'T (degC)']
```

We chose four features – Tdew (degC), sh (g/kg), H2OC (mmol/mol), and T (degC). *Tdew (degC)* is the temperature in Celsius relative to humidity. *sh (g/kg)* is the specific humidity in grams per kilogram. *H2OC (mmol/mol)* is the water vapor concentration in millimoles per mole. And *T (degC)* is the temperature in Celsius. We chose these features for demonstration purposes only. Choice of features should be based on a problem domain.

Create a dataframe to hold the features:

```
mv_features = df[mv_features]
mv_features.index = df['Date Time']
mv_features.head()
```

Visualize features:

```
mv_features.plot(subplots=True)
```

Visualize a single feature:

```
mv_features['T (degC)'].plot(subplots=True)
```

Scale Data

Convert the dataframe to a numpy array:

```
f_np = mv_features.to_numpy()
f_np[:5]
```

Check the number of observations:

```
len(f_np)
```

As expected, we have 420,551 observations.

Scale data as shown in Listing 10-12.

Listing 10-12. Scale data

```
# scale features
f_sc = tf.keras.utils.normalize(f_np)
print ('shape after tf function:', f_sc.shape, br)

# squeeze
f_sq = tf.squeeze(f_sc)

# convert to numpy
f_scaled = f_sq.numpy()
print ('first five scaled observation:')
print (f_scaled[:5])
```

Multistep Model

With relative humidity, we only predicted a single future point. But we can create a model to learn to predict a range of future values, which is what we are going to do with the multivariate data we just established.

Let's say that we want to train our multistep model to learn to predict for the next 6 hours. Since our data time steps are 10 minutes (one observation every 10 minutes), there are six observations every hour. Given that we want to predict for the next 6 hours, our model makes *36* (6 observations × 6) predictions.

Let's also say that we want to show our model data from the last 3 days for each sample. Since there are 24 hours in a day, we have 144 (6 × 24) observations each day. So we have a total of *432* (3 × 144) observations. But we want to sample every hour because we don't expect a drastic change in any of our features within 60 minutes. Thus, *72* (432 observations/6 observations per hour) observations represents each window of data.

Generators

Since we are training multiple features to predict a range of future values, we create a generator function to create train and test splits. A **generator** is a function that returns an object iterator that we iterate over one value at a time. A generator is defined like a normal function, but it generates a value with the *yield* keyword rather than *return*. So adding the yield keyword automatically makes a normal function a generator function.

Generators are easy to implement, but a bit difficult to understand. We invoke a generator function in the same way as a normal function. But, when we invoke it, a generator object is created. We must iterate over the generator object to see its contents. As we iterate over the generator object, all processes within the generator function are processed until it reaches a yield statement. Once this happens, the generator yields a new value from its contents and returns execution back to the for loop. So a generator yields one element from its contents for each yield statement encountered.

Advantages of Using a Generator

Generator functions allow us to declare a function that behaves like an iterator. So we can make iterators in a fast, easy, and clean way. An **iterator** is an object that can be iterated upon. It is used to abstract a container of data to make it behave like an iterable object. As programmers, we use iterable objects like strings, lists, and dictionaries frequently.

Generators save memory space because they don't compute the value of each item when instantiated. Generators only compute a value when explicitly asked to do so. Such behavior is known as *lazy evaluation*. Lazy evaluation is useful when we process large datasets because it allows us to start using data immediately rather than having to wait until the entire dataset is processed. It also saves memory because data is generated only when needed.

Generator Caveats

- A generator creates a single object. So a generator object can only be assigned to a *single* variable no matter how many values it yields.
- Once iterated over, a generator is *exhausted*. So it must be rerun to be repopulated.

Create a Generator Function

Create a generator as shown in Listing 10-13:

Listing 10-13. Generator function

```
def generator(d, t, o, e, w, ts, s):

  # hold features and labels
  features, labels = [], []

  # initialize variables
  data, target = d, t
  origin, end = o, e
  window, target_size = w, ts
  step = s

  # establish starting point that reflects window size
  origin = origin + window

  # enable split for test data
  if end < 0:
    end = len(data) - target_size

  # create feature set of 'window-sized' elements
  for i in range(origin, end):

    # create index set to identify each window
    indices = range(i-window, i, step)

    # create features
    features.append(data[indices])

    # create labels
    labels.append(target[i:i+target_size])

  yield np.array(features), np.array(labels)
```

Generate Train and Test Data

Invoke the generator to create train and test sets as shown in Listing 10-14.

Listing 10-14. Invoke the generator

```
window = 432  # observations for 3 days
future_target = 36  # predictions for the next 6 hours
step = 6  # number of timesteps per hour

train_gen = generator(f_scaled, f_scaled[:, 1], 0,
                      train_split, window,
                      future_target, step)

test_gen = generator(f_scaled, f_scaled[:, 1],
                     train_split, -1, window,
                     future_target, step)
```

Notice that we assign the generator object to a single variable!

Reconstitute Generated Tensors

Remake generated data into numpy arrays. Since train and test data are generators, we must iterate to create features and labels.

Begin with train data by iterating the generator to create features and labels lists:

```
train_f, train_l = [], []
for i, row in enumerate(train_gen):
  train_f.append(row[0])
  train_l.append(row[1])
```

Convert list data to numpy arrays:

```
train_features = np.asarray(train_f, dtype=np.float64)
train_labels = np.asarray(train_l, dtype=np.float64)
```

Inspect shapes:

```
train_features.shape, train_labels.shape
```

As expected, features have 72 observations per window and 4 features. And labels have 36 predictions.

Remove the *1* dimension with the *tf.squeeze* function:

```
train_features, train_labels = tf.squeeze(train_features),\
tf.squeeze(train_labels)

train_features.shape, train_labels.shape
```

Reconstitute test data as shown in Listing 10-15.

Listing 10-15. Reconstitute test data

```
# create test data
test_f, test_l = [], []
for i, row in enumerate(test_gen):
  test_f.append(row[0])
  test_l.append(row[1])

# convert lists to numpy arrays
test_features = np.asarray(test_f, dtype=np.float64)
test_labels = np.asarray(test_l, dtype=np.float64)

# squeeze out the '1' dimension created by the generator
test_features, test_labels = tf.squeeze(test_features),\
tf.squeeze(test_labels)
test_features.shape, test_labels.shape
```

Finish the Input Pipeline

Finish the input pipeline as shown in Listing 10-16.

Listing 10-16. Finish the input pipeline

```
train_mv = tf.data.Dataset.from_tensor_slices(
    (train_features, train_labels))
train_mv = train_mv.cache().shuffle(
    BUFFER_SIZE).batch(BATCH_SIZE).repeat()

test_mv = tf.data.Dataset.from_tensor_slices(
    (test_features, test_labels))
test_mv = test_mv.batch(BATCH_SIZE).repeat()
```

Check tensors:

```
train_mv, test_mv
```

Check a batch:

```
for feature, label in train_mv.take(1):
  print (feature.shape)
  print (label.shape)
```

Each batch contains 256 windows of feature data and 256 labels. Each window has 72 observations with each observation containing 4 features. Each label has 36 predictions.

Check the first training example:

```
for feature, label in train_mv.take(1):
  print ('observations:', len(feature[0]))
  print (feature[0], br)
  print ('predictions:', len(label[0]))
  print (label[0])
```

As expected, the first window has 72 observations with 4 features, and the first label has 36 predictions.

Establish input shape:

```
input_shape_multi = feature.shape[-2:]
input_shape_multi
```

As expected, window size is 72 with 4 features for each window.

Create the Model

Create the model as shown in Listing 10-17.

Listing 10-17. Create the model

```
# clear any previous models
tf.keras.backend.clear_session()

# generate seed to ensure reproducibility
tf.random.set_seed(0)
```

```
neurons = 32  # neurons in GRU layer
outputs = 36  # predictions

gen_model = Sequential([
  GRU(neurons, input_shape=input_shape_multi),
  Dense(outputs)
])
```

Model Summary

Inspect model:

```
gen_model.summary()
```

Here is the breakdown to calculate learnable parameters for the GRU layer:

- $3 \times (32^2 + 32 \times 4 + 2 \times 32)$
- $3 \times (1024 + 128 + 64)$
- 3×1216
- *3,648*

The only difference is that we have four features. So multiply the output (n) dimension by 4.

The dense layer has 1,188 learnable parameters calculated by multiplying neurons from the previous layer (32) by neurons at the current layer (36) and adding the number of neurons at the current layer (36).

Compile the Model

Compile:

```
gen_model.compile(optimizer='adam', loss='mse')
```

Train the Model

Train:

```
num_train_steps = 400
epochs = 10
```

```
gen_history = gen_model.fit(train_mv, epochs=epochs,
                            steps_per_epoch=num_train_steps,
                            validation_data=test_mv,
                            validation_steps=50)
```

Generalize on Test Data

Generalize:

```
test_loss = gen_model.evaluate(test_mv, steps=num_train_steps)
```

Plot Performance

Plot performance as shown in Listing 10-18.

Listing 10-18. Plot performance

```
loss = gen_history.history['loss']
val_loss = gen_history.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.figure()
plt.plot(epochs, loss, 'bo', label='training loss')
plt.plot(epochs, val_loss, 'b', label='validation loss')
plt.title('training and validation loss')
plt.legend()
plt.show()
```

The model is overfitting a bit, but performance is still pretty good.

Plot a Data Window

Create a plotting function as shown in Listing 10-19.

Listing 10-19. Plotting function

```
def multi_step_plot(window, true_future, pred):
  plt.figure(figsize=(12, 6))
  num_in = create_time_steps(len(window))
  num_out = len(true_future)
```

```
  plt.plot(num_in, np.array(window[:, 1]), 'm',
           label='history')
  plt.plot(np.arange(num_out)/step, np.array(true_future),
           'bo', label='true future')
  if pred.any():
    plt.plot(np.arange(num_out)/step, np.array(pred), 'go',
             label='predicted future')
  plt.legend(loc='upper left')
  plt.show()
```

Plot the first training window from the first batch:

```
for x, y in train_mv.take(1):
  multi_step_plot(x[0], y[0], np.array([0]))
```

Make a Prediction

Make a prediction based on the first training window:

```
for x, y in test_mv.take(1):
  y_pred = gen_model.predict(x)[0]
  multi_step_plot(x[0], y[0], y_pred)
```

Not bad.

Index

A

B

D. Paper, *TensorFlow 2.x in the Colaboratory Cloud*, https://doi.org/10.1007/978-1-4842-6649-6

C

D, E

M

N, O, P, Q

R

S

T

U, V

W, X, Y, Z