Artificial Neural Networks with TensorFlow 2

ANN Architecture Machine Learning Projects

Poornachandra Sarang

Apress®

Artificial Neural Networks with TensorFlow 2

ANN Architecture Machine Learning Projects

Poornachandra Sarang

Apress®

Artificial Neural Networks with TensorFlow 2: ANN Architecture Machine Learning Projects

Poornachandra Sarang
Mumbai, India

ISBN-13 (pbk): 978-1-4842-6149-1 ISBN-13 (electronic): 978-1-4842-6150-7
https://doi.org/10.1007/978-1-4842-6150-7

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

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

For information on translations, please e-mail 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/978-1-4842-6149-1. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Poornachandra Sarang started his IT career way back in the late 1980s. During this long career, he had an opportunity to work on a wide variety of technologies. He has taught Computer Science/Engineering at the University of Notre Dame and the University of Mumbai. He has been a Ph.D. advisor for Computer Science and is currently on a Thesis Advisory Committee for students pursuing Ph.D. in Computer Engineering. His current research interests are in machine/deep learning. He has several papers and journal articles to his credit and has presented in several conferences.

About the Technical Reviewer

Vishwesh Ravi Shrimali graduated from BITS Pilani in 2018, where he studied mechanical engineering. Since then, he has worked with Big Vision LLC on deep learning and computer vision and was involved in creating official OpenCV AI courses. Currently, he is working at Mercedes-Benz Research and Development India Pvt. Ltd. He has a keen interest in programming and AI and has applied that interest in mechanical engineering projects. He has also written multiple blogs on OpenCV and deep learning on Learn OpenCV, a leading blog on computer vision. He has also authored *Machine Learning for OpenCV* (2nd edition) by Packt. When he is not writing blogs or working on projects, he likes to go on long walks or play his acoustic guitar.

Acknowledgments

I would first like to thank my past student Prof. Abhijit Gole, who is currently a faculty member in the Department of Computer Science at an autonomous college affiliated to the University of Mumbai. Abhijit initiated me into delivering a postgraduate course in machine learning to his students. This triggered my interest in guiding students in machine/deep learning for their projects at the master's level and in guiding those pursuing Ph.D.

My sincere thanks to my four intern students for their contribution in developing this book. Mukul Rawat graduated this year (2020), and Karan Aryan, Chandrakant Sharma, and Udit Dashore will be graduating next year (2021). All four are brilliant students, and without their help in developing the projects in this book, I would have taken much longer time in completing this book. All of them took painstaking efforts in reviewing the first draft of this book for technical discrepancies.

I would like to thank Mr. Aaron Black, Senior Editor, Apress, who gave me the opportunity to author this book, taking quick decisions on the book's outline and providing me the desired flexibility in finalizing the outline as I kept developing the contents. I would like to thank Jessica Vakili, Coordinating Editor, for her excellent coordination on various fronts, especially on tech review. I would also like to thank the entire Apress team for bringing this book out quickly to market.

Lastly, but not least importantly, I would like to thank Vijay Jadhav who put in lots of effort in running all the project sources to bring out the discrepancies in writing and the code. His efforts in formatting the manuscript as per Apress authoring guidelines require a special mention.

Preface

With recent advances in deep learning and the ever-growing popularity of TensorFlow, you see several domains where machine learning models have caused a tremendous impact. For a developer, it gives a great opportunity to apply their skills in real-world projects across a wide range of domains. Having a knowledge of what is available in the industry today becomes an important asset. This book attempts to provide you with a wide range of real-world applications based on deep neural networks (DNN) and the latest version of TensorFlow. It provides a comprehensive coverage of various deep learning architectures. It is fully project-oriented and contains about 25 fully implemented real-life projects.

Chapter 1 starts with an introduction to TensorFlow 2, describing why it is considered a complete platform for machine learning. It describes the various additions to TensorFlow such as tf.keras integration, eager execution, distributed training, deployment on edge devices, and the use of data pipelines. At the end, it provides guidance for setting up the development environment for running projects in the book.

Chapter 2 goes deeper into TensorFlow 2. As is the convention, we start with a trivial "Hello World" kind of application introducing you to the complete machine learning development cycle. The chapter then deals with a binary classification problem that is a typical application taught at the beginning of any machine learning course. Here, you are introduced to data preprocessing, defining a dense neural network using tf.keras API, training the model, and evaluating its performance using TensorBoard metrics and the traditional confusion matrix. Rather than dealing with only the classification, the project teaches you how to load data at runtime into your development environment, how to preprocess it,

how to develop a machine learning model, and finally how to evaluate the model's performance using TensorBoard analytics and other traditional techniques.

Chapter 3 goes deeper in the tf.keras module. It introduces you to the functional API for model building. It talks about object orientation in TensorFlow and shows how to create a class hierarchy for your models and how to create custom layers by subclassing. You will learn how to save the model using TensorFlow's new SavedModel interface and several ways of saving partial or full models. You will then do a project on a multiclass classification using predefined Convolutional Neural Network (CNN) layers in tf.keras. Again, unlike the traditional ways of teaching, I show you how to experiment with different network architectures to optimize model performance.

As training deep neural networks using multiple CNN layers is resource hungry, in Chapter 4 I show you how to take advantage of somebody else's training. This chapter deals with transfer learning where the knowledge of a domain expert and somebody else's model development efforts are used to your advantage almost effortlessly. The chapter deals with two projects. The first project shows you how to use a pre-trained ImageNet classifier to classify your own set of images. The second project shows you how to construct your own classifier based on a pre-trained model.

Any machine learning training starts with classification and regression problems. In the previous chapters, we tackled classification to a great depth. The question arises: Can we use deep neural networks for regression analysis where there are several successful statistical-based techniques already available for our use? I show you how to use DNN for regression in Chapter 5 with the help of three different dense neural network architectures.

Though the use of pre-trained models eases your custom model development, there are many instances where you cannot find a pre-trained model that fits your purpose. The model development from scratch requires lots of plumbing; the Estimators in TensorFlow come to your help in these situations. Chapter 6 deals with estimators. It contains four projects. The first two projects describe how to use a predefined estimator for classification and then a regression. The third project talks about how to create a custom estimator from scratch, and the last one talks about creating a custom estimator for a pre-trained model.

Chapter 7 talks about text generation techniques. It describes the traditional techniques of RNNs (recurrent neural networks) and LSTM (long short-term memory) for text generation and shows you how to use LSTM for a trivial application of generating baby names. The next project in the chapter uses some advanced text generation techniques to create text matching the linguistics of a famous novel by Tolstoy. As training a text generation model is a very lengthy process, I will show some techniques of performing such long-duration training by showing you how to continue upon an interrupted training, which may be an intentional or unintentional interrupt.

Chapter 8 takes you further in the text generation where I describe the seq2seq model, encoder/decoder architecture, and a recent attention model for language translation. The chapter gives an in-depth coverage of the English to Spanish translator that uses glove word embedding. It shows you how to define your own machine learning model with encoder, decoder, and attention layers.

Chapter 9 describes the state-of-the-art technology for text generation. It describes the latest transformer-based architecture for natural language processing. A complete in-depth project teaches you how to construct a transformer model on your own in Python, and not to depend fully on just the pre-trained models such as BERT (Bidirectional Encoder Representations from Transformers).

After text generation, the book takes you into image processing. Chapter 10 teaches you how to process an image and use an LSTM architecture that you learned in Chapters 7 and 8 to generate a caption for an image. The chapter deals with a hands-on project for image captioning. The project uses a pre-trained model, InceptionV3, for image processing. This is yet another practical use of transfer learning that you learned in Chapter 4. It shows you how to create a decoder with Bahdanau Attention. In the end, you will use this model for captioning any image taken from the Web.

Chapter 11 takes you into another domain of machine learning, and that is time series analysis and forecasting. The chapter gives you two full projects – one for a univariate time series forecasting and the other one for a multivariate time series forecasting.

Chapter 12 deals with yet another aspect of machine learning that can apply the style of a famous painter to your own camera-captured pictures to make them look like the paintings created by the famous painter. We call this style transfer. The chapter contains two projects. The first project shows you how to use a pre-trained model from the TensorFlow hub to perform a quick style transfer. By doing this, you will understand what is a style transfer. The next project takes you deeper into the technique where you would use a predefined VGG16 architecture to extract the features of an image and a famous painting. Then, you will learn how to define an evaluator to compare the content and style losses during a style transfer to create a final stylized picture.

Chapter 13 deals with another important DNN architecture called GAN (Generative Adversarial Networks). It contains a total of three projects. The first one is trivial. It shows you how to use GAN for generating images for handwritten digits. The second one takes you further into the generation of handwritten alphabets. The last one finally takes you into the most advanced use of GANs for generating complex anime colored character images.

Chapter 14 teaches you an important technique of image manipulation, that is, how to colorize a B&W image. You use AutoEncoder for colorizing an image. The chapter describes two projects. The first one shows you how to build an AutoEncoder of your own. The second one uses a pre-trained VGG16 model for encoding to improve the model's performance.

The book gives an exhaustive collection of several deep neural network architectures with an emphasis on their practical usage in real-life scenarios. So, get started!

CHAPTER 1

TensorFlow Jump Start

TensorFlow is an end-to-end open source platform for developing and deploying machine learning applications. We can call it the complete machine learning (ML) ecosystem. All of us have seen face tagging in our photos on Facebook. Well, this is a machine learning application. Autonomous cars use object detection to avoid collisions on the road. Machines now translate Spanish to English. Human voices are converted into text for you to create a digital document. All these are machine learning applications. Even a trivial OCR (optical character reader) application that we use so often uses machine learning. There are many more advanced applications developed today – such as captioning images, generating images, translating images, forecasting a time series, understanding human languages, and so on. All such applications and many more can be developed and deployed on the TensorFlow platform. And exactly, that's what you are going to learn in this book.

It doesn't matter whether you are a beginner or an expert, TensorFlow will help you build your own ML models with ease. In TensorFlow, you are able to define your own neural network architectures, experiment with them, train them, and finally deploy them on the production servers. Not only this, a fully trained model can be deployed on mobiles, on embedded devices, and also on the Web with JavaScript support.

P. Sarang, *Artificial Neural Networks with TensorFlow 2*,
https://doi.org/10.1007/978-1-4842-6150-7_1

You might have used other machine learning development libraries – just to name a few, Keras, Torch, Theano, and Pytorch. A recent research conducted by KDnuggets (www.kdnuggets.com) on "Which Deep Learning framework is growing fastest?" resulted in the findings shown in Figure 1-1.

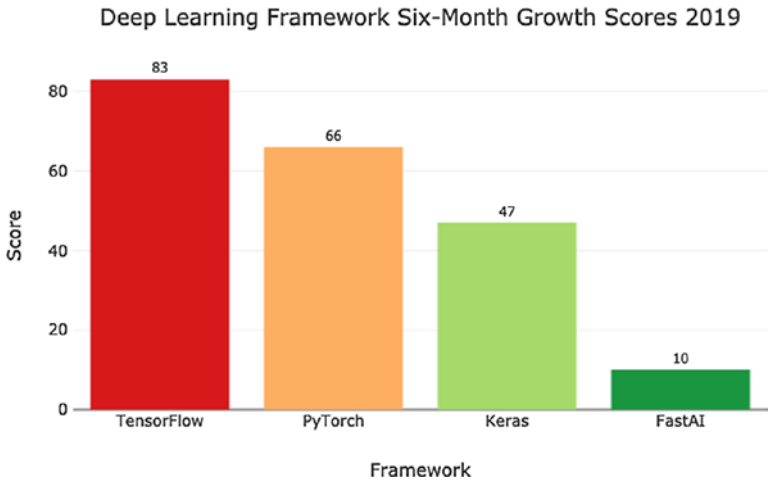


Figure 1-1. *Deep learning framework growth – 2019*

A similar survey in 2018 over several popular frameworks is shown in Figure 1-2.

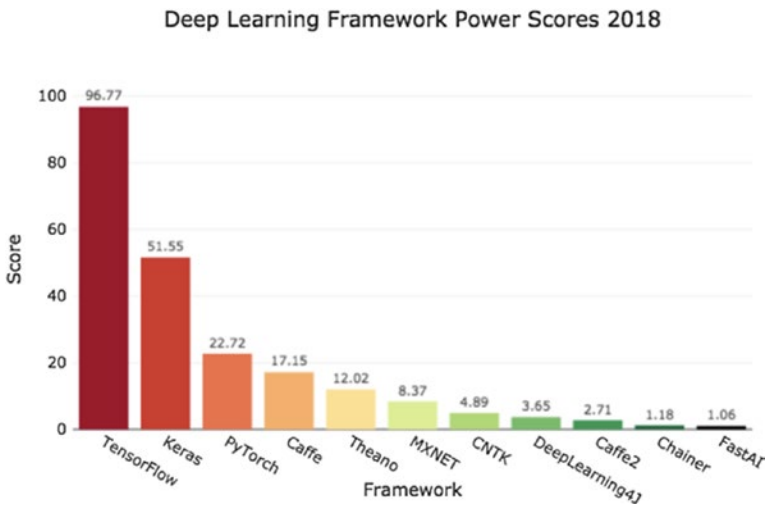


Figure 1-2. *Power scores of several deep learning frameworks*

Very clearly, TensorFlow is the winner among all the deep learning frameworks surveyed. So you have made the right decision in learning and using TensorFlow 2.x for your deep learning application developments. Let us now get started on TensorFlow.

What Is TensorFlow 2.0?

A picture can say a lot more than words. I will give you a simplified conceptual representation of the entire TensorFlow platform.

TensorFlow 2.x Platform

The entire platform is conceptualized using the picture shown in Figure 1-3.

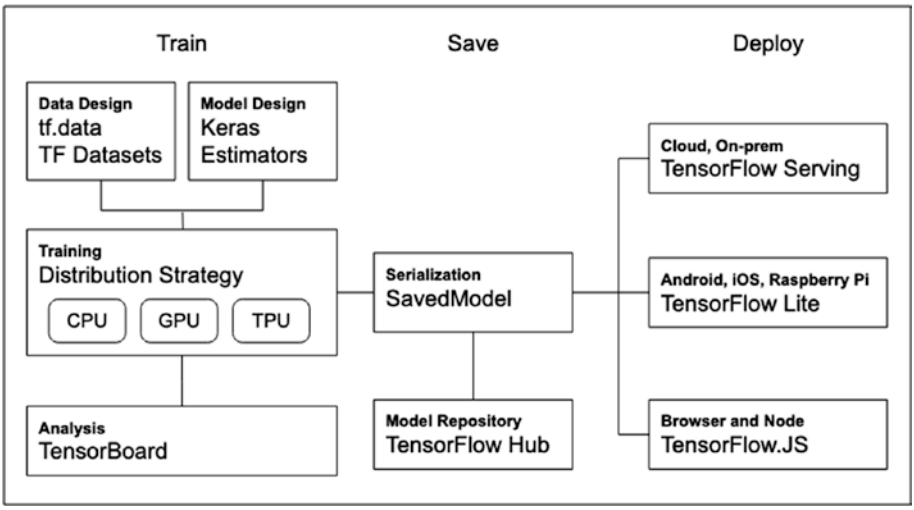


Figure 1-3. *TensorFlow 2.x platform*

As typical of any machine learning project, it consists of three distinct phases. During the first phase, also called the training phase, we define our artificial neural network model and train it on the given data. Also, we test the model using test data and retrain it until we are satisfied with its performance. In the next phase, we save the model to a file, which can later be deployed on a production server. In the third phase of our development, we deploy the saved model on a production server ready to make predictions on unseen data.

I will now describe the individual components of all three phases shown in Figure 1-3.

Training

The training consists of reading data, preparing it in a specific format required by your model, creating the model itself, and running many epochs to train the model. TensorFlow 2.x provides lots of functions, libraries, and tools to facilitate quick training. Generally, the ML model training takes a

considerable amount of time in the entire development process. With the tools and facilities provided in TensorFlow 2.x, you would be able to train the model in a much shorter time as compared to the traditional methods of training. The entire training block is shown in Figure 1-4.

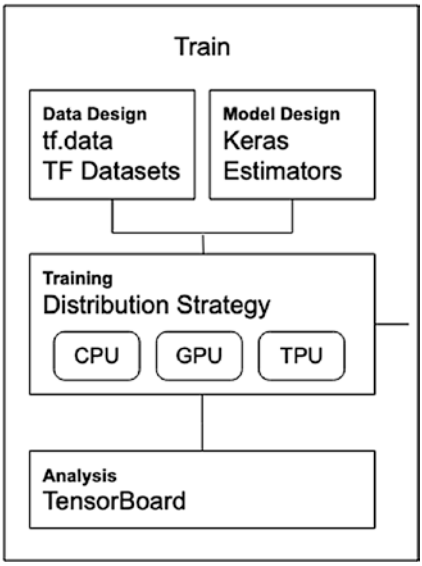


Figure 1-4. *Training block*

I will now explain to you the individual components of the training block.

Data Preparation

Consider the Data Design block shown in Figure 1-5.

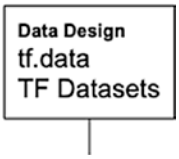


Figure 1-5. *Data Design block*

The Data Design block shows two modules – tf.data and TF Datasets. I will discuss what both contain. First, I will describe tf.data.

Model training requires data be prepared to a specific format as per the model's design. The data preparation requires several steps. Firstly, you need to load the data from an external source and cleanse it. The cleansing process consists of removing rows containing null fields, mapping categorical fields to columns, and scaling numerical values generally to the range of –1 to +1. Next, you will need to decide which columns are to be your features and what the labels are. You will need to eliminate the columns which are not at all relevant to model training. For example, the name and customer ID fields in the database would be totally redundant fields in machine training. Finally, you will need to split the data into training and testing sets. The various functions needed for all these mentioned data operations are provided in the tf.data module.

TensorFlow also provides built-in datasets taken from many popular machine learning libraries ready for use in your tf.data package. There are more than 100 ready-to-use datasets, divided into several categories such as Audio, Image, Video, Text, Translate, and so on. Depending on your requirements, you will load the data from one of the categories and quickly proceed to your model development. In future, more may be added to this module. You will be able to load data from tf.data.dataset using a single program statement that also creates both training and testing sets. So, this saves you a lot of effort in preparing data, and you will be able to quickly focus on model training. Albeit, you may not be able to use dataset as is for feeding it into your machine learning algorithm. Numerical fields may require scaling. Categorical fields may require conversions. The datasets like imdb which is for movie reviews may require encoding to a different format. You may require to reshape (change dimensions) the data. Thus, some sort of preprocessing is almost always required on these built-in datasets. However, they still provide lots of convenience to ML

practitioners. Incidentally, these datasets also support high-performance data pipelines to facilitate quick data transfer between training iterations, resulting in faster training.

Designing Model

The Keras API is now integrated in the TensorFlow library. You can access the entire API using tf.keras. The tf.keras is a high-level API that provides standardization to many APIs used in TF 1.x. Using tf.keras, you will be able to take advantage of several new features introduced in TensorFlow 2.x; as an example, your model development will take advantage of eager execution. Using the functional API of the tf.keras module, you will be able to design models with high complexity.

TensorFlow 2.x also supplies Estimators which can be used to do a quick comparison between different models. The estimators block is shown in Figure 1-6.

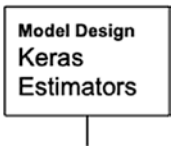


Figure 1-6. *The tf.estimator block*

The estimators are provided under tf.estimator which is a high-level TensorFlow API. They encapsulate the various phases of machine learning like model training, evaluation, predicting, and exporting the model for serving through a production server. There are several premade estimators provided in the library; LinearClassifier and DNNClassifier are two examples of such premade estimators. Besides the use of premade estimators, you will be able to build your own custom estimators. Not only this, the library provides a function called model_to_estimator to convert

the existing model to an estimator. Why would you do this? Converting a Keras model to an estimator would enable you to use TensorFlow's distributed training.

Distribution Strategy

In the entire machine learning process, the most time-consuming part is the training. This can take from a few minutes to several days even on very sophisticated equipment. You may need lots of processing power and memory to train the model. Fortunately, TensorFlow 2.x comes to your rescue here. Look at the distribution strategy block shown in Figure 1-7.

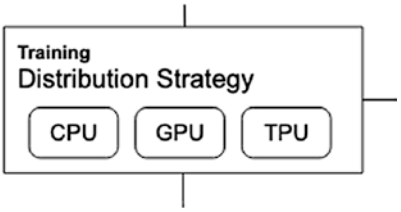


Figure 1-7. *Distribution strategy block*

The model training can now be done on the CPU (central processing unit), GPU (graphics processing unit), or TPU (tensor processing unit). Not only this, but you can distribute your training across the multiple hardware units. This reduces your training time substantially.

Analysis

During the model training phase, you need to analyze the results at different stages of training. Based on this, you will reconfigure your network, modify your loss functions, try different optimizer, and so on. TensorFlow provides a nice analysis tool called TensorBoard for this very purpose, as depicted in Figure 1-8.

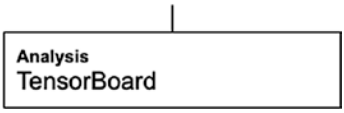


Figure 1-8. *TensorBoard block*

TensorBoard provides the plots of various metrics, such as accuracy and loss, which we widely used during model training. There are several other features available in TensorBoard, which you will keep learning as you read the book further.

Model Saving

The model saving block from the general architecture is depicted in Figure 1-9.

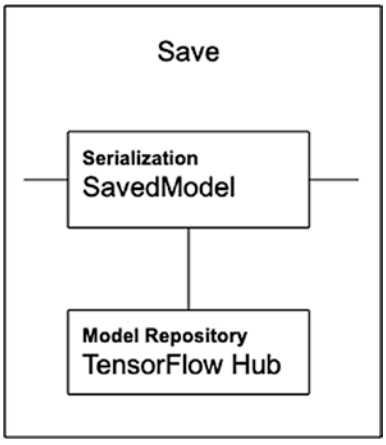


Figure 1-9. *Model saving block*

The model saving consists of two parts – saving the developed model to disk and reusing the pre-trained models from the repository.

After you have trained the model to your desired accuracy level, you save it to disk. In TensorFlow 1.x, there were many ways of saving the model. The TF 2.x standardized the model saving to an abstraction called SavedModel. The saved model can be directly loaded into your ML application, or it can be uploaded on production servers for serving. TensorFlow 2.x models are saved to a standardized format so as to enable it to be deployed on mobiles, embedded devices, and the Web. TensorFlow provides API to deploy once developed models to these different platforms, including the Web with the support of JavaScript and Node.js.

The TensorFlow Hub as it is called is a repository of many pre-trained models. You use transfer learning to reuse and extend these models to meet your requirements. Using pre-trained models gives you the advantage of training your model with a smaller dataset and a quicker training. You will find models for text and image recognition, models trained on Google News dataset, and even modules for Progressive GAN and Google Landmarks Deep Local Features. Unfortunately, as of this writing, most of these models are written for TensorFlow 1.x and require porting to the newer version. Check the TensorFlow site for model updates, and hopefully while you are reading this, most of the models are updated to TensorFlow 2.x.

Next, you will look at the deployment options.

Deployment

The trained model can be deployed on various platforms as shown in Figure 1-10.

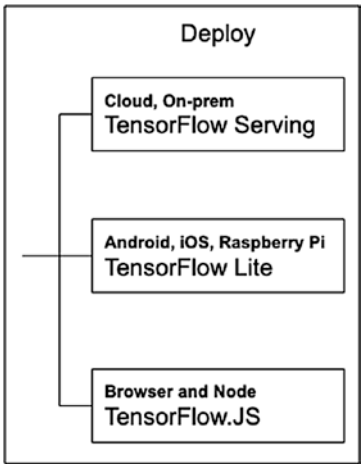


Figure 1-10. *Model deployment options*

The best part of TensorFlow 2.x is that you will be able to deploy the trained model on cloud or on premises. Not only that, using TensorFlow 2.0, you'll be able to deploy the model even on mobile devices such as Android and iOS and also on an embedded device like Raspberry Pi. You will also be able to deploy your model on the Web using Node.js. This will allow you to use the model in your favorite browsers. In general, the deployment may be categorized as follows:

- **TensorFlow Serving** – A library that allows models to be served over HTTP/REST or gRPC/Protocol buffers.
- **TensorFlow Lite** – A lightweight solution for deploying models on Android, iOS, and embedded systems like Raspberry Pi and Edge TPUs.
- **TensorFlow.js** – Enables model deployment in JavaScript environments like web browsers or server side through Node.js. Using TensorFlow.js, you will also be able to define models in JavaScript and train those directly in web browsers using Keras-like API.

With this small introduction to TensorFlow, I will now briefly describe some of the major salient features of TensorFlow 2.x.

What TensorFlow 2.x Offers?

TensorFlow 2.x has introduced many new features compared to its earlier versions. I will briefly summarize the salient features of TensorFlow 2.x. As you read through this book, you will understand their use in a better way.

The tf.keras in TensorFlow

The Keras API is now available through TensorFlow's tf.keras API. This is a high-level API that provides support for TensorFlow-specific functionalities such as eager execution, data pipelines, and estimators. With tf.keras you can build and train models just the way you were doing using Keras, without sacrificing flexibility and performance.

To use tf.keras in your program, you will use the following code:

```
import tensorflow as tf
from tensorflow import keras
```

Once TensorFlow libraries are loaded, you will be able to define your own neural network architectures, create models, and train and test them. You will learn this in Chapter 2 when I discuss the Hello World program of TensorFlow 2.x.

Eager Execution

Prior to TensorFlow 2.x, your machine learning code was divided into two parts:

1. Building the computational graph
2. Creating a session to execute the graph

These steps can be explained using the following code:

```
import tensorflow as tf
a = 2
b = 3
c = tf.add(a, b, name='Add')
print(c)
```

This prints the following on your console:

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

It essentially builds a graph, which can be visualized as shown in Figure 1-11.

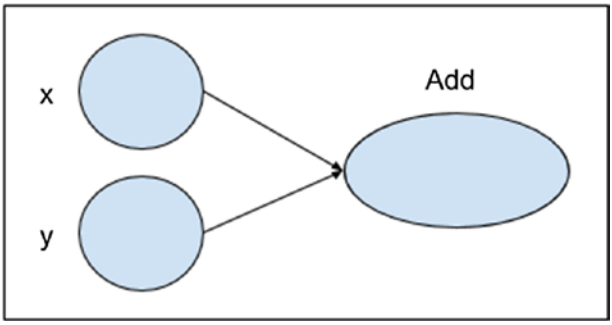


Figure 1-11. *Computational graph*

To run the graph itself, you need to create a session as follows and run it to execute your add function:

```
sess = tf.Session()
print(sess.run(c))
sess.close()
```

When you run the preceding code, the result 5 will be printed on your console.

With TensorFlow 2.x, you can perform the same operation without creating a session. The following code illustrates how this is done:

```
import tensorflow as tf
a = 2
b = 3
c = tf.add(a, b, name='Add')
print(c)
```

The output of executing the preceding code would be

```
tf.Tensor(5, shape=(), dtype=int32)
```

Note that the output tensor value is 5.

Thus, session creation is totally eliminated. This helps a lot in building big models. Typically, during development, a small error may pop up somewhere in the beginning of the model, requiring you to build the entire computational graph one more time. Every time you fix a bug, you need to build the full graph. This causes a lot of inconvenience and is a very time-consuming process. In TensorFlow 2.x, the eager execution allows you to run the partial code without the need of building the full computational graph. This eager execution is implemented by default so that you do not have to take any special considerations while defining the model. You may run the command tf.executing_eagerly() to convince yourself.

With the elimination of session creation, the TensorFlow code can now be run like a Python code. TF 2.x creates what's known as a dynamic computation graph rather than the static computational graph created in TF 1.x.

Distribution

The most expensive operation in ML model building is training the model. TensorFlow 2.x now provides an API called tf.distribute.Strategy to distribute the training across multiple GPUs and TPUs. With this API, you will be able to distribute your existing models and training code with only minimal code changes. The API provides six distribution strategies:

1. MirroredStrategy
2. CentralStorageStrategy
3. MultiWorkerMirroredStrategy
4. TPUStrategy
5. ParameterServerStrategy
6. OneDeviceStrategy

You may dig further into the documentation to learn more about these strategies.

As the tf.distribute.Strategy is integrated into tf.keras, it inherently takes advantage of improved speed during training and also while inferencing.

TensorBoard

TensorBoard is a visualization tool that helps in your experimentation on ML model development. With TensorBoard, some of the things that you can do are

- Visualizing loss and accuracy metrics
- Visualizing model graph

- Viewing histograms of weights, biases, and so on
- Displaying images
- Profiling programs

A typical screenshot of the TensorBoard that displays the accuracy and loss metrics is shown in Figure 1-12.

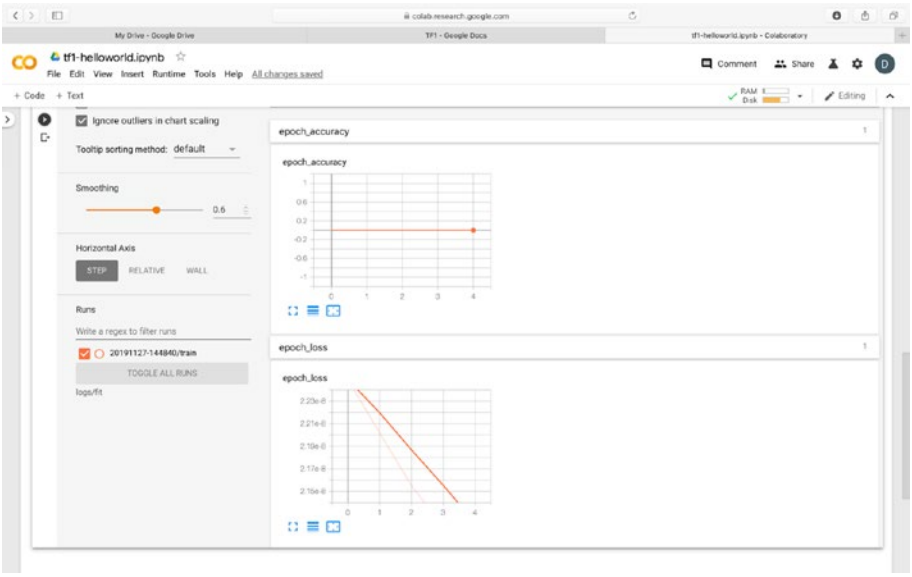


Figure 1-12. *TensorBoard metrics display*

With TensorBoard, you can visualize the model training directly within your Jupyter environment. It provides several useful and exciting features such memory profiling, viewing confusion matrix and conceptual model graph, and so on. It is a tool that allows you to do measurements and view the results in the machine learning workflow.

Vision Kit

With TensorFlow 2.x, you will now be able to design your own IoT devices which are capable of image recognition. These IoT devices will be powered by TensorFlow's machine learning models. With Google AIY Vision Kit, you will be able to build your own intelligent camera that can see and recognize objects. You have an option of creating your own recognition model or using a pre-trained model for this intelligent camera. The entire kit fits in a small handy cardboard cube and is powered by a Raspberry Pi. The kit provides everything that you need to build your own intelligent camera.

Voice Kit

Just the way you build an intelligent camera that provides vision to your IoT devices, with Voice Kit you will be able to provide the listening and answering abilities to your IoT devices. With the help of Google AIY Voice Kit, you will be able to create your own natural language processor that can connect to Google Assistant or the Speech-to-Text service on the cloud. With this, you will be able to issue voice commands to your IoT device or even ask questions and get answers to it. Like Vision Kit, this too fits in a handy cardboard cube and is powered by a Raspberry Pi. The kit includes everything that you need to build an audio-capable IoT device including the Raspberry Pi.

Edge TPU

If you are an IoT device manufacturer, you will be happy to prototype your new ML models on the device itself. Coral has created the Edge TPU board for this very purpose. This is a development board to quickly prototype on-device ML products. It is a single-board computer with a removable system-on-module (SOM). The SOM contains eMMC, SOC, wireless radios, and the Edge TPU. This is also ideally suited for fast on-device ML inferencing required by IoT devices and other embedded systems.

Pre-trained Models for AIY Kits

There are several pre-trained models available for your use on AIY kits. Some of these are listed as follows:

- Face detector
- Dog/cat/human detector
- Dish classifier for identifying food
- Image classifier
- Nature explorer for recognizing birds, insects, and plants

If you develop your own model, you are welcome to submit it to Google for inclusion in the preceding list of pre-trained models displayed on the Google site.

Data Pipelines

As we have seen, with TensorFlow 2.0, the training can be distributed across GPUs and TPUs considerably reducing the time required to execute a single training step. This also calls for providing efficient data transfer between two steps. The new tf.data API helps in building flexible and efficient input pipelines across a variety of models and accelerators. You will use data pipelines in Chapter 2.

Installation

TensorFlow 2.x can be installed on the following platforms:

- macOS 10.12.6 or later
- Ubuntu 16.04 or later

- Windows 7 or later
- Raspbian 9.0 or later

I personally use Mac for my development. All the programs given in this tutorial are developed and tested on Mac.

The installation of TensorFlow is trivial. It requires a pip version >19.0. You can ensure that the latest pip is available on your machine by running the following command in your console window:

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

To install a CPU-only version of TensorFlow, you would run the following command:

```
pip install tensorflow
```

To install a CPU/GPU version, you will use the following command:

```
pip install tensorflow-gpu
```

Installation

To install TensorFlow on Mac, you must have Xcode 9.2 or later – command-line tool available on your machine. The pip package has few dependencies which are installed using the following commands on the command-line tool:

```
pip install -U --user pip six numpy wheel setuptools mock
'future>=0.17.1'
pip install -U --user keras_applications --no-deps
pip install -U --user keras_preprocessing --no-deps
```

Once you have these dependencies installed, you can run pip install to install whatever the version of TensorFlow you want to use.

Docker Installation

If you do not want to do the jugglery of installing the TensorFlow yourself, you can take advantage of a ready-to-use image in a Docker container. To download the Docker image, use the following command:

```
docker pull tensorflow/tensorflow
```

After the Docker container is successfully downloaded, run the following command to start a Jupyter notebook server:

```
docker run -it -p 8888:8888 tensorflow/tensorflow
```

Once the Jupyter environment starts, open a notebook and use TensorFlow as you wish. This is explained next under the section "Testing."

No Installation

So far, I have shown you the ways of installing TensorFlow on a few platforms. Using a Docker image saves you from researching the dependencies. There is yet another easy way to learn and use TensorFlow – and that is using Google Colab. This requires no installation to use TensorFlow. You simply start Google Colab in your browser. Google Colab is a Google research project that essentially provides you a Jupyter notebook environment in a browser. No setup is required and your entire notebook code runs in the cloud. You will be using Google Colab for running the programs in this tutorial.

Testing

As we are going to use Google Colab for the projects in this book, I will show you how to test your TensorFlow installation in Colab. Start Colab by opening the URL – `http://colab.research.google.com`. Assuming that you are logged in to your Google account, you will see the screenshot in Figure 1-13.

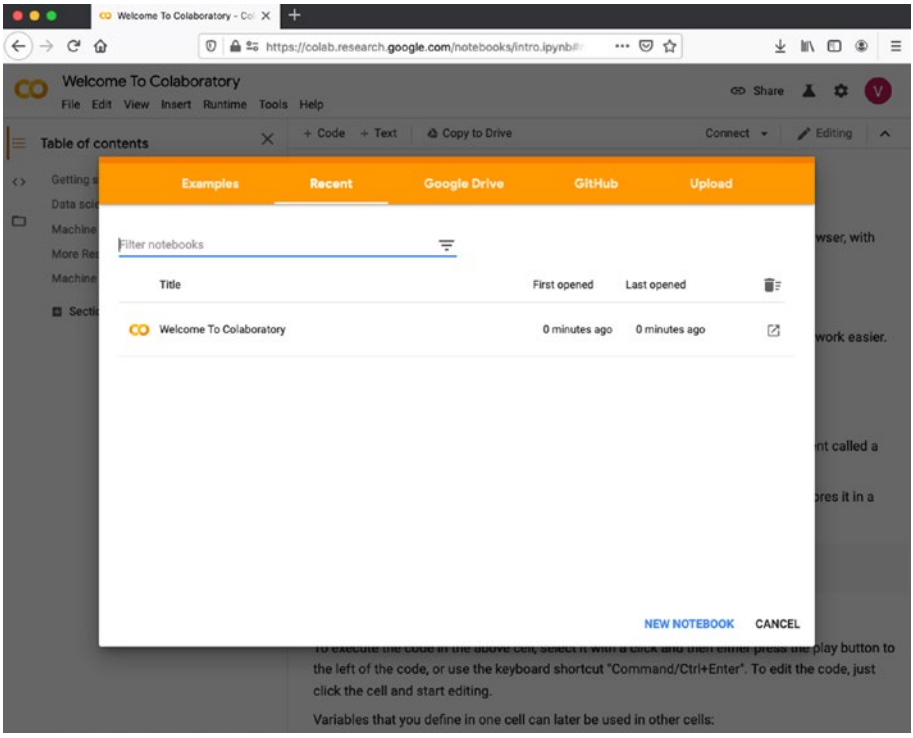


Figure 1-13. *Colab opening a new notebook*

Select the NEW PYTHON3 NOTEBOOK menu. A blank notebook would open in your browser. Type the following two program statements in the code window:

```
%tensorflow_version 2.x
import tensorflow as tf
```

The %tensorflow_version is called the Colab magic that loads TF 2.x instead of the default TF 1.x. The use of this magic is explained later in Chapter 2 when I discuss the TF Hello World application.

Note In the current version of Colab, the use of the %tensorflow_ version is no more required. So this statement is redundant and I have removed this in all subsequent chapters.

Run the code cell and you will see the output as shown in Figure 1-14.

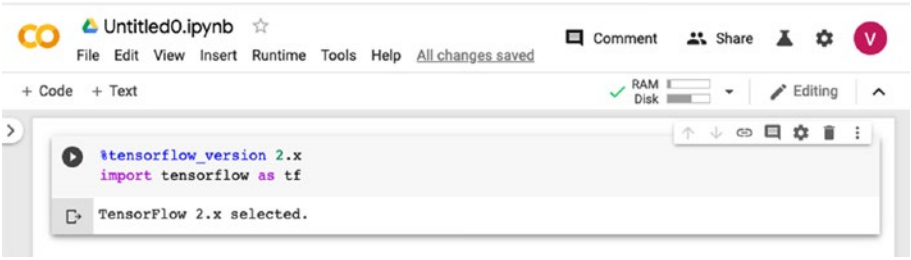


Figure 1-14. *Testing the Colab setup*

The message says that TensorFlow 2.x is selected and is thus available for use in your notebook.

Now add one code cell to your project and write the following code into it:

```
c = tf.constant([[2.0, 3.0], [1.0, 4.0]])
d = tf.constant([[1.0, 2.0], [0.0, 1.0]])
e = tf.matmul(c, d)
print (e)
```

When you run the code, you should see the following output:

```
tf.Tensor(
[[2. 7.]
[1. 6.]], shape=(2, 2), dtype=float32)
```

If you see the preceding output, you are now all set for using TensorFlow 2.x in Colab environment. Note that, unlike the TF 1.x, no sessions are created in the preceding code.

This completes our installation and setup of TensorFlow 2.x.

Summary

TensorFlow 2.x provides a very powerful platform for developing deep machine learning applications. The platform facilitates you right from the data preparation and model building to the final deployment on production servers. It is like using one tool for an end-to-end development. Keras, the popular machine development library, is now fully integrated in TF. Taking advantage of the new features in TF such as eager execution, distribution of training, and inferencing across multiple CPUs, GPUs, TPUs, and efficient data pipelines, you will be able to develop very efficient ML applications in Keras. During development, the TensorBoard provides a useful analysis for you to optimize your model. A fully trained model is saved to a format that can be deployed even on mobiles and embedded devices. TF also provides Vision and Voice Kits for you to deploy your image/video recognition and voice-controlled ML models on embedded devices. There are several pre-trained models contributed by the community that are available for your use. The use of Edge TPU allows you to do inferencing on the device itself. In summary, TF 2.x can be considered as a single platform for machine learning – right from development to deployment.

Toward the end of the chapter, I covered TF installation on a few platforms. I also discussed the use of Google Colab, which provides a cloud-based Python project development environment, for developing TF applications. If you have a good Internet connectivity, which is not a rare commodity in most parts of the world these days, you can rely on Google Colab for all your machine learning applications, and that's what this book does.

In the next chapter, you will start with the actual coding in TF 2.x.

CHAPTER 2

A Closer Look at TensorFlow

In the previous chapter, you saw the capabilities of the TensorFlow platform. Having seen a glimpse of TensorFlow powers, it is time now to start learning how to harness this power into your own real-world applications.

We will start with a trivial application that will teach you the intricacies of a simple ML application development.

A Trivial Machine Learning Application

To get you started on TensorFlow coding, we will start with a trivial Hello World kind of application. In this trivial application, you will develop a machine learning model that does the predictions using statistical regression techniques.

In this application, we will use a fixed set of data points declared within the program code itself. Our data will consist of (x, y) coordinate values. We compute a value called z that has some linear relationship with x and y. For example, the value of z for a given x and y values may be computed using the following mathematical equation:

```
z = 7 * x + 6 * y + 5
```


P. Sarang, *Artificial Neural Networks with TensorFlow 2*,
https://doi.org/10.1007/978-1-4842-6150-7_2

Our task is to make the machine learn on its own to find the best fit for the preceding relationship given a sufficiently large number of x and y values and the corresponding target z values. Once the model is trained, we will use this model to predict **z** for any unseen x and y values. For example, given x equals 2 and y equals 3, the model should predict an output z equal to 37. If it predicts 37 and likewise if it predicts z with 100% accuracy for any not previously known x and y, we say that the model is fully trained with 100% accuracy. Practically, it is never possible to develop a model that predicts with 100% accuracy. So we try to optimize the model performance to achieve this idealistic accuracy level of 100%.

As you can see from the preceding discussion, the problem that we are trying to solve is a classical linear regression case study. To keep things simple, we will create a single-layer network consisting of only one neuron, which is trained to solve a linear regression problem. In practice, your network will always consist of multiple layers with multiple nodes. In this trivial application, I will avoid the use of such deep networks as defining those requires a deeper understanding of Keras API. You will be exposed to those Keras APIs later in this chapter. For this trivial application and all subsequent applications in this book, you will use Google Colab for developments.

Creating Colab Notebook

In this first application, I will guide you through the entire process of creating, testing, and inferring a ML model development in Colab. This is a bit of a detailed explanation of the development process for the benefit of the readers who are new to ML development.

Start Google Colab in your browser by typing the following URL:

```
http://colab.research.google.com
```

You will see the screen shown in Figure 2-1.

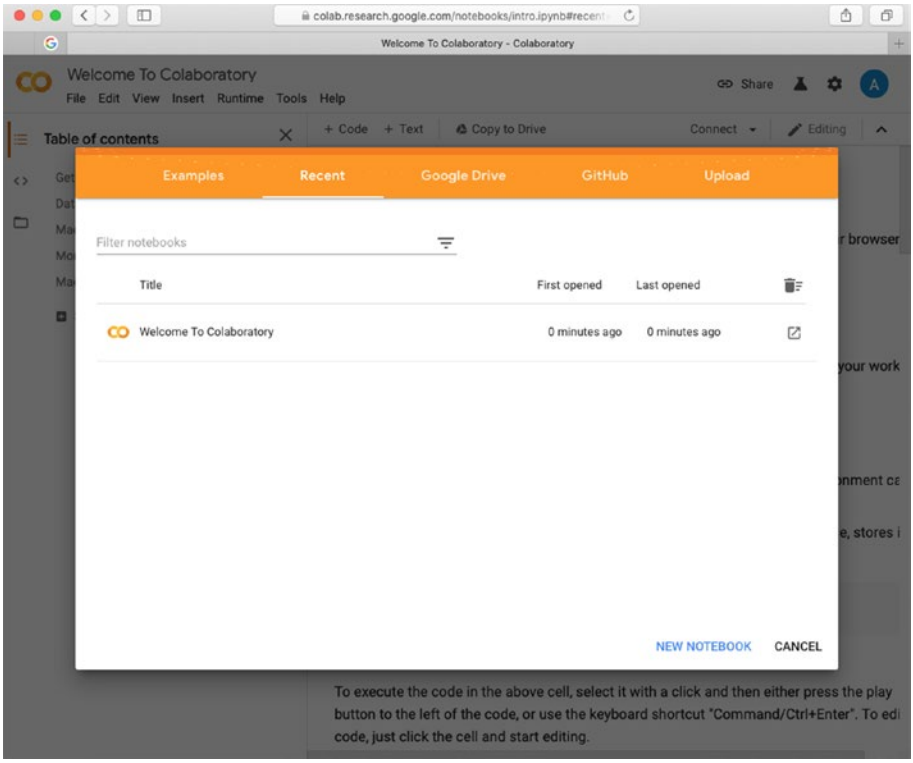


Figure 2-1. *Creating a new Colab notebook*

Select NEW PYTHON3 NOTEBOOK option to open a new Python 3 notebook. Assuming that you are logged in to your Google account, you would see a screen as shown in Figure 2-2.

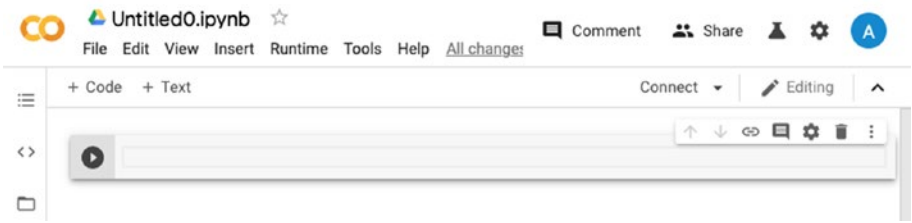


Figure 2-2. *New Colab notebook*

The default name for the notebook starts with Untitledxxx.ipnyb. Change the name to Hello World or whatever you prefer. Next, you will write code to import TensorFlow libraries in your Python code.

Imports

Our trivial program will require three imports – TensorFlow 2.x, numpy library for handling our data, and matplotlib to do some charting.

Importing TensorFlow 2.x

To import TensorFlow in your Python notebook, you would use the following program statement:

```
import tensorflow as tf
```

This imports the default version, which is currently 1.x (at the time of this writing). The output of executing the preceding command is shown in Figure 2-3.

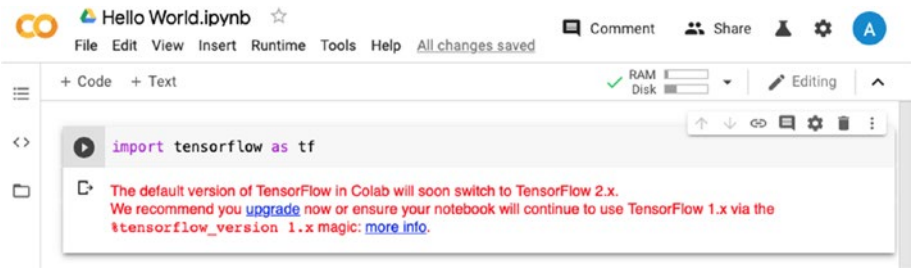


Figure 2-3. *Default TensorFlow library import*

As this book is based on TensorFlow 2.x, we need to import it explicitly. To do so, you must run a tensorflow_version magic. Magic is a feature of Colab and is run using the following statement:

```
%tensorflow_version 2.x
```

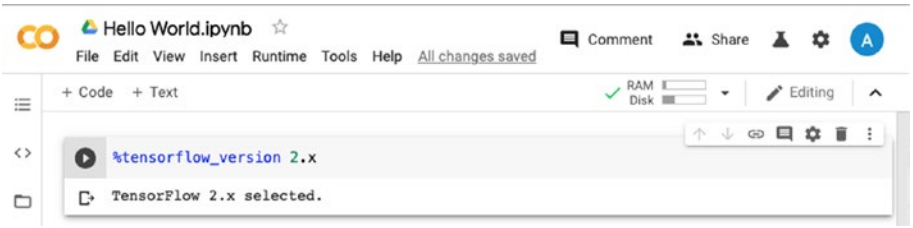


Figure 2-4. *Loading TensorFlow 2.x*

When you run the code, TensorFlow 2.x will be selected. The output is shown in Figure 2-4.

After the TensorFlow 2.x is selected, you would import TensorFlow libraries using the traditional import statement as follows:

```
import tensorflow as tf
```

Note The use of magic is no more required in the current version of Colab.

Keras library is now part of TensorFlow. To use Keras in our application, we need to import it from TensorFlow. This is done using the following import statement:

```
from tensorflow import keras
```

To use Keras modules, you now use tf.keras syntax. Next, you will import other required libraries.

Importing numpy

NumPy is a library for supporting large, multidimensional arrays in Python. It has a collection of high-level mathematical functions to operate on these arrays. Any machine learning model development relies heavily on the use of arrays. You will be using numpy arrays to store the input data required by our network.

To import numpy, you use the following import statement:

```
import numpy as np
```

The matplotlib is a Python library for creating quality 2D plots. You will use this library in our project for plotting accuracy and error metrics.

To import matplotlib, you use the following statement:

```
import matplotlib.pyplot as plt
```

This completes our imports for the application. Next, you will be creating data for our application.

Setting Up Data

We will create a set of 100 data points consisting of x and y coordinates. The count for data points is declared in the Python variable using the statement:

```
number_of_datapoints = 100
```

To generate x and y coordinates, you use the random module in numpy. To generate x values, you use the following program statement:

```
# generate random x values in the range -5 to +5
x = np.random.uniform(low = -5 , high = 5 ,
size = (number_of_datapoints, 1))
```

The low and high parameters in the uniform function define the lower and upper bounds for the random number generator. The size parameter specifies the dimensions of the array, that is, how many values are to be generated. The return value of the preceding program statement is an array consisting of 100 rows and 1 column. You can print the first five values of the generated array using the statement:

```
x[:5,:].round(2)
```

In the output, each value is truncated to two decimal digits by calling the round function. A sample output of the execution of this statement is shown here:

```
array([[ 4.57],
       [-0.68],
       [ 2.64],
       [-3.17],
       [-4.86]])
```

Note that the output varies on every run. Likewise, the y values are generated using a similar statement as shown here:

```
y = np.random.uniform(-5 , 5 , size = (number_of_datapoints , 1))
```

We set up the relation between x and y using a linear equation:

```
z = 7 * x + 6 * y + 5
```

In machine learning terms, the x and y are the features and z is the label. Once our model is trained, we will ask the model to predict z for the given x and y. I have mentioned it earlier that we will be training our network to discover the relationship between x and y. For this, we need to introduce some noise in every value of z that we compute using the

preceding equation. You generate the noise using the random function as earlier with the values ranging from –1 to +1 using the following statement:

```
noise = np.random.uniform(low =-1 , high =1,
        size = (number_of_datapoints, 1))
```

Now, you create the z array using the linear equation and adding noise to it as shown in this statement:

```
z = 7 * x + 6 * y + 5 + noise
```

The input to our neural network is a single dimensional array consisting of 100 rows with each row consisting of another single dimensional array having x and y values as columns. To create the required input data format, you use the column_stack function as follows:

```
input = np.column_stack((x,y))
```

Printing the first five values of the input array produces the following output:

```
array([[-1.9 ,  2.91],
       [-2.14, -0.81],
       [ 4.18,  1.79],
       [-0.93, -4.41],
       [-1.8 , -1.31]])
```

At this point, you are ready with the data for training the network. Our next task is to create a network itself.

Defining Neural Network

As mentioned earlier, our neural network will consist of a single neuron that accepts a single dimensional vector and outputs a single value. The network is depicted in Figure 2-5.

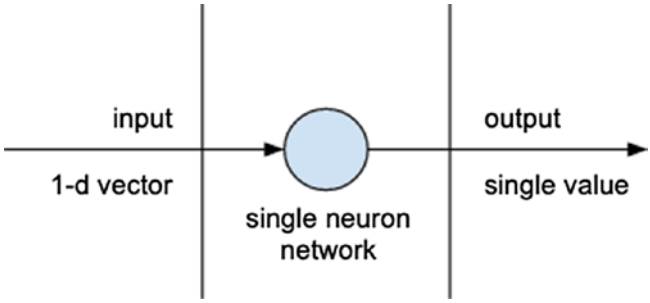


Figure 2-5. *A single layer/single node network*

To define the network models, Keras provides a Sequential API. Using this API, you will be able to construct multilevel sophisticated network architectures. In our current requirement, we need to create an ANN architecture consisting of a single layer with a single neuron. You define the model using the following statement:

```
model = tf.keras.Sequential([keras.layers.Dense(units=1,
input_shape=[1])])
```

The units parameter defines the dimensionality of the output space. Here, by specifying a value of 1, you define a single layer network with a single neuron outputting a single value. The Dense function takes several parameters that allows you to create complex ANN architectures. You would create lots of complex architectures throughout this book using the keras.Sequential API.

After the model is defined, we need to compile it and make it ready for training on our dataset.

Compiling Model

To train a model, we first need to define a learning process. The model compilation is a way of setting up its learning process. The learning process itself consists of a few components:

- Objective loss function
- Optimizer
- Metrics

Firstly, it uses some loss function to determine how far the inference is from the target value. The model tries to minimize the loss during its training. Keras provides several predefined loss functions, to name a few, categorical_crossentropy, mean_squared_error, huber_loss, and poisson. Secondly, to help in reducing the losses, we use an optimizer. An optimizer is an algorithm or a method used to change the attributes of a neural network. The attributes are the weights and the learning rate. By changing these attributes, we try to minimize the losses. Keras provides several predefined optimizers, again to name a few, SGD (stochastic gradient descent), RMSprop, Adagrad, and Adam. Lastly, we define a metric function that is used to judge the performance of the model – to name a few predefined metric functions, MSE (mean squared error), RMSE (root mean squared error), MAE (mean absolute error), and MAPE (mean absolute percentage error).

So we set up the learning process by calling the compile function on the model. The compile takes the abovesaid three parameters as its arguments. The following code segment illustrates the use of compile:

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

Here we specify the stochastic gradient descent as the optimizer, the mean squared error as the loss function, and again the mean squared error as the metrics.

Now, as the model knows the learning process, it is time to feed some training data to it.

Training Network

The model is trained in several iterations. Initially, we assign some preset weights to the various nodes in the network. After the first training iteration, we look at the losses and then adjust these weights for the second iteration where we try to minimize the losses. The process continues through several iterations; we call it epochs. At the end of each epoch, we save and monitor the losses to ensure that we are optimizing the network in the right direction. To save this state, we need to create a history object in Keras, which is done using the following code segment:

```
from tensorflow.keras.callbacks import History
history = History()
```

We will pass the preceding history object to the model training method as a parameter. The training itself is done by calling the model's fit method as shown here:

```
model.fit(input, z , epochs = 15 , verbose = 1,
            validation_split = 0.2, callbacks = [history])
```

The first parameter specifies the stacked input that we have created earlier. The second parameter specifies the target values. The epochs parameter defines the number of iterations. The verbose parameter specifies if you want to observe the training progress. The validation_split parameter specifies that 20% of the given data would be used for validating

the trained model. Lastly, the callbacks parameter specifies where the intermediate monitoring data would be stored. It specifies the callback function which is called at the end of each epoch. The partial output during training is shown in Figure 2-6.

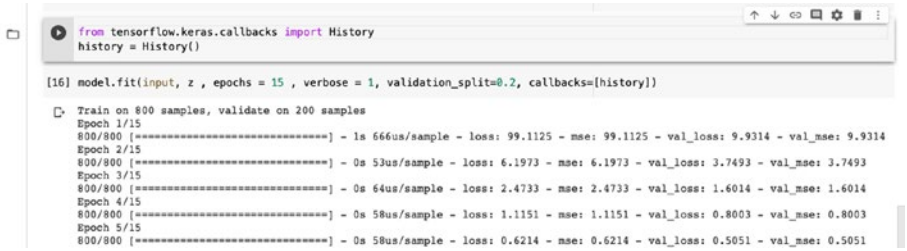


Figure 2-6. *Output during model training*

Once all epochs are completed, the model is supposedly trained. We need to verify that the model is indeed trained to meet our needs. To do so, we will examine the training output by observing the metrics that we have asked the model to create during its learning.

Examining Training Output

We have asked the model to save the status at each epoch in a history variable. We can examine what is recorded in history with the help of the following statement:

```
print(history.history.keys())
```

The output of this print statement would be

```
dict_keys(['loss', 'mse', 'val_loss', 'val_mse'])
```

We see that at each epoch the model has saved the loss and the mse (mean squared error) on both the training and the validation data. The validation loss and mse are indicated with the prefix val_. We will now create a plot for both loss and mse. To print the loss on both the training and validation data, use the following code segment:

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title(Accuracy')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```

Here, we plot the loss and val_loss key values from the recorded history. The output is shown in Figure 2-7.

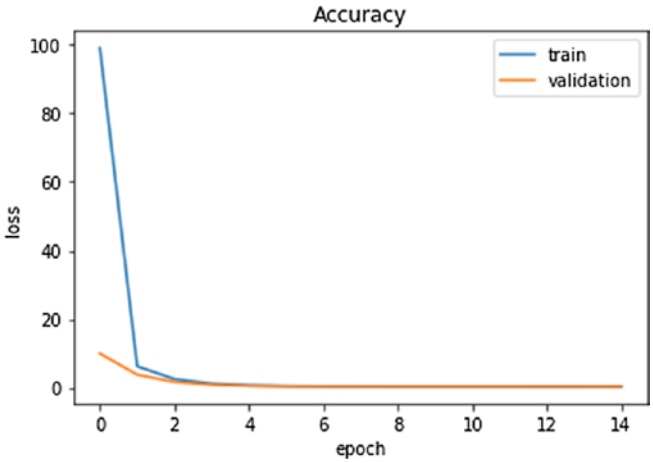


Figure 2-7. *The loss vs. epoch*

We observe from Figure 2-7 that the loss is minimized quite early by the end of the third epoch, and the model is fully trained by the end of 15 epochs that we have specified in the fit method. We will also print the mean squared error to further verify this claim. To plot the mse, use the following code segment:

```
plt.plot(history.history['mse'])
plt.plot(history.history['val_mse'])
plt.title('mean squared error')
plt.ylabel('mse')
plt.xlabel('epochs')
plt.legend(['train' , 'validation'] , loc = 'upper right')
plt.show()
```

The output of the execution of the preceding code is shown in Figure 2-8.

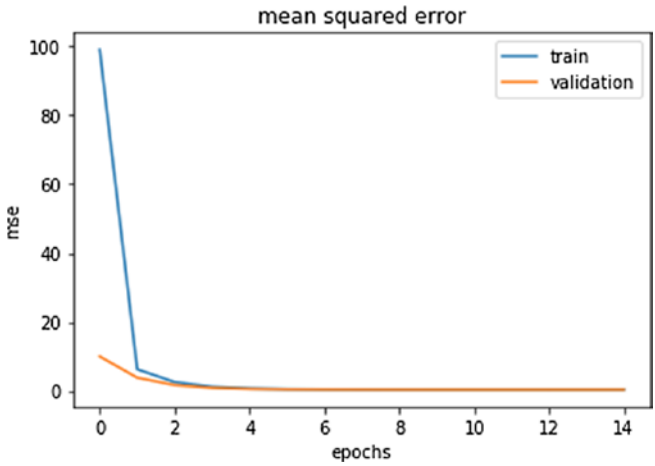


Figure 2-8. *mse vs. epoch*

Lastly, we will also plot the predicted output vs. the real output using the following code segment:

```
plt.plot(np.squeeze(model.predict_on_batch(input)),
np.squeeze(z))
plt.xlabel('predicted output')
plt.ylabel('real output')
plt.show()
```

The program output is shown in Figure 2-9.

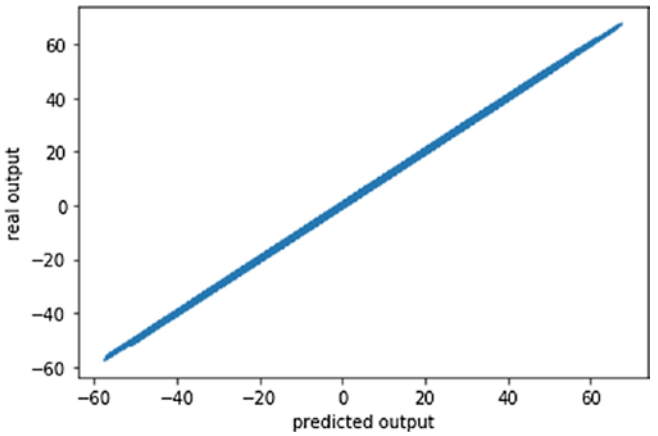


Figure 2-9. *Predicted vs. real output values*

As you observe in Figure 2-9, the output predicted by the model is very close to the expected output. So, we can certainly assume that the model is well trained. To pass the test, we need to test the model's prediction on unseen data, which is done next.

Predicting

To make a prediction on an unseen x and y values, you will use the predict function on the trained model. This is shown in the following program statement:

```
print("Predicted z for x=2, y=3 ---> ",
            model.predict([[2,3]]).round(2))
```

Here, we specify x equals 2 and y equals 3. The result is rounded to two decimal digits. When you execute the preceding print statement, you would see the following output:

```
Predicted z for x=2, y=3 ---> [[36.99]]
```

Now, let us check whether the prediction is close enough to the expected output. To see the expected output, run the following code:

```
# Checking from equation
# z = 7*x + 6*y + 5
print("Expected output: ", 7*2 + 6*3 + 5)
```

The execution prints 37 on the screen. Our model's prediction is 36.99, which is close enough to the expected value. Note that if you run the code, the predicted output would vary on each run because the model's accuracy varies each time. You can test the model's predictions with a few more x and y values to satisfy yourself on the model's training.

Full Source Code

The full source code for our trivial Hello World application described earlier is given in Listing 2-1 for your quick reference.

Listing 2-1. A trivial linear regression application source

```
# Load TensorFlow 2.x in a Colab project.
%tensorflow_version 2.x

# Import required libraries

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# Set up data
number_of_datapoints = 1000
# generate random x values in the range -5 to +5
x = np.random.uniform(low = -5 , high = 5 , size = (number_of_
datapoints, 1))
# generate random y values in the range -5 to +5
y = np.random.uniform(-5 , 5 , size = (number_of_datapoints , 1))
# generate some random error in the range -1 to +1
noise = np.random.uniform(low =-1 , high =1, size = (number_of_
datapoints, 1))
z = 7 * x + 6 * y + 5 + noise

# Print x, y and z sample values for manual verification
x[:5,:].round(2)

y[:2,:].round(2)

z[:2,:].round(2)

# Stack x and y arrays for inputting to neural network
input = np.column_stack((x,y))

# Print few values of input array for demonstration purpose.
input[:2,:].round(2)
```

```
# Create a Keras sequential model consisting of single layer
with a single neuron.
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1)])

# Compile the model with the spcified optimizier, loss function
and error metrics.
model.compile(optimizer = 'sgd' , loss = 'mean_squared_error' ,
metrics = ['mse'] )

# Import History module to record loss and accuracy on each
epoch during training
from tensorflow.keras.callbacks import History
history = History()

model.fit(input, z , epochs = 15 , verbose = 1, validation_
split=0.2, callbacks=[history])

# Print keys in the history just to know their names. These
will be used for plotting the metrics.
print(history.history.keys())

# Plot the loss metric on both training and validation
datasets.
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Accuracy')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()

#Plot the mean squared error on both training and validation
datasets.
plt.plot(history.history['mse'])
```

```
plt.plot(history.history['val_mse'])
plt.title('mean squared error')
plt.ylabel('mse')
plt.xlabel('epochs')
plt.legend(['train' , 'validation'] , loc = 'upper right')
plt.show()

plt.plot(np.squeeze(model.predict_on_batch(input)),
np.squeeze(z))
plt.xlabel('predicted output')
plt.ylabel('real output')
plt.show()

print("Predicted z for x=2, y=3 ---> ", model.predict([[2,3]]).
round(2))


# Checking from equation
# z = 7*x + 6*y + 5
print("Expected output: ", 7*2 + 6*3 + 5)
```

If you have run this trivial project and are getting the preceding output, congratulations! Your setup for deep learning with TensorFlow 2.x is now complete. You will now dive deep into real-world machine learning development. In the next section, you will learn a real-world machine learning development life cycle. You will be using a real dataset, learn how to preprocess it and make it ready for feeding into a neural network, define a multilevel deep neural network, train it, test the model, and plot the accuracy metrics to refine the model. Not only this, you will learn how to use TensorBoard in Colab environment to visualize the metrics for doing analysis on the model training.

So let us start on this real-world project based on TensorFlow.

Binary Classification in TensorFlow

In the previous example, we used the built-in dataset for training our model. Now, we will use a realistic dataset to do some real-world machine learning. You will be solving a classification problem. We will use a dataset from the popular Kaggle site used in one of their challenges. The dataset contains the customer data of a bank. Consider now that the bank has approached you to develop a ML prediction model which provides them some insights on the likelihood of the customer leaving the bank. In the financial terms, this is called churning. Having the knowledge that a certain customer may leave the bank in the near future, the bank can take some preventive measures to retain the customer.

The problem that you are trying to solve is to develop a binary classification model. You will use TensorFlow's deep learning library and will make use of Keras high-level API for implementing the model. More specifically, you will learn the following in this example:

- How to load a CSV data from a local or a remote server?
- How to preprocess it and make it ready for ML algorithm?
- How to define a multilayer ANN using TensorFlow's high-level Keras API?
- How to train the model?
- Evaluating the model's performance on test data.
- Visualizing the results on TensorBoard.
- Doing performance analysis.
- Inferring on unseen data.

So let us start with the project development.

Setting Up Project

Create a new Colab project and name it Binary Classification. The project uses the bank customer data (Churn_Modelling.csv) which is available in the book's source download. Copy the downloaded file to your Google Drive in a folder of your choice. Note that you will need to later on set up the file path appropriately in your program code. If you do not wish to download the data file, you will still be able to run the project taking the data from the GitHub setup for this book. You are now ready to code your project.

Imports

As in the earlier example, include the following imports in the code window:

```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
```

You will be using pandas dataframes for loading external database. You will use sklearn library for preprocessing the data and creating the training/validation datasets. You will use the matplotlib for some charting. To use these libraries, include the following imports in your project code:

```
#loading data
import pandas as pd
#scaling feature values
from sklearn.preprocessing import StandardScaler
#encoding target values
from sklearn.preprocessing import LabelEncoder
#shuffling data
from sklearn.utils import shuffle
#splitting the dataset into training and validation
```

```
from sklearn.model_selection import train_test_split
#plotting curves
import matplotlib.pyplot as plt
```

Next, you need to mount the drive in your program so that the program can access the documents stored in Google Drive.

Mounting Google Drive

To mount Google Drive, type the following code in a new code cell:

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

When you run this code, it will ask you to enter the authorization code to access your drive. You will see the screen shown in Figure 2-10.

Figure 2-10. *Authorization code for Google Drive*

Click the provided link. You will be asked to sign in to your Google account. You will see the authorization code similar to that shown in Figure 2-11.

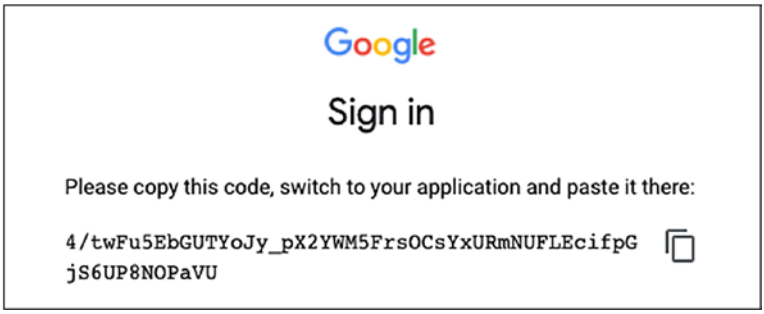


Figure 2-11. *Google sign-in authorization*

Click the icon next to the authorization code to copy it to the clipboard. Paste the code in the earlier shown authorization window. After a successful authorization, you will see this message on your screen:

```
Mounted at /content/drive
```

Now, you are ready to access the contents of your drive through your program code.

Loading Data

To load the data, enter the following program code in a new code cell and execute it:

```
data = pd.read_csv('/content/drive/
            <path to downloaded CSV>/Churn_Modelling.csv')
```

Note you will need to set up the appropriate path to your csv file.

If you have decided to use the data from the book's GitHub, use the following code instead of the preceding code segment:

```
data_url = 'https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch02/Churn_Modelling.csv'
data=pd.read_csv(data_url)
```

The read_csv function loads the data from the specified file and copies it in a pandas dataframe.

Shuffling Data

The data that is collected on field may be in a specific order as per the convenience and the comfort of the data collector. For better machine learning, you should randomize the data so that the learning does not follow the undesired patterns in the data. So, we shuffle the data using the following statement:

```
data=shuffle(data)
```

Examining Data

You can verify that the data is correctly loaded by printing the contents of the data dataframe. Instead of printing the top rows by calling data.head(), I have printed the full dataset so that you will know the number of records and columns present in it. This is shown in Figure 2-12.

```
# Examine loaded data
data
```

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
8940	8941	15658148	Udokamma	657	France	Male	38	7	0.00	2	1	0	185827.74	0
3001	3002	15675949	Fleming	696	Spain	Female	43	4	0.00	2	1	1	66406.37	0
7971	7972	15756848	Edmondson	633	Spain	Male	42	10	0.00	1	0	1	79408.17	0
2701	2702	15797010	Shen	649	France	Female	31	2	0.00	2	1	0	15200.61	0
3682	3683	15572626	Mackenzie	620	Spain	Male	44	8	0.00	2	1	1	15627.51	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1603	1604	15593470	Tu	576	Germany	Female	36	8	156287.85	1	1	1	23305.85	0
9727	9728	15807728	Ferri	530	France	Female	45	1	0.00	1	0	1	190663.89	1
3990	3991	15778752	Johnson	708	France	Male	32	10	86614.06	2	1	1	172129.26	0
6488	6489	15598097	Johnstone	550	France	Male	44	9	0.00	2	1	0	26257.01	0
8028	8029	15775124	Watterston	763	Spain	Male	37	8	0.00	2	1	1	933.38	0

10000 rows × 14 columns

Figure 2-12. *Dataset*

There are 10,000 rows and 14 columns in the database. A brief explanation of the various fields is given here:

- **RowNumber** – Numbers from 1 to 10,000.
- **CustomerId** – A unique identification for the customer.
- **Surname** – Customer's last name.
- **CreditScore** – Customer's credit score.
- **Geography** – Customer's country.
- **Gender** – Male or female.
- **Age** – Customer's age.
- **Tenure** – How long the customer is banking with them?
- **Balance** – Customer's bank balance.
- **NumOfProducts** – The count of bank products the customer is currently using.
- **HasCrCard** – Does the customer hold a credit card?
- **IsActiveMember** – Is the customer currently active?
- **EstimatedSalary** – Customer's current estimated salary.
- **Exited** – A value of 1 indicates that the customer has left the bank.

Now, as you have loaded the data in memory, your next task is to cleanse it before feeding it to our network. We call it data preprocessing, which is discussed next.

Data Preprocessing

The raw real data may not always meet the training requirements of a neural network (ANN). Specifically, you will be checking out data and processing it for the following items:

- Data may contain null values.
- All fields in the database may not be useful for learning.
- The numerical fields may exhibit large variance in their values and thus must be scaled to the same level.
- Certain fields may contain categorical values, for example, male and female; these need to be encoded to 0s and 1s.
- Finally, you will need to decide which fields are to be used as features and what the labels are.

So, let us start processing the data.

Checking Nulls

If the data contains null values, it will severely affect the network training. The easiest way to check for a null value is to call the isnull function. This is done using the following program statement:

```
data.isnull().sum()
```

The output of the preceding statement gives the following result:

```
RowNumber          0
CustomerId         0
Surname            0
CreditScore        0
Geography          0
```

```
Gender              0
Age                 0
Tenure              0
Balance             0
NumOfProducts       0
HasCrCard           0
IsActiveMember      0
EstimatedSalary     0
Exited              0
dtype: int64
```

Computing the sum on a null value generates an error. Very clearly, our dataset does not contain any null values. So there is no question of removing rows from the dataset (the ones containing null fields).

Now comes the major task of selecting fields for machine learning.

Selecting Features and Labels

Not all the fields in our database would be useful for training the algorithm. For example, fields such as CustomerId and Surname do not make any sense to our machine learning. So, we need to drop these columns. This is done using the following statement:

```
X = data.drop(labels=['CustomerId', 'Surname',
              'RowNumber', 'Exited'], axis = 1)
```

Note that we dropped four fields from the dataset. X is a new array containing the fields which are relevant to our model building.

Whether the customer exited the bank marks the output of our model. Thus, the field Exited becomes the label for our model building. This is extracted into the variable y using the following statement:

```
y = data['Exited']
```

At this point, you are ready with features (X) and labels (y) tensors.

Encoding Categorical Columns

You must check if any of the selected columns have categorical values. For this, we will check the data types of all the selected columns using the following statement:

```
X.dtypes
```

The output of the preceding statement is shown as follows:

```
CreditScore          int64
Geography           object
Gender              object
Age                  int64
Tenure               int64
Balance            float64
NumOfProducts        int64
HasCrCard            int64
IsActiveMember       int64
EstimatedSalary    float64
Dtype: object
```

Note that Geography and Gender are object types. You can check the values they contain by printing the first five rows of the features tensor as shown in Figure 2-13.

```
# Examine few records for finding values in categorical columns
X.head()
```

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary
8940	657	France	Male	38	7	0.0	2	1	0	185827.74
3001	696	Spain	Female	43	4	0.0	2	1	1	66406.37
7971	633	Spain	Male	42	10	0.0	1	0	1	79408.17
2701	649	France	Female	31	2	0.0	2	1	0	15200.61
3682	620	Spain	Male	44	8	0.0	2	1	1	15627.51

Figure 2-13. *Top five rows of the features vector*

The Gender takes two categorical values, Male and Female, while Geography has three categorical values – Germany, Spain, and France. You will need to convert these into numerical values before feeding it to the network. The encoding is done by using the LabelEncoder in sklearn's preprocessing module. This is shown in the following code snippet:

```
from sklearn.preprocessing import LabelEncoder
label = LabelEncoder()
X['Geography'] = label.fit_transform(X['Geography'])
X['Gender'] = label.fit_transform(X['Gender'])
```

The one-hot encoding creates dummy variables for categorical columns. As Geography column has three distinct values, it will create three variables – one for each country. Thus, we will have three features pertaining to the country in our training dataset. Having many features increases the training time. To reduce the number of features, you can exclude one of the dummy variables pertaining to the country field and yet achieve the same results. We drop the first variable by calling get_dummies method:

```
X = pd.get_dummies(X, drop_first=True, columns=['Geography'])
```

If you examine the data at this point by printing the top five rows, you will notice that there are only two columns for Geography – Geography_1 and Geography_2.

One more thing that we need to do before feeding data to the network is to scale all the numerical values in the range –1 to 1.

Scaling Numerical Values

As the features in the real data can have a wide range of data values, machine learning would work better if we standardized all these data points to the same scale. Ideally, the mean for each column should be 0, and the standard deviation should be 1 for better results on machine learning. So we transform all our data points using the equation:

```
z = (x - mu) / s
```

where mu is mean and s is standard deviation. This standardization is performed by using the StandardScaler function of sklearn as shown in this code snippet:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

At this point, our preprocessing of data is completed. We are now ready to define and train the model. When we train the model, we also need to validate our training. If the results of the training do not meet our expectations, we will need to further preprocess the data – like adjusting the number of features and so on. For testing, we will reserve a part of the data. Thus, we split our entire preprocessed data – a larger portion used for training and the smaller portion for testing the trained model.

Creating Training and Testing Datasets

To split the data into two portions, we use the train_test_split method of sklearn as in the following program statement:

```
# Split dataset into training and testing
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size = 0.3)
```

The test_size parameter determines what percentage of data should be reserved for testing. The function returns a set of vectors for training and testing.

Many times, the terms validation and test datasets are used interchangeably. To avoid further confusions, I am giving widely accepted unambiguous definitions of these terms:

- **Training dataset** – The part of data that is used for model fitting.
- **Validation dataset** – The part of data used for tuning hyperparameters during training.
- **Test dataset** – The part of data used for evaluating a model's performance after its training.

Now, it is time to define our network.

Defining ANN

After preprocessing, we have 11 features in our dataset. The number of features is determined by computing the shape of the training dataset with the following statement:

```
X_train.shape[1]
```

The expected output of the network is a binary value indicating the likelihood of the customer leaving the bank. The target values are specified in the y_train vector.

You will create a four-layer deep learning network model. In the first layer, you will use 128 nodes, the second one will have 64, the third one will have 32, and the fourth will be a single output node. To create the network, you use tf.keras API, which is a new standard in TensorFlow. You use Sequential API to create a linear stack of layers. You instantiate the model using the following statement:

```
model = keras.models.Sequential()
```

You add the first layer to the stack consisting of 128 nodes using the following statement:

```
model.add(keras.layers.Dense(128, activation = 'relu',
                                input_dim = X_train.shape[1]))
```

The input dimension to this layer is set in the parameter input_dim, which is the number of features defined by the shape of X_train vector. We use ReLU (rectified linear unit) as the activation function. The activation function is used in deciding whether the node is to be activated depending on its weighted sum. ReLU is the most widely used activation function that outputs 0 for negative inputs and 1 otherwise.

Likewise, you add the second layer to the network using the following statement:

```
model.add(keras.layers.Dense(64, activation = 'relu'))
```

The input to this layer comes from the previous layer so there is no need to specify the dimensions of the input vector. The third layer is added using the statement:

```
model.add(keras.layers.Dense(32, activation = 'relu'))
```

Finally, the last layer in the network is added using the statement:

```
model.add(keras.layers.Dense(1, activation = 'sigmoid'))
```

We use sigmoid as the activation function here as this layer is outputting a binary value. A sigmoid function is a type of activation function and is also known as a squashing function. A squashing function limits the output to a range between 0 and 1, making it suitable in predicting probabilities.

You print the network summary by calling the summary function as follows:

```
model.summary()
```

The summary as printed on the screen is shown here:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 128)               1536
_________________________________________________________________
dense_1 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_2 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 33
=================================================================
Total params: 11,905
Trainable params: 11,905
Non-trainable params: 0
_________________________________________________________________
```

Compiling Model

After the model architecture is defined, it needs to be compiled. To compile the model, you call the model's compile method:

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

As the model that we are developing is a binary classifier, we use the binary_crossentropy as our loss function. We use Adam optimizer while training the model as this is suited best in such situations. Later on after the training, if you are not satisfied with the model's performance, you may experiment with other optimizers. The accuracy metrics are collected for analysis by specifying the value for metrics parameter.

I will also show you how to use TensorBoard for analyzing the network performance. For this, we need to define a callback function which will be called at each epoch during training. We will be collecting the logs in the log folder. To clear the earlier log, we use the following action:

```
!rm -rf ./log/
```

You define the callback function using the following code snippet:

```
#tensorboard visualization
import datetime, os
logdir = os.path.join("log",
                      datetime.datetime.now().
                      strftime("%Y%m%d-%H%M%S"))
tensorboard_callback = tf.keras.callbacks.TensorBoard(logdir,
                      histogram_freq = 1)
```

With this setup for training analysis and the compilation of the model, we are now ready to start the training.

Model Training

To train the model, you use the fit method on the model instance:

```
r = model.fit(X_train, y_train, batch_size = 32, epochs = 50,
               validation_data = (X_test, y_test),
               callbacks = [tensorboard_callback])
```

The first parameter to the fit function defines the features vector and the second defines the labels. The batch_size parameter as the name suggests defines the batch size for the training. The epochs parameter determines how many iterations would be performed during training. The test data that we generated during data preprocessing is used for

model validation and is passed to the fit function in the validation_data parameter. Lastly, the callbacks parameter specifies which callback function would be called at the end of each iteration. The partial output during training is shown in Figure 2-14.

```
r = model.fit(X_train, y_train, batch_size = 32, epochs = 50, validation_data = (X_test, y_test), callbacks = [tensorboard_callback])

Train on 7000 samples, validate on 3000 samples
Epoch 1/50
7000/7000 [==============================] - 2s 218us/sample - loss: 0.4244 - accuracy: 0.8219 - val_loss: 0.3801 - val_accuracy: 0.8507
Epoch 2/50
7000/7000 [==============================] - 1s 99us/sample - loss: 0.3514 - accuracy: 0.8541 - val_loss: 0.3689 - val_accuracy: 0.8557
Epoch 3/50
7000/7000 [==============================] - 1s 97us/sample - loss: 0.3352 - accuracy: 0.8620 - val_loss: 0.3668 - val_accuracy: 0.8500
Epoch 4/50
7000/7000 [==============================] - 1s 95us/sample - loss: 0.3304 - accuracy: 0.8600 - val_loss: 0.3618 - val_accuracy: 0.8573
Epoch 5/50
7000/7000 [==============================] - 1s 98us/sample - loss: 0.3242 - accuracy: 0.8640 - val_loss: 0.3616 - val_accuracy: 0.8570
Epoch 6/50
7000/7000 [==============================] - 1s 98us/sample - loss: 0.3178 - accuracy: 0.8677 - val_loss: 0.3779 - val_accuracy: 0.8410
```

Figure 2-14. *Program output during training*

Once the training is over, you can use the collected metrics to evaluate if the model is trained to your desired accuracy.

Performance Evaluation

To evaluate the performance, we will launch the TensorBoard in our Colab environment using the %tensorboard magic. Before this, we need to load the tensorboard using %load_ext magic.

```
%load_ext tensorboard
%tensorboard --logdir log #command to launch tensorboard on
colab
```

Running this magic runs the TensorBoard, and you will see the accuracy and loss metrics plotted on the screen. The accuracy and loss metrics plot is shown in Figure 2-15.

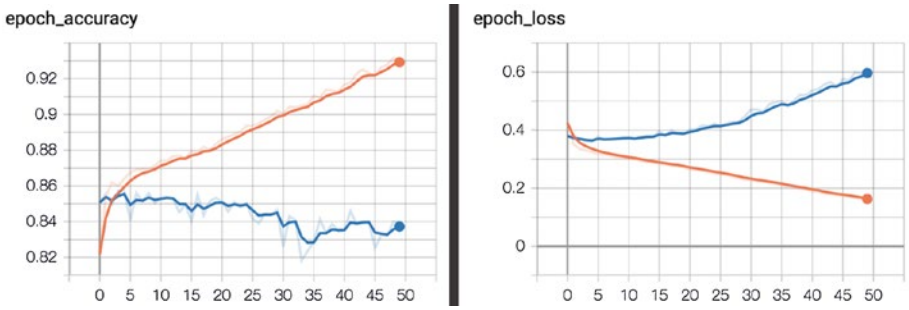


Figure 2-15. *Accuracy and loss metrics in TensorBoard*

The two curves shown here are plotted on the training and the validation data. The examination of accuracy and loss metrics helps you in determining if the model is performing well. The plots basically show you the model's accuracy and the loss at every epoch. If the accuracy is improving on every epoch, your training is in the right direction. Similarly, the loss should keep on reducing on every epoch. These plots can easily detect issues such as overfitting and so on. If you are not satisfied with the performance, you may adjust the model parameters and retrain it to improve the accuracy. You may try a different optimizer and/or introduce regularization to improve the model accuracy.

You can also evaluate the model's performance on the test data by calling the evaluate method and passing the test features and labels vectors as parameters. The program statement for evaluating the model and its output is shown here:

```
test_scores = model.evaluate(X_test, y_test)
print('Test Loss: ', test_scores[0])
print('Test accuracy: ', test_scores[1] * 100)
Test Loss:  0.6143370634714762
Test accuracy:  83.96666646003723
```

The accuracy on the test data which is about 83% indicates that the model will correctly classify 83% of the given data points.

I will also show you how to plot the performance charts on the validation data using the matplotlib – a traditional way of performance evaluation. Use the following code snippet to do so:

```
%matplotlib inline
import matplotlib.pyplot as plt #for plotting curves

plt.plot(r.history['val_accuracy'], label='val_acc')
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()
plt.show()
```

The plot generated by matplotlib is shown in Figure 2-16.

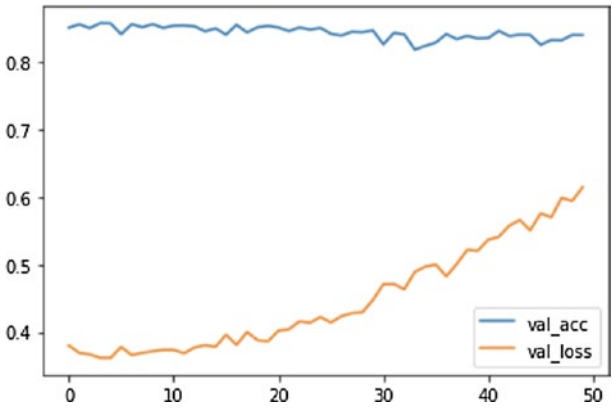


Figure 2-16. *Accuracy/loss matrix on the validation data*

Besides the accuracy and loss metrics, we also used the confusion matrix quite often to evaluate the performance of our network, which is discussed next.

Predicting on Test Data

The confusion matrix requires both the predictions and the True labels. Thus, we first need to generate predictions on our test data. For predicting, use predict_classes method as shown here:

```
y_pred = model.predict_classes(X_test)
```

The method takes the features vector as its argument and returns a tensor of predictions. You may print the predictions on the console. The result is shown as follows:

```
y_pred

array([[1],
       [0],
       [0],
       ...,
       [1],
       [0],
       [0]], dtype=int32)
```

Here the value of 1 at any index value indicates that the customer is going to leave the bank, and the value of 0 indicates that the bank has retained the customer.

You use these prediction results to create and plot a confusion matrix that provides a better visualization of the model's performance.

Confusion Matrix

I will first show you how to generate a confusion matrix and then show you how to interpret the matrix plot. To generate the confusion matrix, you use the sklearn's built-in function as shown here:

```
from sklearn.metrics import confusion_matrix
cf = confusion_matrix(y_test, y_pred)
cf
```

This gives the following output:

```
array([[2175,  209],
       [ 283,  333]])
```

You will plot this matrix to give you a visual effect using the following code:

```
from mlxtend.plotting import plot_confusion_matrix
plot_confusion_matrix(conf_mat = cf, cmap = plt.cm.cmapname)
```

The plot is shown in Figure 2-17.

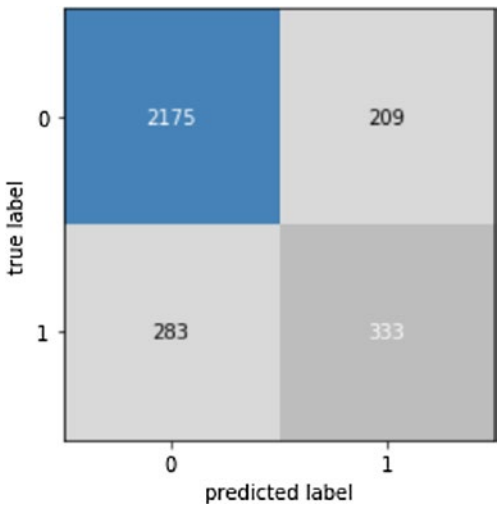


Figure 2-17. *Confusion matrix*

The x-axis represents the predicted labels, and the y-axis represents the True labels. As the chart shows, there are 2175 True positives and 333 True negatives. A True positive indicates that the customers would leave the bank, and they have been correctly classified by our model. Likewise, a True negative indicates that these customers will not leave the bank, and they have been correctly classified. The True positive and True negative help us in determining the accuracy of our model.

The sklearn defines accuracy_score function to compute the accuracy score which is calculated by adding the number of True positives and True negatives and then dividing the sum by the total number of predictions. The following program statement computes the accuracy score for our model:

```
from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_pred)
0.8396666666666667
```

The execution of this statement gives the accuracy score of 83.63%, which is a largely accepted accuracy in machine learning.

As the model is trained to our satisfaction, it is now time to use it on unseen data.

Predicting on Unseen Data

To create an unseen data for our use case, we need to know the data types of all the features to which we will assign some dummy values.

The head dump of our features shows us the column names and the range of values it holds. A partial dump of our features vector is shown in Figure 2-18.

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary
8940	657	France	Male	38	7	0.0	2	1	0	185827.74
3001	696	Spain	Female	43	4	0.0	2	1	1	66406.37

Figure 2-18. *Screenshot of features data*

So, for our unseen test data, we will use the following values:

CreditScore = 615

Gender = Male

Age = 22

Tenure = m5

Balance = 20000

NumOfProducts = 1

HasCrCard = 1

IsActiveMember = 1

EstimatedSalary = 60000

Geography = Spain

You will input this data to our trained model by calling its predict method with the preceding values at appropriate indexes in the parameters list.

```
customer = model.predict([[615, 1, 22, 5, 20000, 5, 1, 1,
60000, 0, 0]])
customer
```

```
if customer[0] == 1:
  print ("Customer is likely to leave")
else:
  print ("Customer will stay")
```

The execution of the preceding code segment gives this output:

```
Customer will stay
```

The value of 0 indicates that this customer is unlikely to leave the bank. Note that the accuracy of this prediction is still about 83% as computed earlier.

After the model is fully trained to your satisfaction, you may save it to disk and deploy it on your production server for real-life use. How this is done is explained in the next chapter when I discuss the tf.keras implementation in depth.

Full Source Code

The complete source code for the project is given in Listing 2-2 for your quick reference.

Listing 2-2. Binary classification full source

```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
import pandas as pd

# Load data from Github
data_url = 'https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch02/Churn_Modelling.csv'
data=pd.read_csv(data_url)

# Shuffle data for taking care of patterns in data collection
from sklearn.utils import shuffle
data=shuffle(data) #shuffling the data

# Examine loaded data
data

# Check for null values
data.isnull().sum()

# Drop irrelevant columns to set up features vector
X = data.drop(labels=['CustomerId', 'Surname', 'RowNumber',
'Exited'], axis = 1)
```

```
# Set up labels vector
y = data['Exited']

# Check data types for finding categorical columns
X.dtypes

# Examine few records for finding values in categorical columns
X.head()

# Encode categorical columns
from sklearn.preprocessing import LabelEncoder
label = LabelEncoder()
X['Geography'] = label.fit_transform(X['Geography'])
X['Gender'] = label.fit_transform(X['Gender'])

# Drop the first column of Geography to reduce the number of
features
X = pd.get_dummies(X, drop_first=True, columns=['Geography'])
X.head()

# Scale all data points to -1 to + 1
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Split dataset into training and validation
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_
size = 0.3)

# Determine number of features
X_train.shape[1]

# Create a stacked layers sequential network
```

```
model = keras.models.Sequential() # Create linear stack of
layers
model.add(keras.layers.Dense(128, activation = 'relu', input_
dim = X_train.shape[1])) # Dense fully connected layer
model.add(keras.layers.Dense(64, activation = 'relu'))
model.add(keras.layers.Dense(32, activation = 'relu'))
model.add(keras.layers.Dense(1, activation = 'sigmoid')) #
activation sigmoid for a single output

# Print model summary
model.summary()

# Compile model with desired loss function, optimizer and
evaluation metrics
model.compile(loss = 'binary_crossentropy', optimizer='adam',
metrics=['accuracy'])

#to clear any other logs if present so that graphs won't
overlap with previous saved logs in tensorboard
!rm -rf ./log/

#tensorboard visualization
import datetime, os
logdir = os.path.join("log", datetime.datetime.now().
strftime("%Y%m%d-%H%M%S"))
tensorboard_callback = tf.keras.callbacks.TensorBoard(logdir,
histogram_freq = 1)

# Perform training
r = model.fit(X_train, y_train, batch_size = 32, epochs = 50,
validation_data = (X_test, y_test), callbacks = [tensorboard_
callback])
```

```
# Load tensorboard in Colab
%load_ext tensorboard
%tensorboard --logdir log #command to launch tensorboard on
colab

# evaluate model performance on test data
test_scores = model.evaluate(X_test, y_test)
print('Test Loss: ', test_scores[0])
print('Test accuracy: ', test_scores[1] * 100)

# Plot metrics in matplotlib
%matplotlib inline
import matplotlib.pyplot as plt #for plotting curves

plt.plot(r.history['val_accuracy'], label='val_acc')
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend()
plt.show()

# Predict on test data
y_pred = model.predict_classes(X_test)
y_pred

# Create confusion matrix
from sklearn.metrics import confusion_matrix
cf = confusion_matrix(y_test, y_pred)
cf

# Plot confusion matrix
from mlxtend.plotting import plot_confusion_matrix
plot_confusion_matrix(conf_mat = cf, cmap = plt.cm.cmapname)

# Compute accuracy score
from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_pred)
```

```
# Predict on unseen customer data
customer = model.predict([[615, 1, 22, 5, 20000, 5, 1, 1,
60000, 0, 0]])
customer
```

```
if customer[0] == 1:
  print ("Customer is likely to leave")
else:
  print ("Customer will stay")
```

Summary

In this chapter, you set up your environment for deep learning using TensorFlow 2.x. You used Colab for developing your Python notebooks. A trivial application helped you learn the development environment. This was followed by a more detailed realistic example. In this realistic example, you loaded an external database and learned how to preprocess the data for making it suitable for machine learning, how to define a deep neural network, how to compile it, how to train it, how to evaluate the model's performance using TensorBoard and plots from matplotlib, and finally how to use the trained model to make predictions on unseen data. Though the model's performance is evaluated, I never talked about how to improve it. In the next chapter, you will learn a few techniques for improving a model's performance. The next chapter covers the TensorFlow Keras integration in more depth and discusses the image classification problem.

CHAPTER 3

Deep Dive in tf.keras

Keras is a high-level neural networks API that runs on top of TensorFlow. Last many years, you were using Keras API with TensorFlow running at the backend. With TensorFlow 2.x, this has changed. TensorFlow has now integrated Keras in tf.keras API. The tf.keras is the TensorFlow's implementation of the Keras API specification. This change is mainly done to bring consistency in using Keras with TF. It also resulted in taking advantage of several TensorFlow features such as eager execution, distributed training and others while using Keras. The latest Keras release as of this writing is 2.3.0. This release adds support for TensorFlow 2.x and is also the last major release of multi-backend Keras. Henceforth, you will be using only tf.keras in all your deep learning applications. You have already used tf.keras in Chapter 2 while getting started on TensorFlow. This chapter will take you deeper in the use of tf.keras.

Getting Started

In creating deep learning applications, the most important task is to define a neural network model. In the previous chapter, while developing a trivial application you created a network consisting of a single layer with a single neuron. Just to recapitulate, this was done using the following program statement:

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1)])
```


P. Sarang, *Artificial Neural Networks with TensorFlow 2*,
https://doi.org/10.1007/978-1-4842-6150-7_3

Though I did not explicitly mention, the above statement uses Keras functional API in TensorFlow. The statement creates a single layer/singe neuron network model. In earlier Keras implementations, you could do the same thing using following two statements:

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

The model is visualized in the screenshot shown in Figure 3-1.

Figure 3-1. *Single layer/single neuron model*

The use of functional API is now the recommended way of defining Keras deep learning models.

Functional API for Model Building

With functional API, you would be able to create extremely complex models with non-linear topology. You will be able to share layers within a model. And you will also be able to create models with multiple inputs and outputs. So, in all the applications in this book, I will be using functional API to define ANN models. I will now show you how to define few architectures using functional API.

Sequential Models

Suppose you want to build a model visualized in the Figure 3-2.

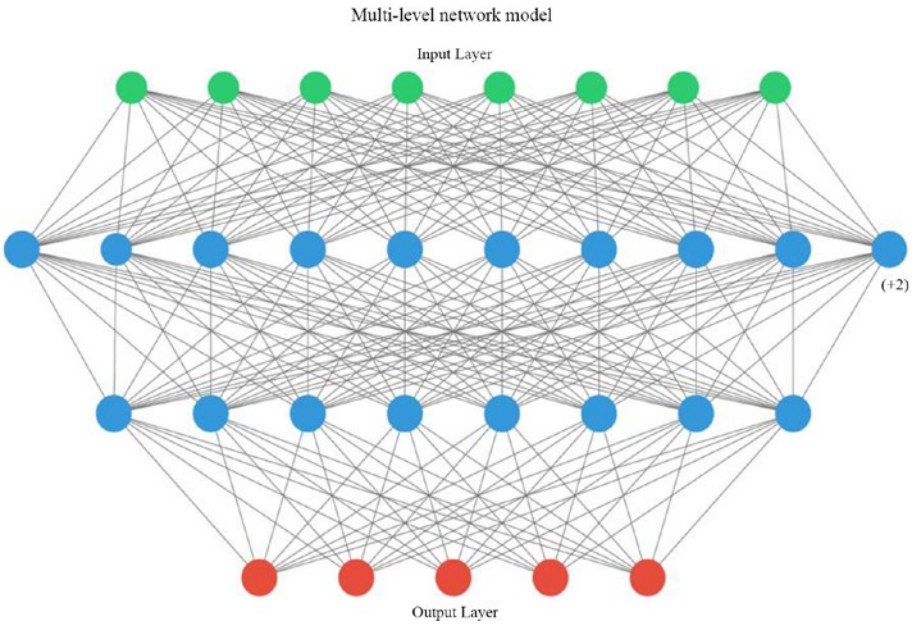


Figure 3-2. *Multilayer model*

As you see in the Figure 3-2, the desired network consists of multiple layers. The Functional API provides a set of tools for building a graph of layers.

Assuming you have the requisite imports as shown in this code:

```
import tensorflow as tf
from tensorflow import keras
```

For creating the model shown in Figure 3-2, first we need to create an input layer, which is done using the following code statement:

```
inputs = keras.Input(shape=(8,), name='image')
```

The above statement returns an inputs tensor of size 8. Next, we will add a Dense layer with 12 nodes using the following statement:

```
x = layers.Dense(12, activation='relu')(inputs)
```

Note that we have passed the inputs tensor as input to the newly added layer. The new layer itself returns a tensor of size 12, which can be input to the next layer. We will now add one more layer having 8 nodes using the following statement:

```
x = layers.Dense(8, activation='relu')(x)
```

Once again, notice that the tensor from the previous layer is input to the new layer. Likewise, you will be able to add any number of layers to the network, each having its own set of nodes and the activation functions. Lastly, we will add the output layer to our network using the following statement:

```
outputs = layers.Dense(5)(x)
```

The output of this layer is tensor of size 5. Thus, we expect our network to output one of the five values given a specific input.

The model can now be defined by using these inputs and outputs as shown in the following statement:

```
model = keras.Model(inputs=inputs, outputs=outputs,
       name='multilayer model')
```

The network plot can be generated using the following statement:

```
keras.utils.plot_model(model, 'multilayer_model.png',
       how_shapes=True)
```

The generated network plot is shown in the Figure 3-3.

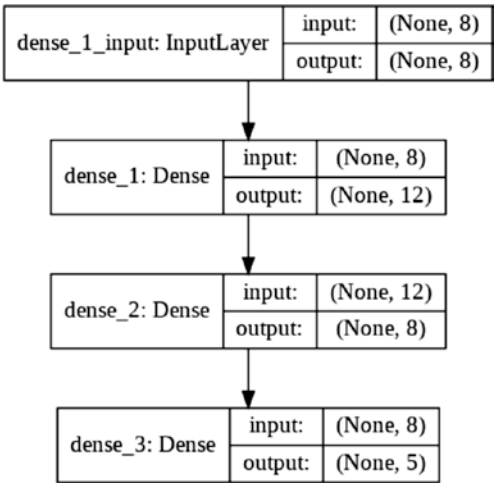
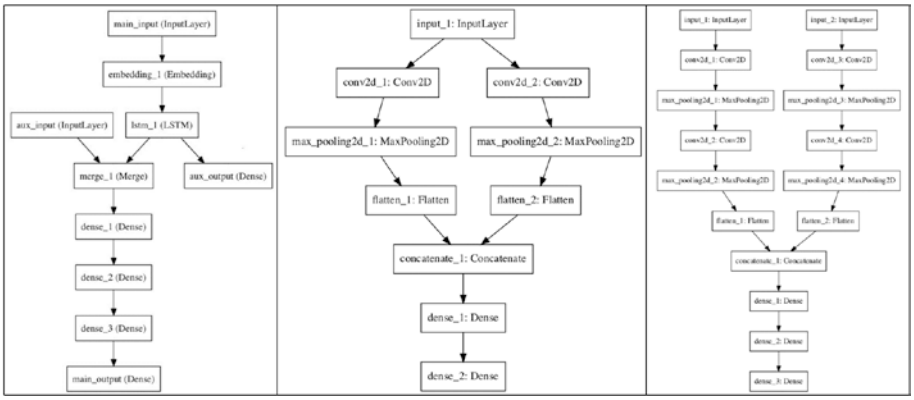


Figure 3-3. *Network plot for multilayer network*

The functional API also allows you to create complex architectures with multiple inputs and outputs, with shared layers, and so on. A few such complex architectures are shown in Table 3-1.

Table 3-1. *A few complex ANN architectures*



Model Subclassing

If you are more of an Object Oriented programmer coming from the background of say Java/C++, you may be wondering if there is a concept of subclassing that will enable you to reuse your previously created models. TensorFlow supports model subclassing. To create your custom class for model definition, you inherit from tf.keras.Model

```
class MyModel(tf.keras.Model):
```

You will need to provide two overridden methods in the class definition – init and call. The init as the name suggests is called during the class instantiation. A typical definition of the init is given in the code fragment below:

```
  def __init__(self, use_dp = False, num_output = 1):
    super(MyModel, self).__init__()
    self.use_dp = use_dp
    self.dense1 = tf.keras.layers.Dense(12, activation=tf.
    nn.relu)
    self.dense2 = tf.keras.layers.Dense(24, activation=tf.
    nn.relu)
    self.dense3 = tf.keras.layers.Dense(4, activation=tf.
    nn.relu)
    self.dense4 = tf.keras.layers.Dense(10, activation=tf.
    nn.sigmoid)

    if self.use_dp:
      use_dp = tf.keras.layers.Dropout(0.3)
```

In the constructor, we define variables for different types of layers that we will be using for constructing our network. All layers here are of type Dense – fully connected layer. The first layer consists of 12 nodes, the second one consists of 24 nodes, the third one 4 nodes and the last one 10 nodes.

The first three layers use ReLU as their activation function, while the last one uses sigmoid. We also define a dropout in case if we decide to add a dropout layer on any of the above Dense layers. The dropout percentage is 30% as indicated by the dropout parameter value of 0.3.

Next, you define call method used during object creation. A typical call method is shown in the code segment below:

```
def call(self, x):
  x = self.dense1(x)
  x = self.dense2(x)
  if self.use_dp:
    x = self.dp(x)
  x = self.dense3(x)
  if self.use_dp:
    x = self.dp(x)
  return self.dense4(x)
```

The network consists of four layers, the output of each one becomes the input to the next one. At the end, we return dense4 which gives us 10 classes. The second and third layers use dropout. This completes our subclassing of the model. Next, we will instantiate this model.

To instantiate the model in your program code, you would use the following program statement:

```
model = MyModel()
```

Once the model object is created, you can call its usual compile method to compile it by passing the desired set of parameters as shown in the code below:

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

Generally, if you love object-oriented programming, you would use model subclassing. Otherwise, there is no specific need to do subclassing, the functional API would meet all your requirements of creating complex architectures.

Predefined Layers

To create network architectures, tf.keras provides Sequential API to which you keep adding network layers. There are several predefined layers for your ready use. The list is quite exhaustive. Each layer is defined as a class that you instantiate in the code and add its instance to your model. Here is a partial list of commonly used predefined layers:

- Dense – Densely-connected layer
- Conv2D – 2D convolutional layer
- InputLayer – An entry point into the network
- LSTM – Long Short-Term memory layer
- RNN – Base class for your custom recurrent layers

The list is really exhaustive and you will find classes such as Dropout, Flatten, LayerNormalization, Multiply, and so on. In addition to these pre-defined classes, you can define your own custom layers.

Custom Layers

The custom class inherits from tf.keras.layers.Layer. You need to override four functions, init, build, call and compute_output_shape. A typical custom class definition is shown in the code below:

```
class MyLayer(tf.keras.layers.Layer):

  def __init__(self, output_dim, ** kwargs):
      self.output_dim = output_dim
```

```
        super(MyLayer, self).__init__( ** kwargs)

    def build(self, input_shape):
        self.W = self.add_weight(name = 'kernel',
        shape = (input_shape[1], self.output_dim),
        initializer = 'uniform',
        trainable = True)
        self.built = True

    def call(self, x):
        return tf.matmul(x, self.W)

    def compute_output_shape(self, input_shape):
        return (input_shape[0], self.output_dim)
```

In the init method we call the superclass constructor and also set the dimensions for our output. In the build we set the initial weight matrix. The matrix shape is set with the help of input parameter – input_shape. The weights are uniformly distributed using the initializer parameter. And we set this matrix to trainable. In the call method, we set whatever the operation that we want to do on the weight matrix. In the current code, we do matrix multiplication of weights with the input vector x. Finally, the compute_output_shape returns the output dimensions.

You can now use this custom layer in any network configuration as illustrated in the code segment below:

```
model = tf.keras.Sequential([tf.keras.layers.Dense(256,
            input_shape=(784,)),
        tf.keras.layers.Dense(256, activation = 'relu'),
        MyLayer(10),
        tf.keras.layers.Dense(10, activation = 'softmax')])
```

Note how the custom layer was added between two Dense layers. As a matter of fact, you can add the custom layer at any desired position in your network. The network plot for the above configuration is shown in Figure 3-4.

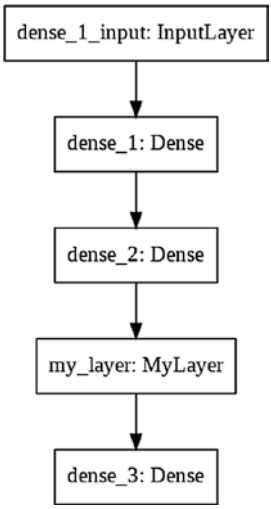


Figure 3-4. *Network containing a custom layer*

You would now compile the model using the usual compile function:

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.binary_crossentropy,
              metrics=['accuracy'])
```

Finally, the model is trained by calling its fit method.

```
model.fit(x,y1, batch_size=32, epochs=30)
```

The entire project code for the custom layers is available in the book's download.

Creating custom layers allows you a great flexibility in designing sophisticated network architectures. Throughout this book, you would be creating a wide variety of network topologies using functional API.

Saving Models

Generally, after a model is fully trained, you would save it to the disk for later deploying it to the production machine. Not only this, but the models that you create in tf.keras can also be saved any time during training. Using this saved model, you can resume training where it left off in the first place. This will help in avoiding long training times in one session. Also, you can share the saved model with others so that they can recreate and continue further on your work. The tf.keras provides several ways of saving your work, which are discussed next.

Whole-Model Saving

When you save the whole model, the following information will be saved.

- The model's architecture
- Weights
- Training configuration that you passed to its compile method
- The optimizer and its state

To save the model you would use the following code:

```
model.save('filename.h5')
```

To recreate the model, you would use the following code:

```
new_model = keras.models.load_model('filename.h5')
```

Export to SavedModel Format

TensorFlow uses a standalone serialization format called SavedModel that can be used by TensorFlow implementations other than Python. This format is also supported by TensorFlow serving. To save the whole model to this format, you would use the following code:

```
model.save('filename', save_format = 'tf')
```

To load the saved model, you would use:

```
new_model = keras.models.load_model('filename')
```

Saving Architecture

In some situations, you may be interested in saving the model's architecture and not its weight values or the optimizer state. In such requirements, you call the get_config method to retrieve the model's configuration and use it later to recreate the model on the saved instance. This is illustrated in the code segment below:

```
# Retrieve and save model configuration
config = model.get_config()

#Recreate model
model = keras.Model.from_config(config)
```

Note that when you recreate the model as shown in this statement, you will lose all the previously learned information.

Saving Weights

In some situations, you may be interested in saving model's training state and not its architecture. In this situation, you would use get_weights and set_weights method to save and retrieve the weights alone. This is illustrated in the code below:

```
# Retrieving model's state
weights = model.get_weights()
# Restoring state
model.set_weights(weights)
```

Note The custom layers do not support model.save() and model.get_config() by default. You will need to override get_config() to support custom layers. However, custom layers support saving weights.

Saving to JSON

Sometimes, you may want to save the model to JSON format. You can do so using the following code fragment:

```
from keras.models import model_from_json
# serialize to JSON
json_model = model_1.to_json()
with open("model_1.json", "w") as json_file:
json_file.write(json_model)
```

```
# load json and re-create model
from keras.models import model_from_json
file = open('model_1.json', 'r')
buffer = file.read()file.close()
model = tf.keras.models.model_from_json(buffer)
```

Consider the following model definition:

```
model_1 = tf.keras.Sequential([
      Conv2D(32, (3, 3), activation = 'relu', padding = 'same',
            input_shape = (32, 32, 3)),
      Conv2D(32, (3, 3), activation = 'relu', padding = 'same'),
      MaxPooling2D((2, 2)),
      Dense(128, activation = 'relu'),
      Dense(10, activation = 'softmax')
])
```

When you save this model architecture to a JSON format, you would see the code shown in Listing 3-1

Listing 3-1. JSON represention of the model

```
{
  "class_name":"Sequential",
  "config":{
    "name":"sequential_3",
    "layers":[
      {
        "class_name":"Conv2D",
        "config":{
          "name":"conv2d_2",
          "trainable":true,
          "batch_input_shape":[
            null,
            32,
            32,
            3
          ],
```

```
"dtype":"float32",
"filters":32,
"kernel_size":[
  3,
  3
],
"strides":[
  1,
  1
],
"padding":"same",
"data_format":"channels_last",
"dilation_rate":[
  1,
  1
],
"activation":"relu",
"use_bias":true,
"kernel_initializer":{
  "class_name":"GlorotUniform",
  "config":{
    "seed":null
  }
},
"bias_initializer":{
  "class_name":"Zeros",
  "config":{

  }
},
"kernel_regularizer":null,
"bias_regularizer":null,
```

```
            "activity_regularizer":null,
            "kernel_constraint":null,
            "bias_constraint":null
          }
        },
      ]
    },
    ...
    "keras_version":"2.2.4-tf",
    "backend":"tensorflow"
}
```

The project code for saving to JSON format is available in the book's download.

With this brief introduction to tf.keras, I will now take you into a more practical approach of learning tf.keras. In the next section, you will be developing an image classification model based on Convolutional Neural Networks (CNN).

Convolutional Neural Networks

Before you start defining the network for image classification, let me give you a few concepts. Look at the image shown in Figure 3-5.

Figure 3-5. *Color image 275x183 pixels*

If I ask you what the image shows? You would immediately say that it shows a bird. This image is of size 275x183 pixels with each pixel representing a color value with RGB components in it. I will now convert the same image into a B&W sketch. The new image is shown in Figure 3-6.

Figure 3-6. *Sketched B&W image 275x183 pixels*

Again, if I asked you what does it show? You would say that this is a bird. Now, I will scale down the image to 32x32 pixels. The new scaled down image is shown in Figure 3-7.

Figure 3-7. *Scaled down image 32x32 pixels*

You can still make out that the image shows a bird. Now, let us come to a point to understand why all these transformations when you have a beautiful picture of a bird for your eyes to appreciate? If our objective is to make the machine learn to interpret (classify) objects in a given image, do we need to feed such high-resolution images to the computer? If a human-being can make the prediction even by looking at a stripped-down version of the image, why can't we have the computer learn the basics of image recognition on such stripped-down images? The reason for doing this is

that the high resolution images contain lots of "bits" information that is redundant for our purpose of object identification. Feeding lots of "bits" into a computer would demand lots of memory and processing power, both of which are very scarce in this world. Thus, to reduce the training time required by a neural network and avoiding the use of heavy resources, we will use stripped-down images in our dataset. The Figure 3-8 shows another picture of an automobile. The original image is of size 200x200. You can still make out in the stripped-down 32x32 image that it is a car.

Figure 3-8. *Car image and the transformations*

A lot of research has already been done on image classification and people have successfully designed ANN architectures to achieve a high level of accuracy. The above-described process of image transformation is called convolution. The exact definition and explanation of image convolution is beyond the scope of this book as this book primarily focuses on building high-performance networks in TF2.x. Researchers have designed networks called Convolutional Neural Networks (CNN) to solve the image classification and object recognition problems. The CNN contains convolutional layers followed by a few more layers of different types. A typical CNN architecture containing two convolutional layers is shown in Figure 3-9.

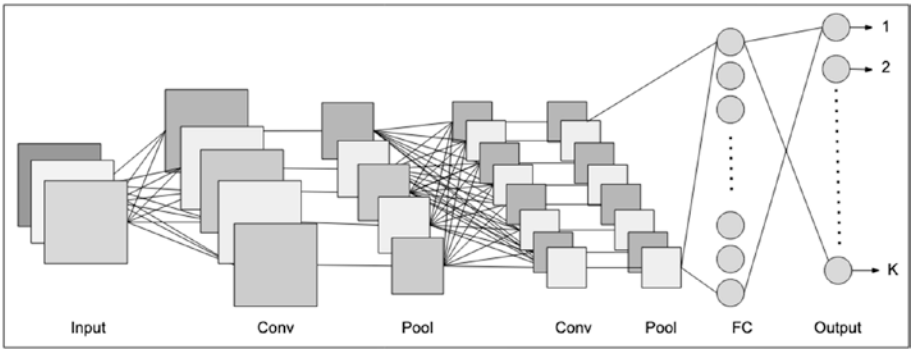


Figure 3-9. *CNN architecture*

Fortunately, as seen earlier, tf.keras provides several ready-to-use layers that include convolution, pooling, flatten, etc. Your task is now to assemble these layers in a proper order and architecture. And that's what you are going to learn in the next section where you will be creating 5 different CNN architectures and evaluate their performances.

Image Classification with CNN

In this section, you will learn how to classify a given image into one of the known types. Suppose, you are shown a certain image and asked to identify an object displayed in the image. For a human-being, this is usually a simple task. But for the machine to interpret the image and classify it to a known object type is not an easy task to learn. Fortunately, a lot of research and development has already been done in this domain and you have many trained ready-to-use ML models that correctly identify an object in a given image with a great degree of accuracy. In this project, you will learn how to develop such a model on your own with the help of libraries provided in tf.keras. So, let us get started with the project.

Creating Project

Open a new Python3 notebook in Colab and name it Ch3-imageClassifier. Load and import TensorFlow 2.x as using the following code:

```
import tensorflow as tf
```

The huge amounts of data are the primary necessity for any machine learning project. To our luck, for image classification somebody has already taken lots of efforts in collecting images and cleansing them so that they are ready for feeding to your image classification algorithm. Your task will now be just to develop a model and keep on refining it so that it can detect objects in a given image with acceptable amounts of accuracy. Before we look into this model development let us look at what dataset is available for our use.

Image Dataset

The image dataset that we are going to use in this project is created by Canadian Institute For Advanced Research (CIFAR). The dataset is available for public use to encourage people to develop techniques of image recognition. Researchers at the CIFAR institute have created two datasets - CIFAR 10 and CIFAR 100. Both datasets consist of 60000 color images of size 32x32 pixels. The CIFAR-10 images are divided in 10 classes containing categories such as cats, birds, ships, airplane, etc. There are 6000 images in each category and they are appropriately randomized for their use as a dataset in machine learning. Note that in Chapter 2, while studying binary classification, we had to randomize the data on our own. The CIFAR-10 dataset is also split appropriately in training and test datasets - the training set consists of 50000 images and the rest 10000 images belong to the test dataset. A set of sample images along with their categories are shown in Figure 3-10 for your quick understanding of the provided image data.

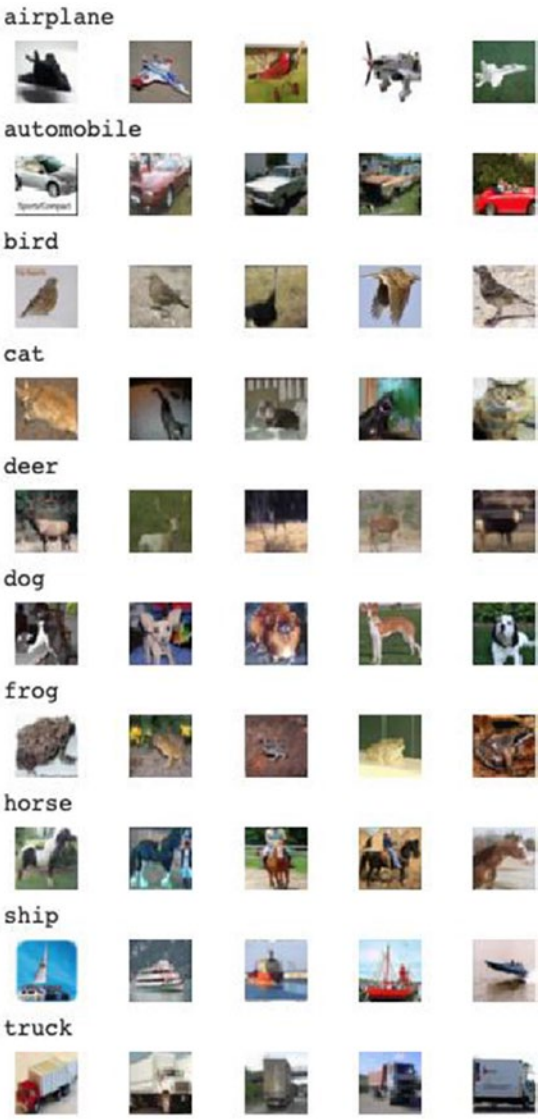


Figure 3-10. *Sample images of CIFAR-10*

Note that all the images are of the fixed size - 32x32 pixels. This is what is required for the machine to learn. You cannot feed the images of the varied sizes to your machine learning algorithm. The input to your

neural network is always of a predetermined fixed size. An image with a variable resolution will contain a variable number of pixels and thus cannot be input to a network of a preset topology. Fortunately, creators of CIFAR datasets have taken the trouble of scaling down all the real images to a fixed miniature size. Note that if you try to train your model on a real world image consisting of say 512x512 pixels or more, the number of input points (pixels) that will be required by your ANN would be very large. This directly implies a large number of "weights" to be trained demanding in huge processing power and time. When the image is scaled down to 32x32, there is obviously loss of data; however, you will find that training on these stripped-down images yet produces great results and identifies unseen objects with extremely acceptable results in the real world.

The CIFAR-100 dataset is similar to CIFAR-10, except that it represents 100 classes instead of 10. Each category is further divided into fine and coarse. For example, a superclass called people contains subclasses baby, boy, girl, man and woman.

We will be using the CIFAR-10 dataset for this project. So, let us start by loading this dataset into our program.

Loading Dataset

The tf.keras integration provides several built-in databases for your ease in model development. These also include CIFAR-10. The databases are available in tf.keras.dataset module. The list of available datasets can be printed on the console using the following two lines of code:

```
import tensorflow_datasets as tfds
print ("Number of datasets: ", len(tfds.list_builders()))
tfds.list_builders()
```

A partial list from the output is shown here for your quick reference.

```
Number of datasets: 141
['abstract_reasoning',
 'aeslc',
 'aflw2k3d',
 'amazon_us_reviews',
 'arc',
 'bair_robot_pushing_small',
 'big_patent',
 'bigearthnet',
 'billsum',
 'binarized_mnist',
 'binary_alpha_digits',
 'c4',
 'caltech101',
 'caltech_birds2010',
 ...
```

As you must have noticed, there are currently 141 datasets available for your use. You will be using cifar10 dataset for developing the current application.

Creating Training/Testing Datasets

In the binary classifier that you developed in the last chapter you used the train_test_split method of sklearn to create training and testing datasets. The keras.datasets now provides a load_data method that creates training/testing datasets in a single call. Use the following statement to create these datasets from the built-in CIFAR dataset.

```
(x_train, y_train), (x_test, y_test) =
                  tf.keras.datasets.cifar10.load_data()
```

After the data is loaded, you will have both features and labels arrays available in the four variables. You may examine the number of images available for training and testing using following code segment:

```
print("x_train dimensions : ",x_train.shape)
print("x_test dimensions  : ",x_test.shape)
print("y_train dimensions : ",y_train.shape)
print("y_test dimensions  : ",y_test.shape)
```

When you execute the above code, you will see the following output:

```
x_train dimensions :  (50000, 32, 32, 3)
x_test dimensions  :  (10000, 32, 32, 3)
y_train dimensions :  (50000, 1)
y_test dimensions  :  (10000, 1)
```

From the size of x_train, you know that there are 50000 images in the training set, each image is of 32x32 pixels and each pixel possesses three values RGB, pertaining to its color. The shape of x_test indicates to you that the testing dataset contains 10000 images. For every image in x_train there is a specific integer value in the range 0 to 9 in y_train. The y_train becomes our labels for machine learning. We will use x_train for training the model and x_test for evaluating the model's performance.

If you are curious in knowing what is loaded in memory, try printing one of the images on the console using the following code:

```
import matplotlib.pyplot as plt
plt.imshow(x_train[40])
```

The execution of the above code shows the image at index value of 40. The image is shown in Figure 3-11.

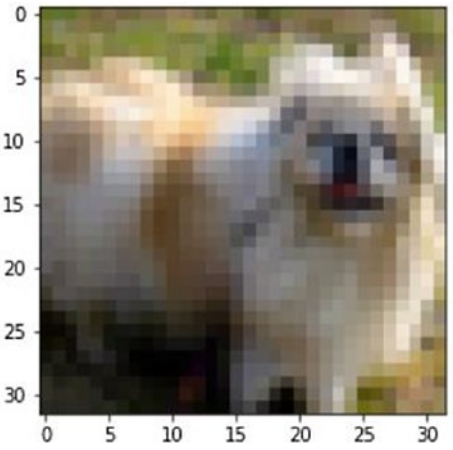


Figure 3-11. *A sample image from the dataset*

Very clearly, this image of size 32x32 barely allows you to guess what it contains? However, as I said, for our machine learning algorithm, the images of these sizes are good enough for training. Now, we will prepare our training dataset for model training.

Preparing Data for Model Training

As said earlier, we will use x_train for training our model. You need to reserve a part of this data for validation during the training phase. We will reserve 5% of the training data for validation.

Creating Validation Dataset

To split the training data into the actual training dataset and validation dataset, we use sklearn's split method as shown in the code segment here:

```
from sklearn.model_selection import train_test_split
x_train, x_val, y_train, y_val = train_test_split(x_train,
                        y_train, test_size = 0.05,
                        random_state = 0)
```

Here, we reserve 5% of the training data for validation during model training. The x_val and y_val represent our validation datasets respectively for features and labels. Next, we will augment our data.

Augmenting Data

Before I explain how to augment data, let me explain what is augmentation and why is it necessary in machine learning? Data augmentation is a strategy used by data scientists to increase the diversity of data available during model training. The raw data collected in the field may exhibit uniformity leading to unperfect training. The augmentation avoids the need of collecting new data for diversity. Some of the techniques used for augmentation are cropping, padding and horizontal flipping. For augmenting image data, Keras preprocessing module provides a ImageDataGenerator class. You instantiate this class using the following code segment:

```
from tensorflow.keras.preprocessing.image import
ImageDataGenerator
datagen = ImageDataGenerator(
  rotation_range=15,
  width_shift_range=0.1,
  height_shift_range=0.1,
  horizontal_flip=True, )
```

The various parameters decide the type of augmentation performed on the specified image. As the parameter names in the example suggest, the image may be rotated, shifted or even flipped to get the needed diversity in the captured data. You will use this datagen instance to augment the training data.

As each RGB value ranges between 0 and 255, these also need to be scaled to the range 0 to 1. For scaling, we define the scaling function as follows:

```
def normalize(data):
    data = data.astype("float32")
    data = data/255.0
    return data
```

We will now do the augmentation and scaling on the training/testing data using the following code segment:

```
x_train = normalize(x_train)
datagen.fit(x_train)
x_val = normalize(x_val)
datagen.fit(x_val)
x_test = normalize(x_test)
```

The normalize function scales the train and validation data. The fit method of ImageDataGenerator augments the data. We just normalize our test data but do not augment it, as we need the testing results on the real images and not the augmented ones.

One last thing that we need to do as a part of data processing is to call the to_categorical method to convert the integer values into a matrix. Note that our labels column has ten categories pertaining to the different object types such as bird and automobile. To convert this column, we use to_categorical function of Keras utils as follows:

```
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)
y_val = tf.keras.utils.to_categorical(y_val, 10)
```

After all these transformations are done, if you wish to see their effect on an image, just print it one more time on the console using the imshow function as given below:

```
plt.imshow(x_train[40])
```

The resultant image is shown in Figure 3-12.

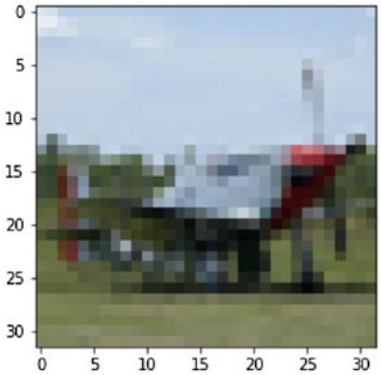


Figure 3-12. *Transformed image at index 40*

Now, as we have completed our data preprocessing, let us print the dimensions of the modified data just to understand the net result of preprocessing. You print the dimensions using the following code segment:

```
print("x_train dimensions : ",x_train.shape)
print("y_train dimensions : ",y_train.shape)
print("x_test dimensions  : ",x_test.shape)
print("y_test dimensions  : ",y_test.shape)
print("x_val dimensions   : ",x_val.shape)
print("y_val dimensions   : ",y_val.shape)
```

The output of execution of above code is as follows:

```
x_train dimensions :  (47500, 32, 32, 3)
y_train dimensions :  (47500, 10)
x_test dimensions  :  (10000, 32, 32, 3)
y_test dimensions  :  (10000, 10)
x_val dimensions   :  (2500, 32, 32, 3)
y_val dimensions   :  (2500, 10)
```

Note that the dimensions on the features vector remains the same, while the dimension of the labels vector has changed to 10. This is due to the fact that we have 10 classes in our dataset.

Now, comes the most important part of our machine learning and that is defining the model. Before I define the model, I am going to write a function that will train, evaluate and print error metrics of a specified model. The reason I am doing this is to show you the effect of different model architectures on the accuracy of results. I will initially start off with a simple architecture and then keep on making additions with the hope that the accuracy would improve. Just in case the changes made to the model result in lesser accuracy, we will discard those changes and go in for the new ones.

Model Development

In this section, we will define several models having different architectures. You will train these models and make predictions on unseen images. You will then do the performance ratings on the models based on their accuracy and prediction results. You will save the best model to disk and use it further in a production environment for performing image classification on unseen images.

To compare the performance of different models, I will define a function which is callable on every model that we define. This function will train the model, evaluate its performance and print the error metrics.

Train/Evaluate/Display Function

We define the function for implementing training, model evaluation and metrics display as follows:

```
def results(model):
```

The function takes a parameter of type model. We will create different types of models and pass it to this function as a parameter.

In the body of the function, we call the model's fit method to begin the training. This is shown in the function call shown here:

```
epoch = 20
r = model.fit(x_train, y_train, batch_size = 32,
                  epochs = epoch, validation_data =
                  (x_val, y_val), verbose = 1)
```

During training, you will see the output of each epoch as shown in Figure 3-13.

```
Train on 47500 samples, validate on 2500 samples
Epoch 1/20
47500/47500 [==============================] - 12s 262us/sample - loss: 1.8401 - accuracy: 0.3402 - val_loss: 1.5944 - val_accuracy: 0.4264
Epoch 2/20
47500/47500 [==============================] - 6s 120us/sample - loss: 1.4749 - accuracy: 0.4737 - val_loss: 1.3695 - val_accuracy: 0.5048
Epoch 3/20
47500/47500 [==============================] - 6s 122us/sample - loss: 1.3041 - accuracy: 0.5380 - val_loss: 1.2184 - val_accuracy: 0.5656
Epoch 4/20
47500/47500 [==============================] - 6s 124us/sample - loss: 1.1634 - accuracy: 0.5889 - val_loss: 1.1211 - val_accuracy: 0.5988
Epoch 5/20
47500/47500 [==============================] - 6s 123us/sample - loss: 1.0602 - accuracy: 0.6282 - val_loss: 1.1006 - val_accuracy: 0.6068
Epoch 6/20
47500/47500 [==============================] - 6s 123us/sample - loss: 0.9761 - accuracy: 0.6566 - val_loss: 1.0766 - val_accuracy: 0.6240
```

Figure 3-13. *Training in progress*

The number of training iterations is declared in the Python variable called epoch which is set to 20. You may change this number for re-training the network after examining the error metrics. The parameters to the fit method are self-explanatory and I have discussed those in the previous chapter while discussing the binary classification project.

After the training is over, we will evaluate the model's accuracy on the test data. This is done by calling the model's evaluate method as shown here:

```
acc = model.evaluate(x_test, y_test)
print("test set loss : ", acc[0])
print("test set accuracy :", acc[1]*100)
```

A typical output of the above code is shown here:

```
test set loss :  1.4717637085914612
test set accuracy : 65.78999757766724
```

We plot the accuracy metrics using the following code segment:

```
# Plot training and validation accuracy
  epoch_range = range(1, epoch+1)
  plt.plot(epoch_range, r.history['accuracy'])
  plt.plot(epoch_range, r.history['val_accuracy'])
  plt.title('Classification Accuracy')
  plt.ylabel('Accuracy')
  plt.xlabel('Epoch')
  plt.legend(['Train', 'Val'], loc='lower right')
  plt.show()
```

The metrics for both training and validation data are plotted. A typical plot is shown in Figure 3-14.

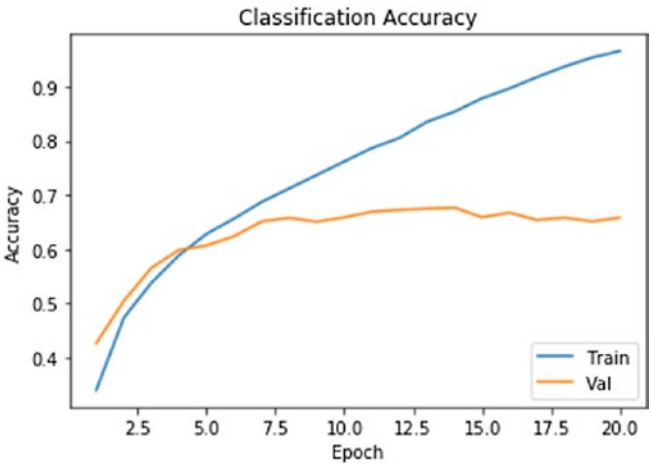


Figure 3-14. *Accuracy metrics plot*

Likewise, we plot the loss metrics using the following code segment:

```
# Plot training & validation loss values
plt.plot(epoch_range,r.history['loss'])
plt.plot(epoch_range, r.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Val'], loc='lower right')
plt.show()
```

A typical plot is shown in Figure 3-15.

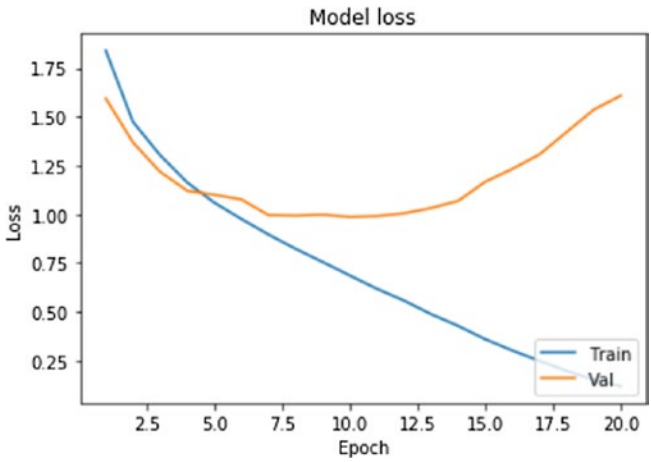


Figure 3-15. *Loss metrics plot*

The above plotting code is self-explanatory and was explained in Chapter 2. This completes the definition of results function. The entire function code is given here for your quick reference.

```
def results(model):
  epoch = 20
  r = model.fit(x_train, y_train, batch_size = 32,
            epochs = epoch, validation_data =
            (x_val, y_val), verbose = 1)
  acc = model.evaluate(x_test, y_test)
  print("test set loss : ", acc[0])
  print("test set accuracy :", acc[1]*100)

  # Plot training and validation accuracy
  epoch_range = range(1, epoch+1)
  plt.plot(epoch_range, r.history['accuracy'])
  plt.plot(epoch_range, r.history['val_accuracy'])
  plt.title('Classification Accuracy')
  plt.ylabel('Accuracy')
  plt.xlabel('Epoch')
```

```
  plt.legend(['Train', 'Val'], loc='lower right')
  plt.show()

  # Plot training & validation loss values
  plt.plot(epoch_range,r.history['loss'])
  plt.plot(epoch_range, r.history['val_loss'])
  plt.title('Model loss')
  plt.ylabel('Loss')
  plt.xlabel('Epoch')
  plt.legend(['Train', 'Val'], loc='lower right')
  plt.show()
```

Before we start defining our various models, let me define one more function that will be used for predicting the class of an object embedded in a specified image.

Predict Function

As our dataset consists of 10 classes, we first define the names of those classes in our code:

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

We define the function called predict_class as follows:

```
def predict_class(filename, model):
```

The function takes two parameters - the first one specifies the name of the image file and the second specifies the model used for classifying the given image. Thus, this is a general purpose function which can be used for comparing the performance of different models.

Within the function we load the image in memory using the code:

```
  img = load_img(filename, target_size=(32, 32))
```

We display the image on screen for our reference:

```
plt.imshow(img)
```

We need to convert the image data into an array that can be input to our model's predict method. The img_to_array method coupled with reshaping does this conversion.

```
img = img_to_array(img)
img = img.reshape(1,32,32,3)
```

We then scale the array elements using the following statements:

```
img = img.astype('float32')
img = img/255.0
```

The image data is now ready for inputting to the model's predict function.

```
result = model.predict(img)
```

The output returned by the predict function is in a matrix form. We will copy the output to a dictionary. We convert the output to a dictionary using a for loop as shown in the statement below:

```
dict2 = {}
for i in range(10):
    dict2[result[0][i]] = classes[i]
```

The prediction result becomes the key and its class becomes the value in the dictionary.

We will now copy the result[0] which is the prediction into a list called res. We sort this array to order the predictions in an ascending order.

```
res = result[0]
res.sort()
```

We pick up the top three predictions for display. Note that the best prediction is stored at the last index.

```
  res = res[::-1]
  results = res[:3]
```

We finally print the result on screen.

```
  print("Top predictions of these images are")
  for i in range(3):
      print("{} : {}".format(dict2[results[i]],
       (results[i]*100).round(2)))
```

We also display the image used in our experiment for reference. Note that the image is already included in the plt object.

```
  print('The image given as input is')
```

This completes the definition of our predict_class function. The entire definition is given here for your quick reference:

```
# Predict the class in a given image
from tensorflow.keras.preprocessing.image
import load_img, img_to_array

classes = ['airplane','automobile', 'bird', 'cat', 'deer',
                   'dog','frog', 'horse','ship','truck']
def predict_class(filename, model):
  img = load_img(filename, target_size=(32, 32))

  plt.imshow(img)

  # convert to array
  # reshape into a single sample with 3 channels
  img = img_to_array(img)
  img = img.reshape(1,32,32,3)
```

```
# prepare pixel data
img = img.astype('float32')
img = img/255.0

#predicting the results
result = model.predict(img)

dict2 = {}
for i in range(10):
    dict2[result[0][i]] = classes[i]

res = result[0]
res.sort()
res = res[::-1]
results = res[:3]

print("Top predictions of these images are")
for i in range(3):
    print("{} : {}".format(dict2[results[i]],
         (results[i]*100).round(2)))

print('The image given as input is')
```

Now it is time to start defining our models.

Defining Models

In this section, we will define 5 models with increasing complexity. After defining each model, we will train it immediately, evaluate its performance and make some predictions. At the end, we will compare the results produced by different models. So, let us start with the first model.

A Model with 2 Convolutional Layers

Our first model in the experimentation would be a simple one consisting of only 2 convolutional layers. To define the models, Keras provides a Sequential API that you have used earlier in Chapter 2. You import the Sequential API using the following statement:

```
from tensorflow.keras.models import Sequential
```

The Sequential API allows you to build a sequential network architecture where you keep adding layers to it in a desired sequence. Keras provides several layer types for your ready use. However, it also allows you to create the custom layers of your own should the need arise. For this application, we will use Keras provided layers. Some of these named layers are Conv2D, Dense, Dropout and Flatten. You import these in your code using following program statement:

```
from tensorflow.keras.layers
import Dense, Dropout, Conv2D, MaxPooling2D,
                  Flatten, BatchNormalization
```

Now, we define our model by instantiating the Sequential class as follows:

```
model_1 = Sequential([
      Conv2D(32, (3, 3), activation = 'relu', padding = 'same',
            input_shape = (32, 32, 3)),
      Conv2D(32, (3, 3), activation = 'relu', padding = 'same'),
      MaxPooling2D((2, 2)),
      Flatten(),
      Dense(128, activation = 'relu'),
      Dense(10, activation = 'softmax')
])
```

As specified in the constructor, the network consists of 6 layers. The first layer is of type Conv2D which is our convolutional layer. This consists of 32 filters of size 3x3 that runs over 32x32x3 images. The input to the network is 32x32x3 which you would remember are the dimensions of our x_train vector. The second layer is once again a convolutional layer. The third is a MaxPooling2D layer followed by Flatten. The last two layers are Dense layers - the first consists of 128 nodes with ReLU activation and the second contains 10 nodes with softmax activation. Remember our network is supposed to output 10 classes and thus the last layer contains 10 nodes. The network plot can be generated using the following code:

```
from tensorflow.keras.utils import plot_model
plot_model(model_1, to_file='model1.png')
```

The generated plot is shown in Figure 3-16.

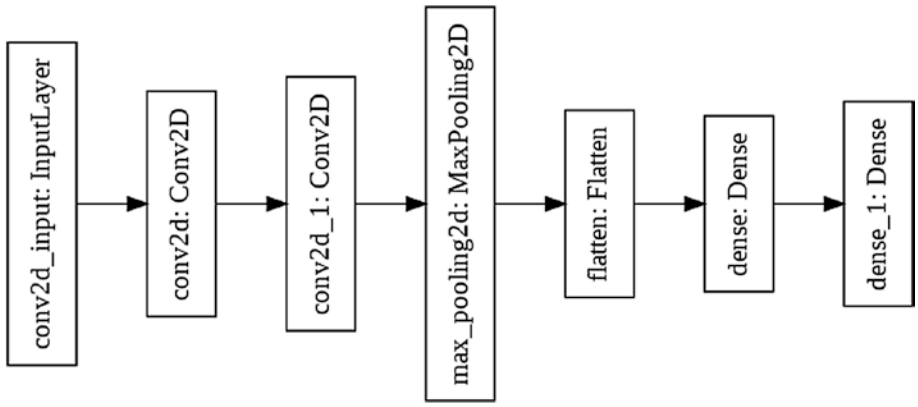


Figure 3-16. *Model 1 network plot*

The network summary can be printed by calling the summary method on the model_1 instance and is shown below:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 32)        896
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 32)        9248
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 32)        0
_________________________________________________________________
flatten (Flatten)            (None, 8192)              0
_________________________________________________________________
dense (Dense)                (None, 128)               1048704
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 1,060,138
Trainable params: 1,060,138
Non-trainable params: 0
```

After defining the network, we set the optimizer to SGD and compile the model with the loss function set to categorical_crossentropy as in the following code fragment:

```
opt = tf.keras.optimizers.SGD(lr=0.001, momentum=0.9)
model_1.compile(optimizer=opt, loss = 'categorical_crossentropy',
                metrics = ['accuracy'])
```

Now, we train, evaluate and display the error metrics by calling our previously defined results function with model_1 as the parameter to it.

```
results(model_1)
```

The output generated is shown in Figure 3-17.

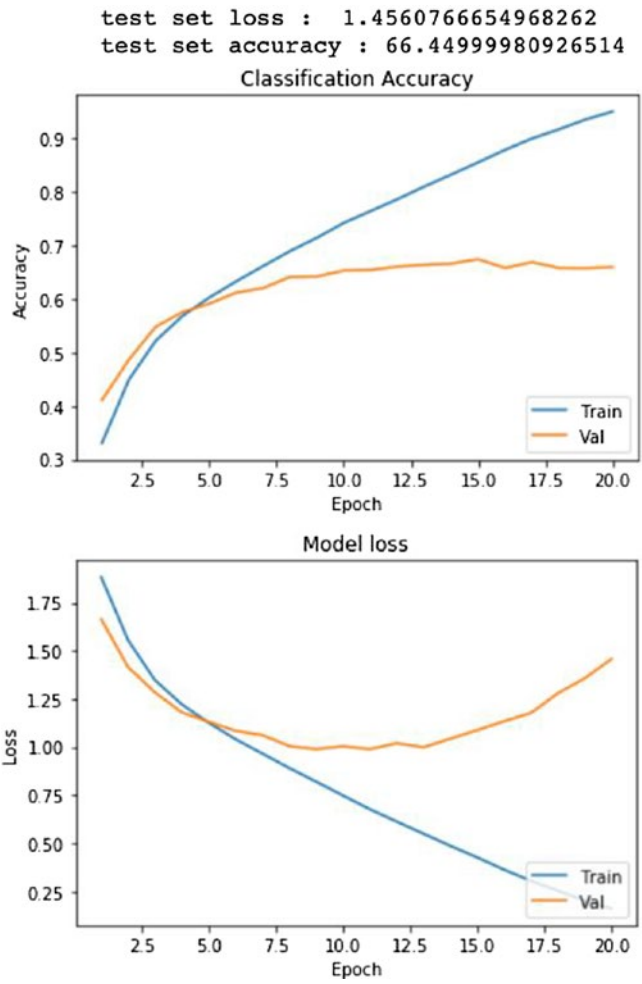


Figure 3-17. *Model 1 error metrics*

As you see the accuracy is about 66% - obviously not within acceptable limits in most cases. Also, there is clearly an overfitting. You can reduce overfitting through parameter optimization and regularization. We will keep applying such techniques to improve the performance of our model.

You will now make some prediction on an unseen image using the following code segment:

```
import urllib
resource = urllib.request.urlopen("https://raw.
githubusercontent.com/Apress/artificial-neural-networks-with-
tensorflow-2/main/ch03/test01.png")

output = open("file01.jpg", "wb")
output.write(resource.read())
output.close()
predict_class("file01.jpg", model_1)
```

The output of predication is shown in Figure 3-18.

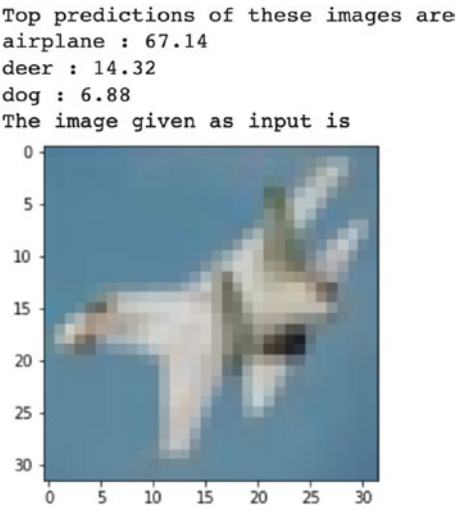


Figure 3-18. *The model_1 performance*

Now with the hope to improve the accuracy, we will add two more convolutional layers to our architecture.

Model_2 with 4 Convolutional Layers

We will now define a new architecture with 4 convolutional layers as shown in the code below:

```
model_2 = Sequential([
  Conv2D(32, (3, 3), activation = 'relu', padding = 'same',
             input_shape = (32, 32, 3)),
  Conv2D(32, (3, 3), activation = 'relu', padding = 'same'),
  MaxPooling2D((2, 2)),
  Conv2D(64, (3, 3), activation = 'relu', padding = 'same'),
  Conv2D(64, (3, 3), activation = 'relu', padding = 'same'),
  MaxPooling2D((2, 2)),
  Flatten(),
  Dense(128, activation = 'relu'),
  Dense(10, activation = 'softmax')
])

opt = tf.keras.optimizers.SGD(lr = 0.001, momentum = 0.9)
model_2.compile(optimizer = opt, loss =
                'categorical_crossentropy',
                metrics = ['accuracy'])
```

The model summary is printed below for your quick reference:

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 32, 32, 32)        9248
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 16, 16, 32)        0
```

```
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 16, 16, 64)        36928
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 4096)              0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               524416
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1290
=================================================================
Total params: 591,274
Trainable params: 591,274
Non-trainable params: 0
```

The network plot is shown in Figure 3-19.

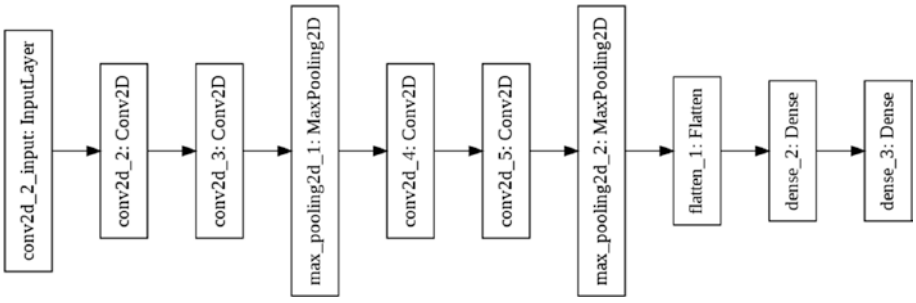


Figure 3-19. *Model 2 network plot*

The model's error metrics are shown in Figure 3-20.

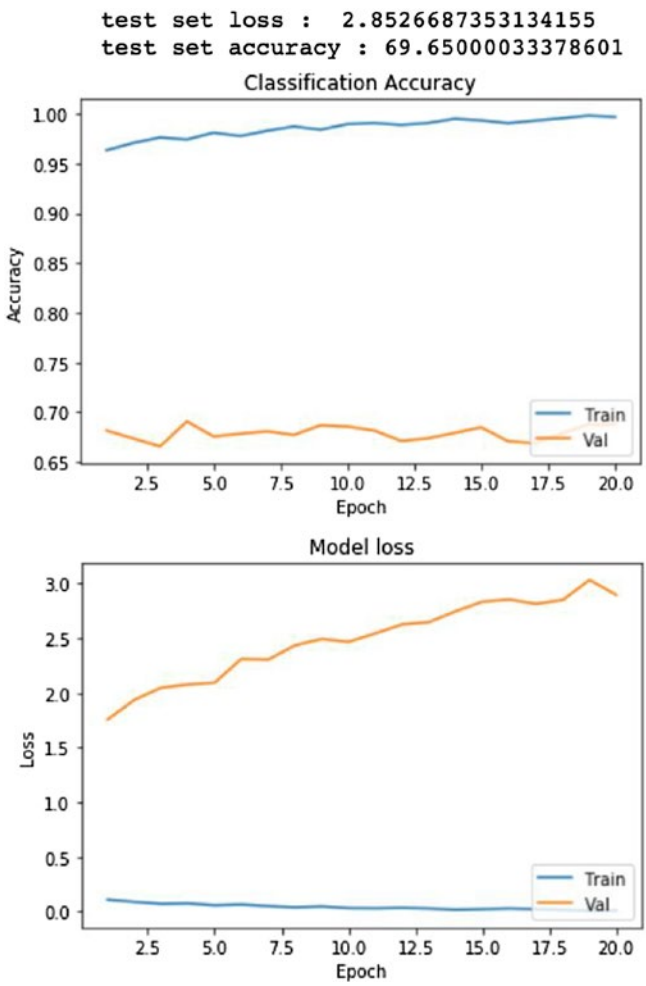


Figure 3-20. *Model 2 error metrics*

The model's predication on the same image used for the first model is shown here:

```
Top predictions of these images are
airplane : 99.95
bird : 0.04
deer : 0.01
```

You observe that the accuracy is marginally increased from 66.44% to about 69.65%. The prediction on the given image almost remains the same.

We will now try to add more convolutional layers to see if the model's accuracy improves further.

Third Model : 6 Convolutional layers with 32, 64 and 128 filters respectively

The modified model definition is given in code below:

```
model_3 = Sequential([
  Conv2D(32, (3, 3), activation = 'relu', padding = 'same',
      input_shape = (32, 32, 3)),
  Conv2D(32, (3, 3), activation = 'relu', padding = 'same'),
  MaxPooling2D((2, 2)),
  Conv2D(64, (3, 3), activation = 'relu', padding = 'same'),
  Conv2D(64, (3, 3), activation = 'relu', padding = 'same'),
  MaxPooling2D((2, 2)),
  Conv2D(128, (3, 3), activation = 'relu', padding = 'same'),
  Conv2D(128, (3, 3), activation = 'relu', padding = 'same'),
  MaxPooling2D((2, 2)),
  Flatten(),
  Dense(128, activation = 'relu'),
  Dense(10, activation = 'softmax')
])

opt = tf.keras.optimizers.SGD(lr = 0.001, momentum = 0.9)
model_3.compile(optimizer = opt, loss = 'categorical_
crossentropy',
       metrics = ['accuracy'])
```

The model's summary is as follows:

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_6 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 32, 32, 32)        9248
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 16, 16, 64)        36928
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 8, 8, 128)         73856
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 8, 8, 128)         147584
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 4, 4, 128)         0
_________________________________________________________________
flatten_2 (Flatten)          (None, 2048)              0
_________________________________________________________________
dense_4 (Dense)              (None, 128)               262272
_________________________________________________________________
dense_5 (Dense)              (None, 10)                1290
```

```
=================================================================
Total params: 550,570
Trainable params: 550,570
Non-trainable params: 0
```

The network plot for this model is shown in Figure 3-21.

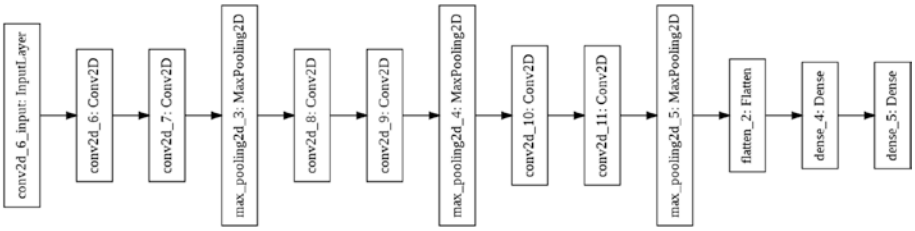


Figure 3-21. *Model 3 network plot*

The model's error metrics are shown in Figure 3-22.

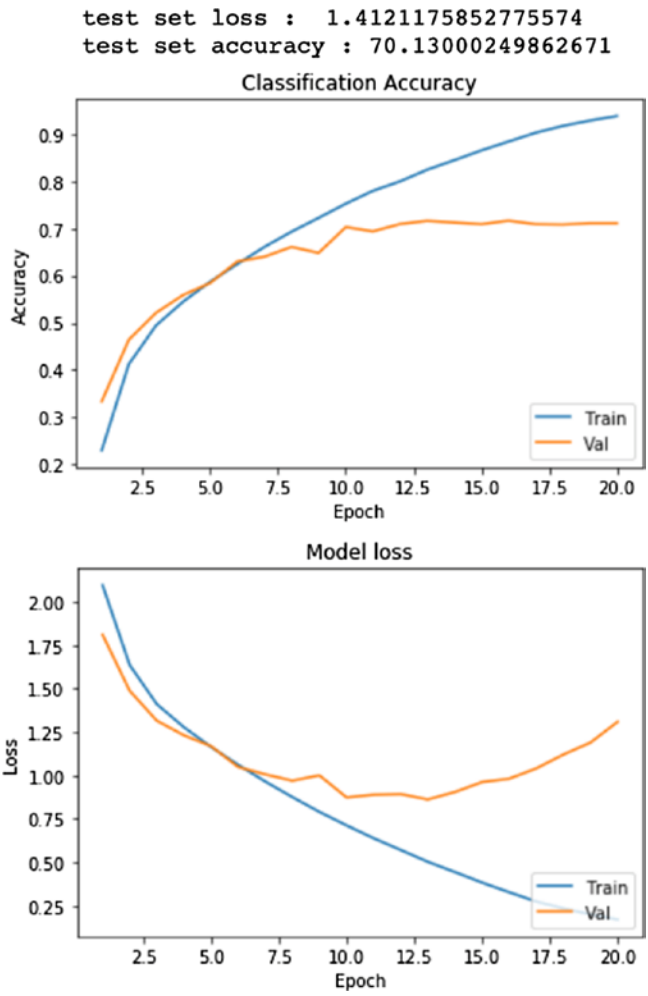


Figure 3-22. *Model 3 error metrics*

The predictions made by this model on our earlier image are as follows:

```
Top predictions of these images are
deer : 87.18
bird : 8.07
airplane : 3.81
```

The model's accuracy (70.13%) has almost remained at the earlier model's accuracy level (69.65%). Thus, merely adding more convolutional layers is not getting to help us creating a better model.

Let us now try by adding a dropout layer to our model.

Fourth Model : Addition of dropout layer

The idea in dropout is to randomly drop a few units along with their connections from the network during training. The reduction in number of parameters in each step of training has an effect of regularization. The modified model definition is given here:

```
model_4 = Sequential([
  Conv2D(32, (3, 3), activation = 'relu', kernel_initializer =
           'he_uniform', padding = 'same', input_shape =
           (32, 32, 3)),
  Conv2D(32, (3, 3), activation = 'relu', kernel_initializer =
           'he_uniform', padding = 'same'),
  MaxPooling2D((2, 2)),
  Dropout(0.2),
  Conv2D(64, (3, 3), activation = 'relu', kernel_initializer =
           'he_uniform', padding = 'same'),
  Conv2D(64, (3, 3), activation = 'relu', kernel_initializer =
           'he_uniform', padding = 'same'),
  MaxPooling2D((2, 2)),
  Dropout(0.2),
  Conv2D(128, (3, 3), activation = 'relu', kernel_initializer =
           'he_uniform', padding = 'same'),
  Conv2D(128, (3, 3), activation = 'relu', kernel_initializer =
           'he_uniform', padding = 'same'),
  MaxPooling2D((2, 2)),
  Dropout(0.3),
  Flatten(),
```

```
  Dense(128, activation = 'relu'),
  Dense(10, activation = 'softmax')
])

opt = tf.keras.optimizers.SGD(lr = 0.001, momentum = 0.9)
model_4.compile(optimizer = opt, loss = 'categorical_
crossentropy',
           metrics = ['accuracy'])
```

The model summary is shown here:

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_12 (Conv2D)           (None, 32, 32, 32)        896
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 32, 32, 32)        9248
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 16, 16, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 16, 16, 64)        18496
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 16, 16, 64)        36928
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 8, 8, 64)          0
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 8, 8, 128)         73856
```

```
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 8, 8, 128)         147584
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 4, 4, 128)         0
_________________________________________________________________
dropout_2 (Dropout)          (None, 4, 4, 128)         0
_________________________________________________________________
flatten_3 (Flatten)          (None, 2048)              0
_________________________________________________________________
dense_6 (Dense)              (None, 128)               262272
_________________________________________________________________
dense_7 (Dense)              (None, 10)                1290
=================================================================
Total params: 550,570
Trainable params: 550,570
Non-trainable params: 0
```

The network plot for the model is shown in Figure 3-23

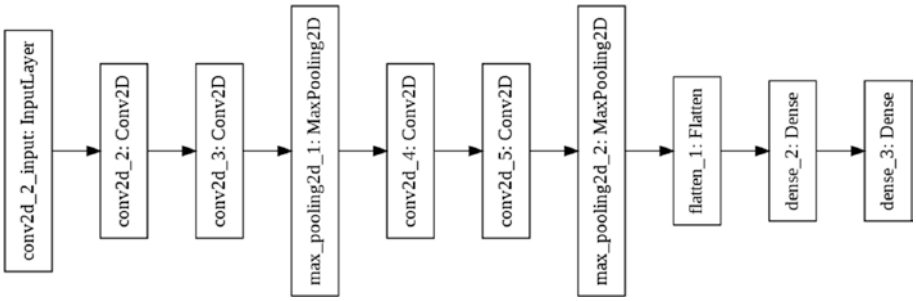


Figure 3-23. *Model 4 network plot*

The model's error metrics are shown in Figure 3-24.

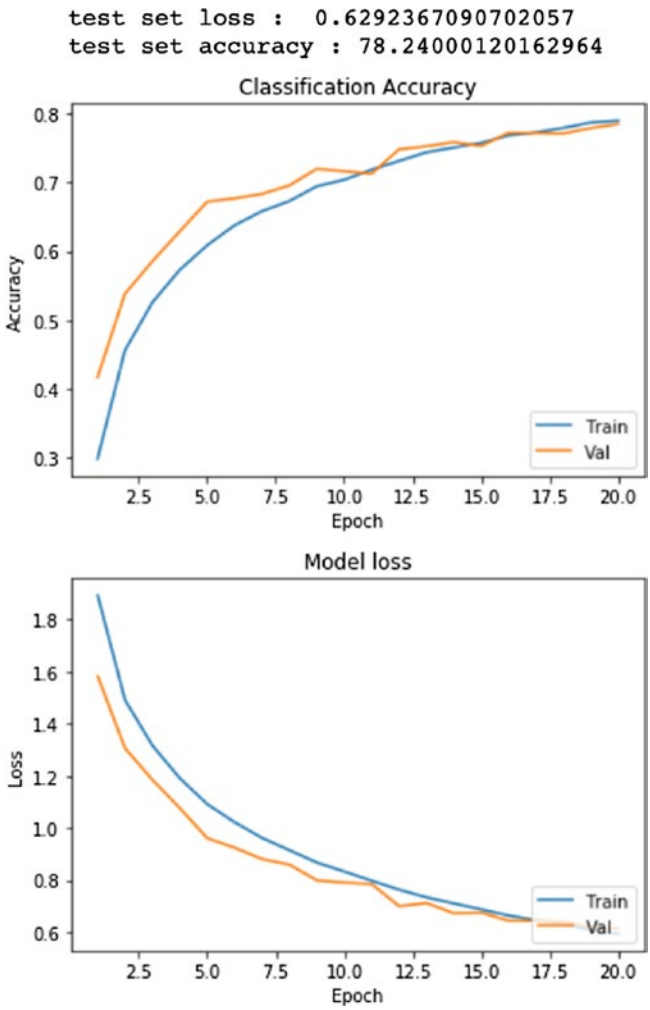


Figure 3-24. *Model 4 error metrics*

The prediction result on the same previously used image is shown here:

```
Top predictions of these images are
cat : 26.33
dog : 25.52
airplane : 21.61
```

The accuracy has now improved to 78.24%, a much bigger gain than the previous cases. We will now create one more model with batch normalization and regularization.

Model 5

The BatchNormalization works the same way as a regular normalization, except that it works on batches of data. The activation of the previous layer occurs at each batch to maintain the mean activation close to 0 and the standard deviation close to 1. Thus, it reduces the number of training steps thereby speeding up the training process. In some situations, it may also result in eliminating the need for dropout used in the previous model.

The Regularization is a technique used to reduce the error by fitting a function appropriately on the training set so as to avoid overfitting.

We will apply these techniques in our new model definition.

The model definition is shown in the code below:

```
weight_decay = 1e-4
model_5 = Sequential([
  Conv2D(32, (3, 3), activation = 'relu', padding = 'same',
      kernel_regularizer = tf.keras.regularizers.l2(weight_
      decay),
      input_shape = (32, 32, 3)),
  BatchNormalization(),
  Conv2D(32, (3, 3), activation = 'relu', kernel_regularizer =
      tf.keras.regularizers.l2(weight_decay), padding = 'same'),
  BatchNormalization(),
  MaxPooling2D((2, 2)),
  Dropout(0.2),
  Conv2D(64, (3, 3), activation = 'relu', kernel_regularizer =
      tf.keras.regularizers.l2(weight_decay), padding = 'same'),
```

```
  BatchNormalization(),
  Conv2D(64, (3, 3), activation = 'relu', kernel_regularizer =
      tf.keras.regularizers.l2(weight_decay), padding = 'same'),
  BatchNormalization(),
  MaxPooling2D((2, 2)),
  Dropout(0.3),
  Conv2D(128, (3, 3), activation = 'relu', kernel_regularizer =
      tf.keras.regularizers.l2(weight_decay), padding = 'same'),
  BatchNormalization(),
  Conv2D(128, (3, 3), activation = 'relu', kernel_regularizer =
      tf.keras.regularizers.l2(weight_decay), padding = 'same'),
  BatchNormalization(),
  MaxPooling2D((2, 2)),
  Dropout(0.3),
  Flatten(),
  Dense(128, activation = 'relu'),
  Dense(10, activation = 'softmax')
])

opt = tf.keras.optimizers.SGD(lr = 0.001, momentum = 0.9)
model_5.compile(optimizer = opt, loss = 'categorical_
crossentropy',
      metrics = ['accuracy'])
```

The model summary is as follows:

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_18 (Conv2D)           (None, 32, 32, 32)        896
_________________________________________________________________
batch_normalization (BatchNo (None, 32, 32, 32)        128
```

```
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 32, 32, 32)        9248
_________________________________________________________________
batch_normalization_1 (Batch (None, 32, 32, 32)        128
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 16, 16, 32)        0
_________________________________________________________________
dropout_3 (Dropout)          (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 16, 16, 64)        18496
_________________________________________________________________
batch_normalization_2 (Batch (None, 16, 16, 64)        256
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 16, 16, 64)        36928
_________________________________________________________________
batch_normalization_3 (Batch (None, 16, 16, 64)        256
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 8, 8, 64)          0
_________________________________________________________________
dropout_4 (Dropout)          (None, 8, 8, 64)          0
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 8, 8, 128)         73856
_________________________________________________________________
batch_normalization_4 (Batch (None, 8, 8, 128)         512
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 8, 8, 128)         147584
_________________________________________________________________
batch_normalization_5 (Batch (None, 8, 8, 128)         512
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 4, 4, 128)         0
```

```
_________________________________________________________________
dropout_5 (Dropout)          (None, 4, 4, 128)         0
_________________________________________________________________
flatten_4 (Flatten)          (None, 2048)              0
_________________________________________________________________
dense_8 (Dense)              (None, 128)               262272
_________________________________________________________________
dense_9 (Dense)              (None, 10)                1290
=================================================================
Total params: 552,362
Trainable params: 551,466
Non-trainable params: 896
```

The network plot is shown in Figure 3-25

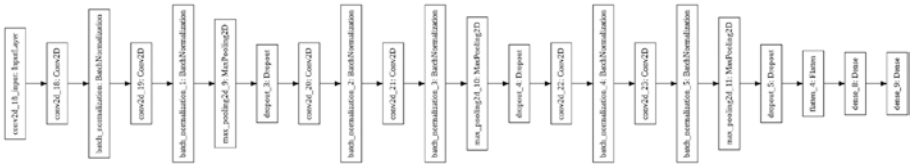


Figure 3-25. *Model 5 network plot*

The error metrics plots are shown in Figure 3-26.

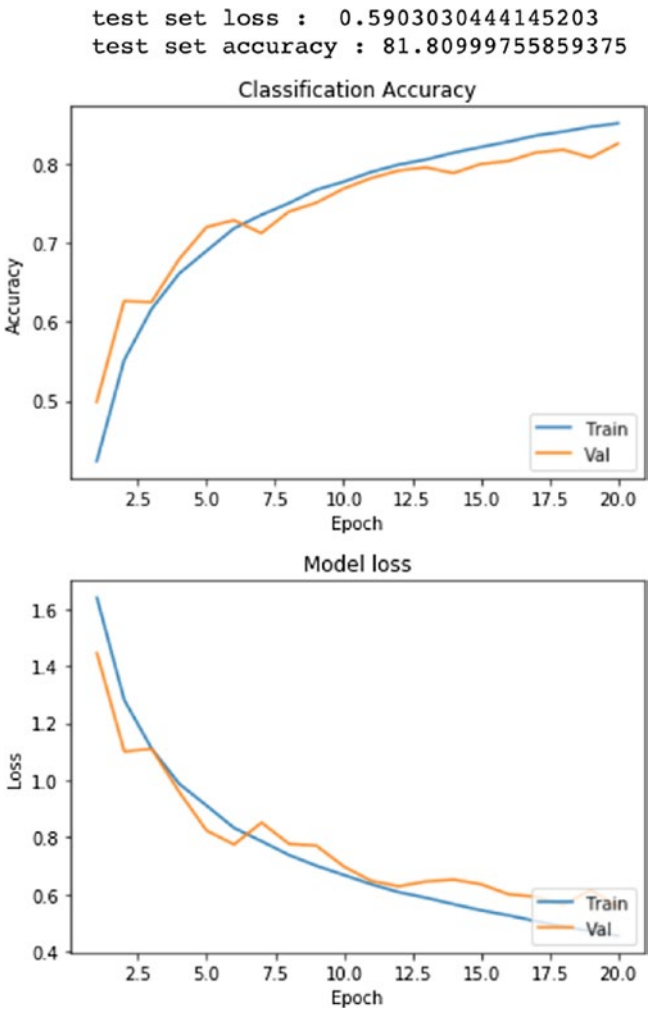


Figure 3-26. *Model 5 error metrics*

The prediction results for the same image used in earlier examples are as follows:

```
Top predictions of these images are
airplane : 83.3
cat : 5.25
ship : 4.34
```

The accuracy is now improved to 81.80%, a great improvement over all other earlier designs. The comparison of the five different models is given in Table 3-2.

Table 3-2. *Performance comparision on different models*

Modules	Model 1	Model 2	Model 3	Model 4	Model 5
Loss	1.4560	2.8526	1.4121	0.6292	0.5903
Accuracy	66.4499	69.6500	70.1300	78.2400	81.8099

Clearly, the model 5 gives us the best accuracy amongst all the models that we tested. Also, notice that model 3 and model 2 provide the same level of accuracy indicating that merely adding a greater number of convolutional layers does not help in improving the model's accuracy.

Saving Model

As you have developed models with different levels of accuracy, you can select the appropriate one for your production use. For this, you will be required to save the model to a file. Earlier, I have discussed the various ways of saving models along with their attributes and states.

You may save any of the above models to disk by calling the save method on the model instance as shown in the code here:

```
model_5.save("model_5.h5")
```

The model is saved to HDF5 format and can be reloaded any time by using the following statement:

```
m = load_model("model_5.h5")
```

The newly loaded model can then be used for predicting any unseen image

Predicting Unseen Images

Now, we will use our saved model to predict three unseen images of different sizes. The three images are shown in Figure 3-27.

Figure 3-27. *Unseen images for testing*

Note that all the images are not of square size. Our model was trained on square images of size 32x32 and require inputs of the same size. Let us try to see if the model predicts the objects in these images accurately. Use the following code fragment to load the image from the project GitHub and call our predict_class function to make the predictions.

```
# unseen image 1
resource = urllib.request.urlopen("https://raw.
githubusercontent.com/Apress/artificial-neural-networks-with-
tensorflow-2/main/ch03/unknown01.png")
output.write(resource.read())
output.close()
predict_class("/content/unknown01.jpg", m)

# unseen image 2
resource = urllib.request.urlopen("https://raw.
githubusercontent.com/Apress/artificial-neural-networks-with-
tensorflow-2/main/ch03/unknown02.png")
output = open("/content/unknown02.jpg","wb")
output.write(resource.read())
```

```
output.close()
predict_class("/content/unknown02.jpg", m)

# unseen image 3
resource = urllib.request.urlopen("https://raw.
githubusercontent.com/Apress/artificial-neural-networks-with-
tensorflow-2/main/ch03/unknown03.png")
output = open("/content/unknown03.jpg","wb")
output.write(resource.read())
output.close()
predict_class("/content/unknown03.jpg", m)
```

The predictions made by the model on the three images are given below:

```
Top predictions of these images are
airplane : 98.56
bird : 0.66
deer : 0.64

Top predictions of these images are
automobile : 99.94
airplane : 0.06
truck : 0.0

Top predictions of these images are
bird : 99.9
cat : 0.08
airplane : 0.01
```

You see in all three cases our model has predicted the object correctly with almost 99% accuracy. Congratulations! You have learned to develop a ML model using tf.keras for image classification. The full source for this project is available in the project's download.

So far in this and the last chapter, you have created models of your own, which is definitely a time-consuming process that requires lots of efforts too. So, why not just reuse models trained by others on your own datasets for your needs? And that's what the next chapter covers. So, keep reading!

Summary

The Keras API is now fully implemented in TF and is accessed through tf.keras module. With the functional API provided in tf.keras you are now able to create complex network architectures having multiple inputs/outputs, non-linear topology, models with shared layers, and so on. You learned how to define such architectures using pre-defined and custom layers. You also learned about how to subclass a model for object-oriented coding. You learned how to save a model, its state and weights. To apply these learnings, the chapter covered a complete example on image classification using CNN. With a brief theory behind CNN, the application develops a full-fledged application for image classification based on CIFAR-10 dataset. You started out with a simple model and kept improving its performance by applying different techniques such as increasing the number of Conv2D layers, adding dropouts, BatchNormalization and Regularization. You learned how to save the model and use it later for inference on unseen images.

CHAPTER 4

Transfer Learning

In the previous chapter, you developed a binary image classifier. With 60,000 images, it took a while to train the model. The accuracy we achieved was about 80 to 90%. If you want a higher accuracy, more images would be required for training. As a matter of fact, a deep learning network learns better with a higher number of data points. The ImageNet (https://devopedia.org/imagenet), the first of its kind in terms of scale, consists of 14,197,122 images organized into 21,841 subcategories which are further classified into 27 subtrees. To classify the images in ImageNet, many machine learning models were developed; these were developed mainly as a way of research and competitions. In 2017, one such model achieved an error rate of as low as 2.3%. The underlying network was very complex. Considering the number of trainable points for such a complex network, imagine the number of resources and time it would have taken to train the model.

The question now is can we reuse whatever these networks have learned and gain from their knowledge to our own benefits? And exactly, that's what this chapter is all about. The technology behind this is called transfer learning. It is like human beings transferring their learnings to younger generations. In this chapter, you will learn how to transfer the learnings of other networks to your own benefits and develop your own new models based on their work.

To be concise, this chapter covers the following:

- What do we mean by knowledge transfer?
- What is TensorFlow Hub?

P. Sarang, *Artificial Neural Networks with TensorFlow 2*,
https://doi.org/10.1007/978-1-4842-6150-7_4

- What pre-trained models are available?
- How to use a pre-trained model?
- How transfer learning techniques are used for creating a dog breed classifier?

So, let us start with the question: What is knowledge transfer?

Knowledge Transfer

Last several decades, developers have been using binary libraries for reusing their own code or the code shared by others. In machine learning, can we adapt the same concept by reusing the models developed by others for our own advancements? It is not as simple as it may sound. In software libraries, it is only the code that gets shared. In machine learning, it is more than the code that needs sharing. The four important ingredients of a trained model are listed here:

- Algorithm
- Data
- Training
- Expertise

Firstly, it is the algorithm that is developed by somebody to train a neural network. This is the code part of the knowledge transfer. The second is the data. Generally, a model learns with the help of a huge set of data points. The third is the training – to train a model you require tremendous processing power and time. Lots of resources are consumed while training a network. Lastly, it is the knowledge and expertise of a domain expert that gets inherently embedded in a trained model. So, how do we transfer these four aspects to a new model? To facilitate this, researchers at TensorFlow were inspired to develop a TensorFlow Hub.

TensorFlow Hub

TensorFlow Hub is a platform provided by TensorFlow where the pre-trained models are published. Other users can discover and reuse parts of the machine learning modules from these models. So, what's a module? A module is visualized as depicted in Figure 4-1.

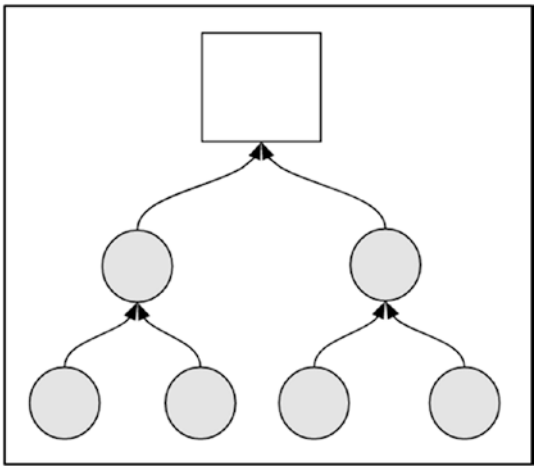


Figure 4-1. *Module graph*

A module is essentially a self-contained piece of a TensorFlow graph, along with its training weights. This graph can be reused elsewhere to perform similar tasks. A developer can add more layers to this graph to create their own model. The new model can then be trained using a smaller dataset that can result in faster training. In a way, this also results in generalization where the generalized modules can be reused across several models. You may think of a model like a binary which is the final executable in software engineering and a module like a generalized library used for creating executables.

Modules are composable as depicted in Figure 4-2.

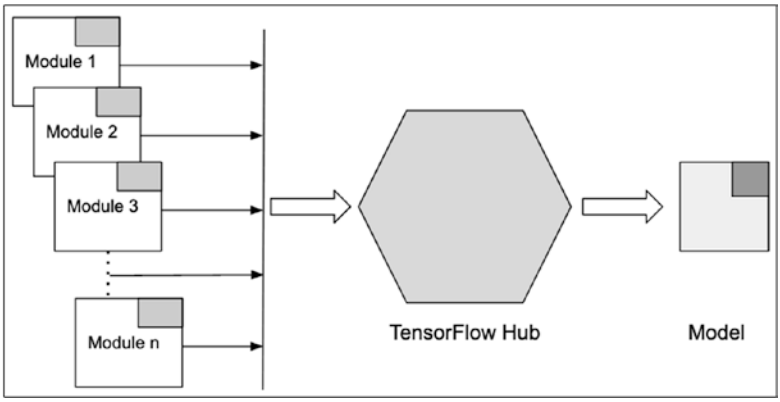


Figure 4-2. *TensorFlow Hub architecture*

You can add your own layers on top of the pre-trained modules to create your own network. The entire network can then be trained including the embedded pre-trained modules.

Modules are reusable and retrainable with a single parameter passed to a function call. When I say they are retrainable, it means that you can back-propagate through them just like a usual neural network. Note that if you do retraining, ensure a low learning rate; otherwise, the existing weights may go haywire, producing totally unexpected results.

When you create your own model and if you feel that the model could be used by the community, you may submit it to Google for inclusion in TensorFlow Hub. There are many third-party modules currently available in the Hub. Google warns you to use a third-party module with caution, especially if you do not trust the source.

So, what modules are available for your use today?

Pre-trained Modules

There are several pre-trained modules provided by Google and the channel partners in TensorFlow Hub. These are classified in three categories – Image, Text, and Video. One of the categories – Image domain – is shown in Figure 4-3.

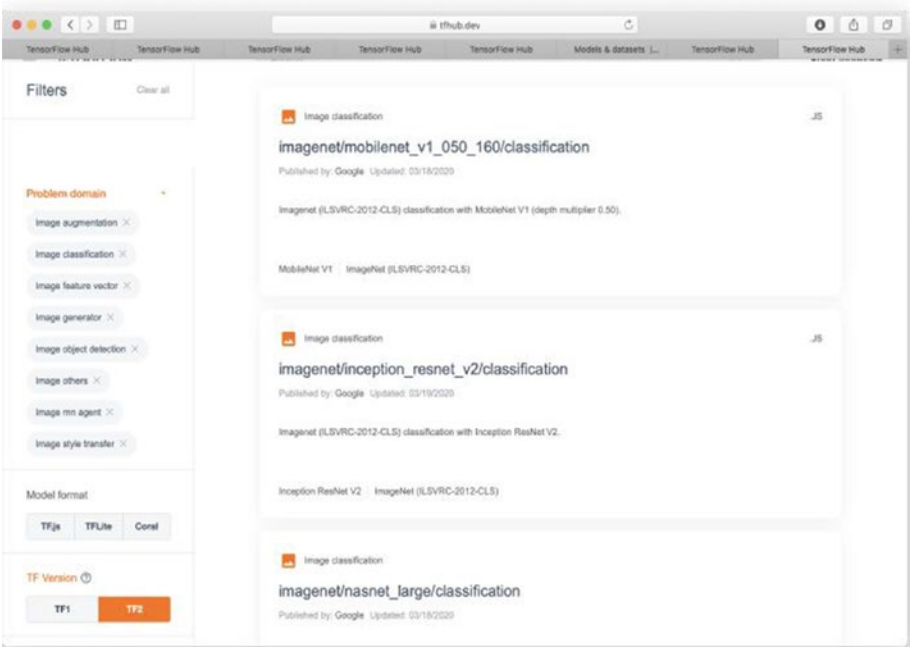


Figure 4-3. *Predefined models in TensorFlow*

Some of the modules in the Image domain are

- ImageNet classification with MobileNet/Inception ResNet V2/NASNet-A
- Feature vectors of images with MobileNet/Inception/ ResNet/PNASnet

The abovementioned modules support TF2. There is an exhaustive list of modules developed by Google and other partners that run under TF1 and are waiting for migration to TF2. Some of the modules under this TF1 category that you may find of your interest are

- Progressive GAN
- Fast arbitrary image style transfer
- Image augmentation module performing random crop, small rotation, and color distortions
- PlaNet that predicts the rough geolocation of arbitrary photos
- Google landmarks - deep local features (DELF)

Under the Text category, you will find modules such as

- ELMO
- BERT
- Universal Sentence Encoder
- Token-based text embedding trained on English Wikipedia corpus

In the Video category, as of this writing, there are no models available for TF2. Under TF1, there are a few models created by Google and DeepMind.

Once again, the total number of modules available under the TF1 category is exhaustive. You may see the entire list of modules available by visiting the TensorFlow Hub site (`https://tfhub.dev/`).

Now, you know what is available for your use. The next question is how to use such a pre-trained module.

Using Modules

To use a pre-trained module in your program code, select the desired module from the tfdev website. Look for the Model format; under the Saved Model, you will find the module's URL as shown in the screenshot in Figure 4-4.

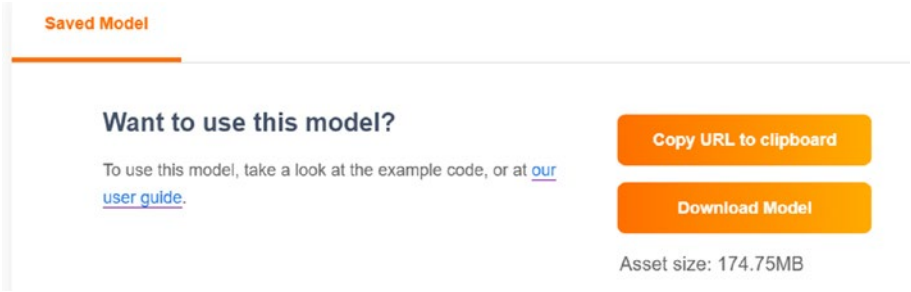


Figure 4-4. *Module's URL*

Copy the URL and use it in your program code as shown here:

```
module_url = "https://tfhub.dev/google/imagenet/mobilenet_
v2_100_160/feature_vector/4"
my_model = hub.KerasLayer(module_url)
```

You may then add more layers to my_model. Train your new model using the fit method, just the way you train your other models. Once trained, use it for your predictions.

Having seen the capabilities of pre-trained modules, it is time to try it out.

ImageNet Classifier

In this project, you will use an ImageNet classifier called mobilenet_v2 provided by Google. The classifier provides 1001 different classifications. The model was trained on more than one million images. It used the initial learning rate of 0.045 and learning decay rate of 0.98 per epoch. It used 16 GPU asynchronous workers and a batch size of 96. You can imagine the number of resources involved in creating this model. You will use the transfer learning to transfer this knowledge into building your own image classifier on top of this. You will be doing this over the next two projects in this chapter. The first one will teach you how to use the MobileNet model to classify your own images. The second one will show you how to expand it to add more classifications of your own.

First, I will show you how to use the ImageNet classifier to classify your own images. Essentially, in this project, you will use the MobileNet classifier as is without adding any layers to it. You will just learn to load a pre-trained model and use it on your own images. The later project (dog breed classifier) in this chapter will go into lots of detail where you would be adding your classification layer to the ImageNet module and do lots of studies to learn the benefits offered by transfer learning.

So, start by creating a project called ImageNetClassifier in Colab.

Setting Up Project

Set up TensorFlow and import the required packages using the following code segment:

```
import tensorflow as tf

# other imports
import tensorflow_hub as hub
import numpy as np
import matplotlib.pyplot as plt
```

Note you need to import tensorflow_hub. The other two imports numpy and matplotlib serve their usual purpose; you have used them in your earlier projects.

Classifier URL

We will be using the pre-trained MobileNet classifier for this project. Locate this classifier on the TensorFlow Hub. The screenshot of a search done on the tfhub.dev site is shown in Figure 4-5.

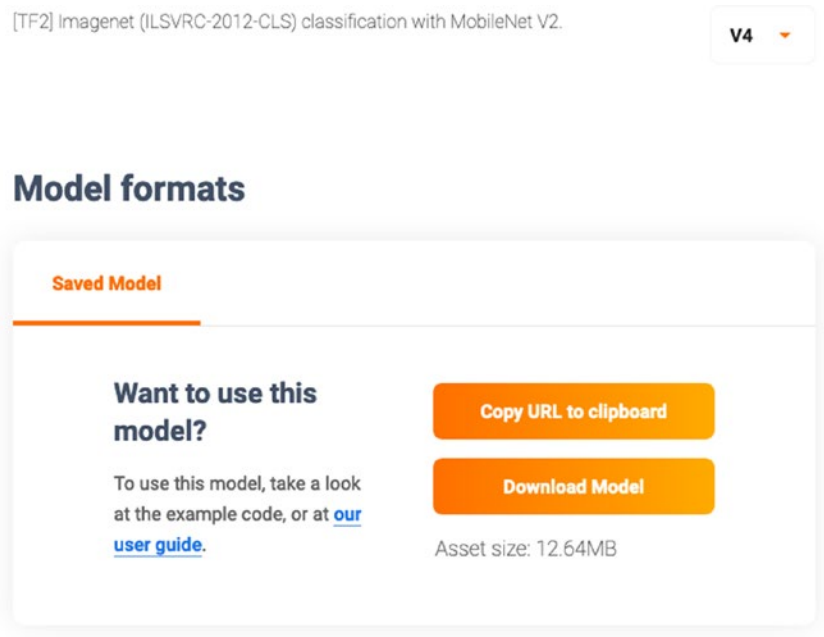


Figure 4-5. *MobileNet model on TensorFlow Hub*

Copy the model's URL by clicking the "Copy URL" button. You will get the following URL:

```
https://tfhub.dev/google/tf2-preview/mobilenet_v2/
classification/4
```

Note you may get a later version than the one shown here.

Declare two variables in your project code as follows:

```
classifier_url = "https://tfhub.dev/google/tf2-preview/
mobilenet_v2/classification/4"
IMAGE_SHAPE = (224, 224)
```

The input to this model requires images of 224x224 pixels, as described in the model's description. The model's description is available in the user guide, which you can read by clicking the "our user guide" link shown in Figure 4-5.

Creating Model

Next, you create the model by using the Sequential API as follows:

```
# create a model
classifier = tf.keras.Sequential([
    hub.KerasLayer(classifier_url, input_shape = IMAGE_SHAPE+(3,))
])
```

The hub.KerasLayer returns the module that is added to our sequential model. All the layers included in this module are now available to our network. We do not add any more layers of our own to this model. So, our model definition is now complete. As this is a fully trained model, we do not need to call the fit method on our classifier for training it further. You will simply call its predict method to infer any given input image. Before you call predict, let us prepare a few images for inputting to our model.

Preparing Images

I have uploaded a few images on the book's download site. The URLs for these images are listed here. Add this code into your project:

```
# set up URLs for image downloads
image_url1 = "https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/Ch04/bulck_cart.jpg"
image_url2 = "https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/Ch04/flower.jpg"
image_url3 = "https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/Ch04/swordweapon.jpg"
image_url4 = "https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/Ch04/tiger.jpg"
image_url5 = "https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/Ch04/tree.jpg"
```

You will download these images to your drive using the following code:

```
# download images
!pip install wget
import wget
wget.download(image_url1,'image1.jpg')
wget.download(image_url2,'image2.jpg')
wget.download(image_url3,'image3.jpg')
wget.download(image_url4,'image4.jpg')
wget.download(image_url5,'image5.jpg')
```

We use wget for downloading the five image files. These are stored in the `/content/` folder of your drive with the names image1, image2, and so on.

As these images have varied sizes, you need to resize them to 224x224 after loading them into the memory. You also need to rescale the data in the range 0 to 1 for better machine learning. All these things are done in the following code:

```
# load images and reshape to 224x224 required by the model
import PIL.Image as Image

image1 = tf.keras.utils.get_file("/content/image1.jpg",
image_url1)
image1 = Image.open(image1).resize(IMAGE_SHAPE)
# load images and reshape to 224x224 required by the model
import PIL.Image as Image

image1 = tf.keras.utils.get_file("/content/image1.jpg",
image_url1)
image1 = Image.open(image1).resize(IMAGE_SHAPE)
# scale the array
image1 = np.array(image1)/255.0

image2 = tf.keras.utils.get_file("/content/image2.jpg",
image_url2)
image2 = Image.open(image2).resize(IMAGE_SHAPE)
image2 = np.array(image2)/255.0

image3 = tf.keras.utils.get_file("/content/image3.jpg",
image_url3)
image3 = Image.open(image3).resize(IMAGE_SHAPE)
image3 = np.array(image3)/255.0

image4 = tf.keras.utils.get_file("/content/image4.jpg",
image_url4)
image4 = Image.open(image4).resize(IMAGE_SHAPE)
image4 = np.array(image4)/255.0
```

```
image5 = tf.keras.utils.get_file("/content/image5.jpg",
image_url5)
image5 = Image.open(image5).resize(IMAGE_SHAPE)
image5 = np.array(image5)/255.0
```

Now, you are ready to infer the images. So, let us start with the first image. You infer the image by calling the predict method of the model:

```
result = classifier.predict(image1[np.newaxis, ...])
```

The probability predictions are now available in the result tensor. Print the shape of the result using the shape method:

```
result.shape
(1, 1001)
```

You see that there are 1001 values in the result array indicating that there are 1001 classes in the output. The probability predictions are arranged in an ascending order. So pick up the last one as the highest prediction made by our model.

```
predicted_class = np.argmax(result[0], axis=-1)
predicted_class
293
```

The predicted class as you can see is 293. This integer does not make any sense to us. Fortunately, the mapping of these integers to the labels is provided by Google.

Loading Label Mappings

The label names are available on the Google site in the file ImageNetLabels.txt which is loaded into our program using the following code:

```
labels_path = tf.keras.utils.get_file('ImageNetLabels.
txt','https://storage.googleapis.com/download.tensorflow.org/
data/ImageNetLabels.txt')
imagenet_labels = np.array(open(labels_path).read().
splitlines())
```

Print the label names and array's length using the following two print statements:

```
print (imagenet_labels)
print ("Number of labels: " , len(imagenet_labels))
```

```
['background' 'tench' 'goldfish' ... 'bolete' 'ear' 'toilet
tissue']
Number of labels:  1001
```

The output shown on the next two lines lists a few names at the start and end of the array. It also tells you that there are 1001 names, each one for the output class of our model.

Displaying Prediction

To display the prediction, we plot the image and the predicted class name using the following code:

```
plt.imshow(image1)
plt.axis('off')
predicted_class_name = imagenet_labels[predicted_class]
_ = plt.title("Prediction: " + predicted_class_name.title())
```

The result is shown in Figure 4-6.

Figure 4-6. *Test image with predicted title*

As you can see, our model predicted the image correctly, saying that it is a tiger. Now, do the predictions on other four images by writing the code similar to what you did for image1.

For this, you may use the predict and display image function shown here:

```
def predict_display_image(imagex):
  result = classifier.predict(imagex[np.newaxis, ...])
  predicted_class = np.argmax(result[0], axis=-1)
  plt.imshow(imagex)
  plt.axis('off')
  predicted_class_name = imagenet_labels[predicted_class]
  _ = plt.title("Prediction: " + predicted_class_name.title())
```

The function receives a preprocessed image as an argument, infers it, and prints the prediction result along with the image itself.

You will call this function on the four remaining images as follows:

```
# predict and print results for the specified image
predict_display_image(image2)
predict_display_image(image3)
predict_display_image(image4)
predict_display_image(image5)
```

The four prediction results are shown in Figure 4-7.

Figure 4-7. *Predictions for the other four images*

Before I discuss the output in Figure 4-7, let us first look at the list of classes that the ImageNet classifier classifies.

Listing All Classes

To see the list of all classes predicted by ImageNet, use the following code:

```
for i in range (len(imagenet_labels)):
 print (imagenet_labels[i])
```

The first few items from the output are shown here:

```
background
tench
goldfish
great white shark
```

```
tiger shark
hammerhead
electric ray
stingray
cock
Hen
```

Now, I will discuss the results shown in Figure 4-7.

Result Discussions

The first picture in the output is interpreted as a cart, which is correct. The second picture is interpreted as a letter opener, which is close in looks to a sword. Note the actual picture is of the sword. A letter opener resembles in looks to a sword. So, this inference may be considered okay. The third picture is a tree, which is interpreted as a pot. The last one which is a bunch of roses is interpreted as face powder – absolutely incorrect. So, what do we derive from this discussion? To differentiate between a Teddy and a bunch of roses, the ImageNet would require further training with more images. This is obviously beyond our reach. However, we will be able to extend the ImageNet to add our own classifications. And that's what I am going to show in the next topic.

Dog Breed Classifier

Consider you have a dataset of dog pictures with its breed as the label. You may not be in a position to collect enough pictures in each class. You want to develop a model to classify the dogs as per their breed. You can think of using the ImageNet classifier discussed in the previous program to extract the features of a dog image. You will then add a layer on top of this to differentiate dogs based on these extracted features. In short, you will transfer the learning of ImageNet to classify the dog types and save the trouble of training your model for feature extractions.

Image classification models have millions of parameters. Training them from scratch is usually a challenging task. It requires lots of training data and is computationally expensive. Transfer learning shortens this task by taking a piece of a model that has already been trained. You simply need to add your classification layers on top of it.

This project will demonstrate how to build a Keras model for classifying the breed of dogs using a pre-trained TF2 saved model from TensorFlow Hub MobileNet V2 which takes an input size of (224,224,3) for image feature extraction followed by the output Dense layer of our model for classification using softmax.

Let us start building the project.

Project Description

The dog breed project will use transfer learning to define a new model for classifying different breeds of dogs. This is a multiclass classifier that classifies the given dog image into several predefined classes. Differentiating between a multiclass and a binary classification, one can say, a dog vs. cats or a human being vs. a horse is the binary classification. Tesla uses a multiclass image classification in their self-driving cars.

For this project, we will use a dataset from the Kaggle dog breed identification competition (https://www.kaggle.com/c/dog-breed-identification/overview.) It consists of a collection of 10,000+ labeled images of 120 different dog breeds. We will need to preprocess the data by converting the image data into Tensors. Our machine learning model will find patterns in the input Tensors. Kaggle, being a competition, has provided the training and testing data separately. Obviously, the test data does not have labels.

Creating Project

Create a new Colab project and rename it as DogBreedClassifier. Import TensorFlow and TensorFlow Hub using the following two statements:

```
import tensorflow as tf
import tensorflow_hub as hub
```

Next, you will load data into the project.

Loading Data

I have kept the entire data (source: www.kaggle.com/c/dog-breed-identification/data) on the project site so as to enable you to run the project in one go without worrying about downloading the data separately. To download the data into your project, run the following code fragment:

```
! wget --no-check-certificate -r 'https://drive.google.com/
uc?export=download&id=11t-eBwdXU9EWriDhyhFBuMqHYiQ4gdae' -O
dogbreed
```

Note that in the preceding download, the download site is assumed to be trusted. If you have any doubts, you may download the file with your own tools and include the appropriate path in the project for inputting to the model.

The downloaded zip file contains the labels and the images for training and testing. The structure of the zip file is shown in Figure 4-8.

```
.
├── labels.csv
├── sample_submission.csv
├── test
│   ├── 00a3edd22dc7859c487a64777fc8d093.jpg
│   ├── 00a558277e1f03b71d8c813e03344ddf.jpg
│   ├── 00a6892e5c7f92c1f465e213fd904582.jpg
│   ├── 0a01f3b0c6d250c08785716c562e2d4d.jpg
│   ├── 0a053ff8908375e9579ead9f3965e0ce.jpg
│   ├── 0a0b97441050bba8e733506de4655ea1.jpg
│   ├── 0a0f82d4557f70718efa26f8a8a7773f.jpg
```

Figure 4-8. *Folder structure for downloaded data*

Unzip the downloaded file using the command:

```
!unzip dogbreed
```

The zip file also contains the labels for the training images. You can check the labels by loading the labels.csv file and printing the first five records:

```
# Checkout the labels
import pandas as pd
labels_csv = pd.read_csv("/content/labels.csv")
labels_csv.head()
```

This will show the output given in Figure 4-9.

	id	breed
0	000bec180eb18c7604dcecc8fe0dba07	boston_bull
1	001513dfcb2ffafc82cccf4d8bbaba97	dingo
2	001cdf01b096e06d78e9e5112d419397	pekinese
3	00214f311d5d2247d5dfe4fe24b2303d	bluetick
4	0021f9ceb3235effd7fcde7f7538ed62	golden_retriever

Figure 4-9. *A few labels*

To describe the table's contents, call the describe method on the dataframe. The command and its output are shown in Figure 4-10.

```
2  labels_csv.describe()
```

	id	breed
count	10222	10222
unique	10222	120
top	f679cd9c45865bd983920f79a2d85de3	scottish_deerhound
freq	1	126

Figure 4-10. *Labels' table structure*

As you can see from the output, there are 10,222 images and 120 categories. To print the distribution of data, use the following statement:

```
# How many images are there of each breed?
labels_csv["breed"].value_counts().plot.bar(figsize=(20, 10));
```

The execution will produce the chart shown in Figure 4-11.

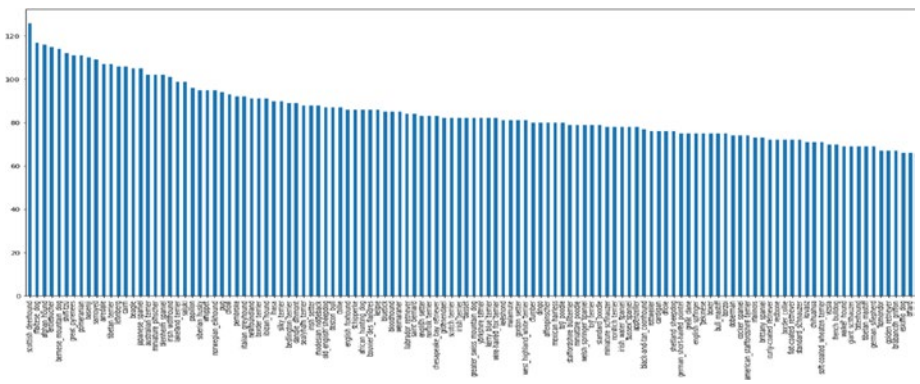


Figure 4-11. *Image data distribution*

As you can see in Figure 4-11, each category has more than 60 images. Google recommends a minimum of ten images per class to get started with image classification. The more the number of images, the better are the chances of figuring out the patterns between them.

You may now try printing a sample image from the dataset using the following command:

```
from IPython.display import display, Image
Image("/content/train/000bec180eb18c7604dcecc8fe0dba07.jpg")
```

The output is shown in Figure 4-12.

Figure 4-12. *Sample image from the downloaded dataset*

Now, you have all the data for training and testing your model. This data needs to be converted to a specific format for inputting to our model.

Setting Up Images and Labels

We will now set up the paths to images for preparing data for training. This is done using the following two program statements:

```
# Define our training file path for ease of use
train_path = "/content/train"
```

```
# Create path names from image ID's
filenames = [train_path + '/'+fname + ".jpg" for fname in
labels_csv["id"]]
```

You can verify that the file path is set correctly by printing one of the images as follows:

```
# Check an image directly from a file path
Image(filenames[9000])
```

You should see the image as shown in Figure 4-13.

Figure 4-13. *Sample image from the new path*

Now that we have set up the images for processing, we will set up the labels array. We will read the labels from labels_csv and convert them into numpy array.

```
import numpy as np
labels = labels_csv["breed"].to_numpy()
```

As we know, the number of labels is 10,222. We need to extract the unique values from these labels, which we do using the following code:

```
unique_breeds = np.unique(labels)
len(unique_breeds)
```

The output is 120 indicating that there are 120 classes in our dataset. In other words, there are 120 breeds of dogs which are classified. You can print this list by calling the list method:

```
# print class names
list(unique_breeds)
```

The partial output is shown in Figure 4-14.

```
['affenpinscher',
 'afghan_hound',
 'african_hunting_dog',
 'airedale',
 'american_staffordshire_terrier',
 'appenzeller',
 'australian_terrier',
 'basenji',
 'basset',
 'beagle',
 'bedlington_terrier',
 'bernese_mountain_dog',
 'black-and-tan_coonhound',
 'blenheim_spaniel',
 'bloodhound',
 'bluetick',
```

Figure 4-14. *Class (labels) list*

Next, encode target labels with values between 0 and n_classes-1, using the following code:

```
# Encode target labels with values between 0 and 120
from sklearn.preprocessing import LabelEncoder
labels = LabelEncoder().fit_transform(labels).reshape(-1,1)
labels
```

The output is shown in Figure 4-15.

```
array([[19],
       [37],
       [85],
       ...,
       [ 3],
       [75],
       [28]])
```

Figure 4-15. *Encoded labels*

As you can see from Figure 4-15, the first dog is classified as #19, the second one is #37, and so on. You now use OneHotEncoder to transform these categorical values into columns for model training.

```
# Use one-hot encoding to convert categorical values
from sklearn.preprocessing import OneHotEncoder
boolean_labels = OneHotEncoder().fit_transform(labels).
toarray()
```

You may see the effect of one-hot encoding by printing one of the values in the array as shown here:

```
Boolean_labels[5]
```

The output of the execution is shown in Figure 4-16.

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0.])
```

Figure 4-16. *A typical label data after one-hot encoding*

Notice that you have now 120 fields added after the encoding is done. For each column (field), the value is 0, except for the one where the value is 1. This is the class for the given image.

Now that we have preprocessed the labels and made it ready for machine learning, we will preprocess the images and make them ready for learning by our model.

Preprocessing Images

You have so far converted labels into numeric format. However, the images are still just the filenames. You must read the image data and convert it to a format suitable for inputting to the model. We will represent the image data in the form of Tensors for faster processing using GPUs. To do so, we will write a function that does the following:

1. Take an image filename as input
2. Load the image (a jpeg file) in its binary format
3. Turn the image data into Tensors
4. Resize the image to be of shape (224, 224)
5. Return the image Tensor

Why do we need to resize the image to 224x224? Remember that the MobileNet model requires an image input of shape (224, 224, 3) where 3 represents the RGB dimension. So, let us look at the various steps.

We first set up variables for our features and labels:

```
# Setup variables
X = filenames
y = boolean_labels
```

Next, we split the training dataset into training and validation:

```
from sklearn.model_selection import train_test_split
# Split them into training and validation
X_train, X_val, y_train, y_val = train_test_split(
    X, y, test_size=0.2, random_state=42)
```

Out of the 10,222 images in the training folder, 2045 images would now be used for validation.

Now, you will write a function for preprocessing the image, that is, to load the image from a given path, read its data, and convert it into a Tensor.

Processing Image

We first define a variable for the size of the image:

```
IMG_SIZE = 224
```

We define the function called process_image as follows:

```
def process_image(image_path):
```

The function takes one argument, which is the path to the image file. To read the image, you use the read_file function of tf.io.

```
image = tf.io.read_file(image_path)
```

The read_file function reads the data from the jpeg image into a binary array. The data read is decoded using the decode_jpeg method of the tf.image.

```
image = tf.image.decode_jpeg(image, channels=3)
```

The image data is then converted to a float value by calling the convert_image_dtype method of tf.image.

```
image = tf.image.convert_image_dtype(image, tf.float32)
```

We resize the image by calling the resize function of tf.image.

```
image = tf.image.resize(image, size=[IMG_SIZE, IMG_SIZE])
```

Finally, the function returns the processed image data to the caller.

```
return image
```

The entire function code for process_image is given in Listing 4-1.

Listing 4-1. The process_image function

```
# Define image size
IMG_SIZE = 224

def process_image(image_path):
 """
 Takes an image file path and turns it into a Tensor.
 """
 # Read image file
 image = tf.io.read_file(image_path)
 # Turn the jpeg image into numerical Tensor
 image = tf.image.decode_jpeg(image, channels=3)
 # Convert the color channel values from 0-225 values to 0-1
values
 image = tf.image.convert_image_dtype(image, tf.float32)
 # Resize the image to our desired size (224, 244)
 image = tf.image.resize(image, size=[IMG_SIZE, IMG_SIZE])
 return image
```

Next, we will write a function to associate the image with a label.

Associating Labels to Images

We will write a function that takes an image path as an argument, process an image with a call to our process_image function, and associate a label with it. The function definition is given here:

```
# Create a simple function to return a tuple (image, label)
def get_image_label(image_path, label):
 """
 Takes an image file path name and the associated label,
 processes the image and returns a tuple of (image, label).
 """
 image = process_image(image_path)
 return image, label
```

The function returns a tuple containing an image in Tensor form and its associated label.

We will now write a function to convert our data into an input pipeline.

Creating Data Batches

Let us first understand what a batch is. A batch is a small portion of your data – images and their labels. Typically, a batch is of size 32; it means there are 32 images and 32 corresponding labels in a batch. In deep learning, instead of finding patterns in an entire dataset at the same time, you often find them in one batch at a time. Loading and processing 10,000+ images in one go would require a huge amount of memory and processing power. So, we process them in small batches, one at a time. We will now write a function to divide our data into batches and create Tensors consisting of image data and its associated label.

The function for creating batches is shown in Listing 4-2.

Listing 4-2. Function for creating data batches

```
# Define the batch size, 32 is a good default
BATCH_SIZE = 32

# Create a function to turn data into batches
def create_data_batches(x, y = None, batch_size = BATCH_SIZE,
data_type = 1):
 """
 Creates batches of data out of image (x) and label (y) pairs.
 Shuffles the data if it's training data but doesn't shuffle it
 if it's validation data.
 Also accepts test data as input (no labels).
 """
 # If the data is a test dataset, we don't have labels
 if data_type == 3:
   print("Creating test data batches...")
   data = tf.data.Dataset.from_tensor_slices((tf.constant(x)))
          # only filepaths
   data_batch = data.map(process_image).batch(BATCH_SIZE)
   return data_batch
  # If the data if a valid dataset, we don't need to shuffle it
 elif data_type == 2:
   print("Creating validation data batches...")
   data = tf.data.Dataset.from_tensor_slices((tf.constant(x),
                                              # filepaths
                                             tf.constant(y)))
                                              # labels
   data_batch = data.map(get_image_label).batch(BATCH_SIZE)
   return data_batch
```

```
else:
  # If the data is a training dataset, we shuffle it
  print("Creating training data batches...")
  # Turn filepaths and labels into Tensors
  data = tf.data.Dataset.from_tensor_slices((tf.constant(x),
                                             # filepaths
                                            tf.constant(y)))
                                             # labels

  # Shuffling pathnames and labels before mapping image
    processor function
  # is faster than shuffling images
  data = data.shuffle(buffer_size = len(x))

  # Create (image, label) tuples
  # (this also turns the image path into a preprocessed image)
  data = data.map(get_image_label)

  # Turn the data into batches
  data_batch = data.batch(BATCH_SIZE)

return data_batch
```

Depending on the input value, the function creates batches on training, validation, and testing datasets. In the training dataset, we shuffle the data to achieve some randomness in the data. The function from_tensor_slices converts the data into input pipelines. The map function converts the (image, label) tuples into a data input pipeline. The batch function splits the data into batches.

We will now write a function to display a few images from the set. This will be useful to us during testing.

Display Function for Images

We now write a function to display 25 images from the dataset. The full code for the function show_25_images is given in Listing 4-3. Assuming that you are familiar with matplotlib plotting, the code is self-explanatory and does not require any further comments.

Listing 4-3. The display image function

```
# Function for viewing images in a data batch
import matplotlib.pyplot as plt

def show_25_images(images, labels):
 """
 Displays 25 images from a data batch.
 """
 # Setup the figure
 plt.figure (figsize = (10, 10))
 # Loop through 25 (for displaying 25 images)
 for i in range(25):
   # Create subplots (5 rows, 5 columns)
   ax = plt.subplot(5, 5, i+1)
   # Display an image
   plt.imshow(images[i])
   # Add the image label as the title
   plt.title(unique_breeds[labels[i].argmax()])
   # Turn grid lines off
   plt.axis("off")
```

Having prepared ourselves with the various functions for data preprocessing and displaying, we are now ready to define our machine learning model.

Selecting Pre-trained Model

As said earlier, we will use the transfer learning in our model definition. For this, we need to look out for an appropriate model in TensorFlow Hub. The Hub classifies the models under different categories. If you select the image classification category, you will find several models listed under it. Some of these models are 1.x specific. As we are using TF2, select a model that supports TF2. One such model is mobilenet_v2_130_224 as of this writing. The screenshot of the selection is shown in Figure 4-17.

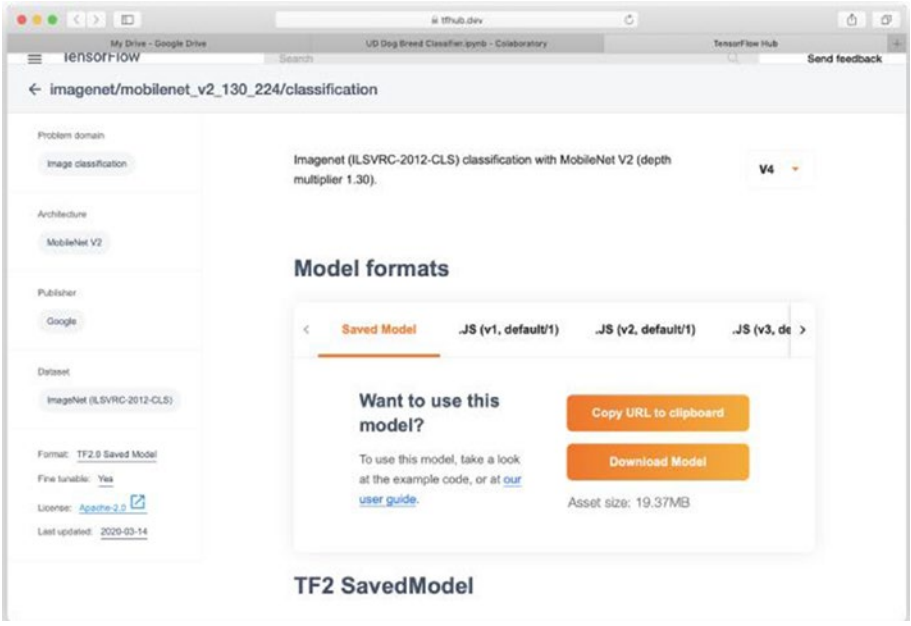


Figure 4-17. *Selecting MobileNet from tfhub*

Looking at the model documentation, you would discover that the model takes the input of shape (224, 224, 3).

Defining Model

To define a model, we need three important pieces of information:

- The input shape (images, in the form of Tensors)
- The desired number of classes (number of dog breeds)
- The URL of the pre-trained model that we want to use

We declare the variables for these using the following code segment:

```
# Setup input shape to the model
INPUT_SHAPE = [None, IMG_SIZE, IMG_SIZE, 3] # batch, height,
width, colour channels

# Setup output shape of the model
OUTPUT_SHAPE = len(unique_breeds) # number of unique labels

# Setup model URL from TensorFlow Hub
MODEL_URL = "https://tfhub.dev/google/imagenet/mobilenet_
v2_130_224/classification/4"
```

To define the model, we will write a function called create_model:

```
def create_model(input_shape=INPUT_SHAPE,
                 output_shape=OUTPUT_SHAPE,
                 model_url=MODEL_URL):
 print("Building model with:", MODEL_URL)
```

The create_model function takes the input shape, the output shape, and the pre-trained model URL as parameters. I have created a function

for model building so that later on you may experiment with different pre-trained models that may require different input and output shapes.

Next, we will use the Sequential API to define two layers in our model.

```
# Setup the model layers
 model = tf.keras.Sequential([
   hub.KerasLayer(MODEL_URL), # TensorFlow Hub layer
   tf.keras.layers.Dense(units=OUTPUT_SHAPE,
                         activation="softmax") # output layer
 ])
```

The first layer is the entire pre-trained model taken from the Hub. The second layer is the softmax classification layer that classifies the dogs into 120 categories that we are seeking.

Next, you will compile the model using the following statement:

```
# Compile the model
 model.compile(
     loss=tf.keras.losses.CategoricalCrossentropy(),
     optimizer=tf.keras.optimizers.Adam(),
     metrics=["accuracy"]
 )
```

We use CategoricalCrossentropy for our loss function and Adam optimizer. We capture the accuracy metrics to evaluate the model's performance.

We build the model by calling the build method that takes the input Tensor as its parameter:

```
  model.build(INPUT_SHAPE)
```

Finally, we returned the compiled model to the caller:

```
return model
```

The full function definition for create_model is given in Listing 4-4.

Listing 4-4. The create_model function

```
# Create a function which builds a Keras model
def create_model(input_shape=INPUT_SHAPE,
                 output_shape=OUTPUT_SHAPE,
                 model_url = MODEL_URL):
 print("Building model with:", MODEL_URL)
F
 # Setup the model layers
 model = tf.keras.Sequential([
   hub.KerasLayer(MODEL_URL), # TensorFlow Hub layer
   tf.keras.layers.Dense(units=OUTPUT_SHAPE,
                         activation="softmax") # output layer
 ])

 # Compile the model
 model.compile(
     loss=tf.keras.losses.CategoricalCrossentropy(),
     optimizer=tf.keras.optimizers.Adam(),
     metrics=["accuracy"]
 )

 # Build the model
 model.build(INPUT_SHAPE)

 return model
```

You can test this function by creating a model variable and printing its summary as follows:

```
model = create_model()
model.summary()
```

The model summary that is the output generated by the preceding code is shown in Figure 4-18.

```
Building model with: https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/classification/4
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
keras_layer (KerasLayer)     multiple                  5432713
_________________________________________________________________
dense (Dense)                multiple                  120240
=================================================================
Total params: 5,552,953
Trainable params: 120,240
Non-trainable params: 5,432,713
_________________________________________________________________
```

Figure 4-18. *Model summary*

Now that we have our model ready for training, let us create datasets for training it.

Creating Datasets

We have already split our training dataset into training and validation. We now just need to preprocess those using our previously defined create_data_batches function.

```
train_data = create_data_batches(X_train, y_train)
Val_data = create_data_batches(X_val,y_val)
```

If you just want to visually inspect a few records of the dataset, use our previously developed function show_25_images:

```
train_images, train_labels = next(train_data.as_numpy_
iterator())
show_25_images(train_images, train_labels)
```

You would see the output as shown in Figure 4-19.

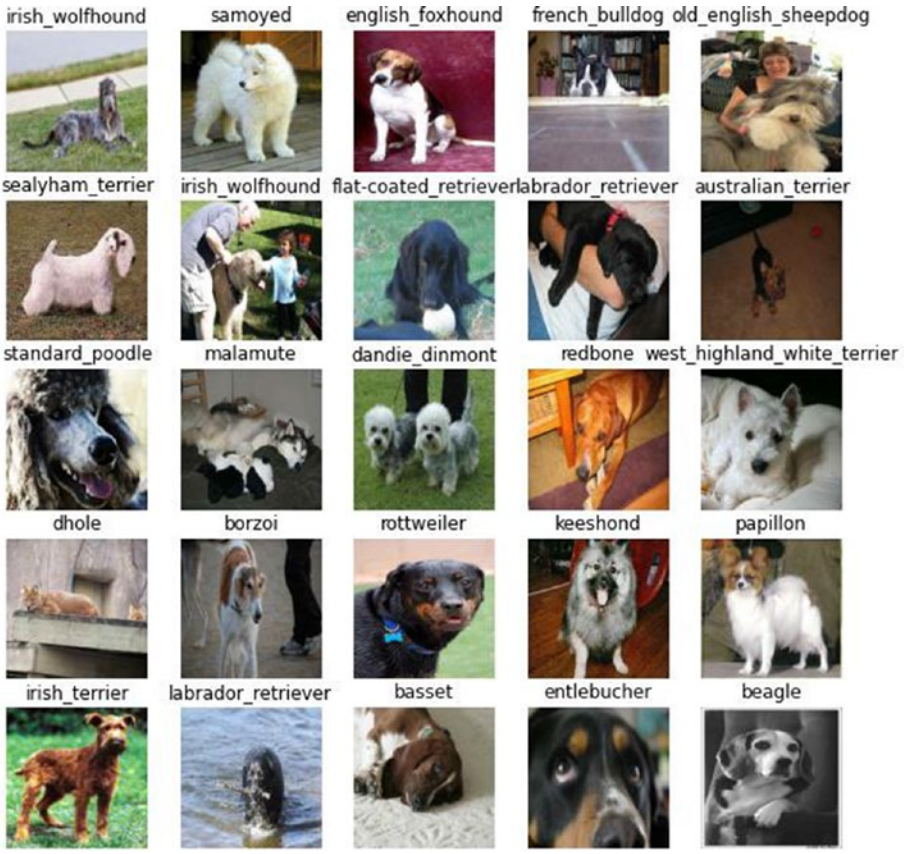


Figure 4-19. *Sample images from the dataset*

Note that in your case, the output may show different pictures because we do shuffle the training dataset on every run.

Our dataset for model training is now ready. We need to do just one last thing before training the model, and that is to set up TensorBoard for analytics.

Setting Up TensorBoard

First, you need to load the TensorBoard extension.

```
%load_ext tensorboard
```

Next, you will clean up the logs from the previous run, if any.

```
!rm -rf ./logs/  # cleaning the previous log
```

You will now create a callback function that will be called after every epoch.

```
import datetime
import os

# Create a function to build a TensorBoard callback
def create_tensorboard_callback():
 # Create a log directory for storing TensorBoard logs
 logdir = os.path.join("logs",
                           # Timestamp the log
 datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
 return tf.keras.callbacks.TensorBoard(logdir)
```

Note that in the function we create a logs directory and store the current time on each log.

We will use the EarlyStopping function to monitor the accuracy on the validation dataset. The following code will create two variables for logging and monitoring:

```
# TensorBoard callback
model_tensorboard = create_tensorboard_callback()

# Early stopping callback
model_early_stopping = tf.keras.callbacks.EarlyStopping(monitor
="accuracy",
# stops after 3 rounds
# of no improvements
                                patience=3)
```

We will pass these values to the callbacks parameter in the fit method. If, during training, the accuracy on validation dataset does not show much improvement over the last three cycles, defined by the patience parameter, the training would stop.

Now, we are ready to code for the actual training.

Model Training

We first build the model by calling our previously defined function create_model.

```
model = create_model()
```

We define a variable for the number of epochs used during training.

```
NUM_EPOCHS = 100
```

I have declared the number of epochs to be very large. I am going to demonstrate to you one important feature of TF2 that stops the training automatically when a saturation level is reached on the model's accuracy, indicating that further training is not going to help in getting any better accuracy. When you actually do the model training, you will notice that the training halts much earlier than the set 100 epochs.

We now start training by calling the fit method.

```
model.fit(x = train_data,
              epochs = NUMBER_OF_EPOCHS,
              validation_data = val_data,
              callbacks = [model_tensorboard,
                           model_early_stopping])
```

The training output is verbosed on your screen. When I ran the training, it stopped after 14 iterations. The screenshot of the training progress just before it stopped the training is shown in Figure 4-20.

```
Epoch 9/100
256/256 [==============================] - 31s 122ms/step - loss: 0.0355 - accuracy: 0.9976 - val_loss: 0.7079 - val_accuracy: 0.8098
Epoch 10/100
256/256 [==============================] - 31s 122ms/step - loss: 0.0294 - accuracy: 0.9980 - val_loss: 0.7175 - val_accuracy: 0.8147
Epoch 11/100
256/256 [==============================] - 31s 122ms/step - loss: 0.0237 - accuracy: 0.9993 - val_loss: 0.7229 - val_accuracy: 0.8127
Epoch 12/100
256/256 [==============================] - 35s 138ms/step - loss: 0.0231 - accuracy: 0.9984 - val_loss: 0.7452 - val_accuracy: 0.8161
Epoch 13/100
256/256 [==============================] - 36s 141ms/step - loss: 0.0180 - accuracy: 0.9990 - val_loss: 0.7534 - val_accuracy: 0.8122
Epoch 14/100
256/256 [==============================] - 36s 140ms/step - loss: 0.0160 - accuracy: 0.9991 - val_loss: 0.7820 - val_accuracy: 0.8122
<tensorflow.python.keras.callbacks.History at 0x7f12d8aa11d0>
```

Figure 4-20. *Training output just before EarlyStopping*

Note that on the 12/13/14 iterations, the accuracy on the validation dataset was 0.8161/0.8122/0.8122, indicating the saturation in training. Testing the accuracy on the validation data shows you how much your model works in general for cases outside the training set.

Examining Logs

You can now open TensorBoard in your Colab environment using the following magic command:

```
%tensorboard --logdir logs
```

The screenshot in Figure 4-21 shows the accuracy and loss plots for both training and validation data.

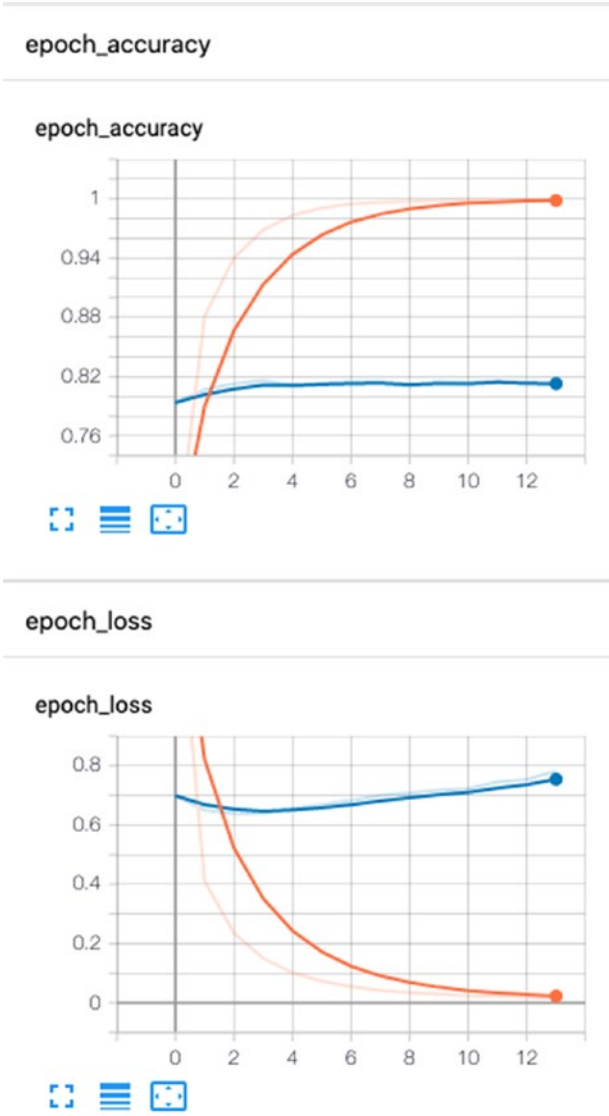


Figure 4-21. *Accuracy and loss metrics*

Evaluating Model Performance

You can evaluate the model's performance by calling its evaluate method.

```
model.evaluate(val_data)
```

In my run, it showed me an accuracy of 80.88% as seen in the output shown in Figure 4-22.

```
64/64 [==============================] - 7s 108ms/step - loss: 0.7830 - accuracy: 0.8122
[0.7830359155777842, 0.8122249]
```

Figure 4-22. *Model evaluation output*

Now, you will do predictions on test images to see if the model performs to our satisfaction.

Predicting on Test Images

The downloaded dataset contains the images for testing in a separate test folder. You just need to set up the path to these images and call our create_data_batches function to prepare a dataset for processing.

```
# set up path to test images
test_path = "/content/test"
test_filenames = [test_path +'/'+ fname for fname in
os.listdir(test_path)]
# prepare test dataset
test_data = create_data_batches(test_filenames, data_type = 3)
```

You make the predictions by calling the predict method on the trained model.

```
# Make predictions
test_predictions = model.predict(test_data,
                                 verbose=1)
```

When you run the preceding code, you will get an array of predictions. You can check the size of this array by printing its shape.

```
test_predictions.shape
```

You will get an output of (10357,120) indicating that 10,357 images were analyzed. Each item in an array is another array of 120 items. Each item indicates the probability of the type of dog with the corresponding index value. For example, the following line would print the predictions for the first image in the test data:

```
test_predictions[0]
```

The output in my case is shown in Figure 4-23.

```
array([1.71625504e-06, 2.09769784e-08, 4.42356409e-08, 3.60783137e-09,
       6.83360646e-10, 3.30916805e-08, 2.28535613e-09, 5.69529739e-08,
       1.12093312e-09, 1.28260762e-08, 8.87338480e-10, 1.12767259e-06,
       2.72967480e-08, 1.75928699e-06, 7.62196764e-11, 2.46849332e-08,
       2.15978602e-07, 2.86165952e-10, 3.18566293e-08, 5.58064350e-09,
       3.46805145e-08, 2.05301305e-08, 1.93678787e-07, 1.10779412e-08,
       4.09313365e-08, 5.94809269e-09, 2.17959713e-08, 5.79143240e-08,
       4.13444889e-10, 1.95744946e-07, 1.79607351e-09, 1.95080951e-09,
       1.01097652e-09, 6.40962469e-07, 4.43499848e-10, 2.93363597e-11,
       1.50097392e-08, 3.50538730e-11, 1.70451464e-09, 8.80133584e-08,
       1.56203242e-07, 7.27263028e-10, 9.05519479e-11, 8.22270516e-11,
       2.50760905e-07, 3.76196425e-08, 4.52130638e-10, 3.47058382e-09,
       1.23620314e-09, 3.41306472e-09, 7.42496198e-09, 4.11943439e-08,
       7.67591057e-09, 1.03463926e-09, 5.51114709e-09, 3.05814609e-08,
       7.21103248e-08, 5.35525457e-10, 3.54033114e-09, 5.31774624e-10,
       8.41837249e-08, 5.17785429e-06, 1.08501794e-07, 1.11155103e-08,
       4.11728540e-10, 1.00259268e-09, 2.02230996e-08, 7.91160970e-10,
       5.65018032e-11, 8.55700288e-10, 1.55159352e-09, 6.42869891e-09,
       4.19546797e-09, 3.46384006e-08, 5.19582727e-07, 2.08382822e-08,
       1.77174229e-07, 3.15960480e-09, 3.40442408e-09, 9.31709287e-09,
       1.71757530e-09, 1.13689913e-09, 8.17984847e-10, 1.01354489e-08,
       9.99965549e-01, 1.57419588e-08, 2.75883593e-07, 1.29112177e-05,
       3.02005532e-09, 9.60315050e-10, 4.79819462e-10, 2.03560968e-09,
       1.59662164e-07, 3.72057656e-08, 1.54300839e-09, 1.36166378e-07,
       4.08241085e-09, 2.83839636e-08, 1.39534592e-07, 2.19978847e-06,
       1.49835174e-07, 1.88222664e-08, 3.91109688e-06, 6.00249006e-09,
       5.31374695e-08, 2.38341569e-10, 1.84611926e-09, 1.26122968e-09,
       5.42101697e-10, 2.70874096e-07, 1.22152613e-07, 1.76050214e-08,
       2.92270439e-08, 4.73288964e-09, 2.40588860e-09, 6.44375362e-08,
       3.84022556e-08, 1.86857574e-09, 2.89643296e-08, 6.90489230e-07],
      dtype=float32)
```

Figure 4-23. *The 120 probabilities for the first image*

You can find out the maximum probability along with the index value at which this occurs by using the argmax function of numpy. The following code snippet would print the prediction for the first image:

```
# the max probability value predicted by the model
print(f"Max value: {np.max(predictions[0])}")
# the index where the max value in predictions[0] occurs
print(f"Max index: {np.argmax(predictions[0])}")
# the predicted label
print(f"Predicted label: {unique_breeds[np.
argmax(predictions[0])]}") # the predicted label
```

When you run the code, you will see the following output:

```
Max value: 0.9999655485153198
Max index: 84
Predicted label: papillon
```

Now, I will provide you with the code so that you can print the dog's image, the predicted class, and the distribution chart of the top ten predictions. That way you will be able to visualize and interpret the test results better.

Visualizing Test Results

We will create two plots in each row of our output. The first plot will give the dog's image along with its predicted class name. The second plot would be for the distribution bar chart.

The first plotting function is given in Listing 4-5.

Listing 4-5. Function for plotting image and prediction

```
def plot_pred(prediction_probabilities, images):
 image = process_image(images)
 pred_label = unique_breeds[np.argmax(prediction_
probabilities)]
 plt.imshow(image)
 plt.axis('off')
 plt.title(pred_label)
```

The function plot_pred takes the predicted probabilities and images arrays as arguments. The image is retrieved by calling the process_image function, which is then plotted. The label for the image is retrieved from the unique_breeds array, which is then printed as the plot title.

The function for printing the bar chart of probabilities is given in Listing 4-6.

Listing 4-6. Function for plotting predictions on a given image

```
def plot_pred_conf(prediction_probabilities):
 top_10_pred_indexes = prediction_probabilities.argsort()
 [-10:][::-1]
 top_10_pred_values = prediction_probabilities[top_10_pred_
 indexes]
 top_10_pred_labels = unique_breeds[top_10_pred_indexes]
 top_plot = plt.bar(np.arange(len(top_10_pred_labels)),
                    top_10_pred_values,
                    color="grey")
```

```
 plt.xticks(np.arange(len(top_10_pred_labels)),
            labels=top_10_pred_labels,
            rotation="vertical")
 top_plot[0].set_color("green")
```

We obtain the top ten prediction indices by sorting the predictions array and then picking up the last ten entries from the sorted array. Then, we plot the bar chart and provide the x-axis labels as vertical text.

You will now print the first three images along with their prediction results using the following code snippet:

```
num_rows = 3
plt.figure(figsize = (5 * 2, 5 * num_rows))
for i in range(num_rows):
 plt.subplot(num_rows, 2, 2*i+1)
 plot_pred(prediction_probabilities=predictions[i],
           images=test_filenames[i])

 plt.subplot(num_rows, 2, 2*i+2)
 plot_pred_conf(prediction_probabilities=predictions[i])
plt.tight_layout(h_pad=1.0)
plt.show()
```

The output of running this code is given in Figure 4-24.

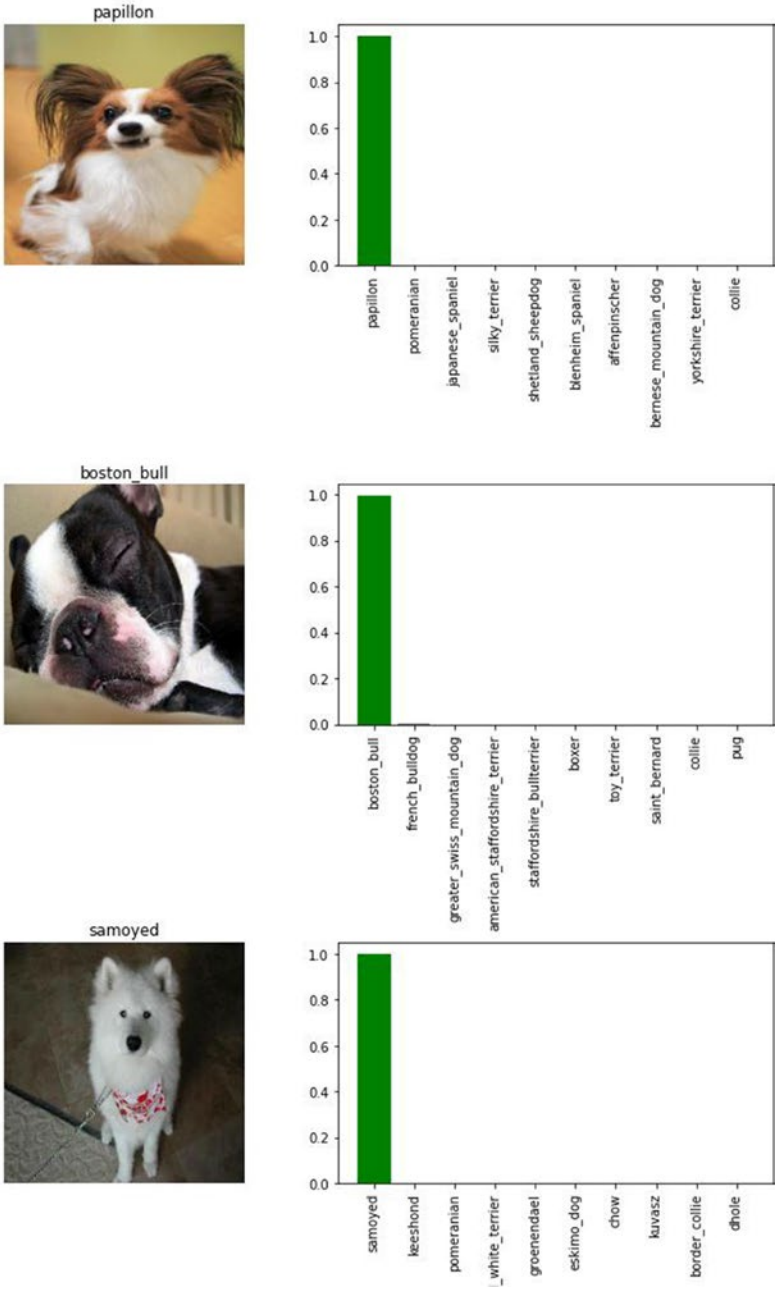


Figure 4-24. *Images and prediction probabilities*

Having seen the results on test data, we will now try to predict an unknown image; let this be an animal other than a dog.

Predicting an Unknown Image

Suppose you input a tiger image shown in Figure 4-25 to our network, what would the network infer?

Figure 4-25. *Test image*

To test this, first load the image from the book's site using the following code snippet:

```
!pip install wget
url='https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/Ch04/tiger.jpg'
import wget
wget.download(url,'tiger.jpg')
```

Then, prepare the image for our model by calling our create_data_batches function:

```
data=create_data_batches(['/content/tiger.jpg'],batch_
size=1,data_type=3)
```

Now, do the prediction using the model's predict method:

```
result = model.predict(data)
```

Get the predicted class and its name:

```
predict_class_index = np.argmax(result[0],axis=-1)
predict_class_name = unique_breeds[(predict_class_index)]
```

Next, you can get the probability predictions by calling the predict_proba method on the model:

```
result_proba = model.predict_proba(data,batch_size=None)
```

You can then check the maximum value for the predictions. If this value is less than a certain threshold, you may conclude that the given image is not of a dog at all.

```
if result_proba.max() > 0.7:
 print(pred_label)
else:
 print('Not a dog breed as the predicted probability is {}'.
format(result_proba.max()))
```

The output of running this code is shown in Figure 4-26.

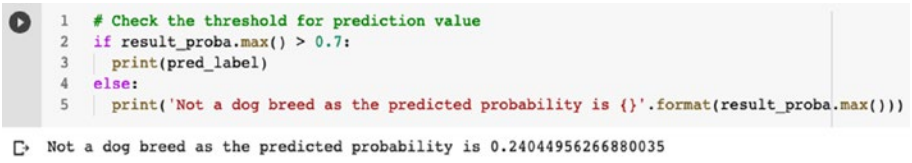


Figure 4-26. *Prediction result on an unseen image*

Now, you will be able to appreciate the usefulness of transfer learning. As the MobileNet model was trained on a wide variety of categories, we could use its knowledge in inferring that the given input image was not of a dog. Had it been a dog image, it would have used our classification layer to classify its breed into one of the known 120 classifications.

There is one more important usefulness of transfer learning I would like to show to you. And that is whether we can work with a smaller dataset.

Training with Smaller Datasets

As it may be difficult to collect a huge number of data points in every situation, we will check whether the transfer learning will help us develop usable models with smaller datasets.

You will first write a function called train_model which can be called for a set of models. The function definition is given here:

```
model_performances = []

# function for training the given model on specified
# number of images
def train_model (model, NUM_IMAGES):
 model.fit(x=train_data,
              epochs=NUM_EPOCHS,
              validation_data=val_data,
              callbacks=[model_tensorboard,
                         model_early_stopping])
 # append the results
 model_performances.append(model.evaluate(val_data))
```

The train_model function takes two parameters; the model parameter specifies the model to be trained, and the NUM_IMAGES specifies the number of images on which the model will be trained. The function simply calls the model's fit method for training and then evaluates the performance by calling its evaluate method. The evaluation results are added to an array for later comparisons.

The function itself is called using the following code snippet:

```
# Training
NUM_EPOCHS = 100
# Create models and test for 1000,2000, 3000, 4000 images
for NUM_IMAGES in range(1000, 5000, 1000):
 model = create_model()
 x_train,x_val,y_train,y_val=train_test_split(X[:NUM_
IMAGES],y[:NUM_IMAGES],test_size=0.2,random_state=10)
 train_data=create_data_batches(x_train,y_train,batch_size=10)
 val_data=create_data_batches(x_val,y_val,batch_size=10,data_
type=2)
 train_model(model,NUM_IMAGES)
 model_performances.append(model.evaluate(val_data))
```

Before calling the train_model, we create a model by calling our create_model function. We split the training dataset into training and validation for the specified number of images. We preprocess the data as before and then train the created model on the processed data. Remember, after the training, the evaluation results are stored in an array. You will now create a pandas dataframe for loss and accuracy metrics:

```
import pandas as pd
comp = pd.DataFrame(model_performances,index =
[1000,2000,3000,4000], columns = ['val_loss', 'val_acc'])
```

You may plot the results for visualization using the following code:

```
# plot the table
import matplotlib.pyplot as plt
plt.style.use('ggplot')
comp.plot.bar()
plt.xlabel('number of images')
plt.ylabel('performance values')
plt.show()
```

The output plot is shown in Figure 4-27.

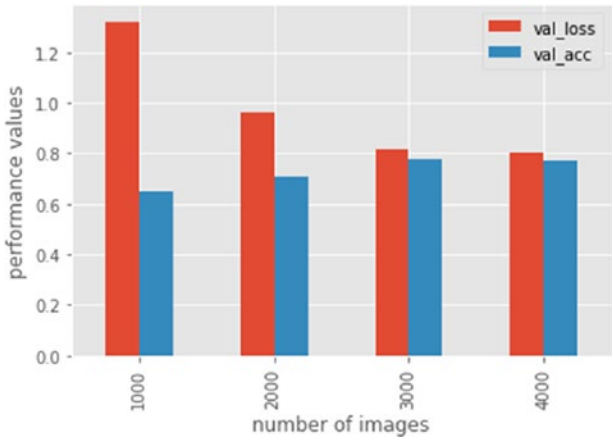


Figure 4-27. *Error metrics on different sizes of dataset*

From these plots, you can see that the accuracy score saturates after about 2000 images. Thus, instead of training the model on a full dataset of 10,000 images, you could train it on a much smaller dataset and yet achieve acceptable results. This will be one of the big advantages of transfer learning, especially when you do not have enough data points available for training.

Now, as you have done creating the new model for dog breed classification, you will like to put this into a real use. For this, I will show you how to save the trained model and later on reuse it for inferring unseen images.

Saving/Reloading Model

To save the trained model in h5 format, call its save method:

```
model.save('model.h5') #saving the model
```

To load the saved model, use the load_model method.

```
from tensorflow.keras.models import load_model
model=load_model('model.h5',custom_objects={"KerasLayer":hub.
KerasLayer})
```

You can check the model's summary to ensure that the model is correctly loaded:

```
model.summary()
```

You will see the output in Figure 4-28.

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
keras_layer_5 (KerasLayer)   multiple                  5432713
_________________________________________________________________
dense_5 (Dense)              multiple                  120240
=================================================================
Total params: 5,552,953
Trainable params: 120,240
Non-trainable params: 5,432,713
_________________________________________________________________
```

Figure 4-28. *Summary for the loaded model*

Now, you are free to use the loaded model for further inferring just the way we did for an unknown image. Enjoy!

Submitting Your Work

Are you not excited with what you learned so far? TensorFlow allows you to submit your own creations for inclusion in the tfhub repository. The tensorflow_hub library is used for loading models from this repository. The HTTP-based protocol supports the retrieval of the model's documentation and provides the endpoint to fetch the model itself. To load the model from the repository, you use load_model method as you had seen in the examples in this chapter. You may create your own repository of models that are loadable with the tensorflow_hub library. For this, there is a certain protocol that needs to be followed by your HTTP distribution service.

After your model is fully trained to your satisfaction, you will save the graph and the parameter values using the following code:

```
saver = tf.train.saver()
saver.save (sess, 'my_model')
```

The saved model can then be put on production servers for public serving. Once beta-tested, you may add it to the repository for community benefit through the tfhub library. Further details of this specification are beyond the scope of this book. The interested reader is referred to more information about this on the TensorFlow site under "Hosting Your Own Models."

Further Work

I have shown you one example of image classification that uses a pre-trained model from TensorFlow Hub. As said earlier, the Hub provides models in other domains, too. For example, you can easily develop your own object detection classifier, a text classifier, a universal sentence encoder, and so on. Suggest you to visit the TensorFlow Hub site to explore these pre-trained models.

There are several commercially deployed case studies listed on the TensorFlow site. To give you an example, Airbnb classified their images to improve the guest experience by retraining the ResNet50 model. Somebody used TensorFlow.js to make Amazon Echo respond to sign language for the benefit of disabled persons. Coca-Cola used SqueezeNet CNN for OCR in their mobile proof-of-purchase application. Google pixel phones use the MobileNet module of TF Hub for camera image recognition and to enhance its camera capabilities. There are endless possibilities for using transfer learning in your own applications. So, keep experimenting with the pre-trained models provided in TensorFlow Hub and apply your minds in using those in your own applications.

Summary

Transfer learning is an important technique of knowledge transfer in machine learning. In software engineering, people use binary libraries to reuse the code. In machine learning, the trained models contain the algorithms, the data, the processing power, and the expert's domain knowledge. All these need to be transferred to the new model. That's what the transfer learning provides. In TensorFlow Hub, there are several pre-trained models available spanning various domains and developers. In this chapter, you first learned to reuse one of such models for classifying your own images. In the latter part, you developed a multiclass image classifier that runs on top of a model pre-trained by Google, taking advantage of their expertise, processing time, and power. There are several other models available in the Hub for you to explore.

In the next chapter, you will learn the intricacies of machine learning with an example on regression model development.

CHAPTER 5

Neural Networks for Regression

So far, we looked at the classification model in deep learning. Can we apply whatever techniques you learned so far as a part of deep learning to a regression problem, which is considered probably the simplest in data analytics? Is it worth even attempting to use deep learning in the areas of regression, considering the overheads of deep learning? Is there an advantage in using deep learning over the traditional statistical techniques – especially in the case of regression modeling? You will find answers to these and similar questions in this chapter.

You already know that a simple linear regression is the simplest thing in machine learning, and many a times this is the first topic that is taught in machine learning. Statistical regression models have been around for many years and are providing useful predictions in many real-world applications. These models are equally used in industry, business, and scientific environments. At the same time, the deep learning networks that we have learned so far in this book are seen to have successfully solved the most complex problems of our world. They provide very accurate predictions that resemble the learning process of our brains. Now, the question is can we use deep learning networks to run a regression

P. Sarang, *Artificial Neural Networks with TensorFlow 2*,
https://doi.org/10.1007/978-1-4842-6150-7_5

problem? And, is there any advantage that we could get instead of using the traditional statistical coding techniques of running a regression? A brief answer to this question is a big Yes. To understand why and how, keep reading!

Let me first define what regression is.

Regression

I will start with the regression definition followed by its applications in real-world situations. Then, I will explain to you the statistical modeling of regression followed by the various regression types.

Definition

In statistical modeling, a regression analysis is a set of statistical processes. These processes try to establish a relationship between a dependent variable (target) and an independent variable (feature). The dependent variable is also called an outcome variable, and the independent variable is sometimes called a predictor or a covariate. In simple situations, there exists only a single predictor, and in more complex data distributions, there will be multiple covariates. The simplest form of regression is the linear regression where the analyst finds a straight-line relationship between the predictor and the outcome. But in many practical situations, the relationship cannot be represented by a straight line. The relationship may be a set of lines resulting in some sort of a polynomial. Determining hyperplanes in complex datasets is a big challenge for data analysts. There exist several types of regression such as linear, logistic, stepwise, and so on. Estimating the type of relationship and coding the same is not a simple task for the developers.

Now, where do you apply regression modeling?

Applications

Regression analysis is primarily used to address the following areas:

- Prediction and forecasting
- Inferring causal relationships

To estimate the pricing of a house is the predictive power of a mostly linear regression model. A multivariate model may forecast the future price of a given stock. The causal relationship is to answer the question – what event caused another or what brought a certain change? What caused an upsurge in a website's traffic, What is the cause of a malfunction in an assembly line, Whether a drug caused an improvement in certain medical conditions – are the examples of establishing causality. To address such areas of applications, a data analyst must first investigate whether a relationship has a predictive power. Why or why not a relationship between two variables has a causal interpretation? Answering these questions is not an easy task for a statistician or a data analyst in modern terms. Neural networks will come to your aid in answering these questions, and that is what I am going to show to you in this chapter.

Regression Problem

In a regression problem, we predict a value or a probability of a dependent variable, given an input of a set of continuous values. Contrasting this with a classification model that you have studied so far, the problem was to select a class within a list of predefined classes. For example, in Chapter 4, you did a dog breed classifier where the input image was inferred as one of the classes within a set of 120 predefined breeds.

In regression analysis, you model the relationship between a dependent variable and one or more independent variables. This relationship can be expressed by a simple mathematical equation:

$$\gamma = \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \ldots + \beta_k X_k + \varepsilon$$

where γ is a dependent variable that you are trying to predict, and X_1, X_2, …, X_k are the independent variables. β_1, β_2 … β_k are the coefficients or the weights in neural networks, and ε is the error or the bias as it is called in ML terminology. Now, let us look at the types of regressions.

Regression Types

The regression analysis can be classified into the following categories:

- Linear
- Polynomial
- Logistic
- Stepwise
- Ridge
- Lasso
- ElasticNet

A linear regression fits a case when there is a linear relationship between the dependent and independent variables – a straight line. In the case of polynomial regression, the dependent variable is best fitted by a polynomial, that is, a curve or a series of curves. In such models, outliers can distort the prediction and thus are prone to what is known as overfitting in machine learning terms. In the case of logistic regression, the predicted value is a binary and strictly follows a binomial distribution. When you have a large number of independent variables, in other words,

a high dimensionality, you use stepwise regression to detect which variables are significant and drop the insignificant ones to maximize the prediction power of your model. When the independent variables are highly correlated, also called multicollinearity, they make variances large enough to cause a large deviation in the predicted value. The technique of ridge regression adds a bias term to the regression estimate penalizing the coefficients or the weight values. It uses least squares to shrink coefficients to ensure that they do not reach zero. This is a form of regularization – an L2 regularization. The lasso regularization is similar to the ridge regression and helps in solving the multicollinearity problem the same way by shrinking the regression coefficients. However, now the absolute values are shrinked rather than the least squares. It also means that some of the weights can shrink to zero, eliminating totally the output of that particular node. This is useful in feature selection because it essentially picks up the one out of a group of dependent variables. This is also a type of regularization – an L1 regularization. Lastly, ElasticNet is a combination of ridge and lasso. The model is trained successively with L1 and L2 regularizations, resulting in a trade-off between the two techniques. As a result, ElasticNet may pick more than one correlated variable.

Have all these things confused you totally? Not only understanding the statistical techniques but also considering various kinds of regression, you can easily imagine the complexity involved in coding them. Can the deep learning techniques come to your rescue? Keep reading!

Regression in Neural Networks

To show you that you can indeed use neural networks in solving even the simplest regression problem, I am going to demonstrate this with a trivial example. For our program, we will use a dataset from a Kaggle competition for simple linear regression. You can download the dataset from here (www.kaggle.com/luddarell/101-simple-linear-regressioncsv). The data

consists of only two columns – GPA and SAT. The GPA column represents the student's Grade Point Average, and the SAT column represents the student's SAT (Scholastic Aptitude Test) score. We will develop a linear regression model to establish a relationship between a student's GPA and their SAT score. Once a model is trained, we would use it to predict how much a student would score in a SAT examination given their GPA.

Setting Up Project

Create a new Colab project and name it LinearRegression. Add the following imports in the project code:

```
import tensorflow as tf
from tensorflow import keras
import pandas as pd
```

The data file is available at the book's site. To download the file in your project, we will use wget. Add the following code to install wget:

```
!pip install wget
import wget
```

Now, to download the file, use the following code:

```
url = 'https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/Ch05/student.csv'
wget.download(url,'data.csv')
```

The downloaded file will be stored in the /content/ folder with the name data.csv. You can examine its contents by first loading the file in a pandas dataframe and then printing the first few records with the following code:

```
import pandas as pd
df=pd.read_csv('/content/data.csv')
df.head(10)
```

You will see the output shown in Figure 5-1.

```
import pandas as pd
df=pd.read_csv('/content/data.csv')
df.head(10)
```

	SAT	GPA
0	1714	2.40
1	1664	2.52
2	1760	2.54
3	1685	2.74
4	1693	2.83
5	1670	2.91
6	1764	3.00
7	1764	3.00
8	1792	3.01
9	1850	3.01

Figure 5-1. *Sample rows in the loaded dataset*

Next, you will extract features and the label from this dataset.

Extracting Features and Label

The dataset contains only two columns – GPA and SAT. We will use GPA as our feature and SAT as our label. Note we are trying to predict how much a student will score on SAT given their GPA. To extract the features and label, use the following code:

```
# Extract features and label
dataset = df.values
x = dataset[:,1]
y = dataset[:,0]
```

To keep this program simple enough, I will not create the training and validation datasets. Also, we will not reserve any data for testing. Next, we define our model.

Defining/Training Model

To define the model, we use the Sequential API as in the earlier cases:

```
model = tf.keras.Sequential([tf.keras.layers.Dense
                (units=1, input_shape=[1])])
```

Our network model consists of only a single layer with a single neuron. The input to this model is a single dimensional tensor. Next, we compile the model:

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

We use stochastic gradient descent as our optimizer and mean squared error for our loss function.

The model is trained using the usual fit method:

```
model.fit(X, y, epochs = 15)
```

Note that in this simple example, I have not captured the error metrics for evaluation of the model's performance.

Predicting

Now, as the model is trained, we are ready to do some predictions. Suppose you want to find out how much a student with GPA 5.0 will score in SAT. You would use the predict method of the model and print its result as follows:

```
result = model.predict([5.0])
print("Expected SAT score for GPA 5.0: {:.0f}"
                 .format(result[0][0]))
```

To find out how much a student with GPA 3.2 would score on SAT, you would use the following:

```
result = model.predict([3.2])
print("Expected SAT score for GPA 3.2: {:.0f}"
                .format(result[0][0]))
```

Note that we have not taken the trouble of verifying the accuracy of these predictions. But it proves to us the point that neural networks can be used to create ML models for even the simplest linear regression problems.

Next, I will deal with a more practical problem of Multicollinearity in Regression Analysis.

Wine Quality Analysis

In this project, you would use regression analysis for determining the wine quality based on certain features. You will use the white wine quality dataset provided in the UCI Machine Learning Repository. The model's goal is to predict the wine quality based on a given set of input features. The dataset provides the following input features, which are based on physicochemical tests.

List of Features:

1. Fixed acidity
2. Volatile acidity
3. Citric acid
4. Residual sugar
5. Chlorides
6. Free sulfur dioxide
7. Total sulfur dioxide

8. Density
9. pH
10. Sulfates
11. Alcohol

The Quality field in the database will be used as the label. The quality value ranges between 0 and 10.

```
Output Label:
     Quality
```

Creating Project

Create a new Colab project and rename it to WineQuality. Load TensorFlow 2.x and import the required libraries using the following code:

```
import tensorflow as tf

import pandas as pd
import requests
import io
import matplotlib.pyplot as plt
```

Data Preparation

The white wine quality dataset is available at the UCI website.

Downloading Data

We will declare a variable in our code to refer to the UCI Machine Learning Repository:

https://raw.githubusercontent.com/Apress/artificial-neural-networks-with-tensorflow-2/main/Ch05/winequality-white.csv

Preparing Dataset

You will use the pandas library to read the CSV file into a dataframe.

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

You may display a few records in the dataframe by calling the head or tail method on it. The result of calling tail is shown in Figure 5-2.

```
dataset.tail()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
4893	6.2	0.21	0.29	1.6	0.039	24.0	92.0	0.99114	3.27	0.50	11.2	6
4894	6.6	0.32	0.36	8.0	0.047	57.0	168.0	0.99490	3.15	0.46	9.6	5
4895	6.5	0.24	0.19	1.2	0.041	30.0	111.0	0.99254	2.99	0.46	9.4	6
4896	5.5	0.29	0.30	1.1	0.022	20.0	110.0	0.98869	3.34	0.38	12.8	7
4897	6.0	0.21	0.38	0.8	0.020	22.0	98.0	0.98941	3.26	0.32	11.8	6

Figure 5-2. *Last few rows of the dataset*

As you can see from Figure 5-2, there are 4897 records in the database. Each record contains 12 fields; the last one is the quality that we will be using as the label. We extract the features and labels using the following two statements:

```
x = dataset.drop('quality', axis = 1)
y = dataset['quality']
```

Next, you will create datasets.

Creating Datasets

For machine learning, you need training and testing datasets. The training dataset is further divided into training and validation sets. To create these datasets, we use the train_test_split method of sklearn as follows:

```
# creating training, validation and testing datasets
from sklearn.model_selection import train_test_split
x_train_1 , x_test , y_train_1 , y_test =
     train_test_split
     (x , y , test_size = 0.15 , random_state = 0)
x_train , x_val , y_train , y_val =
     train_test_split(x_train_1 , y_train_1 ,
     test_size = 0.05 , random_state = 0)
```

Note that 15% of the data is reserved for testing. The training data is split into 95:5 ratio – 95% for training and 5% for validation.

Generally, the data in different fields show a wide variance in their values. The neural network will learn better if these data items are scaled down to a fixed scale. Thus, we need to preprocess the entire data. This preprocessing mainly consists of feature scaling through standardization or Z-score normalization. You rescale the feature values so as to have a standard normal distribution with a mean of zero and a standard deviation of one. This is shown in Figure 5-3.

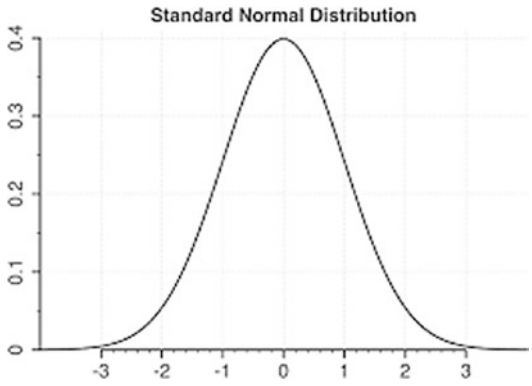


Figure 5-3. *Desired data distribution*

We now proceed to scale our data so as to ideally achieve the distribution curve illustrated in Figure 5-3.

Scaling Data

To scale our input data or to be more precise to center it, you subtract the mean and then divide the result by the standard deviation:

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

where **μ** is the mean and **σ** is the standard deviation.

You first do the transformation on the training set. For this, we use the StandardScaler class of sklearn and call its fit_transform method. The following code does this transformation:

```
from sklearn.preprocessing import StandardScaler
sc_x = StandardScaler()
x_train_new = sc_x.fit_transform(x_train)
```

You may check the effect of this transformation by plotting the raw data before and after the transformation. The following code generates these two plots for the fixed_acidity field:

```
fig, (ax1, ax2) = plt.subplots
                (ncols = 2, figsize = (20, 10))

ax1.scatter(x_train.index,
               x_train['fixed acidity'],
               color = c,
               label = 'raw',
               alpha = 0.4,
               marker = m
               )

ax2.scatter(x_train.index,
               x_train_new[: , 1],
               color = c,
               label = 'adjusted',
               alpha = 0.4,
               marker = m
               )

ax1.set_title('Training dataset')
ax2.set_title('Standardized training dataset')

for ax in (ax1, ax2):
   ax.set_xlabel('index')
   ax.set_ylabel('fixed acidity')
   ax.legend(loc ='upper right')
   ax.grid()

plt.tight_layout()

plt.show()
```

The output is shown in Figure 5-4.

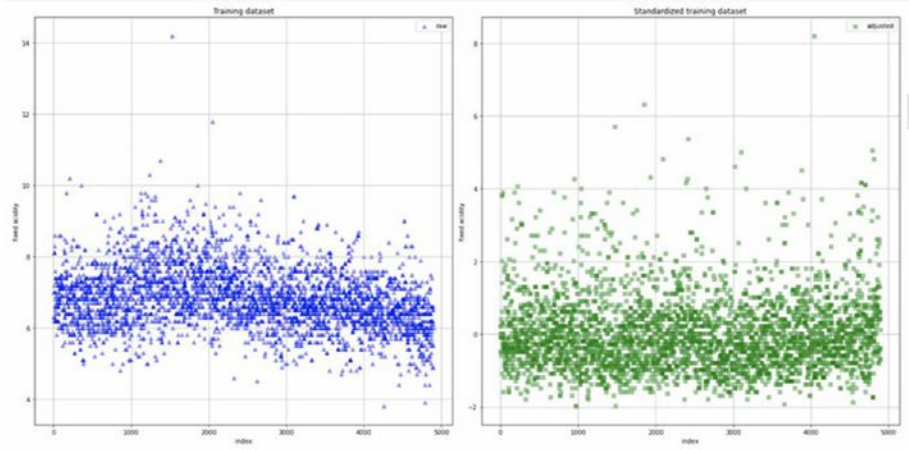


Figure 5-4. *Data distribution before and after standardization*

The plot on the LHS shows the raw data, where you can see that the fixed acidity field values range from about 0 to 14 with the mean somewhere around 7. The plot on the RHS shows the same data after the transformation. Note that the mean is about 0 with the data variation between –2 and +2, indicating a standard deviation of +/–1. If you check the plots for other fields, you will again notice that the mean is brought to zero, and there is a somewhat even distribution on either side of zero equal to the standard deviation.

You can try one more plot – this time plotting the total sulfur dioxide on the y axis and residual sugar on the x axis. The following code generates these plots:

```
fig, (ax1, ax2) =
     plt.subplots(ncols = 2, figsize = (20, 10))

for l, c, m in zip(range(0, 2),
                ('blue', 'red'), ('^', 's')):
```

```
    ax1.scatter(x_train['residual sugar'],
                x_train['total sulfur dioxide'],
                color = c,
                label = 'class %s' % l,
                alpha = 0.4,
                marker = m
                )

for l, c, m in zip(range(0, 2),
                   ('blue', 'green'), ('^', 's')):
    ax2.scatter(x_train_new[: , 3],
                x_train_new[: , 6],
                color = c,
                label = 'class %s' % l,
                alpha = 0.4,
                marker = m
                )

ax1.set_title('Training dataset')
ax2.set_title('Standardized training dataset')

for ax in (ax1, ax2):
    ax.set_xlabel('residual sugar')
    ax.set_ylabel('total sulfur dioxide')
    ax.legend(loc ='upper right')
    ax.grid()

plt.tight_layout()


plt.show()
```

The output is shown in Figure 5-5.

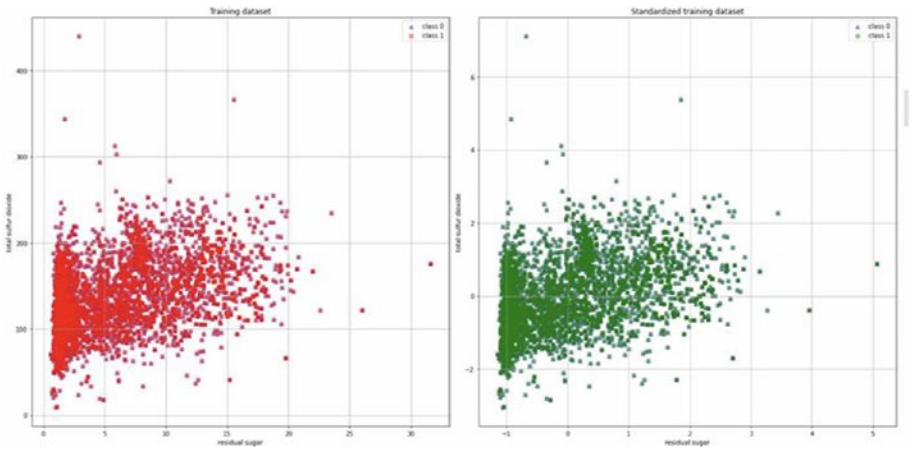


Figure 5-5. *Data distribution for two fields after standardization*

As you can see from the LHS plot in Figure 5-5, the residual sugar values vary between 0 and 30, while the total sulfur dioxide value has a large variation between 0 and 400. After the transformation as seen in the plot on RHS of Figure 5-5, both fields show a mean value of 0 and even distribution between –2 and +2 for the rest of the data points.

This kind of transformation helps neural networks learn better. So, we do these transformations on all our features. The fit_transform not only transforms the data but also stores the values of **μ** and **σ** in internal variables. For very obvious reasons, you must apply the same transformation that you applied to the training dataset to your testing and cross-validation sets. Thus, retaining the values of these parameters is an important goal behind using the fit_transform method. The method fits the values, remembers them, and then applies the transformation. Now, to transform the testing and validation datasets, we simply call the transform method and not the fit_transform method. This is done using the following two statements:

```
x_test_new = sc_x.transform(x_test)
x_val_new = sc_x.transform(x_val)
```

With all fields now standardized to a mean value of zero with the even distribution of other data points determined by the standard deviation, you are now ready for building the model.

Model Building

Before we build the model, I am going to write a small function for visualizing the training results that can be used for the several models that we are going to train.

Visualization Function for Metrics

Rather than using TensorBoard for analysis, I am going to use matplotlib to draw the analysis metrics. I am going to define a function that will be repeatedly called after every trial of a new model. If you decide to use TensorBoard instead, remember you may be required to reset its state or provide a separate log folder after every plot and the model run. So, it is probably more convenient to use a plotting function.

The entire plotting function is given in Listing 5-1. As before, the function does not need any further comments; it is self-explanatory.

Listing 5-1. Metrics visualization function

```
import matplotlib.pyplot as plt
epoch = 30
def plot_learningCurve(history):
 # Plot training & validation accuracy values
 epoch_range = range(1, epoch+1)
 #plotting the mae vs epoch of training set
 plt.plot(epoch_range, history.history['mae'])
 #plotting the val_mae vs epoch of the validation dataset.
 plt.plot(epoch_range, history.history['val_mae'])
```

```
plt.ylim([0, 2])
plt.title('Model mae')
plt.ylabel('mae')
plt.xlabel('Epoch')
plt.legend(['Train', 'Val'], loc = 'upper right')
plt.show()

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

# Plot training & validation loss values
plt.plot(epoch_range, history.history['loss'])
plt.plot(epoch_range, history.history['val_loss'])
plt.ylim([0, 4])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Val'], loc = 'upper right')
plt.show()
```

Now, we will start our model building exercise. You will be creating three models with increasing complexity. After defining the model, you will train it and evaluate its performance on a test data. We will use the same test data to evaluate the performance of the various models that we build in this exercise. We will try to experiment with a different optimizer for a large model in our case studies. I will show you when the overfitting occurs and how to detect it. Then, I will give you some clues on how to mitigate it.

So, let us start our experiment with the small model.

Small Model

In the small model, we will construct a model that has only one hidden layer.

Definition

The input to the model is a tensor of 11 input features, and the output is a single layer that gives out the wine quality. We will set 16 neurons in the hidden layer with the ReLU activation on each neuron. The model is defined using the following statement:

```
small_model = tf.keras.Sequential([
                tf.keras.layers.Dense(16 ,
                    activation = 'relu' ,
                    input_shape = (11 , )),
                tf.keras.layers.Dense(1)
])
```

We will use the Adam optimizer and mean_squared_error (mse) as a loss function. We will use the mean absolute error (mae) for our analysis. The model is compiled using the following statement:

```
small_model.compile(optimizer = 'adam' ,
                    loss = 'mse' ,
                    metrics = ['mae'])
```

Training

We train the model by calling its fit method. We use the batch size of 32 during training. Note that we have 4000 and odd number of records available in our dataset. Thus, there is a scope to create enough number of batches during training. The following statement performs the training and captures the results in the history_small variable:

```
history_small = small_model.fit
                (x_train_new, y_train ,
                 batch_size = 32,
                 epochs = 30,verbose = 1 ,
                 validation_data =
                     (x_val_new , y_val))
```

Evaluation

After training is over, we plot the evaluation metrics by calling our previously defined plotting function:

```
plot_learningCurve(history_small)
```

The output is shown in Figure 5-6.

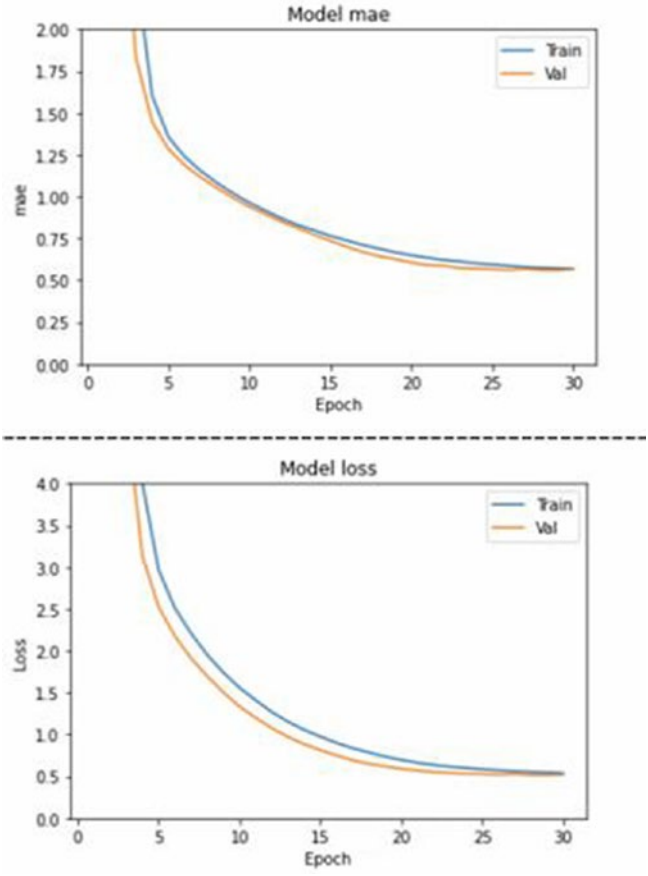


Figure 5-6. *Small model evaluation metrics*

You may evaluate the model's performance on the test data that we have previously created by calling its evaluate method. The method call and its output are shown here:

```
s_test_loss , s_test_mae = small_model.evaluate
                (x_test_new , y_test ,
                batch_size = 32 , verbose = 1)
print("small model test_loss : {}"
                .format(s_test_loss))
print("small model test_mae : {} "
                .format(s_test_mae))
```

```
small model test_loss : 0.6353380084037781
small model test_mae : 0.6149751543998718
```

Evaluation on Unseen Data

To evaluate the model's performance on an unseen data, we will need to create such a data item. For this, I picked up the data point from the test data with the id equal to 2125. I removed the label (the wine quality) from it and created the unseen data using the following declaration:

```
unseen_data = np.array([[6.0 , 0.28 , 0.22 , 12.15 ,
                         0.048 , 42.0 , 163.0 ,
                         0.99570 , 3.20 , 0.46 ,
                         10.1]])
```

You may then do the prediction on this test data and print its result as follows:

```
y_small = small_model.predict
                (sc_x.transform(unseen_data))
print ("Wine quality on unseen data
                 (small model): ", y_small[0][0])
```

```
Wine quality on unseen data (small model):  5.618517
```

The wine quality is predicted to be 5.62, which is close to the actual label value of 5.0 as seen in the training dataset for the data item of id 2125.

Now, we will proceed with a more complex model.

Medium Model

In this model, we will increase the number of hidden layers from 1 to 3. We will also increase the number of neurons in each of these layers to 64. We will continue using the earlier used Adam optimizer and the mse as our loss function. We will use mae for our evaluation metrics.

Model Definition/Training

The following code defines the model, compiles it, and performs training on it:

```
medium_model = tf.keras.Sequential([
                    tf.keras.layers.Dense
                      (64 , activation = 'relu' ,
                              input_shape = (11, )),
                    tf.keras.layers.Dense
                       (64 , activation = 'relu'),
                    tf.keras.layers.Dense
                       (64 , activation = 'relu'),
                    tf.keras.layers.Dense(1)
])

medium_model.compile(loss = 'mse' ,
                      optimizer = 'adam' ,
                      metrics = ['mae'])
```

```
history_medium = medium_model.fit
                  (x_train_new , y_train ,
                    batch_size = 32,
                    epochs = 30, verbose = 1 ,
                    validation_data =
                    (x_val_new , y_val))
```

In the medium_model definition, we have simply added two Dense layers, making it a total of three hidden layers, each consisting of 64 neurons. As before, each neuron is activated by the ReLU function. The idea behind adding these layers is to check whether adding more layers to the network would help in getting better accuracy. So, let us test it out by checking the evaluation results.

Model Evaluation

As in the earlier case, we plot the evaluation metrics by calling our plotting function.

```
plot_learningCurve(history_medium)
```

The plot is shown in Figure 5-7.

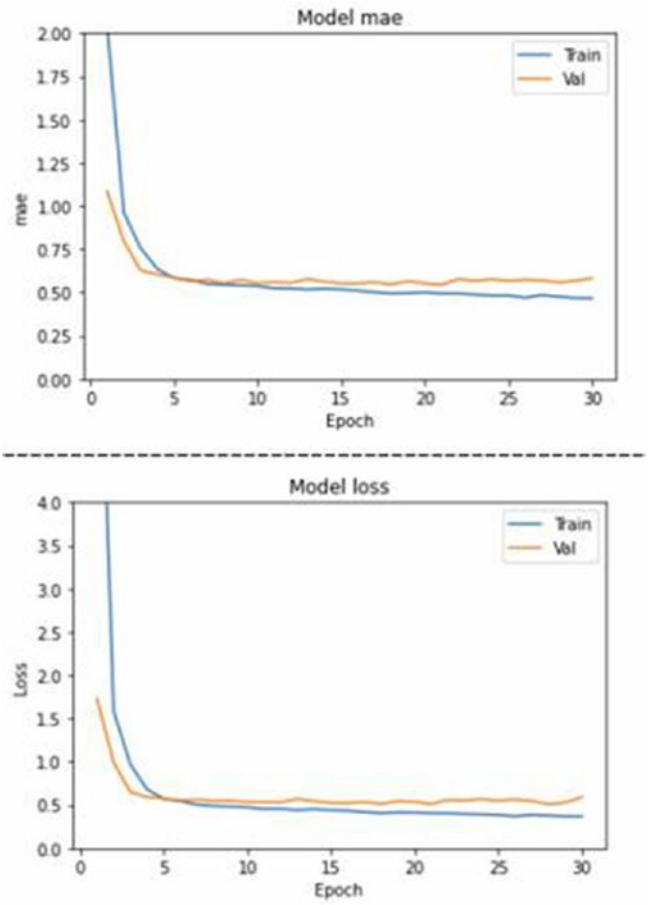


Figure 5-7. *Medium model error metrics*

As you can see from Figure 5-7, the training saturates after about five epochs. Consider this with the case of the small model, where we did not see the saturation until about 20 epochs. In the medium_model, we also observe some overfitting. If you see a divergence in the two curves shown in each plot, it would be the case of overfitting.

Let us now evaluate the model's performance on the test data:

```
m_test_loss , m_test_mae = medium_model.evaluate
                (x_test_new , y_test ,
                batch_size = 32 , verbose = 1 )
print("medium model test_loss : {}".format
                           (m_test_loss))
print("medium model test_mae : {}".format
                            (m_test_mae))
```

You will see the following output:

```
medium model test_loss : 0.6351445317268372
medium model test_mae : 0.6231803894042969
```

Compared to the small model, both test_loss and test_mae have reduced in value. Finally, let us check the evaluation on an unseen data:

```
y_medium = medium_model.predict
                (sc_x.transform(unseen_data))

Print ("Wine quality on unseen data
                 (medium model): ", y_medium[0][0])

Wine quality on unseen data (medium model):  5.246436
```

The predicted wine quality is now 5.25 closer to the actual value of 5.0 and definitely lower than the quality predicted by the small_model which is 5.62.

Let us now proceed with a more complex model.

Large Model

Now, we will add two more hidden layers to our medium model, making it a total of four hidden layers. We will also increase the number of neurons in each layer to 128. Note that adding the layers and number of neurons increases the number of parameters to be tuned. We once again continue using Adam, mse, and mae as before.

Model Definition/Training

In the code for model definition and training, not much changes, except for the two additional layers and the number of neurons. The following code segment gives the model definition and its training:

```
large_model = tf.keras.Sequential([
                  tf.keras.layers.Dense
                      (128 , activation = 'relu' ,
                      input_shape = (11, )),
                  tf.keras.layers.Dense
                       (128 , activation = 'relu'),
                  tf.keras.layers.Dense
                       (128 , activation = 'relu'),
                  tf.keras.layers.Dense
                       (128 , activation = 'relu'),
                  tf.keras.layers.Dense(1)
])

large_model.compile
            (loss = 'mse' , optimizer = 'adam' ,
           metrics = ['mae'])

history_large = large_model.fit
                     (x_train_new , y_train ,
                     batch_size = 32, epochs = 30,
                     verbose = 1 , validation_data =
                     (x_val_new , y_val))
```

Now, let us look at the model's performance evaluation.

Model Evaluation

The plot of the error metrics is shown in Figure 5-8 which is generated by calling our plot_learningCurve method:

```
plot_learningCurve(history_large)
```

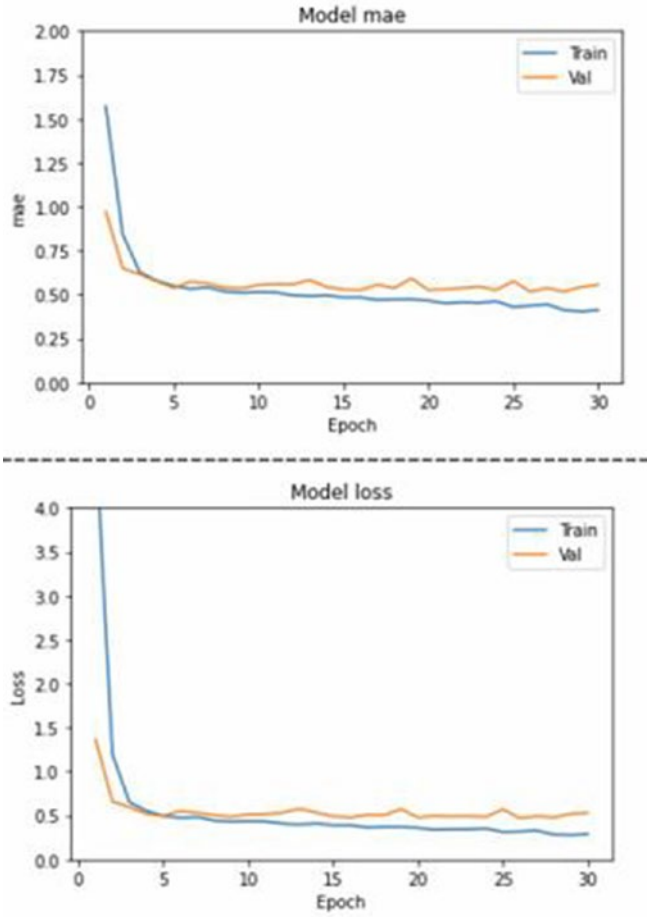


Figure 5-8. *Large model error metrics*

From Figure 5-8, you observe that the saturation is achieved much earlier after about three epochs. You would also notice that this is a larger case of overfitting. I will show you techniques of overcoming this overfitting in the following sections. Let us now look at the results of the evaluation on the test data which are shown here:

```
l_test_loss , l_test_mae = large_model.evaluate
                (x_test_new , y_test ,
                batch_size = 32 , verbose = 1)
print("large model test_loss : {}"
                .format(l_test_loss))
print("large model test_mae : {}"
                .format(l_test_mae))
```

```
large model test_loss : 0.5520739555358887
large model test_mae : 0.5552783012390137
```

The loss is now 0.57 and mae too is 0.57. I will provide you a table later on to compare the results of the three models. Let us now check the results on unseen data.

```
y_large = large_model.predict(sc_x.transform
           (np.array([[6.0 , 0.28 , 0.22 , 12.15 ,
                       0.048 , 42.0 , 163.0 ,
                       0.99570 , 3.20 , s0.46 ,
                      10.1]])))
Print ("Wine quality on unseen data (large model): ",
           y_large[0][0])
```

```
Wine quality on unseen data (large model):  5.389405
```

This predicts a wine quality of 5.39, somewhat larger than the one predicted by the medium model. I will have a full discussion of the results later in the chapter. Currently, let us look at how to fix the overfitting problem.

Fixing Overfitting

Before I show you a few techniques of fixing overfitting, let me discuss what is overfitting and to that matter what is a good fit and what is underfitted.

What Is Overfitting?

A diagram in Figure 5-9 visually shows you the difference between overfitting, good fit, and underfitted.

Figure 5-9. *Various modes of model fitting*

Generally, the overfitting occurs when the model accuracy is high on the data used during the training and drops significantly with the new data. In other words, the model knows the training data well but is not able to generalize its inference. As you can see from Figure 5-9, if the curve is not balanced, there is a chance of overfitting. One of the techniques of reducing the overfitting is by adding dropout layers.

So, let us try it out.

Adding Dropout Layers

In the large model, the overfitting is very clearly observed. Let us try by adding dropout layers to this model to check if it reduces the overfitting. The new model with the dropout layers added is given in the following code snippet:

```
large_model_overfit = tf.keras.Sequential([
                tf.keras.layers.Dense
                (128 , activation = 'relu' ,
                     input_shape = (11, )),
                tf.keras.layers.Dropout(0.4),
                tf.keras.layers.Dense
                (128 , activation = 'relu'),
                tf.keras.layers.Dropout(0.3),
                tf.keras.layers.Dense
                (128 , activation = 'relu'),
                tf.keras.layers.Dropout(0.2),
                tf.keras.layers.Dense
                (128 , activation = 'relu'),
                tf.keras.layers.Dense(1)
])
large_model_overfit.compile(loss = 'mse' ,
           optimizer = 'adam' , metrics = ['mae'])
history_large_overfit = large_model_overfit.fit
      (x_train_new , y_train , batch_size = 32,
      epochs = 30,verbose = 0 , validation_data =
      (x_val_new , y_val))
plot_learningCurve(history_large_overfit)
```

We have added 40%, 30%, and 20% dropouts after the first three hidden Dense layers. The rest of the code remains the same as in our testing of the large model. Now, let us look at the error metrics generated by this model.

Figure 5-10 gives the plots of error metrics for the two networks – one without the dropout layer is shown on the left-hand side, and the other one with the dropout layers is shown on the right-hand side.

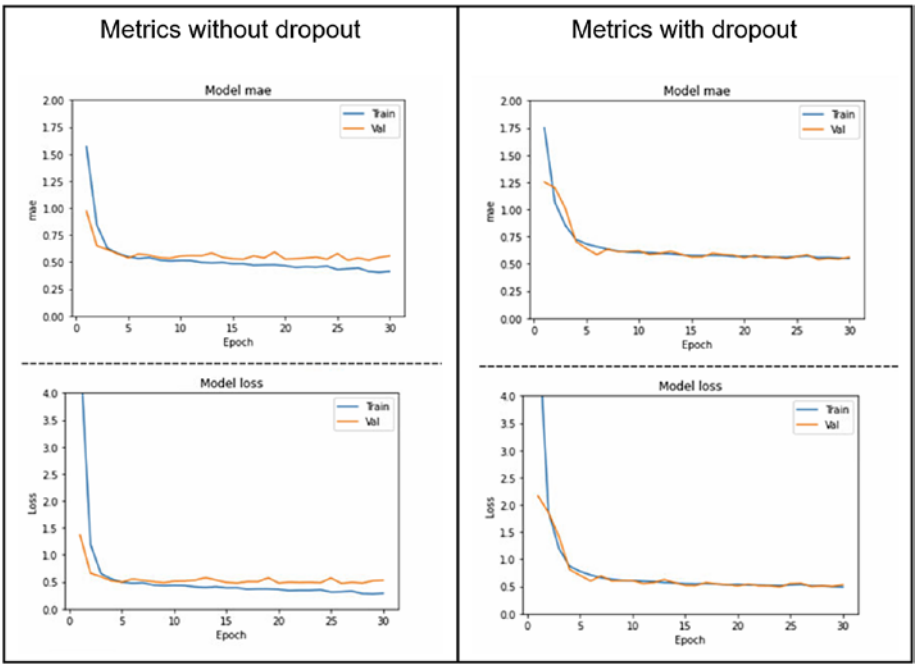


Figure 5-10. *Effect of adding dropout in the large model*

You can clearly see from Figure 5-10 that adding the dropout layers eliminates or at least reduces the overfitting.

The dropout layers are usually added to the large networks as they have enough number of neurons available in each layer from which you can afford to drop out the outputs of a few.

Using RMSprop Optimization

The small model did not show any overfitting. We can still optimize its training by using the RMSprop optimizer, and that's what I am going to demonstrate now. The following code snippet uses the RMSprop optimization:

```
model_small = tf.keras.Sequential([
               tf.keras.layers.Dense(16 ,
                   activation = 'relu' ,
                   input_shape = (11 , )),
             tf.keras.layers.Dense(1)
])
optimizer = tf.keras.optimizers.RMSprop(0.001)
model_small.compile(loss = 'mse' , optimizer =
                optimizer , metrics = ['mae'])
history_small_overfit = model_small.fit
            (x_train_new , y_train , batch_size = 32,
                      epochs = 30, verbose = 0 ,
                      validation_data =
                      (x_val_new , y_val))
plot_learningCurve(history_small_overfit)
```

The only change that we have done in the case study of our small model is changing the optimizer from Adam to RMSprop. We use a very slow learning rate of 0.001. Figure 5-11 now shows the results of adding this optimizer.

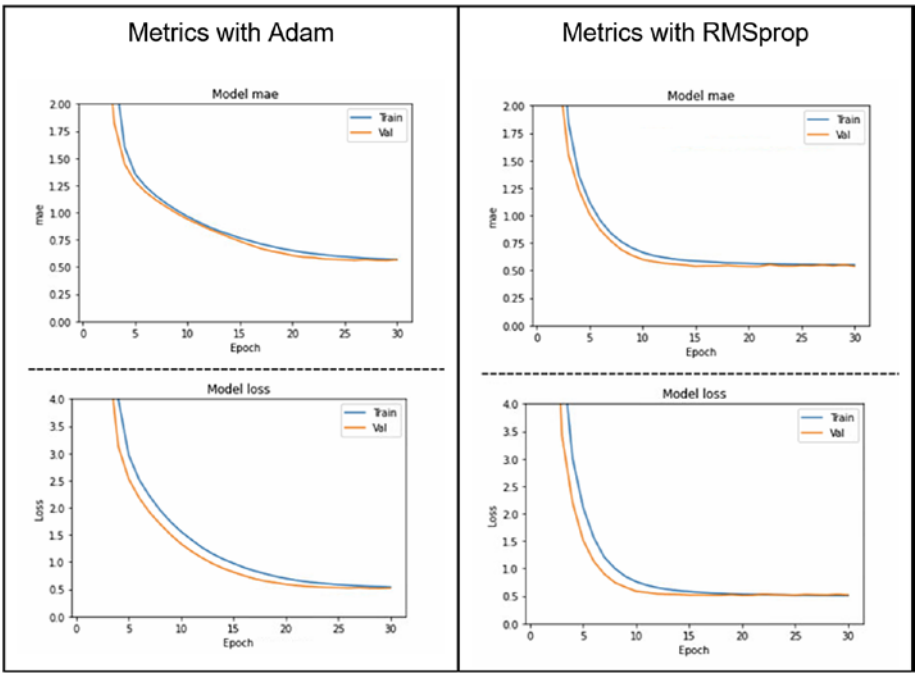


Figure 5-11. *Adam vs. RMSprop error metrics*

The LHS of Figure 5-11 shows the metrics for the model that uses the Adam optimizer, and the one on the right shows the one that uses the RMSprop optimizer. The curves on the RHS are smoother as compared to those on the LHS, indicating less of an overfitting. Also, the charts clearly show that using RMSprop reduces the number of epochs considerably. This will reduce the training time, which is vital in the case of huge datasets. Another way of reducing the training time would be to use the EarlyStopping feature of TensorFlow that you have learned in the previous chapter.

Result Discussion

I am presenting here the results of one of my runs for the three models that we tested. Note that your results and the results on every run will differ each time.

```
small model test_loss : 0.6353380084037781
small model test_mae : 0.6149751543998718
Wine quality on unseen data (small model):  5.618517
Trainable params: 209

medium model test_loss : 0.6351445317268372
medium model test_mae : 0.6231803894042969
Wine quality on unseen data (medium model):  5.246436
Trainable params: 9,153

large model test_loss : 0.5520739555358887
large model test_mae : 0.5552783012390137
Wine quality on unseen data (large model):  5.389405
Trainable params: 51,201
```

The results are summarized in a tabular format in Table 5-1 for a quick comparison.

Table 5-1. *Different model error comparisons*

	MSE	MAE	Unseen data wine quality prediction	Trainable parameters
Small model	0.6353	0.6149	5.6185	209
Medium model	0.6351	0.6231	5.2464	9153
Large model	0.5520	0.5552	5.3894	51201

As you can see from Table 5-1, the loss and mae reduce initially and later on show a higher value for a larger model. However, the differences are marginal. Similarly, the predictions on an unseen data are almost in the same range. At the same time, increasing the model's complexity resulted in overfitting. Now, if you look at the trainable parameters, those increase from 209 to a whopping 51,201. So, is there a necessity of using a complex model in such a simple regression case? One can conclude that it is okay to use a small model in such situations of a small dataset.

To summarize our studies so far, we can conclude that neural networks and deep learning techniques can be used for solving regression problems. When you have multiple numeric features with different ranges, ensure that each feature is scaled independently to the same range as a part of data preprocessing. If you do not have enough data, use a small network with a few hidden layers to avoid overfitting.

In our regression projects, we used the mean squared error (mse) as a loss function and mean absolute error (mae) as evaluation metrics. These are common in regression models. However, the tf.keras library provides many more loss functions for the use of regression models. I will now discuss some of these so that in your experimentation, you may try using them to create a better performing model.

Loss Functions

A loss function is a measure of how good your model predicts the expected outcome. There is not a single loss function that can be applied to all sorts of data. Depending on the distribution of data and the presence of outliers, you would need to use a different loss function. The loss functions that you use for a regression problem may differ from those you use for classification problems. I will describe a few loss functions

which are specifically used for regression problems and for what kind of data distribution are those applied. I will discuss the following five loss functions:

- Mean squared error (mse)/quadratic loss/L2 loss
- Mean absolute error (mae)/L1 loss
- Huber loss
- Log cosh loss
- Quantile loss

Mean Squared Error

This is probably the most commonly used loss function. Mathematically, it is expressed as

$$MSE = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - \hat{y}_i\right)^2$$

You take the difference between the prediction and the truth, square it, and average it out across the whole dataset. This loss function is ideally useful for eliminating outliers as the errors are magnified due to squaring. The function is available as tf.keras.losses.MSE in the tf.keras library.

Mean Absolute Error

Mathematically, mae is expressed as

$$MAE = \frac{1}{n}\sum_{j=1}^{n}\left|y_j - \hat{y}_j\right|$$

Here, you take the absolute value of the difference between the prediction and the truth and then average it out across the whole dataset. The function does not give too much importance to the outliers and provides an even measure across all data points. The function is available as tf.keras.losses.MAE in the tf.keras library.

Huber Loss

The mse detects outliers while mae ignores them. There can be situations where none of these will give a desirable prediction. Consider a data distribution where say 80% of the data have y1 as the true target, while the remaining 20% have y2 as a true target. The mae model as it works on the average will consider 20% as outliers, while the mse which amplifies the error may give y2 as the prediction in many cases. Huber loss provides a solution in between mae and mse. Mathematically, it is expressed as

$$L_{\delta}\left(y, f\left(x\right)\right) = \begin{cases} \frac{1}{2}\left(y - f\left(x\right)\right)^{2} & \text{for } \left|y - f\left(x\right)\right| \le \delta, \\ \delta\left|y - f\left(x\right)\right| - \frac{1}{2}\delta^{2} & \text{otherwise} \end{cases}$$

As you can see from the equation, the Huber loss approaches mae when δ approaches 0 and mse when δ approaches infinity. It is less sensitive to outliers and provides a solution to this in between mae and mse. The Huber loss function is provided as a part of the tf.keras library and is specified using tf.keras.losses.Huber().

Log Cosh

Mathematically, this is expressed as

$$L\left(y,y^{p}\right)=\sum_{i=1}^{n}\log\left(\cosh\left(y_{i}^{p}-y_{i}\right)\right)$$

The function takes the logarithmic of the hyperbolic cosine of the error. For small errors, the log(cosh(x)) approximates to $\frac{(error)^2}{2}$, and for large errors, it is abs(error) – log(error). So, mostly, it works like a mse. It is twice differentiable everywhere and has all the advantages of the Huber loss. The function is provided as tf.keras.losses.LogCosh() in the tf.keras library.

Quantile

Consider a data distribution shown in the LHS of Figure 5-12.

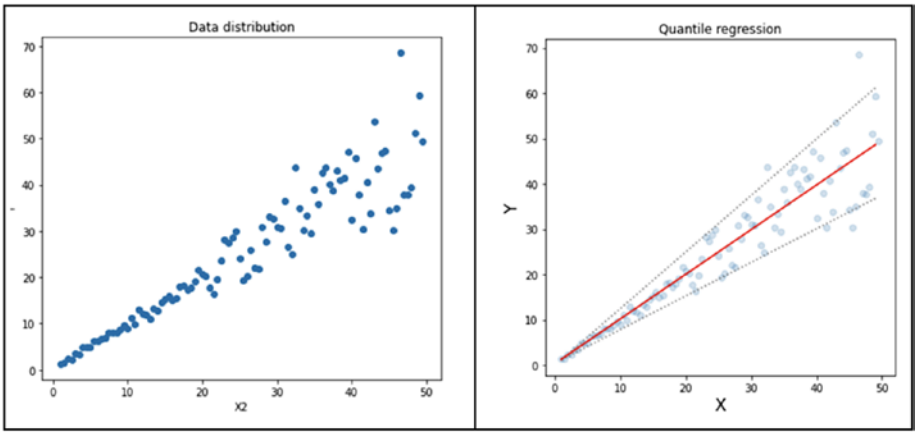


Figure 5-12. *Quantile regression plots*

Rather than modeling a single regression line, you may like to model this data with multiple regression lines as shown in the RHS of Figure 5-12. In such situations, the quantile loss comes to your rescue. Mathematically, it is specified as

$$L\left(y,y^{p}\right)=\sum_{i=y_i<y_i^p}(\gamma-1).\left|y_i-y_i^p\right|+\sum_{i=y_i\geq y_i^p}(\gamma).\left|y_i-y_i^p\right|$$

where γ takes the value between 0 and 1. The quantile loss is actually an extension of mae. It becomes mae when γ takes the value of 0.5. The values on either side provide penalties to either overestimate or underestimate.

The quantile loss functions are useful when you want to predict an interval instead of a point prediction. This can significantly improve decision making in many business problems.

Now, having seen the various loss functions that can be used in the context of regression, let me tell you what optimizers are available in tf.keras, which can be used along with these loss functions.

Optimizers

Optimizers are essentially the algorithms used to reduce losses during the model training. They update the weight parameters to minimize the loss function which continuously tells the optimizer if it is moving in the right direction to reach the global minimum. Unlike loss functions, optimizers are not specific to regression, and all available optimizers can be applied to regression problems. The various optimizers available in tf.keras are listed here:

- Adagrad
- RMSprop
- Adam

- SGD
- Adadelta
- Adamax
- Nadam

You have already used some of these in our earlier projects. Each optimizer has its own purpose. I strongly recommend that you go through the tf.keras documentation to understand the importance of each one. Depending on the need, try using different optimizers to improve the model's performance. Once again, these are not specific to regression problems and apply across all types of deep learning projects.

Summary

To summarize whatever you have learned in this chapter, it can be said that if an existing statistical model of regression meets your purpose, do not bother to use neural networks.

If you are modeling a complex dataset with a large number of features having complex hyperplane relationships to the target values, use deep learning neural networks to get that extra prediction power. The traditional regression functions are available in R, scikit-learn, and other such similar libraries. For example, the scikit-learn library besides linear regression provides several other regressors such as KNeighboursRegressor, DecisionTreeRegressor, and RandomForestRegressor. On the other hand, neural networks have a bit of overhead, but provide a prediction power that is incomparable to any of the regressors mentioned earlier. It is very rare to find a regression equation that perfectly fits the given datasets. The deep learning network, on the contrary, will try to find the best fit on its own without any additional efforts from your side. Thus,

to conclude, henceforth, think of using neural networks for solving even small regression problems; who knows, this small problem (dataset) may become huge over a period of time where keeping tab of traditional statistical techniques will become a nightmare.

The next chapter will take you through some of the techniques of quicker machine learning.

CHAPTER 6

Estimators

Introduction

Any machine learning project consists of many stages that include training, evaluation, prediction, and finally exporting it for serving on a production server. You learned these stages in previous chapters where the classification and regression machine learning projects were discussed. To develop the best performing model, you played around with different ANN architectures. Basically, you experimented with several different prototypes to achieve the desired results. Prior to TF 2.0, this entire experimentation was not so easy as for every change that you make in the code, you were required to build a computational graph and run it in a session. The estimators that you are going to study in this chapter were designed to handle all this plumbing. The entire process of creating graphs and running them in sessions was a time-consuming job and posed lots of challenges in debugging the code.

Additionally, after the model is fully developed, there was a challenge of deploying it to a production environment where you may wish to deploy it in a distributed environment for better performance. Also, you may like to run the model on a CPU, GPU, or a TPU. This necessitated code changes. To help you with all these issues and to bring everything under one umbrella, estimators were introduced, albeit prior to TF 2.0. However, the estimators, which you are going to learn in this chapter and

P. Sarang, *Artificial Neural Networks with TensorFlow 2*,
https://doi.org/10.1007/978-1-4842-6150-7_6

will be using in all your future projects, are capable of taking advantage of the many new facilities introduced in TF 2.x. For example, there is a clear separation between building a data pipeline and the model development. The deployment on a distributed environment too occurs without any code change. The enhanced logging and tracing make debugging easier. In this chapter, you will learn how estimator achieves this.

In particular, you will learn the following:

- What is an estimator?
- What are premade estimators?
- Using a premade estimator for classification problems
- Using a premade estimator for regression problems
- Building custom estimators on Keras models
- Building custom estimators on tfhub modules

Estimator Overview

A TensorFlow estimator is a high-level API that unifies several stages of machine learning development under a single umbrella. It encapsulates the various APIs for training, evaluation, prediction, and exporting your model for production use. It is a high-level API that provides a further abstraction to the existing TensorFlow API stack.

API Stack

After the introduction of estimators, the new API stack for TensorFlow can be depicted as shown in Figure 6-1.

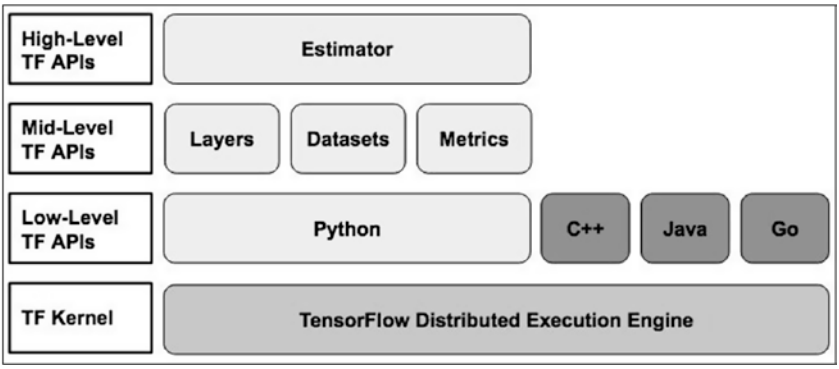


Figure 6-1. *TensorFlow API stack*

So far, you have mostly used mid-level APIs; the use of low-level APIs becomes a rare necessity when you need a finer control over the model development. Now, after learning the estimator API, you may not even use the mid-level APIs for your model developments. But then what happens to the models which you have already developed – can they benefit from using this API? And if so, how to make them use this API? Fortunately, the TensorFlow team has developed an interface that allows you to migrate the existing models to use the estimator interface. Not only that, they have created a few estimators on their own to get you quickly started. These are called premade estimators. They are not just the start-up points. They are fully developed and tested for you to use in your immediate projects. If at all these do not meet your purpose or if you want to migrate existing models to facilitate the benefits offered by the estimator, you can develop custom estimators using the estimator API. Before you learn how to use premade estimators and to build your own ones, let me discuss its benefits in more detail.

Estimator Benefits

To give you a quick overview of the major benefits offered by an estimator, I am first going to list them here:

- Providing a unified interface for train/evaluate/predict
- Handling data input through an Input function
- Creating checkpoints
- Creating summary logs

These need not be the only benefits, but are certainly the most important ones. I will now discuss each one in further detail. To understand the discussion, keep the estimator interface depicted in Figure 6-2 in your mind.

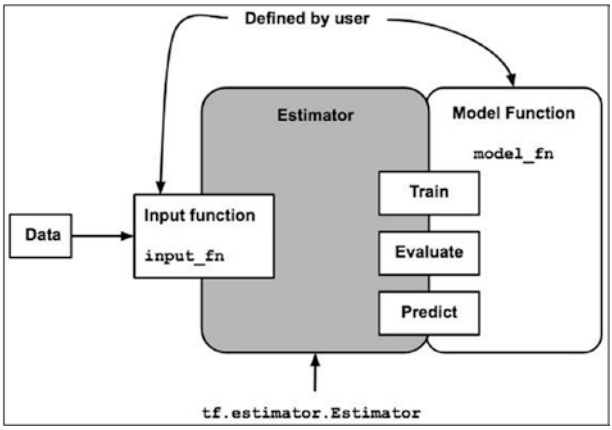


Figure 6-2. *Estimator interface*

As you can see in Figure 6-2, the estimator class provides three interface methods for training, evaluation, and predictions. So, once you develop an estimator object, you will be able to call the train, evaluate, and predict methods on the same object. Note that you need to send

different datasets for each. This is achieved with the help of an input function. I have explained the structure of this input function in more detail later in this section. It is sufficient to say at this point that the introduction of the Input function simplifies your experimentation with different datasets.

The checkpoints created during training allow you to roll back to a known state and continue the training from this checkpoint. This would save you lots of training time, especially when the errors occur toward the fag end of an epoch. This also makes the debugging quicker. After the training is over, the summary logs created during evaluation can be visualized on TensorBoard, giving you quick insights on how good the model is trained.

After the model is trained to your full satisfaction, next comes the task of deployment. Deploying a trained model on CPU/GPU/TPU, even mobiles and the Web, and a distributed environment previously required several code changes. If you use an estimator, you will deploy the trained model as is or at the most with minimal changes on any of these platforms.

Having said all these benefits, let us first look at the types of estimators.

Estimator Types

The estimators are classified into two categories:

- Premade estimators
- Custom estimators

The classification can be visualized in the diagram shown in Figure 6-3.

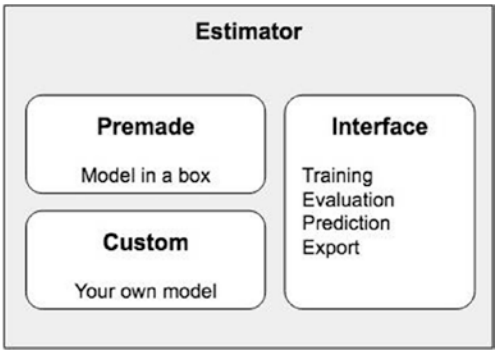


Figure 6-3. *Estimator classifications*

The premade estimators are like a model in a box where the model functionality is already written by the TensorFlow team. On the other hand, in custom estimators, you are supposed to provide this model functionality. In both cases, the estimator object that you create will have a common interface for training, evaluation, and prediction. Both can be exported for serving in a similar way.

The TensorFlow library provides a few premade estimators for your immediate go; if these do not meet your purpose and/or you want to migrate your existing models to get the benefits of estimators, you would create your own estimator class. All estimators are subclasses of the tf.estimator.Estimator base class.

The DNNClassifier, LinearClassifier, and LinearRegressor are the few examples of premade estimators. The DNNClassifier is useful for creating classification models based on dense neural networks, while the LinearRegressor is for handling linear regression problems. You will learn how to use both these classes in the upcoming sections in this chapter.

As part of building a custom estimator, you will be converting an existing Keras model to an estimator. Doing so will enable you to take advantage of the several benefits offered by estimators, which you have

seen earlier. You will be building a custom estimator for the wine quality regression model that you developed in the previous chapter. Finally, I will also show you how to build a custom estimator based on tfhub modules.

To work with the estimators, you need to understand two new concepts – Input functions and Feature columns. The input function creates a data pipeline based on tf.data.dataset that feeds the data to the model in batches for both training and evaluation. You may also create a data pipeline for inference. I will show you how to do this in the DNNClassifier project. Feature columns specify how the data is to be interpreted by the estimator. Before I discuss these Input function and Feature column concepts, I will give you an overview of the development of an estimator-based project.

Workflow for Estimator-Based Projects

The various steps required in the development of an estimator-based project are listed here:

- Loading data
- Data preprocessing
- Defining Features columns
- Defining the Input function
- Model instantiation
- Model training
- Model evaluation
- Judging a model's performance on TensorBoard
- Using a model for predictions

As part of the learning that you have done in all previous chapters, you are definitely familiar with many steps in the preceding workflow. What needs attention is defining Features columns and the Input function. I will now describe these requirements.

Features Column

Features column provide a bridge between raw data and an estimator. It transforms a diverse range of raw data into formats that estimators need. You use the tf.feature_column module to build a list of Feature columns. This then becomes an input to the estimator constructor. The estimator object uses this list while interpreting the data coming from the Input function. The whole process can be visualized as shown in Figure 6-4.

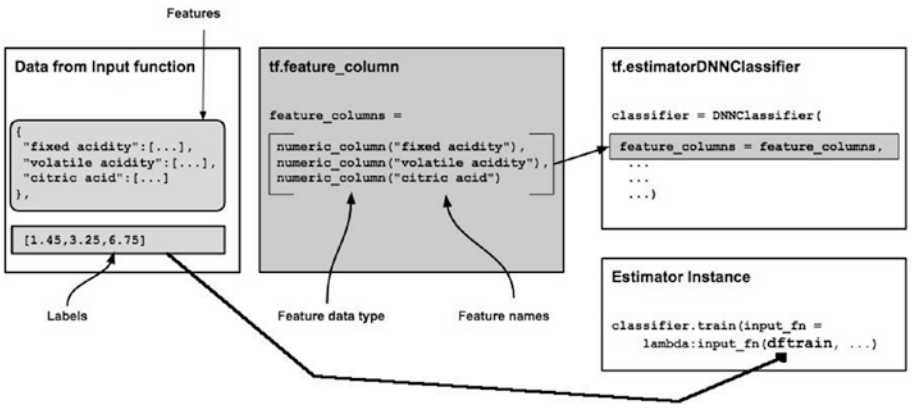


Figure 6-4. *How Features columns are used*

As seen in the central block of Figure 6-4, the list of Feature columns is as follows:

- Fixed acidity (numeric)
- Volatile acidity (numeric)

- Citric acid (numeric)
- ...

These are the features of our wine quality model that you developed in the previous chapter. Each element in the list is of type tf.feature_column.numeric_column. The following code snippet tells you how to build such a list:

```
# build numeric features array
numeric_feature = []
for col in numeric_columns:
 numeric_feature.append
    (tf.feature_column.numeric_column(key=col))
```

Sometimes, your dataset may contain categorical fields which you want to use as the features in your model building. The following code snippet tells you how to build a Features column for such categorical fields:

```
categorical_features = []
for col in categorical_columns:
 vocabulary = data[col].unique()
 cate = tf.feature_column.
        categorical_column_with_vocabulary_list
            (col, vocabulary)
 categorical_features.append
        (tf.feature_column.indicator_column(cate))
```

Note we first obtain the vocabulary by calling the categorical_column_with_vocabulary_list method and then append the indicator columns to the list.

This list is passed as a parameter to the estimator constructor, as seen in Figure 6-4. If your model requires both numerical and categorical features, you will need to append both to your destination Features column.

The left block in Figure 6-4 shows data which is actually built by an Input function. This data will be input to the estimator's train/evaluate/predict method calls.

Next, I will describe how to write the Input function.

Input Function

The purpose of the Input function is to return the following two objects for the use of our estimator model object:

- A dictionary of feature names (keys) and Tensors or Sparse Tensors (values) containing the corresponding feature data
- A Tensor containing one or more labels

The basic skeleton looks like this:

```
def input_fn (dataset):
    # create dictionary with feature names
    # and Tensors with corresponding data
    # create Tensor for label data
    return dictionary, label
```

You may write separate functions based on this prototype for training/evaluation/inferencing.

I will now present you the practical implementation of the Input function taken from the example that is later discussed in this chapter. This would further clarify the concepts behind the Input function in your minds.

The input function is a Python function with the following prototype:

```
def input_fn(features, labels, training =
             True, batch_size = 32):
```

Here the features and labels arguments represent tensors containing the features and the labels data. Within the function, the data is converted into tensor by calling the function from_tensor_slices of the tf.data.Dataset module.

```
#converts inputs to a dataset
Dataset = tf.data.Dataset.from_tensor_slices
            ((dict(features),labels))
```

The input parameter to the function is a Python dictionary of features and the corresponding labels. Finally, the function returns this data to the caller in batches by using the batch method of tf.data.Dataset.

```
return dataset.batch(batch_size)
```

The function structure is visualized in Figure 6-5 for further understanding.

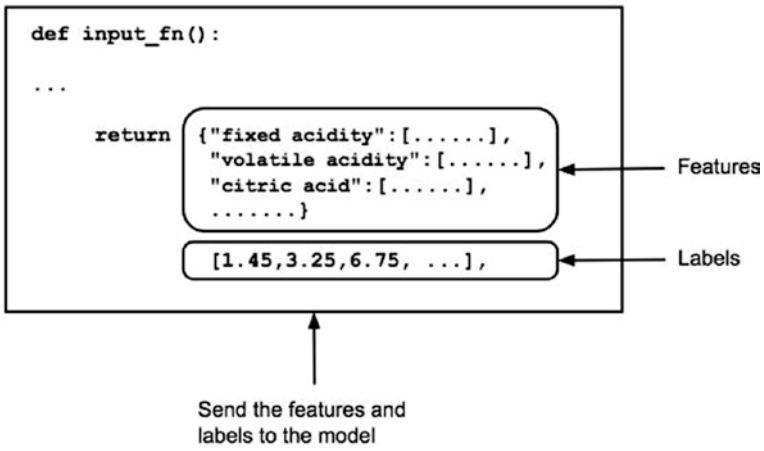


Figure 6-5. *The Input function structure*

The whole matter may look quite complicated; a practical example will help in clarifying the entire implementation, and that is what I am going to do next.

Premade Estimators

I will be discussing two types of premade estimators – one for classification and the other for a regression type of problems. First, I will describe a project for classification. We will use the premade DNNClassifier for this project. The DNNClassifier defines a deep neural network and classifies its input into several classes. You will be using the MNIST database for classifying the handwritten digits into ten numeric digits. The second project that I am going to describe uses the premade estimator called LinearRegressor. The project uses the Airbnb database for the Boston region. The database consists of several houses listed on Airbnb. For each listed house, several features are captured, and the price at which the house is sold/saleable is listed. Using this information, you will develop a regression model to predict the price at which a newly listed house would probably be sold.

The use of these two different models will give you a good insight into how to use the premade estimators on your own model development problems. So let us start with a classification model first.

DNNClassifier for Classification

Create a new Colab project and rename it to DNNClassifier-estimator. As usual, import the TensorFlow using the following statement:

```
import tensorflow as tf
```

We will use the MNIST database for this project. The dataset is available in the sklearn kit. It is the database of the handwritten digits. Our task is to use the premade classifier to recognize the digits embedded in these images. The output consists of ten classes, pertaining to the ten digits.

Loading Data

To load the MNIST data from the sklearn, you use the following code:

```
from sklearn import datasets
digits = datasets.load_digits()
```

You can examine the contents of the loaded data by plotting a few images. The following code will display the first four images:

```
#plotting sample image
import matplotlib.pyplot as plt
plt.figure(figsize=(1,1))
fig, ax = plt.subplots(1,4)
ax[0].imshow(digits.images[0])
ax[1].imshow(digits.images[1])
ax[2].imshow(digits.images[2])
ax[3].imshow(digits.images[3])
plt.show()
```

You will see the output shown in Figure 6-6.

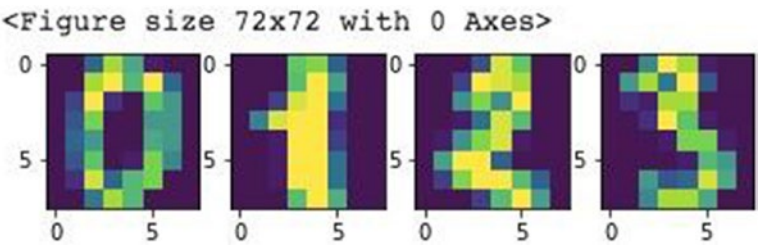


Figure 6-6. *Sample images*

Each image is of size 8 by 8 pixels.

Preparing Data

All images are color images. We do not need the color component to identify the digit. The grayscale images would be fine for our needs. So, we remove the color component by reshaping the images:

```
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))
```

If you check the shape of data, you would notice that there are 1797 images, each consisting of totally 64 pixel values.

We split the data into training and testing using the following code:

```
from sklearn.model_selection
               import train_test_split
X_train, X_test, y_train, y_test =
                   train_test_split(
    data, digits.target, test_size =
                   0.05, shuffle=False)
```

At this point, we are now ready to define our input function for the estimator.

Estimator Input Function

As you have seen earlier, the input function requires data in a particular format. We need to specify a list of columns followed by the actual data as an input to the estimator. The image data consists of 64 pixels. Each pixel will be represented as a numeric column during the model training. So, to create a tensor consisting of this pixel data, first we create names for our pixel columns:

```
# create column names for our model input function
columns = ['p_'+ str(i) for i in range(1,65)]
```

The column names would be simply p_1, p_2, and so on. We will now construct our Features columns by appending a numeric_column type from the tf.feature_column class into an array called feature_columns.

```
feature_columns = []
for col in columns:
 feature_columns.append
    (tf.feature_column.numeric_column(key = col))
```

We define the Input function as follows:

```
def input_fn(features, labels, training =
                True, batch_size = 32):
```

The first parameter defines the features data, the second defines the target values, and the third parameter specifies whether the data is to be used for training or evaluation. The data is processed in batches – the batch size is decided by the last parameter.

We now convert this data into a tensor dataset for more efficient processing by the estimator.

```
#converts inputs to a dataset
 dataset = tf.data.Dataset.from_tensor_slices
                ((dict(features),labels))
```

The data is converted into tensors by calling the from_tensor_slices method. The method takes an input consisting of the dictionary of features and the corresponding labels.

If the input data is to be used for training, we shuffle the dataset.

```
#shuffle and repeat in a training mode
 if training:
   dataset=dataset.shuffle(1000).repeat()
```

The shuffle method shuffles the dataset. We specify the buffer size of 1000 for shuffling. To handle datasets that are too large to fit in memory, the shuffling is done in batches of data. If the buffer size is greater than the number of data points in the dataset, you get a uniform shuffle. If it is 1, you get no shuffling at all.

Finally, we return the batches of data:

```
#giving inputs in batches for training
 return dataset.batch(batch_size)
```

Now comes the time to create an estimator instance.

Creating Estimator Instance

We use the premade DNNClassifier estimator for our purpose. The instance is created using the following statement:

```
classifier = tf.estimator.DNNClassifier
               (hidden_units = [256, 128, 64],
                feature_columns = feature_columns,
                          optimizer = 'Adagrad',
                          n_classes = 10,
                          model_dir = 'classifier')
```

The constructor takes five parameters. The first parameter defines the network architecture. Here, we have defined three hidden layers in our architecture; the first layer contains 256 neurons, the second layer contains 128, and the third layer contains 64. The second parameter specifies the list of features that the data will have. Note that we have earlier created the feature_columns vector, so this is set as a default parameter. The third parameter specifies the optimizer to be used, set to Adagrad by default here. Adagrad is an algorithm for gradient-based optimization that does just this: It adapts the learning rate to the parameters, performing smaller updates. The n_classes parameter defines the number of output classes.

In our case, it is 10, which is the number of digits 0 through 9. The last parameter model_dir specifies the directory name where the logs will be maintained.

Next comes the important part of model training.

Model Training

The estimator model that we created will be trained with our usual train method. Before we start the training, we need to create the input dataset for training. For this, we create a pandas dataframe consisting of the training data and the list of features using the following statement:

```
# create dataframes for training
import pandas as pd
dftrain = pd.DataFrame(X_train, columns = columns)
```

We start the training by calling the train method on the classifier object that we created earlier:

```
classifier.train(input_fn = lambda:input_fn
                              (dftrain,
                            y_train,
                            training = True),
                            steps = 2000)
```

The method takes our input_fn as the parameter. The input_fn itself takes the features and labels data as the first two parameters. The training parameter is set to True so that the data will be shuffled. The number of steps is defined to be 2000. Let me explain what this means. In machine learning, an epoch means one pass over the entire training set. A step corresponds to one forward pass and back again. If you do not create batches in your dataset, a step would correspond to exactly one epoch. However, if you have split the dataset into batches, an epoch will contain many steps – note that a step is an iteration over a single batch of data. You

can compute the total number of epochs executed during the full training cycle with the formula:

$$\text{Number\_of\_epochs} = (\text{batch\_size} * \text{number\_of\_steps}) / (\text{number\_of\_training\_samples})$$

In our current example, the batch size is 32, the number of steps is 2000, and the number of training samples is 1707. Thus, it will take (32 * 2000) / 1707, that is, 38 epochs, to complete the training.

Model Evaluation

To evaluate the model's performance, you will create a dataframe as before, but this time with the testing data as the input:

```
dftest = pd.DataFrame(X_test, columns = columns)
```

The evaluation is performed by calling the evaluate method on the classifier.

```
eval_result = classifier.evaluate(
   input_fn = lambda:input_fn(dftest, y_test,
                              training = False)
)
```

Note that the training parameter is set to false. You can print the evaluation result by simply printing the value of eval_result. The output is shown in the screenshot given in Figure 6-7.

```
eval_result
{'accuracy': 0.93333334,
 'average_loss': 0.2026415,
 'global_step': 2000,
 'loss': 0.19606882}
```

Figure 6-7. *Model evaluation results*

After the evaluation is done, you can check out the various parameters in TensorBoard by loading the logs in the folder that you had specified as a value to the model_dir parameter of the DNNClassifier constructor.

```
%load_ext tensorboard
%tensorboard --logdir ./classifier
```

A sample loss plot from the logs is shown in Figure 6-8.

Figure 6-8. *Sample evaluation metric*

Next, you will learn how to predict an unseen data using our trained estimator.

Predicting Unseen Data

To predict an unseen data, first we will create an input function called pred_input_fn as follows:

```
# An input function for prediction
def pred_input_fn(features, batch_size = 32):
   # Convert the inputs to a Dataset without labels.
   return tf.data.Dataset.from_tensor_slices
(dict(features)).batch(batch_size)
```

The function returns tensor batches of data consisting of only the features and not the labels. We will create two data points for testing the prediction functionality by picking up the items from the dftest dataset as follows:

```
test = dftest.iloc[:2,:]
```

We will pick up the target values for these two data items from the y_test dataset:

```
expected = y_test[:2].tolist()
```

We do the actual prediction by calling the predict method on the estimator object:

```
pred = list(classifier.predict(
   input_fn =  lambda:pred_input_fn(test))
)
```

The input function uses the test data that we create for testing the two unseen data items.

Finally, we print the predicted class for these two items, along with the probability of prediction and the actual target value, using the following loop:

```
for pred_dict, expec in zip(pred, expected):
   class_id = pred_dict['class_ids'][0]
   probability = pred_dict['probabilities']
                   [class_id]
   print('predicted class {} ,
            probability of prediction {} ,
                expected label {}'.format
                (class_id,probability,expec))
```

The output of the execution of the preceding statement is given here:

```
predicted class 8 , probability of prediction
0.9607188701629639 , expected label 8

predicted class 4 , probability of prediction
0.9926437735557556 , expected label 4
```

Experimenting Different ANN Architectures

It is very easy to experiment with different ANN architectures in evaluating a model's performance. For example, you can add one more layer to the previous architecture by changing the value of the hidden_units parameter in the estimator constructor. I tried the following configuration in our existing code:

```
hidden_units = [256, 128, 64, 32],
```

The execution produced the evaluation results shown in Figure 6-9.

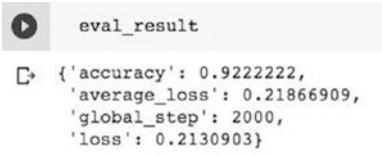


Figure 6-9. *Evaluation results for more dense network*

Not only the number of hidden layers and the neurons in it, you may also add the dropouts by adding a parameter in the estimator instantiation code as shown here:

```
classifier = tf.estimator.DNNClassifier
            (hidden_units = [256, 128, 64, 32],
             feature_columns = feature_columns,
                       optimizer='Adagrad',
                       n_classes=10,
                       dropout = 0.2,
                       model_dir='classifier')
```

The evaluation results for this configuration are shown in Figure 6-10.

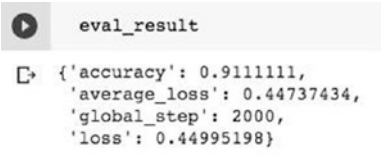


Figure 6-10. *Evaluation metrics after adding dropout*

Thus, you can easily experiment with several model architectures. You may also experiment with different datasets – for example, by changing the number of features included in the Features column list. Once satisfied with the model's performance, you can save it to a file and then take the saved file straightaway to the production server for everybody's use.

Project Source

The full code for the project is given in Listing 6-1 for your quick reference.

Listing 6-1. DNNClassifier-Estimator full source

```
import tensorflow as tf

from sklearn import datasets
digits = datasets.load_digits()

#plotting sample image
import matplotlib.pyplot as plt
plt.figure(figsize=(1,1))
fig, ax = plt.subplots(1,4)
ax[0].imshow(digits.images[0])
ax[1].imshow(digits.images[1])
ax[2].imshow(digits.images[2])
ax[3].imshow(digits.images[3])
plt.show()

# reshape the data to two dimensions
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))

data.shape

# split into training/testing
from sklearn.model_selection
    import train_test_split
        X_train, X_test, y_train, y_test =
            train_test_split(
        data, digits.target, test_size = 0.05,
                shuffle=False)

# create column names for our model input function
  columns = ['p_'+ str(i) for i in range(1,65)]
```

```
feature_columns = []
for col in columns:
 feature_columns.append
    (tf.feature_column.numeric_column(key=col))

def input_fn(features, labels,
             training = True, batch_size = 32):
  #converts inputs to a dataset
  dataset = tf.data.Dataset.from_tensor_slices(
               (dict(features),labels))
  #shuffle and repeat in a training mode
  if training:
    dataset=dataset.shuffle(1000).repeat()

  #giving inputs in batches for training
  return dataset.batch(batch_size)

classifier = tf.estimator.DNNClassifier
             (hidden_units = [256, 128, 64],
              feature_columns = feature_columns,
                        optimizer = 'Adagrad',
                        n_classes = 10,
                        model_dir = 'classifier')

# create dataframes for training
import pandas as pd
dftrain = pd.DataFrame
            (X_train, columns = columns)

classifier.train(input_fn =
           lambda:input_fn(dftrain,
                           y_train,
                           training = True),
                           steps = 2000)
```

```
# create dataframe for evaluation

dftest = pd.DataFrame(X_test, columns = columns)

eval_result = classifier.evaluate(
    input_fn = lambda:input_fn
                (dftest, y_test, training = False)
)

eval_result

%load_ext tensorboard
%tensorboard --logdir ./classifier

# An input function for prediction
def pred_input_fn(features, batch_size = 32):
# Convert the inputs to a Dataset without labels.
    return tf.data.Dataset.from_tensor_slices
              (dict(features)).batch(batch_size)

test = dftest.iloc[:2,:]
#1st two data points for predictions

expected = y_test[:2].tolist()
#expected labels

pred = list(classifier.predict(
    input_fn =  lambda:pred_input_fn(test))
)

for pred_dict, expec in zip(pred, expected):
    class_id = pred_dict['class_ids'][0]
    probability = pred_dict['probabilities'][class_id]
    print('predicted class {} ,
       probability of prediction {} ,
```

```
        expected label {}'.
        format(class_id,probability,expec))
```

Now, as you have seen how to use a dense neural network classification estimator, I will show you how to use a premade classifier for regression problems.

LinearRegressor for Regression

As I mentioned in the previous chapter, neural networks can be used to solve regression problems. To support this claim, we also see a premade estimator in the TF libraries for supporting regression model development. I am going to discuss how to use this estimator in this section.

Project Description

The regression problem that we are trying to solve in this project is to determine the estimated selling price of a house in Boston. We will use the Airbnb dataset for Boston (www.kaggle.com/airbnb/boston) for this purpose. The dataset is multicolumnar and the various columns must be examined carefully for their suitableness as a feature in model development. Thus, for the regression model development, you would need to have a stringent preprocessing of data so that we sufficiently reduce the number of features and yet achieve a great amount of accuracy in predicting the house's price.

Creating Project

As usual, create a Colab project and rename it to DNNRegressor-Estimator. Import the following libraries:

```
import tensorflow as tf
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

Loading Data

You can download the data into your project using the URL specified in the following code:

```
url = 'https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch06/listings.csv'
data = pd.read_csv(url)
```

The data is read into a pandas dataframe. You may check the data size by calling its shape method. You will know that there are 3585 rows and whopping 95 columns. Finding out a regression relationship between these 95 columns is not an easy task even for a highly skilled data analyst. That's where once again you will realize that neural networks can come to your aid.

Out of the 95 columns that we have in the database, obviously not all will be of use to us in training our model. So, we need some data cleansing to remove the unwanted columns and standardize the remaining. We now do the data preprocessing as described further.

Features Selection

The first thing that we do is to list out all columns, which we do by calling the columns method on the data. When you do this, you will see the output in Figure 6-11.

```
# listing column names
data.columns
```

```
Index(['id', 'listing_url', 'scrape_id', 'last_scraped', 'name', 'summary',
       'space', 'description', 'experiences_offered', 'neighborhood_overview',
       'notes', 'transit', 'access', 'interaction', 'house_rules',
       'thumbnail_url', 'medium_url', 'picture_url', 'xl_picture_url',
       'host_id', 'host_url', 'host_name', 'host_since', 'host_location',
       'host_about', 'host_response_time', 'host_response_rate',
       'host_acceptance_rate', 'host_is_superhost', 'host_thumbnail_url',
       'host_picture_url', 'host_neighbourhood', 'host_listings_count',
       'host_total_listings_count', 'host_verifications',
       'host_has_profile_pic', 'host_identity_verified', 'street',
       'neighbourhood', 'neighbourhood_cleansed',
       'neighbourhood_group_cleansed', 'city', 'state', 'zipcode', 'market',
       'smart_location', 'country_code', 'country', 'latitude', 'longitude',
       'is_location_exact', 'property_type', 'room_type', 'accommodates',
       'bathrooms', 'bedrooms', 'beds', 'bed_type', 'amenities', 'square_feet',
       'price', 'weekly_price', 'monthly_price', 'security_deposit',
       'cleaning_fee', 'guests_included', 'extra_people', 'minimum_nights',
       'maximum_nights', 'calendar_updated', 'has_availability',
       'availability_30', 'availability_60', 'availability_90',
       'availability_365', 'calendar_last_scraped', 'number_of_reviews',
       'first_review', 'last_review', 'review_scores_rating',
       'review_scores_accuracy', 'review_scores_cleanliness',
       'review_scores_checkin', 'review_scores_communication',
       'review_scores_location', 'review_scores_value', 'requires_license',
       'license', 'jurisdiction_names', 'instant_bookable',
       'cancellation_policy', 'require_guest_profile_picture',
       'require_guest_phone_verification', 'calculated_host_listings_count',
       'reviews_per_month'],
      dtype='object')
```

Figure 6-11. *List of columns*

If you prefer, you may use data.info() to get a better understanding of the data structure. You can easily notice that there are many fields that may be irrelevant to us. To get a better idea on what columns can be eliminated, get the data description by calling the describe method. The screen output is shown in Figure 6-12.

```
# understanding data
data.describe()
```

	id	scrape_id	host_id	host_listings_count	host_total_listings_count	neighbourhood_group_cleansed	latitude
count	3.585000e+03	3.585000e+03	3.585000e+03	3585.000000	3585.000000	0.0	3585.000000
mean	8.440875e+06	2.016091e+13	2.492311e+07	58.902371	58.902371	NaN	42.340032
std	4.500787e+06	8.516813e-01	2.292781e+07	171.119663	171.119663	NaN	0.024403
min	3.353000e+03	2.016091e+13	4.240000e+03	0.000000	0.000000	NaN	42.235942
25%	4.679319e+06	2.016091e+13	6.103425e+06	1.000000	1.000000	NaN	42.329995
50%	8.577620e+06	2.016091e+13	1.928100e+07	2.000000	2.000000	NaN	42.345201
75%	1.278953e+07	2.016091e+13	3.622147e+07	7.000000	7.000000	NaN	42.354685
max	1.493346e+07	2.016091e+13	9.385411e+07	749.000000	749.000000	NaN	42.389982

Figure 6-12. *Data description*

After the careful examination of the data, I decided to use only 17 columns for our analytics. In practice, you would use any of the known feature selection techniques – just to say univariate selection or correlation matrix with heatmap. These columns are selected by explicitly typing out their names as in the following statement:

```
# Selecting only few columns as our features
data = data[['property_type','room_type',
         'bathrooms','bedrooms','beds','bed_type',
         'accommodates','host_total_listings_count',
         'number_of_reviews','review_scores_value',
         'neighbourhood_cleansed','cleaning_fee',
         'minimum_nights','security_deposit',
         'host_is_superhost','instant_bookable',
         'price']]
```

After selecting our features, we need to ensure that the data it contains is clean.

Data Cleansing

We first check if the data contains any null values. You do this by using the isnull function and taking the sum on its output. The screenshot in Figure 6-13 shows the columns containing null values.

```
data.isnull().sum() #nan values
```

```
property_type                 3
room_type                     0
bathrooms                    14
bedrooms                     10
beds                          9
bed_type                      0
accommodates                  0
host_total_listings_count     0
number_of_reviews             0
review_scores_value         821
neighbourhood_cleansed        0
cleaning_fee               1107
minimum_nights                0
security_deposit           2243
host_is_superhost             0
instant_bookable              0
price                         0
dtype: int64
```

Figure 6-13. *Checking null values*

As you can see in Figure 6-13, seven columns have null values, where the sum does not equal to zero. As the security_deposit column has 2243 nulls out of 3585 records, a very large percentage, we will eliminate this column in our analysis.

```
data = data.drop('security_deposit' , axis = 1)
```

We can examine the data by printing a few records. The command and its output are shown in Figure 6-14.

```
data.head()
```

```
data.head()
```

	property_type	room_type	bathrooms	bedrooms	beds	bed_type	accommodates	host_total_listings_count	number_of_reviews	review_sc
0	House	Entire home/apt	1.5	2.0	3.0	Real Bed	4	1	0	
1	Apartment	Private room	1.0	1.0	1.0	Real Bed	2	1	36	
2	Apartment	Private room	1.0	1.0	1.0	Real Bed	2	1	41	
3	House	Private room	1.0	1.0	2.0	Real Bed	4	1	1	
4	House	Private room	1.5	1.0	2.0	Real Bed	2	1	29	

Figure 6-14. *Examining data*

When you examine this data, you will notice that the property_type field is categorical. We will deal with it separately during our cleansing operation. You also notice that the cleaning_fee, security_deposit, and price columns also contain a $ sign. These values must be converted into a floating-point number after stripping the $ symbol. Also, notice that the amount fields also contain a comma in between the digits. This also needs to be stripped. For cleaning up the values in these three columns, we write a transform function as shown here:

```
# function to remove $ and , characters
def transform(x):
 x = str(x)
 x = x.replace('$','')
 x = x.replace(",","")
 return float(x)
```

We now apply this transformation on the three fields in a for loop:

```
for col in ["cleaning_fee","price"]:
  data[col] = data[col].apply(transform)
  #filling nan with mean value
  data[col].fillna(data[col].mean(),inplace = True)
```

In the loop, we also replace the null values with the field's mean value.

For the rest of the columns, we just replace null values with the column's mean using the following code:

```
#filling nan values with mean value
for feature in ["bathrooms","bedrooms","beds",
                "review_scores_value"]:
 data[feature].fillna(data[feature].mean(),
               inplace = True)
```

Next, we look into the categorical column – property_type. As this is a categorical column, we cannot simply replace null values with a mean. We need to find an appropriate value for the replacement of the null fields. For this, let us check what the unique values are in this column. The unique values can be seen in Figure 6-15.

```
data['property_type'].unique()
```

```
array(['House', 'Apartment', 'Condominium', 'Villa', 'Bed & Breakfast',
       'Townhouse', 'Entire Floor', 'Loft', 'Guesthouse', 'Boat', 'Dorm',
       'Other', nan, 'Camper/RV'], dtype=object)
```

Figure 6-15. *Unique values in categorical columns*

We can check the frequency of each value by calling the value_counts method. This is shown in Figure 6-16.

```
data['property_type'].value_counts()
```

```
Apartment          2612
House               562
Condominium         231
Townhouse            54
Bed & Breakfast      41
Loft                 39
Other                17
Boat                 12
Villa                 6
Entire Floor          4
Dorm                  2
Guesthouse            1
Camper/RV             1
Name: property_type, dtype: int64
```

Figure 6-16. *Frequency distribution for property_type values*

As the Apartment word occurs most frequently, we will use this as a replacement text to our null values in this column. We do this replacement using the following code statement:

```
# replacing nan with Apartment
data['property_type'].fillna('Apartment',
                             inplace = True)
```

Creating Datasets

Now, we are ready to extract features and labels from the preprocessed data. We do this using the following two statements:

```
feature = data.drop('price', axis = 1)
#input data
target = data['price']
```

There is one last thing that I would like to show you as a part of data preprocessing. We want to predict the price for a newly signed up apartment. So, let us look at the price distribution in our database. You can plot the histogram of the prices using the following code:

```
# price value histogram
data['price'].plot(kind='hist',grid = True)
plt.title('price distribution')
plt.xlabel('price')
```

The histogram is shown in Figure 6-17.

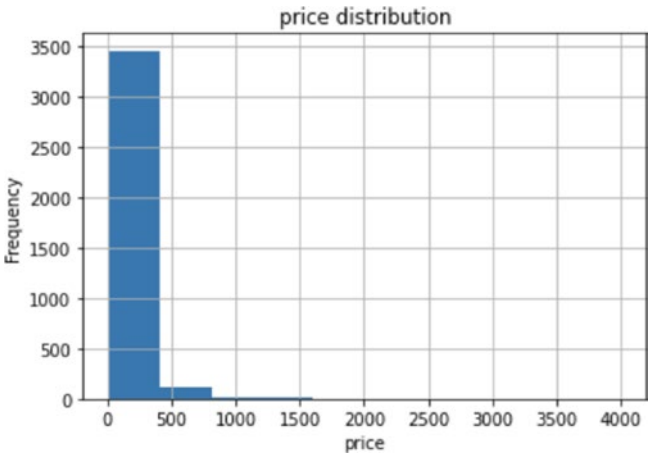


Figure 6-17. *Original price distribution*

As you can see in Figure 6-17, most of the price values are in the lower range of the price spectrum. For better learning, these prices should have a better distribution rather than the skewed distribution that they have now. This is done by taking the logarithm of the price as shown in the following code:

```
target = np.log(data.price) #output data
target.hist()
plt.title
    ('price distribution after log transformation')
```

After the transformation, the distribution looks like the one shown in Figure 6-18.

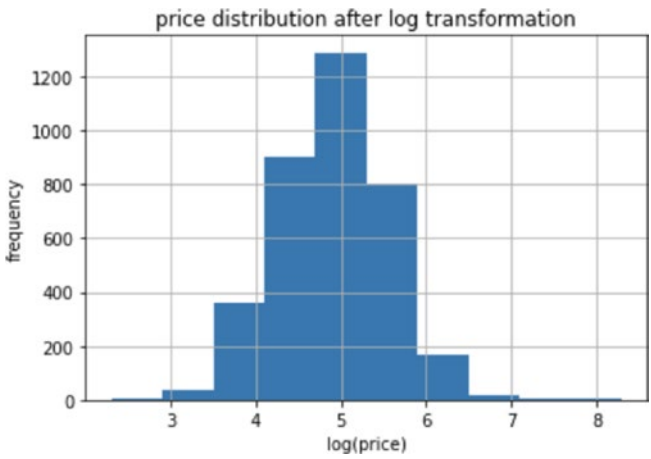


Figure 6-18. *Price distribution after transformations*

At this point, your data processing is complete. We now create the training and testing datasets.

To create the training and testing datasets, we use the train_test_split method of sklearn.

```
#splitting input and output data for training and testing
from sklearn.model_selection import train_test_split
xtrain,xtest,ytrain,ytest=train_test_split
    (feature, target, test_size = 0.2,
     random_state = 42)
```

We reserve 20% of our data for testing. The last thing that we need to do before building our estimator is to create Feature columns, which we do next.

Building Feature Columns

In the previous application, you created a Features column as a simple list of feature names. All the columns in that list were numeric. The current dataset has both numeric and categorical features. First, we build the list of numeric columns:

```
# selecting numerical feature columns
numeric_columns = feature.select_dtypes
      (include = np.number).columns.tolist()
numeric_columns
```

We include all columns with type np.number from our feature list and create a new list called numeric_columns. When you print this list, you would see the output shown in Figure 6-19.

```
['bathrooms',
 'bedrooms',
 'beds',
 'accommodates',
 'host_total_listings_count',
 'number_of_reviews',
 'review_scores_value',
 'cleaning_fee',
 'minimum_nights']
```

Figure 6-19. *Numeric features array*

Note the names of all numeric columns from our feature list are added to the new list. Next, we will convert this array into a format required by our input function for the estimator. The input function requires that the list of features with their data types (tf.feature_column) should be included in the list. We build this list using a for loop as follows:

```
# build numeric features array
numeric_feature = []
```

```
for col in numeric_columns:
 numeric_feature.append
    (tf.feature_column.numeric_column(key=col))
numeric_feature
```

If we print this list, you would see the output shown in Figure 6-20.

```
[NumericColumn(key='bathrooms', shape=(1,), default_value=None, dtype=tf.float32, normalizer_fn=None),
 NumericColumn(key='bedrooms', shape=(1,), default_value=None, dtype=tf.float32, normalizer_fn=None),
 NumericColumn(key='beds', shape=(1,), default_value=None, dtype=tf.float32, normalizer_fn=None),
 NumericColumn(key='accommodates', shape=(1,), default_value=None, dtype=tf.float32, normalizer_fn=None),
 NumericColumn(key='host_total_listings_count', shape=(1,), default_value=None, dtype=tf.float32, normalizer_fn=None),
 NumericColumn(key='number_of_reviews', shape=(1,), default_value=None, dtype=tf.float32, normalizer_fn=None),
 NumericColumn(key='review_scores_value', shape=(1,), default_value=None, dtype=tf.float32, normalizer_fn=None),
 NumericColumn(key='cleaning_fee', shape=(1,), default_value=None, dtype=tf.float32, normalizer_fn=None),
 NumericColumn(key='minimum_nights', shape=(1,), default_value=None, dtype=tf.float32, normalizer_fn=None),
 NumericColumn(key='security_deposit', shape=(1,), default_value=None, dtype=tf.float32, normalizer_fn=None)]
```

Figure 6-20. *Features column*

Likewise, we will create a list of categorical columns present in our feature list. We select and build the list of column names having categorical attributes as follows:

```
#selecting categorical feature columns
categorical_columns = feature.select_dtypes
          (exclude=np.number).columns.tolist()
categorical_columns
```

When you print this list, you will see the output shown in Figure 6-21.

```
['property_type',
 'room_type',
 'bed_type',
 'neighbourhood_cleansed',
 'host_is_superhost',
 'instant_bookable']
```

Figure 6-21. *Categorical columns list*

Note that there are six columns having categorical values. Now, we build the Features column having these categorical fields with the following for loop:

```
categorical_features = []
for col in categorical_columns:
 vocabulary = data[col].unique()
 cate = tf.feature_column.
          categorical_column_with_vocabulary_list
          (col, vocabulary)
 categorical_features.append
          (tf.feature_column.indicator_column(cate))
```

We first extracted the unique values in the categorical column, then built the vocabulary list, and finally added it to the categorical_features list. If you print this list, you would see the output shown in Figure 6-22.

Figure 6-22. *Features column for categorical columns*

Finally, we combine both numeric and categorical features into a single list for inputting as a parameter to our input function that is defined next.

```
# combining both features as our final features list
features = categorical_features +
            numeric_feature
```

Defining Input Function

The input function is defined as follows:

```
# An input function for training and evaluation
def input_fn(features,labels,training = True,batch_size = 32):
 #converts inputs to a dataset
 dataset=tf.data.Dataset.from_tensor_slices
            ((dict(features), labels))

 #shuffle and repeat in a training mode
 if training:
   dataset=dataset.shuffle(10000).repeat()

 # return batches of data
 return dataset.batch(batch_size)
```

As seen in the previous project, the function takes features and labels as the first two parameters. The training parameter determines whether the data is to be used for training; if so, the data is shuffled. The from_tensor_slices function call creates a tensor for the data pipeline to our model. The function returns the data in batches.

Creating Estimator Instance

We now create the instance of a premade estimator using the following statement:

```
linear_regressor = tf.estimator.LinearRegressor(
   feature_columns = features,
   model_dir = "linear_regressor")
```

Note we use the LinearRegressor class as a premade estimator for regression. The first parameter is the Features columns, which specify the combined numerical and categorical columns list. The second parameter is the name of the directory where the logs would be maintained.

Model Training

We train the model by calling the train method:

```
linear_regressor.train(input_fn = lambda:input_fn(xtrain,
                       ytrain,
                       training = True),
                       steps = 2000)
```

The input function takes the features data in the xtrain dataset and the target values in the ytrain dataset. The training parameter is set to true that enables data shuffling. The number of steps would decide the number of epochs in the training phase.

Model Evaluation

We evaluate the model by calling the evaluate method:

```
linear_regressor.evaluate(
   input_fn = lambda:input_fn
              (xtest, ytest, training = False)
)
```

This time, the input function uses xtest and ytest datasets for evaluation purposes. A typical evaluation output would be as follows:

```
{'average_loss': 0.18083459,
 'global_step': 2000,
 'label/mean': 4.9370537,
 'loss': 0.1811692,
 'prediction/mean': 4.956979}
```

After the evaluation completes, we can see the various metrics by loading the TensorBoard.

```
%load_ext tensorboard
%tensorboard --logdir ./linear_regressor
```

A loss curve displayed on the TensorBoard is shown in Figure 6-23.

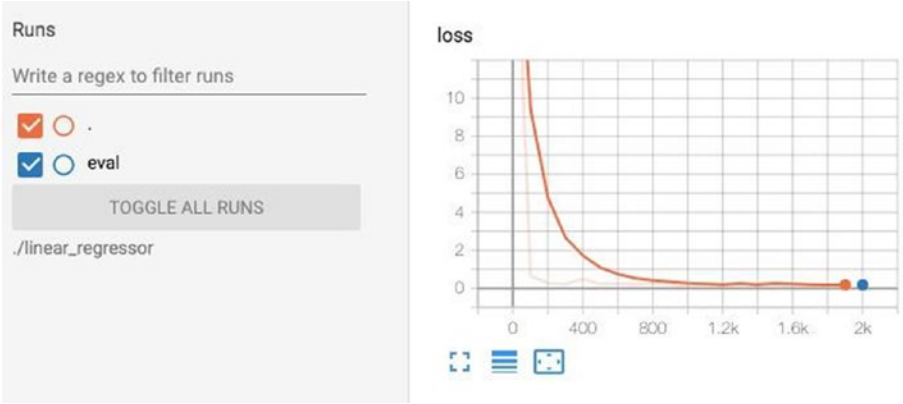


Figure 6-23. *Loss metrics*

Project Source

The full code for the project is given in Listing 6-2 for your quick reference.

Listing 6-2. LinearRegressor-Estimator full source

```
import tensorflow as tf
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

```
url = 'https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch06/listings.csv'
data = pd.read_csv(url)

# listing column names
data.columns

# understanding data
data.describe()

# Selecting only few columns as our features
data = data[['property_type','room_type',
          'bathrooms','bedrooms','beds','bed_type',
          'accommodates','host_total_listings_count',
          'number_of_reviews','review_scores_value',
          'neighbourhood_cleansed','cleaning_fee',
          'minimum_nights','security_deposit',
          'host_is_superhost','instant_bookable',
          'price']]

data.isnull().sum() #nan values

data.head()

# function to remove $ and , characters
def transform(x):
  x = str(x)
  x = x.replace('$','')
  x = x.replace(",","")
  return float(x)
```

```
for col in ["cleaning_fee","security_deposit","price"]:
   data[col] = data[col].apply(transform)
# apply function
   #filling nan with mean value
   data[col].fillna(data[col].mean(),inplace = True)

#filling nan values with mean value
for feature in ["bathrooms","bedrooms","beds",
                "review_scores_value"]:
  data[feature].fillna(data[feature].mean(),
                inplace = True)

data['property_type'].unique()

# get frequency of all unique values
data['property_type'].value_counts()

# replacing nan with Apartment
data['property_type'].fillna
           ('Apartment', inplace = True)

feature = data.drop('price', axis = 1)
#input data
target = data['price']

# price value histogram
data['price'].plot(kind='hist',grid = True)
plt.title('price distribution')
plt.xlabel('price')

# make a log transformation to remove skew.
target = np.log(data.price) #output data
target.hist()
plt.title('price distribution after log transformation')
```

```
#splitting input and output data for training and testing
from sklearn.model_selection
             import train_test_split
             xtrain,xtest,ytrain,ytest =
             train_test_split
             (feature, target, test_size = 0.2,
             random_state = 42)

# selecting numerical feature columns
numeric_columns = feature.select_dtypes
           (include = np.number).columns.tolist()
numeric_columns

# build numeric features array
numeric_feature = []
for col in numeric_columns:
  numeric_feature.append
        (tf.feature_column.numeric_column(key=col))
numeric_feature

#selecting categorical feature columns
categorical_columns = feature.select_dtypes
      (exclude=np.number).columns.tolist()
categorical_columns

categorical_features = []
for col in categorical_columns:
  vocabulary = data[col].unique()
  cate = tf.feature_column.
            categorical_column_with_vocabulary_list
            (col, vocabulary)
  categorical_features.append
         (tf.feature_column.indicator_column(cate))
```

```
categorical_features

# combining both features as our final features list
features = categorical_features +
           numeric_feature

# An input function for training and evaluation
def input_fn(features,labels,training = True,batch_size = 32):
  #converts inputs to a dataset
  dataset=tf.data.Dataset.from_tensor_slices(
                (dict(features), labels))

  #shuffle and repeat in a training mode
  if training:
    dataset=dataset.shuffle(10000).repeat()

  # return batches of data
  return dataset.batch(batch_size)

linear_regressor = tf.estimator.LinearRegressor(
    feature_columns = features,
    model_dir = "linear_regressor")

linear_regressor.train(input_fn = lambda:input_fn(xtrain,
                ytrain,
                training = True),
                steps = 2000)

linear_regressor.evaluate(
    input_fn = lambda:input_fn
            (xtest, ytest, training = False)
)

%load_ext  tensorboard
%tensorboard --logdir ./linear_regressor
```

This project has demonstrated how to use a premade LinearRegressor for solving the linear regression problems. Next, I will describe how to build a custom estimator of your own.

Custom Estimators

In the previous chapter, you developed a wine quality classifier. You used three different architectures – small, medium, and large – to compare their results. I will now show you how to convert these existing Keras models to a custom estimator to take advantage of the facilities provided by the estimator.

Creating Project

Create a new Colab project and rename it to ModelToEstimator. Add the following imports to the project:

```
import tensorflow as tf
import pandas as pd
from sklearn.model_selection
             import train_test_split
from sklearn.preprocessing
             import StandardScaler
```

Loading Data

We will use the white wine quality that you have used previously. The data is downloaded to your project from the UCI Machine Learning Repository using the following code:

```
data_url='https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/Ch05/winequality-
white.csv'
data=pd.read_csv(data_url,delimiter=';')
```

As you are already familiar with the data, I am not going to describe it again. I will not also include any data preprocessing here. I will straightaway move on to features selection.

Creating Datasets

As you know from the previous chapter, the quality field in the database is used as a target label, and the rest of the fields are used as features. To separate out features and the target label, we used the following code:

```
x = data.iloc[:,:-1]
y = data.iloc[:,-1]
```

All selected features, which are numeric, must now be scaled to a normal distribution.

```
sc = StandardScaler()
x = sc.fit_transform(x)
```

We now create the training/testing datasets by using the train_test_split method.

```
xtrain, xtest, ytrain, ytest =
      train_test_split(x, y, test_size =
                       0.3,random_state = 20)
```

We capture the shape of xtrain in a variable; this is required while defining the Keras sequential model.

```
input_shape = xtrain.shape[1]
```

Defining Model

We use the small model definition from our previous example. The model definition is given here:

```
small_model = tf.keras.models.Sequential([
             tf.keras.layers.Dense
                (64,activation = 'relu',
                input_shape = (input_shape,)),
             tf.keras.layers.Dense(1)
])
```

The output consists of a single neuron which emits a float value for the wine quality. Note that we are treating this problem as a regression problem like the wine quality problem discussed in the previous chapter.

We compile the model by calling its compile method:

```
small_model.compile
            (loss = 'mse', optimizer = 'adam')
```

We consider this as a regression problem and thus use mse for our loss function. We will use this compiled model in the estimator instantiation. Before that, we define the input function.

Defining Input Function

The input function like in the preceding example is defined as follows:

```
def input_fn(features, labels,
           training = True, batch_size = 32):
 #converts inputs to a dataset
 dataset = tf.data.Dataset.from_tensor_slices
           (({'dense_input':features},labels))
  #shuffle and repeat in a training mode
```

```
if training:
  dataset = dataset.shuffle(1000).repeat()

#giving inputs in batch for training
return dataset.batch(batch_size)
```

The function definition is identical to the earlier example and does not need any further explanation.

Now, it is time for us to convert our model to the estimator.

Model to Estimator

To convert the existing Keras models to an estimator instance, the TF libraries provide a function called model_to_estimator. The function call is shown in the following code statement:

```
keras_small_estimator = tf.keras.estimator.model_to_estimator(
   keras_model = small_model,
        model_dir = 'keras_small_classifier')
```

The function takes two arguments; the first argument specifies the previously existing compiled Keras model, and the second argument specifies the folder name where the logs would be maintained during the training. The function call returns an estimator instance which you would use like in the earlier example to train, evaluate, and predict.

Model Training

We train the model by calling the train method on the estimator instance like what we did in the earlier example.

```
keras_small_estimator.train
            (input_fn = lambda:input_fn
            (xtrain, ytrain), steps = 2000)
```

For training, the input function takes the features data in xtrain and the labels in ytrain. The number of steps is set to 2000, which decides the number of training epochs. The mode set by the training argument takes the default value of True.

Evaluation

After the training completes, you can evaluate the model's performance by calling the evaluate method on the estimator instance.

```
eval_small_result =
            keras_small_estimator.evaluate(
   input_fn = lambda:input_fn
           (xtest, ytest, training = False),
                steps=1000)
print('Eval result: {}'.format
           (eval_small_result))
```

The input function this time specifies the value for the training argument which is set to False. The testing data created earlier is used for evaluation. After the evaluation, the results are printed on the console. You may check out the various metrics generated by the evaluation using TensorBoard.

```
%load_ext tensorboard
%tensorboard --logdir ./keras_small_classifier
```

Note that we pick up the metrics from the earlier specified logs folder.

Project Source

The full code for the project is given in Listing 6-3 for your quick reference.

Listing 6-3. ModelToEstimator full source

```
import tensorflow as tf
import pandas as pd
from sklearn.model_selection
        import train_test_split
from sklearn.preprocessing
        import StandardScaler

data_url='https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/Ch05/winequality-
white.csv'
data=pd.read_csv(data_url,delimiter=';')

x = data.iloc[:,:-1]
y = data.iloc[:,-1]

sc = StandardScaler()
x = sc.fit_transform(x)

xtrain, xtest, ytrain,
        ytest = train_test_split
        (x, y, test_size = 0.3,random_state = 20)

input_shape = xtrain.shape[1]

small_model = tf.keras.models.Sequential([
              tf.keras.layers.Dense
                 (64,activation = 'relu',
                  input_shape = (input_shape,)),
              tf.keras.layers.Dense(1)
])
```

```
small_model.compile
        (loss = 'mse', optimizer = 'adam')

def input_fn(features, labels, training =
             True, batch_size = 32):
  #converts inputs to a dataset
  dataset = tf.data.Dataset.from_tensor_slices
         (({'dense_input':features},labels))

  #shuffle and repeat in a training mode
  if training:
    dataset = dataset.shuffle(1000).repeat()

  #giving inputs in batch for training
  return dataset.batch(batch_size)

keras_small_estimator = tf.keras.estimator.model_to_estimator(
    keras_model = small_model, model_dir =
                  'keras_small_classifier')

keras_small_estimator.train
    (input_fn = lambda:input_fn
                (xtrain, ytrain), steps = 2000)

eval_small_result =
         keras_small_estimator.evaluate(
    input_fn = lambda:input_fn
               (xtest, ytest, training = False),
                    steps=1000)
print('Eval result: {}'.format
        (eval_small_result))

%load_ext tensorboard
%tensorboard --logdir ./keras_small_classifier
```

Custom Estimators for Pre-trained Models

In Chapter 4, you saw the use of pre-trained models from TF Hub. You have seen how to reuse these models in your own model definitions. It is also possible to convert these models into an estimator while extending their architecture. The following code shows you how to extend the VGG16 model and create a custom estimator on the extended model. VGG16 is a state-of-the-art deep learning which is trained to classify an image into 1000 categories. Suppose you just want to classify your datasets into just two categories – cats and dogs. So, we need just a binary classification. For this, we will need to change the output from categorical to binary. This example shows you how to replace the existing output layer of VGG16 with a single neuron Dense layer.

Creating Project

Create a new Colab project and rename it to tfhub-custom-estimator. Import TensorFlow as usual.

```
import tensorflow as tf
```

Importing VGG16

Import the VGG16 trained model into your project with the following statement:

```
keras_Vgg16 = tf.keras.applications.VGG16(
   input_shape=(160, 160, 3), include_top=False)
```

The input shape to the model is set as per the original model specification. The include_top parameter specifies that the top layer of the model be eliminated. Obviously, we do not want to regenerate the weights for this already trained model. So, we set the trainable parameter to false:

```
keras_Vgg16.trainable = False
```

Building Your Model

We will now build our model on top of the loaded VGG16 model:

```
estimator_model = tf.keras.Sequential([
   keras_Vgg16,
   tf.keras.layers.GlobalAveragePooling2D(),
   tf.keras.layers.Dense(256),
   tf.keras.layers.Dense(1)
])
```

We build a sequential model by taking the VGG16 as the base layer. On top of this, we add a pooling layer, followed by two Dense layers. The last layer is a binary output layer.

You may print the summaries of the two models to see the changes you have made.

The original model summary is printed with the following statement:

```
keras_Vgg16.summary()
```

The output is shown in Figure 6-24.

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 160, 160, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 160, 160, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 160, 160, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 80, 80, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 80, 80, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 80, 80, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 40, 40, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 40, 40, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 40, 40, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 40, 40, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 20, 20, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 20, 20, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 20, 20, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 20, 20, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 10, 10, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 10, 10, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 10, 10, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 10, 10, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 5, 5, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
```

Figure 6-24. *Model summary for VGG16*

As you can see in Figure 6-24, the VGG16 models have several hidden layers, and the total number of trainable parameters equals more than 14 million. You can imagine the time and resources it would have taken to train this model. Obviously, in your application, when you use this model, you will never think of retraining this model.

You can now print the summary of your newly built model.

```
estimator_model.summary()
```

The output is shown in Figure 6-25.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Model)                (None, 5, 5, 512)         14714688
_________________________________________________________________
global_average_pooling2d (Gl (None, 512)               0
_________________________________________________________________
dense (Dense)                (None, 256)               131328
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 257
=================================================================
Total params: 14,846,273
Trainable params: 131,585
Non-trainable params: 14,714,688
_________________________________________________________________
```

Figure 6-25. *Model summary for the extended model*

In your model, you have only about 100,000 trainable parameters.

Compiling Model

The model is compiled as usual, with the desired optimizer, loss function, and metrics:

```
# Compile the model
estimator_model.compile(
   optimizer = 'adam',
   loss=tf.keras.losses.BinaryCrossentropy
            (from_logits = True),
   metrics = ['accuracy'])
```

Creating Estimator

You create an estimator using the model_to_estimator method:

```
est_vgg16 = tf.keras.estimator.model_to_estimator
            (keras_model = estimator_model)
```

Before you train the estimator, you need to process your images.

Processing Data

The model requires images of size 160x160 pixels. We preprocess the image by using the following preprocess function given in TF samples:

```
IMG_SIZE = 160
import tensorflow_datasets as tfds
def preprocess(image, label):
 image = tf.cast(image, tf.float32)
 image = tf.image.resize
            (image, (IMG_SIZE, IMG_SIZE))
 return image, label
```

The following Input function will define the data pipeline for the dogs and cats training data, which is available in the TensorFlow built-in datasets:

```
def train_input_fn(batch_size):
 data = tfds.load('cats_vs_dogs',
          as_supervised=True)
 train_data = data['train']
 train_data = train_data.map(preprocess).shuffle(500).batch
                      (batch_size)
 return train_data
```

Training/Evaluation

You train the estimator by calling its train method.

```
est_vgg16.train(input_fn =
       lambda: train_input_fn(32), steps = 500)
```

After the model is trained, you can evaluate the model's performance:

```
est_vgg16.evaluate(input_fn = lambda: train_input_fn(32),
steps=10)
```

Note that I have used the same training dataset with a different step size for model evaluation, as no separate testing dataset is available for the purpose.

The evaluation produces the following results:

```
{'accuracy': 0.96875, 'global_step': 500, 'loss': 0.27651623}
```

Project Source

The full project source is given in Listing 6-4.

Listing 6-4. VGG16-custom-estimator full source

```
import tensorflow as tf

keras_Vgg16 = tf.keras.applications.VGG16(
   input_shape=(160, 160, 3), include_top=False)
keras_Vgg16.trainable = False

estimator_model = tf.keras.Sequential([
   keras_Vgg16,
   tf.keras.layers.GlobalAveragePooling2D(),
```

```
   tf.keras.layers.Dense(256),
   tf.keras.layers.Dense(1)
])

keras_Vgg16.summary()

estimator_model.summary()
# Compile the model
estimator_model.compile(
   optimizer = 'adam',
   loss=tf.keras.losses.BinaryCrossentropy
           (from_logits = True),
   metrics = ['accuracy'])

est_vgg16 = tf.keras.estimator.model_to_estimator
              (keras_model = estimator_model)

IMG_SIZE = 160
import tensorflow_datasets as tfds
def preprocess(image, label):
 image = tf.cast(image, tf.float32)
 image = tf.image.resize
            (image, (IMG_SIZE, IMG_SIZE))
 return image, label

def train_input_fn(batch_size):
 data = tfds.load('cats_vs_dogs',
          as_supervised = True)
 train_data = data['train']
 train_data = train_data.map(preprocess).shuffle(500).batch
          (batch_size)
 return train_data
```

```
est_vgg16.train(input_fn = lambda: train_input_fn(32),
steps=500)

est_vgg16.evaluate(input_fn = lambda: train_input_fn(32),
steps=10)
```

This trivial example has demonstrated how you can use well-trained models into your own models.

Summary

The estimators facilitate model development by providing a unified interface for training, evaluation, and prediction. They provide a separation of data pipeline from the model development, thus allowing you to experiment with different datasets easily. The estimators are classified as premade and custom. In this chapter, you learned to use the premade estimators for both classification and regression types of problems. The custom estimators are used for migrating the existing models to the estimator interface to take advantage of the benefits offered by estimators. An estimator-based model can be trained on a distributed environment or even on a CPU/GPU/TPU. Once you develop your model using the estimator, it can be deployed easily even on a distributed environment without any code changes. You also learned to use a well-trained model like VGG16 in your own model building. It is recommended that you use the pre-trained model wherever possible. If it does not meet your purpose, then think of creating custom estimators.

CHAPTER 7

Text Generation

In this book, so far you have used vanilla neural networks, the networks that do not have the ability to remember their past. They accept a fixed-sized vector as input and produce a fixed-sized output. Consider the case of image classification where the input is an image and the output is one of the classes for which the model has been trained. Now, consider a situation where the prediction requires the knowledge of the previous predictions. To give you an example, consider you are watching a movie. Your mind keeps guessing what will be the next scene. The guess depends on what has happened not just in the near past but something that also happened 15 minutes ago or even an hour for a long movie. Our vanilla neural networks, in the way they work, do not have memory to remember what happened in the past and to apply that knowledge to the current guess.

Take another example; I say I am from India. Okay, you have gained the knowledge that I am from India. After some time, you are asked to guess what language I speak. The probability of answering it to be the national language of India depends largely on the fact that you still remember that I am from India. The neural networks that we have seen so far are in no way in a position to solve such kind of problems. That is where the recurrent neural networks (RNNs) and a special version of it called long short-term memory (LSTM) came into the picture. In this chapter, you will be introduced to both RNN and LSTM, followed by an example on text generation. All subsequent chapters too will use these networks in

P. Sarang, *Artificial Neural Networks with TensorFlow 2*,
https://doi.org/10.1007/978-1-4842-6150-7_7

solving the problems in different domains. So a good understanding of this chapter will help you understand the rest better.

In this chapter, we will train RNN character-level language models. We will give a chunk of text as an input to the model, train the model to understand the sequencing of characters in the text, and then later on ask it to predict the probability of occurrence of a character for a user-defined sequence of characters. In short, if you are asked to predict the character that follows the character sequence "hell", you would most likely come up with the answer "o". That is the probability of the character "o" being the highest among the rest of the alphabets. We will train our network to perform such predictions.

Specifically, I will discuss two text generation applications; the first application is a trivial one for you to get introduced to the nitty-gritties of the process. In this trivial application, you would train the model to generate baby names, based on its knowledge of the existing names. The second application is a more daring attempt where we will try to create a new novel after making the model learn the semantics and style of writing from a well-known novel.

So, first let us try to understand what a recurrent neural network is.

Recurrent Neural Networks

Like Convolutional Neural Networks that you used in your image classification problems, RNNs have been around for decades. Researchers could exploit their full potential only recently due to the growing computational resources that are made available to them now. RNNs are now used in transcribing speech to text, used to perform language translation and generate handwritten text, used in time series forecasting, and have been used as powerful language models. We find RNNs in computer vision applications that perform frame-level video classification, image captioning, visual question answering, and so on. There are numerous applications in which RNNs and their variants are used in the

current days. In the remainder of this book, you will learn to use RNNs for developing such various kinds of applications.

RNNs allow us to operate over sequences of vectors, which can be in the form of input, output, or in the most general case both. The three models are explained using Figure 7-1.

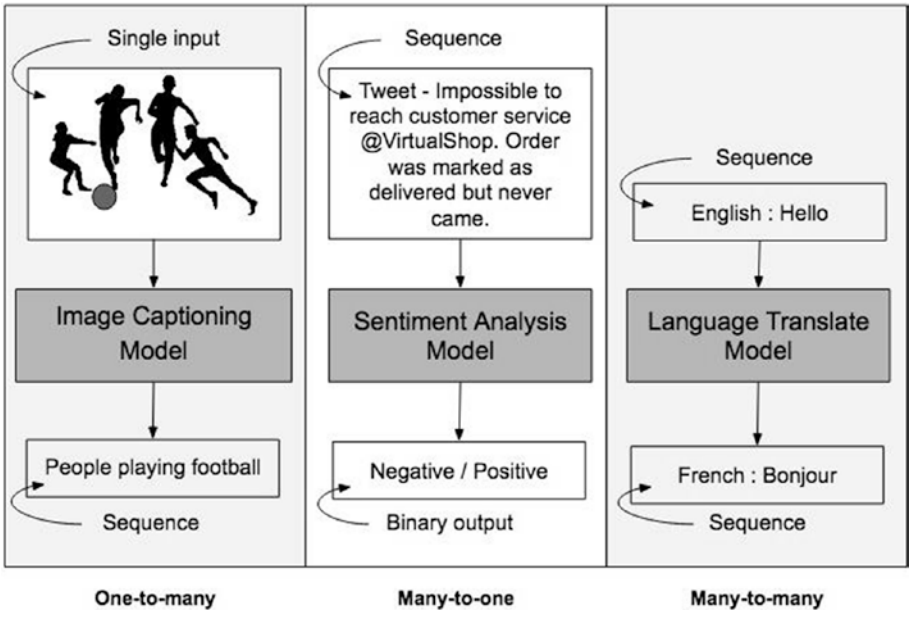


Figure 7-1. *Caption: NN models with various inputs/outputs*

Look at the Image Captioning neural networks (NN) Model shown in Figure 7-1. It takes a single input, which is an image. It generates a caption for the image, which is a sequence of words/characters. In the Sentiment Analysis Model shown in Figure 7-1, the input to the model is a tweet. It consists of a sequence of words/characters. The output produced by the model is either a negative or positive. The output is binary, which is a single output. Finally, consider the case of a model that accepts a sequence as an input and outputs another sequence. The Language Translate Model shown in Figure 7-1 accepts a word/sentence in English and spits out its translation in French, which is again a sequence of words/sentences.

Having said the benefits of RNNs, let me describe the architecture of a simple RNN.

Simple RNN

A RNN may be best described with the help of its architecture drawing shown in Figure 7-2.

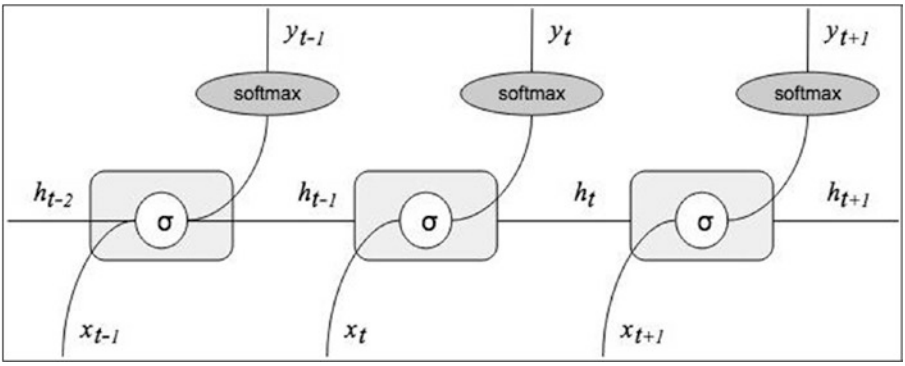


Figure 7-2. *RNN architecture*

At each node, X_i is the input vector and h_i is the output. Each node except the first one in the entire network receives the output of the previous node as an additional input. Likewise, the network consists of nodes where the output of each is not just influenced by its input alone but also by the inference made by the earlier nodes. Thus, RNNs can be considered as the type of artificial neural networks that are designed to recognize patterns in a sequence of data. In short, RNN not only takes the current input but also what they have perceived previously in time, so the network takes two inputs, the present value and the past value. The question is to what depth in time we will ask RNN to take into account the past values. When you train RNNs for larger depths, you will face a well-known vanishing gradient problem. In network training with a large number of layers, the vanishing and exploding gradient problems are well known. RNNs inherently work on sequences which may be of sufficiently

large sizes and are therefore prone to vanishing gradients. Before I discuss the solution to this problem, let me briefly describe what is meant by a vanishing and exploding gradient.

Vanishing and Exploding Gradients

During neural network model training, gradients are back-propagated all the way to the initial layer. At each layer, they go through a continuous matrix multiplication. For deeper networks, due to long chains, the gradients shrink exponentially approaching values much below 1, and eventually they vanish, making the model stop learning further. This is known as the vanishing gradient problem. On the other hand, if they have large values greater than 1, they get larger as they progress through the chain and eventually blow up to crash the model. This is called the exploding gradient problem.

From this discussion, you can understand easily why RNNs are prone to vanishing gradient problems. To overcome this, a new model called long short-term memory (LSTM) was proposed, which I am going to discuss next.

LSTM – A Special Case

Long short-term memory networks, usually called LSTMs, are a special kind of RNN. They are capable of learning long-term dependencies and yet avoid the problem of vanishing gradients that occur in simple RNN models. Hochreiter and Schmidhuber introduced them first in 1997. Later many people refined and popularized it. The LSTMs are now widely used to solve a large variety of problems.

The main motivation for developing LSTMs is that the model should remember information for long periods of time. This should be their default behavior and not something that they have to struggle with.

LSTMs like any other recurrent neural network have the form of a chain of repeating modules, except that the internal architecture of a repeating module varies from that of a standard RNN. A standard RNN module and its chain were already depicted in Figure 7-2. Each module in RNN has a simple structure, such as a single tanh layer as depicted in Figure 7-2. Contrary to this, LSTM modules have a more complicated structure. The LSTM module is shown in Figure 7-3.

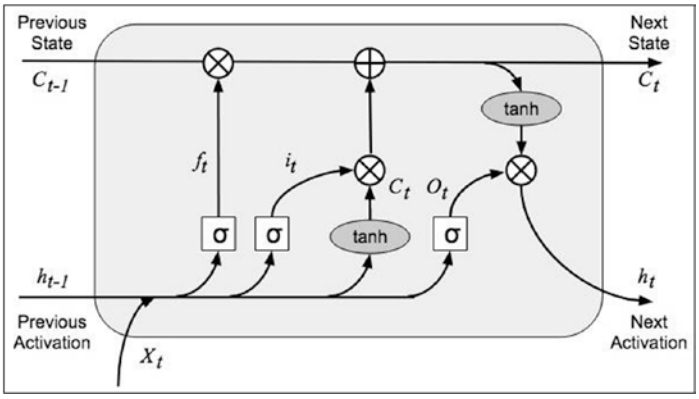


Figure 7-3. *LSTM architecture*

Each repeating module has four network layers. They interact with each other in a very special way. In LSTM, information flows through a mechanism known as cell states. The entire flow can be explained with the help of the four states of the following cells:

- Forget gate
- Input gate
- Update gate
- Output gate

I will briefly describe the purpose of each.

Forget Gate

The forget gate is highlighted in our LSTM block as seen in Figure 7-4.

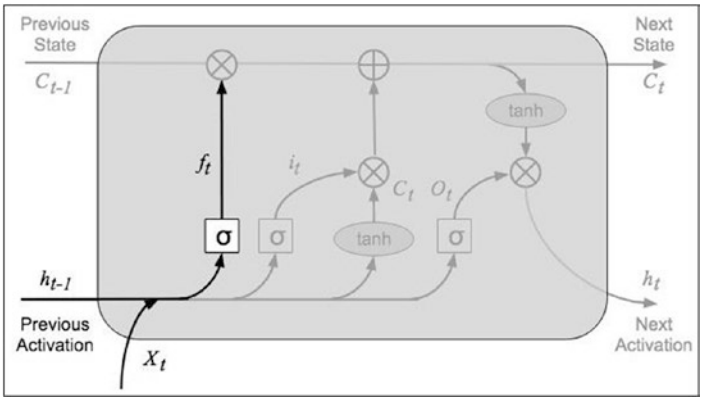


Figure 7-4. *Forget gate*

The forget gate layer has a sigmoid function (σ) to decide what information to throw away or keep. The inputs to this layer are the previous hidden state (h_{t-1}) and the current input (x_t), and the output is binary. The False output tells the network to forget the information, and True output asks it to keep the information for each number in the cell state C_{t-1}. Mathematically, this is expressed with the following equation:

$$f_t = \sigma\left(W_f . \left[h_{t-1,} x_t\right] + b_f\right)$$

Input Gate

The input gate is highlighted in Figure 7-5.

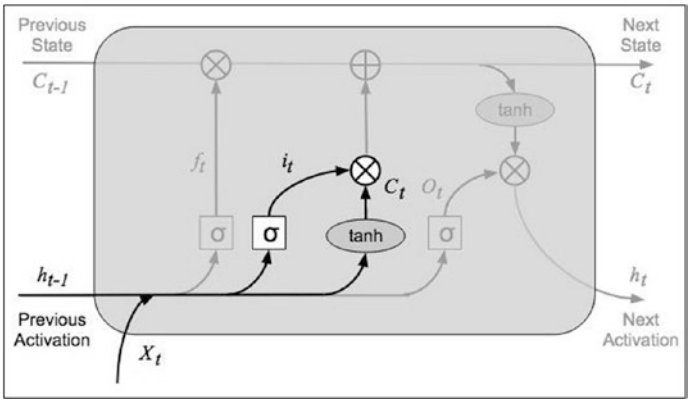


Figure 7-5. *Input gate*

The sigmoid input layer decides which values we will update, and the tanh activation layer creates a vector of new candidate values, $\tilde{C}_t$. Mathematically, this is expressed as

$$i_t = \sigma\left(W_i.\left[h_{t-1,}x_t\right]+b_i\right)$$

$$\tilde{C}_t = tanh\left(W_{c*}\left[h_{t-1,}x_t\right]+b_c\right)$$

Update Gate

The update gate is highlighted in Figure 7-6.

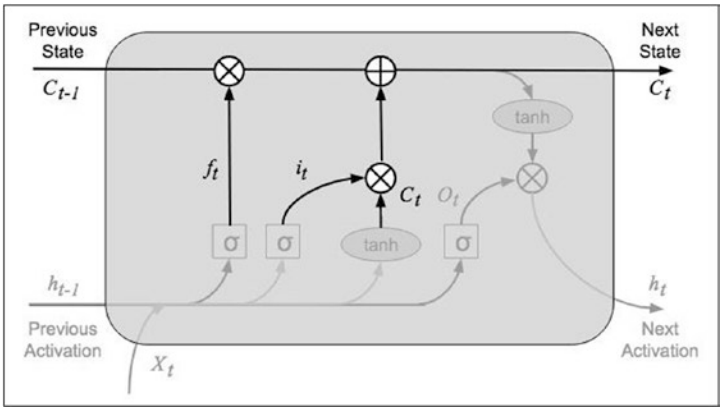


Figure 7-6. *Update gate*

Here, the old cell state C_{t-1} is multiplied by f_t and $i_t * C_t$ is added to it to update the old cell state to a new cell state $\tilde{C}_t$. Mathematically, this is expressed as

$$C_t = f_t * C_t + i_t * \tilde{C}_t$$

Output Gate

The output gate is highlighted in Figure 7-7.

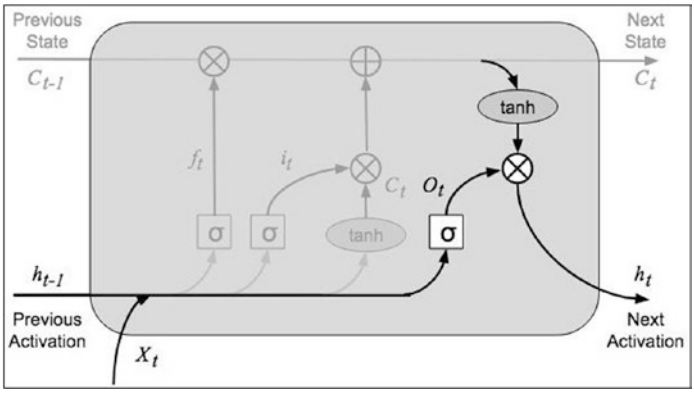


Figure 7-7. *Output gate*

First, the sigmoid layer decides which part of the cell state we are going to keep. Then we feed the cell state through a tanh activation function and multiply it from the output of the sigmoid gate to get the final output. Mathematically, this is expressed as

$$o_t = \sigma\left(W_o\left[h_{t-1,}x_t\right]+b_o\right)$$

$$h_t = o_t * tanh\left(C_t\right)$$

So, with this enhancement of RNN block, LSTM achieves a long short-term memory and overcomes the problem of vanishing gradients observed in simple RNN.

With the ability of a long-term memory, LSTMs are widely used in many domains. In this chapter, I will focus on applications of LSTM for language models. When I say language models, there are again numerous applications, some of which are listed as follows:

- Predict the next character or the word while typing
- Word or sentence completion
- Learn the syntactic and semantic meanings of a large chunk of text, for example, learn the style of Shakespeare and Agatha Christie and generate new stories in their styles
- Neural Machine Translation

After all these theories, I will now move forward to demonstrate to you the application of LSTMs in generating text.

Text Generation

A language sentence is a series of characters that convey a syntactic and semantic meaning to the reader. Every author also has their own style of writing. Our task in text generation is to create a new text that follows the author's style of writing and also has syntactic and semantic meaning to the created text. For example, you may like to write a new novel based on Shakespeare's writing style or create a legal document based on the previously judged legal cases, observing the legal style of writing such documents. You may even like to generate a computer source code in Python after understanding the syntax and the language structure from the other Python programs. The possibilities are endless. Most of these would require lots of processing in terms of resources and time. In the example that follows, I will show you how to generate baby names – a trivial example where text generation can be applied followed by a more realistic example of generating text based on a text from a well-known novel.

Let me first describe some theory behind the text generation.

Model Training

As we want to predict at the character level, we divide the entire text into groups of characters called sequences. Consider the passage shown here taken from the text that is to be used for our training:

> *"Investors so much about their startup hubs. As a lot of mind I don't know the more airborning case of the European of the schedule, and from such sites …"*

We will split this input into sequences of 25 characters each. This is illustrated in Figure 7-8.

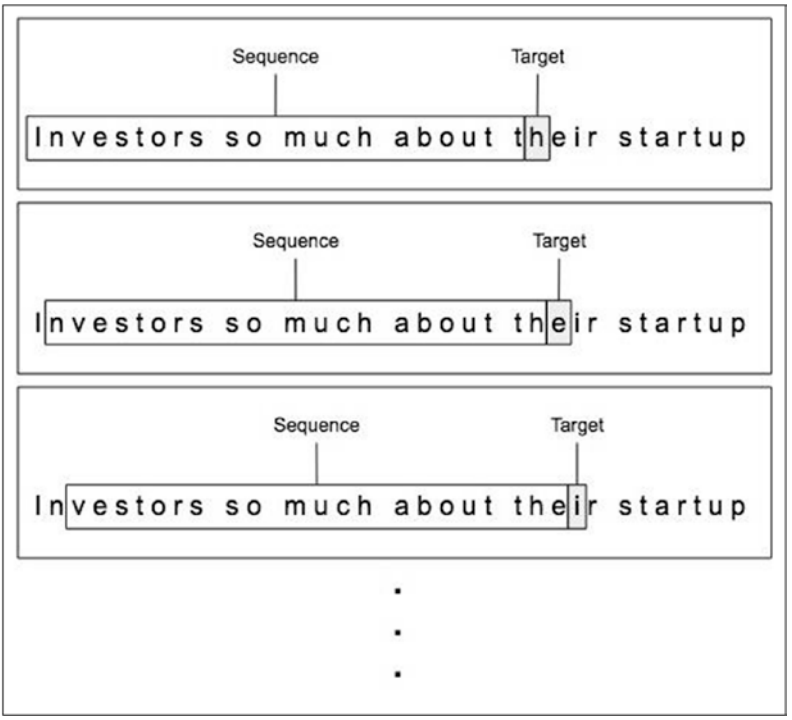


Figure 7-8. *Slicing input text into sequences*

The first 25 characters would be grouped together in the order in which they occur to create the first sequence. This sequence is followed by the character h. Thus, we train the network saying that for this given sequence of characters, the next character should be h. Then, we move the window by one character to the right, and for the new sequence, we tell the model that the next character for this sequence is e. This is now followed by another shift in window where, as shown in the last sequence, the next character would be i. Likewise, we keep training the model for the entire corpus, telling it what would be the next character for any given sequence of 25 characters in the text. Now, you will appreciate why the simple DNN

(deep neural network) cannot be used for such applications, as these applications require a long-term memory to memorize the sequences and the characters following them.

Once the model is trained, hopefully it has learned the writing style, syntax, and semantics of the text. Obviously, the more the corpus that you feed, the better will be the understanding gained by the model. This is equivalent to the fact that after reading several novels written by Agatha Christie, a person starts understanding her style of writing. After the training is complete, how do we use the model to generate a new text in the style of what it has learned?

Inference

We follow the steps similar to what we did during training. We start with a seed consisting of a predefined sequence. Let us say we define a window size of 25 characters like what we did during the training. The window size used for inference need not be the same as the one used during the training. Based on the given sequence of 25 characters, we ask the model to predict the 26th character. We store the prediction somewhere as part of the predicted text. We now shift the window to the right by one character, this time taking the predicted character as the last 25th character in the new sequence. For this newly created sequence, we ask the model to predict the next character. The new prediction will now be added to the next sequence and so on. Likewise, we can ask the model to predict any number of characters following the given input seed. If the model has learned the sequences well enough, it would produce the meaningful text. I will show you the stepwise results during a long training in the large corpus example in this chapter.

Model Definition

The network model that we used for training will mainly consist of multiple LSTM layers as depicted in Figure 7-9.

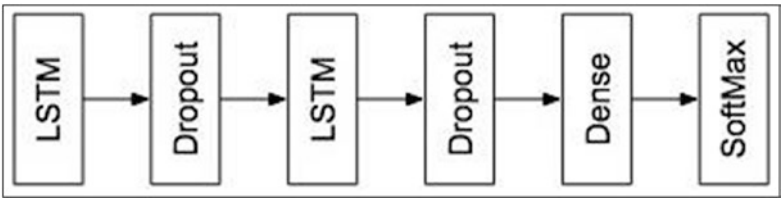


Figure 7-9. *Neural network for text generation*

Each LSTM layer would consist of a large number of nodes. The more the number of nodes, the better would be the model's ability to memorize in the long term. This however would mean the large number of weights to be trained. The model can learn the complex data well by making the network deep, that is, adding more number of LSTM layers. For each LSTM layer, the "return_sequences" parameter would be set to True because it connects its output to the next LSTM layer. For the last LSTM layer, this parameter would be set to False where the output is fed to a Dense layer for character classification based on the probability score of each possible character. We use softmax for this purpose.

With all these theories covered, now let us move on to a practical example to have a better understanding of these theories. The application that I am going to describe next is to generate baby names given a set of known names.

Generating Baby Names

For this project, you will be using the dataset given in the famous Andrej Karpathy blog on text generation – *The Unreasonable Effectiveness of Recurrent Neural Networks* (`http://karpathy.github.io/2015/05/21/rnn-effectiveness/`). The dataset consists of several known baby names

separated by a newline character. You will use these names to train the LSTM model so that it memorizes the sequences defined in the set. After the training, you will ask the model to predict a few new names based on the semantics it has learned.

Creating Project

Create a new Colab project and rename it to TextGenerationBabyNames. Import the required libraries.

```
import sys
import re
import requests
import numpy as np
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.callbacks
                import ModelCheckpoint
from tensorflow.keras.layers
                import Dense, Activation,
                        Dropout, LSTM
```

Downloading Text

To download the dataset, make a HTTP request on the URL specified in the following command:

```
r = requests.get('https://cs.stanford.edu/people/karpathy/
namesGenUnique.txt')
```

On successful completion, you can extract the dataset by copying the text from the response into a local variable:

```
raw_txt = r.text
```

Check the length of the data read by calling the len method on the raw_txt:

```
len(raw_txt)
```

You will get the output as 52127, indicating that there are totally 52,127 characters in the dataset. To see what the data looks like, simply print the raw_txt on the terminal:

```
raw_txt
```

The command and its output are shown in Figure 7-10.

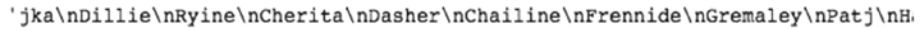


Figure 7-10. *Data file contents*

You see a long list containing the names separated by a newline character. You can check out the first few names by printing them on individual lines using the print statement:

```
print(raw_txt[:100])
```

You will see the following output:

```
jka
Dillie
Ryine
Cherita
Dasher
Chailine
Frennide
```

```
Gremaley
Patj
Handi
Gully
Wennie
Ferentra
Jixandli
```

The program has printed the first 100 characters. The names obviously are of variable lengths.

Processing Text

For inputting the data to our model for training, we must get rid of the \n character. We replace it with a space by using the following command:

```
raw_txt = raw_txt.replace('\n' , ' ')
```

We now extract all the unique characters from the raw_txt by creating a set on it:

```
set(raw_txt)
```

The partial output is shown in Figure 7-11.

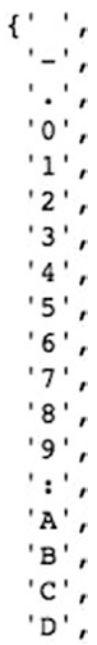


Figure 7-11. *Unique character set*

As you can see, the set contains characters such as space, dash (-), dot (.), colon (:), and numerics (0 through 9). Before training the model, these characters should be removed from the set as they are not useful in the new names that the model would be generating. We remove these characters by using the regular expressions.

```
raw_txt = re.sub('[-.0-9:]' , '' , raw_txt)
```

Also, we do not need the presence of both uppercase and lowercase characters in the generated baby names. So, we convert all the characters to lowercase by calling the lower method:

```
raw_txt1 = raw_txt.lower()
set(raw_txt1)
```

Try printing the set again, and you will notice that it contains only the lowercase alphabets and the space character.

You would be wondering why I did this entire exercise of data processing if, finally, we wanted only a set of lowercase alphabets and the space character. For generating the baby names, the destination character set just contains the alphabets and the space to separate out the names. However, in more advanced text generation applications such as generating documents containing mathematical equations, legal documents, science abstracts, and so on, your destination character set would be much larger, containing all sorts of characters. However, taking a large destination character set would also result in an exponential increase in the training period. So, generally, we will strip a few unwanted characters from the original text. Such text processing for removing the undesired characters would result in faster training.

You can now check the size of this new set:

```
len1 = len(set(raw_txt1))
print (len1)
```

It would print 27 on your terminal.

As the model works on numbers and not on the alphabets, you need to map the alphabets to distinct numbers. Also, when the model outputs a prediction, it will send you a set of numbers which must be converted back to alphabets for you to make sense. Thus, we create two arrays for these mappings. This is done using the following code snippet:

```
chars = sorted(list(set(raw_txt1)))

arr = np.arange(0, len1)

char_to_ix = {}
ix_to_char = {}
for i in range(len1):
   char_to_ix[chars[i]] = arr[i]
   ix_to_char[arr[i]] = chars[i]
```

The char_to_ix array will provide a mapping from characters in the set to unique integers, and ix_to_char will provide the reverse mappings from integer to characters. Try printing the ix_to_char array, and you will see the partial output shown in Figure 7-12.

```
ix_to_char

{0: ' ',
 1: 'a',
 2: 'b',
 3: 'c',
 4: 'd',
 5: 'e',
 6: 'f',
 7: 'g',
 8: 'h',
 9: 'i',
 10: 'j',
 11: 'k',
 12: 'l',
 13: 'm',
```

Figure 7-12. *Unique character set after preprocessing*

Now, you will create the input and output sequences using the following code snippet:

```
maxlen = 5
x_data = []
y_data = []
for i in range(0, len(raw_txt1) - maxlen, 1):
   in_seq  = raw_txt1[i: i + maxlen]
   out_seq = raw_txt1[i + maxlen]

   x_data.append([char_to_ix[char]
                  for char in in_seq])
   y_data.append([char_to_ix[out_seq]])
```

```
nb_chars = len(x_data)
print('Text corpus: {}'.format(nb_chars))
print('Sequences # ', int(len(x_data) / maxlen))
```

Note that we define a sequence length of 5. So the first five characters would be input, and the sixth character would be the target. In the next loop, characters from 2 to 6 would be the input sequence, and character 7 would be the target and so on. Thus, in a for loop, we create the x_data for training our model, y_data being the target values used while training. The output of the preceding code snippet would be as follows:

```
Text corpus: 52038
Sequences #  10407
```

The dataset consists of 52,038 characters, which is divided into 10,407 sequences, each of length 5.

Next, we transform the data into numpy arrays for inputting to our model and also normalize the training data to a scale of 0 to 1.

```
x = np.reshape(x_data , (nb_chars , maxlen , 1))
x = x/float(len(chars))
```

We convert the target sequence into categorical columns.

```
y = tf.keras.utils.to_categorical(y_data)
y[:1]
```

In the preceding statement, when you print one of the items in the y_data after conversion, you would see an output shown in Figure 7-13.

```
array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]], dtype=float32)
```

Figure 7-13. *A sample target data after categorization*

In this array, one of the values is 1, while the rest are 0. The value 1 corresponds to the character in the char_to_ix array at that particular index in the array.

You may now print the shape of x_data:

```
x.shape
```

The output is

```
(52038, 5, 1)
```

This indicates that the input has 52,038 sequences, each of length 5. You may also check the size of y by calling y.shape.

```
y.shape
```

The output is

```
(52038, 27)
```

There are 27 categories in the output.

Defining Model

We define our model as follows:

```
model = tf.keras.Sequential([
               tf.keras.layers.LSTM(256,
                        input_shape = (maxlen, 1),
                          return_sequences = True),
               tf.keras.layers.LSTM(256,
                          return_sequences = True),
               tf.keras.layers.Dropout(0.2),
               tf.keras.layers.LSTM(64),
               tf.keras.layers.Dropout(0.2),
               tf.keras.layers.Dense(len(y[1]),
                          activation='softmax')
])
```

The model summary is shown as follows:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 5, 256)            264192
_________________________________________________________________
lstm_1 (LSTM)                (None, 5, 256)            525312
_________________________________________________________________
dropout (Dropout)            (None, 5, 256)            0
_________________________________________________________________
lstm_2 (LSTM)                (None, 64)                82176
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense (Dense)                (None, 27)                1755
=================================================================
Total params: 873,435
Trainable params: 873,435
Non-trainable params: 0
```

As you can see, the model has three LSTM layers each consisting of 500 nodes. Each LSTM layer is followed by a dropout layer with 20% dropout. The last layer is a Dense layer with 27 nodes classified by softmax activation. Note that our data has 27 categorical outputs. The visual representation of the model is shown in Figure 7-14.

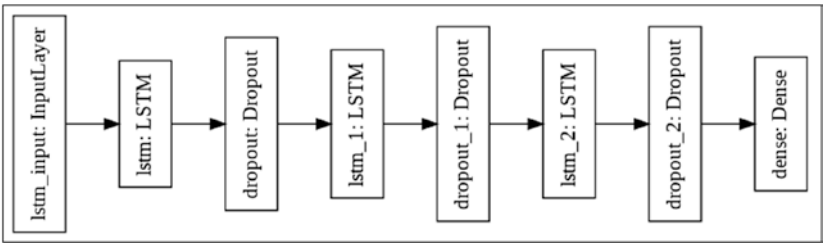


Figure 7-14. *Model architectural layers*

Compiling

We compile the model using the categorical cross-entropy and Adam optimizer.

```
model.compile(loss = 'categorical_crossentropy',
                              optimizer = 'adam')
```

Note that for these kinds of language model problems, there is no test dataset. We model the entire dataset to predict the probability of each categorical character in a sequence. The model accuracy in predicting the next character perfectly is not important to us. Rather, we are interested in minimizing the chosen loss function. Thus, we are trying to achieve a balance between generalization and overfitting that is short of memorization.

Creating Checkpoints

Training an LSTM network typically takes a long time. Due to the nature of the network itself, the loss after each epoch may increase or decrease. The lowest loss will finally give us the best results in our predictions. Thus, we need to capture the model weights for the epoch which has given us the lowest loss. This is done by using the ModelCheckPoint method and setting the callback after each epoch.

```
filepath = "model_weights_babynames.hdf5"
checkpoint = ModelCheckpoint(filepath,
```

```
                monitor = 'loss', verbose = 1,
                save_best_only = True, mode = 'min')
model_callbacks = [checkpoint]
```

You have earlier used callbacks like early stopping in a project in Chapter 4, so this is another type of callback which will be called after every epoch.

Training

We now train the model by calling the fit method.

```
model.fit(x,y, epochs = 300, batch_size = 32 ,
               callbacks = model_callbacks)
```

I have set the number of epochs to 10 and a batch size of 32 sequences. I will later on show you the effect of increasing both the values of these variables. When I ran this code on a GPU, the training time was roughly 6 seconds per epoch, reasonable for us. I shall like to say here, the reason I am so concerned with the training time is that training LSTMs on large corpus and having large categorical outputs require several hours to complete even when you use distributed training across a set of GPUs.

After the training is over, let us try some predictions.

Prediction

We will first need to create an input sequence. For the time being, we define an input sequence from the original database itself. It is defined using the following code snippet:

```
pattern = []
seed = 'handi'
for i in seed:
   value = char_to_ix[i]
   pattern.append(value)
```

The sequence that we used is "handi" – having length equal to 5. Note that our model definition expects an input with sequence size of 5. Each character in the seed sequence is converted to its integer value by using the char_to_ix array that we created earlier.

We will now set up a for loop to predict the next 100 characters following the given sequence "handi." For the reference, we first print the seed and set the number of characters in our vocabulary to the n_vocab variable.

```
print(seed)
n_vocab = len(chars)
```

We set up the loop for doing 100 predictions.

```
for i in range(100):
```

We first reshape this input pattern and normalize its contents using the following two statements:

```
    X = np.reshape(pattern , (1, len(pattern) , 1))
    X = X/float(n_vocab)
```

We feed this to our model and ask it to predict the next character following the given pattern:

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

We extract the index of the character having the maximum probability of prediction and convert it to a character by using our previously created ix_to_char array.

```
    index = np.argmax(int_prediction)
    prediction = ix_to_char[index]
```

We print the predicted character on the terminal:

```
    sys.stdout.write(prediction)
```

We append this character to our pattern and recreate a new pattern by extracting the last five characters, which is our input sequence length. With this new pattern, we ask the model to do the next prediction. Likewise, we will ask the model to do 100 predictions starting with the input sequence defined by us as the seed.

```
pattern.append(index)
pattern = pattern[1:len(pattern)]
```

The entire for loop for generating 100 characters is given in Listing 7-1 for your quick reference.

Listing 7-1. Loop for predicting 100 characters

```
print(seed)
n_vocab = len(chars)
for i in range(100):
   X = np.reshape(pattern , (1, len(pattern) , 1))
   X = X/float(n_vocab)
   int_prediction = model.predict(X , verbose = 0)
   index = np.argmax(int_prediction)
   prediction = ix_to_char[index]
   sys.stdout.write(prediction)
   pattern.append(index)
   pattern = pattern[1:len(pattern)]
```

Full Source – TextGenerationBabyNames

The full source for the baby name generation project is given in Listing 7-2.

Listing 7-2. TextGenerationBabyNames full source

```
import sys
import re
import requests
```

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.callbacks
                import ModelCheckpoint
from tensorflow.keras.layers
                import Dense, Activation,
                        Dropout, LSTM

r = requests.get('https://cs.stanford.edu/people/karpathy/
namesGenUnique.txt')

raw_txt = r.text
len(raw_txt)

raw_txt

print(raw_txt[:100])

set(raw_txt)

len(set(raw_txt))

raw_txt = raw_txt.replace('\n' , ' ')
set(raw_txt)

len(set(raw_txt))

raw_txt = re.sub('[-.0-9:]' , '' , raw_txt)
len(set(raw_txt))

raw_txt1 = raw_txt.lower()
set(raw_txt1)

len1 = len(set(raw_txt1))
print (len1)

len1
```

```
chars = sorted(list(set(raw_txt1)))

arr = np.arange(0, len1)

char_to_ix = {}
ix_to_char = {}
for i in range(len1):
   char_to_ix[chars[i]] = arr[i]
   ix_to_char[arr[i]] = chars[i]

char_to_ix

ix_to_char

#print("Total length of file  : {}".format(len(raw_txt1)))

maxlen = 5
x_data = []
y_data = []
for i in range(0, len(raw_txt1) - maxlen, 1):
   in_seq  = raw_txt1[i: i + maxlen]
   out_seq = raw_txt1[i + maxlen]

   x_data.append([char_to_ix[char]
                     for char in in_seq])
   y_data.append([char_to_ix[out_seq]])
nb_chars = len(x_data)
print('Text corpus: {}'.format(nb_chars))
print('Sequences # ', int(len(x_data) / maxlen))

#y_data[:5]

#x_data[1][:]

x = np.reshape(x_data , (nb_chars , maxlen , 1))
x = x/float(len(chars))
```

```
y = tf.keras.utils.to_categorical(y_data)
y[:1]

x.shape

y.shape

model = tf.keras.Sequential([
               tf.keras.layers.LSTM(256,
                         input_shape = (maxlen, 1),
                           return_sequences = True),
               tf.keras.layers.LSTM(256,
                           return_sequences = True),
               tf.keras.layers.Dropout(0.2),
               tf.keras.layers.LSTM(64),
               tf.keras.layers.Dropout(0.2),
               tf.keras.layers.Dense(len(y[1]),
                           activation='softmax')
])

model.summary()

model.compile(loss = 'categorical_crossentropy',
                             optimizer = 'adam')

filepath = "model_weights_babynames.hdf5"
checkpoint = ModelCheckpoint(filepath,
                monitor = 'loss', verbose = 1,
                save_best_only = True, mode = 'min')
model_callbacks = [checkpoint]

model.fit(x,y, epochs = 300, batch_size = 32 ,
                      callbacks = model_callbacks)

pattern = []
```

```
seed = 'handi'
for i in seed:
    value = char_to_ix[i]
    pattern.append(value)

print(seed)
n_vocab = len(chars)
for i in range(100):
   X = np.reshape(pattern , (1, len(pattern) , 1))
   X = X/float(n_vocab)
   int_prediction = model.predict(X , verbose = 0)
   index = np.argmax(int_prediction)
   prediction = ix_to_char[index]
   sys.stdout.write(prediction)
   pattern.append(index)
   pattern = pattern[1:len(pattern)]

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

cd 'My Drive'

model.save('baby_names_model.h5')

from tensorflow.keras.models import load_model
saved_model = load_model('baby_names_model.h5')

pattern = []
seed = 'bgajm'
for i in seed:
    value = char_to_ix[i]
    pattern.append(value)

print(seed)
n_vocab = len(chars)
```

```
for i in range(100):
   X = np.reshape(pattern , (1, len(pattern) , 1))
   X = X/float(n_vocab)
   int_prediction = saved_model.predict
                      (X , verbose = 0)
   index = np.argmax(int_prediction)
   prediction = ix_to_char[index]
   sys.stdout.write(prediction)
   pattern.append(index)
   pattern = pattern[1:len(pattern)]
```

When I ran the code, I got the output shown in Figure 7-15.

```
handi
 saree carie carie carie carie carie carie carie carie carie carie carie
```

Figure 7-15. *Baby names for seed "handi"*

The network has probably done a good job of generating some meaningful names. You will now try increasing the number of epochs to check if there are any improvements in the predictions.

When I retrained the model for 50 epochs and a batch size of 128, I got the output shown in Figure 7-16 for the same seed "handi."

```
handi
a craskie llonpre trestor allussea wande cherita merylee gilphon salda gerrek
```

Figure 7-16. *Baby names after 50 epochs*

This may be better looking depending on the individual's choice of names. At least, the names are not repeated.

The point to understand here is that by running the training for a large number of epochs, increasing the number of nodes in LSTM layers, increasing the number of LSTM layers, adjusting sequence length, and so on may result in improving the quality of predictions. In fact, larger

networks with exhaustive training will theoretically result in giving the same output as in the original text for a known seed from the original text. This is as good as saying that a person having a picturesque memory will be able to reproduce the original text as it in the same order as the original. What you gain from this discussion is that LSTM neural networks can be effectively used for generating quality text on its own that mimics the original passage of text.

As far as our application of generating baby names is concerned, it does not make sense in using the network to generate the names which already exist in our database. To generate names which do not exist in our database and sounding likewise to existing names, you need to seed the network with an input sequence that does not exist in the original text. For this, generally people generate a random seed. I tried a random seed "bgajm" and got the output shown in Figure 7-17.

```
bgajm
 sherine margoria solly adisa naril rurodore je viyette alady goyne jasel minco
```

Figure 7-17. *Names generated with a random seed*

Before I move on to a more complex text generation problem, let me point out one more important dependency of the network training time on the batch size. If the batch size is small, it will take more time to cover all those characters in the entire text corpus, resulting in an increased training time. At the same time, increasing the batch size would demand additional resources in terms of system memory. Thus, it will be a compromise between the batch size and the resources that you can allocate to training while getting the optimal training time for the network. Considering the large training time involved in the case of LSTMs, it is advised that you save the model after training and reuse it later for different seeds. I will now show you how to save and reuse the model for later predictions.

Saving/Reusing Model

You will save the trained model to your Google Drive. You will have to mount the drive for this.

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

During mounting, it will ask you to enter the authorization code. After the drive is mounted, change to the folder where you would like to save the model.

```
cd 'drive/My Drive/TextGenerationDemo'
```

Now, call the model's save method to save it to a desired filename.

```
model.save('baby_names_model.h5')
```

You can reload the saved model at any later point of time by calling the load_model.

```
from tensorflow.keras.models import load_model
saved_model = load_model('baby_names_model.h5')
```

The loaded model is available in the variable saved_model which can be used for your next predictions or further trainings.

With this introduction to text generation by using a trivial example, let us move on to a more realistic example with a huge text corpus.

Advanced Text Generation

In this application, we will use a text from the famous novel by Leo Tolstoy, *War and Peace* (https://cs.stanford.edu/people/karpathy/char-rnn/warpeace_input.txt). The novel certainly uses many special characters such as question/exclamation marks, quotes, and periods. In our earlier

example, we stripped such characters because we just wanted to generate names. In this project, I will not remove all such special characters because we want to create another novel or at least a passage that contains all such characters.

The time required to train a complex LSTM model on such a huge corpus is very high; you need to make provisions for storing the model's training state periodically. I will show you how to save the model's state at every epoch. So in case of disconnection during training, you can continue the training from a previously known checkpoint. Also, I am going to add another facility of testing the model's performance by asking it to predict on a fixed seed at the end of every epoch. We will be storing the predictions to a file in Google Drive. Thus, you can keep on training the model in the background and keep checking its performance periodically by examining the contents of the prediction file on your disk.

Creating Project

Create a new Colab project and rename it to LargeCorpusTextGeneration. Import the required libraries.

```
import sys
import requests
import numpy as np
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.callbacks
                import ModelCheckpoint
from tensorflow.keras.layers
                import Dense, Activation,
                       Dropout, LSTM
```

You will be saving the checkpoint data and the model's predictions at the end of every epoch during training. For this, you need to mount Google Drive and point to the appropriate folder for saving your data.

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

During mounting, you will be asked for the authorization. When the drive is mounted, you set up the folder for storing your files.

```
cd '/content/drive/My Drive/TextGenerationDemo'
```

Loading Text

We load the novel text into the project by making the following HTTP request:

```
r = requests.get("https://cs.stanford.edu/people/karpathy/char-
rnn/warpeace_input.txt")
```

We read the novel text into a local variable from the response object.

```
raw_txt = r.text
```

We get a list of unique characters in the text and print the corpus size and the number of output categories:

```
chars = sorted(list(set(raw_txt)))
print("Corpus: {}".format(len(raw_txt)))
print("Categories: {}".format(len(chars)))
```

You will see the following output:

```
Corpus: 3258246
Categories: 87
```

The novel has more than 3 million characters and contains 87 unique characters in the text. Both these figures are much larger than the ones in our baby names model.

Processing Data

Like in the earlier example, we need to map all the unique characters to integers for processing by our model. We also need to provide reverse mapping for interpreting the model's output. We do this by creating the following two arrays:

```
ix_to_char = {ix:char for ix,
                    char in enumerate(chars)}
char_to_ix = {char:ix for ix,
                    char in enumerate(chars)}
```

We then split the entire text into sequences using the following code segment:

```
maxlen = 10
x_data = []
y_data = []
for i in range(0, len(raw_txt) - maxlen, 1):
   in_seq  = raw_txt[i: i + maxlen]
   out_seq = raw_txt[i + maxlen]
   x_data.append([char_to_ix[char]
                    for char in in_seq])
   y_data.append([char_to_ix[out_seq]])
nb_chars = len(x_data)
print('Number of sequences:',
                    int(len(x_data)/maxlen))
```

This code is similar to whatever you saw in the previous example. With a sequence size of 10, the number of created sequences is 325,823. This is the output of the preceding code:

```
Number of sequences: 325823
```

We now scale and reshape the input data to make it suitable for feeding to our network.

```
# scale and transform data
x = np.reshape(x_data , (nb_chars , maxlen , 1))
n_vocab = len(chars)
x = x/float(n_vocab)
```

We convert the output to its categories by calling the to_categorical method.

```
y = tf.keras.utils.to_categorical(y_data)
```

You may check the size of the input and output using the print statements:

```
print("The shape of x_training data : " ,x.shape)
print("The shape of y_training data : " ,y.shape)
```

The output is

```
The shape of x_training data :  (3258236, 10, 1)
The shape of y_training data :  (3258236, 86)
```

As you can see, we have large numbers of sequences. The number of inputs to the network will be 10 and 86 categorical outputs.

Defining Model

We define the model using the following code:

```
Model = tf.keras.Sequential([
                  tf.keras.layers.LSTM(800 ,
                  input_shape = (len(x[1]) , 1) ,
                        return_sequences = True),
                  tf.keras.layers.Dropout(0.2),
                        tf.keras.layers.LSTM(800,
                        return_sequences = True),
                  tf.keras.layers.Dropout(0.2),
                  tf.keras.layers.LSTM(800),
                  tf.keras.layers.Dropout(0.2),
                  tf.keras.layers.Dense(len(y[1]),
                        activation = 'softmax')
])
```

The model definition is the same as the one used in the previous example, except that I have increased the number of nodes in each layer considering the corpus size of the input text.

The model is compiled using the typical cross-entropy and Adam optimizer.

```
Model.compile(loss = '
                categorical_crossentropy' ,
                optimizer = 'adam')
```

Now, I will describe the most important portion of this project, and that is saving the model's state and its predictions at the end of each epoch.

Creating Checkpoints

For creating checkpoints, you need to create a custom callback for the training method. We first assign the name for our checkpoint file.

```
filepath = "model_weights_saved.hdf5"
```

We use the ModelCheckpoint method to create a callback method:

```
checkpoint = ModelCheckpoint(filepath,
                monitor = 'loss', verbose = 1,
                save_best_only = True, mode = 'min')
```

In the callback function, we will monitor the loss and save the model's training weights for the minimum loss value.

We create a variable to list the number of callbacks, which in our case is only one.

```
model_callbacks = [checkpoint]
```

CustomCallback Class

We will now create another callback for writing the model's predictions. We will be storing the predictions at the end of each epoch in a text file. We create a global variable for tracking the epoch number.

```
epoch_number = 0
```

We declare a filename for storing predictions:

```
filename = 'predictions.txt'
```

We overwrite the contents of the file if it preexists.

```
file = open(filename , 'w')
file.truncate()
file.close()
```

We declare the custom class as follows:

```
class CustomCallback(tf.keras.callbacks.Callback):
```

We define a method called on_epoch_end with the following statement:

```
def on_epoch_end(self , epoch , logs = None):
```

This method will be called at the end of each epoch. In the method body, we first increment our global epoch count:

```
global epoch_number
epoch_number = epoch_number + 1
```

We open the prediction file in the append mode to add our predictions:

```
filename = 'predictions.txt'
file = open(filename , 'a')
```

We declare our seed:

```
seed = "looking fo"
```

We create a pattern from this seed in a simple for loop:

```
pattern = []
for i in seed:
    value = char_to_ix[i]
    pattern.append(value)
```

We first write the epoch number to the file:

```
file.seek(0)
file.write("\n\n Epoch number :
            {}\n\n".format(epoch_number))
```

We set up a loop for doing 100 predictions:

```
for i in range(100):
```

We reshape and scale the input data:

```
X = np.reshape(pattern ,
            (1, len(pattern) , 1))
X = X/float(n_vocab)
```

We ask the model to predict the next character for a given seed:

```
int_prediction = Model.predict(X ,
                        verbose = 0)
```

We pick up the character with the max probability and copy it to the prediction variable:

```
index = np.argmax(int_prediction)
prediction = ix_to_char[index]
```

We write the predicted character to the file:

```
file.write(prediction)
```

We add the predicted character to the pattern and extract the last ten characters (the size of our seed) to create a new pattern:

```
pattern.append(index)
pattern = pattern[1:len(pattern)]
```

We now iterate for the next prediction. After we predict for 100 times, we close the file:

```
file.close()
```

Model Training

After creating the two callback functions, we now train the model using the following statement:

```
Model.fit(x, y , batch_size = 200,
                 epochs = 10 ,
                 callbacks = [CustomCallback() ,
                 model_callbacks])
```

We define a sufficiently large batch size of 200 sequences. The two callback functions are specified in the fit call. The custom callback function stores the predictions, and the model callback function stores the model's state.

When I ran the training, it took me approximately 550 seconds for each epoch on a GPU. Fortunately, the results are saved at the end of every epoch. So, there was no worry for timeout or disconnection as I was able to see the model performance till the last break and also continue further training from the last breakpoint, with the code discussed after the "Results" section.

Results

The predictions after the first, fifth, and tenth epoch were as follows:

```
Epoch number : 1
r the soldiers were all the same time the soldiers were all the
same time the soldiers were all the
```

```
Predictions after 5 epochs:
```

```
Epoch number : 5
```

```
r the first time to the countess was a serious and the servants
and the servants and the servants an

Predictions after 10 epochs:

Epoch number : 10

r the first time they had been sent to the countess was a
still more than the countess was a still m
```

You can notice that the model's performance keeps improving on every epoch.

Training Continuation

To continue training from the last known checkpoint, use the following code:

```
try:
   Model.load_weights(filepath)
except Exception as error:
   print("Error loading in model : {}".format(error))
```

We simply load the weights from the stored checkpoint file and then call the model's fit method to continue the training.

```
Model.fit(x, y , batch_size = 200, epochs = 10 ,
                 callbacks = [CustomCallback() ,
                 model_callbacks])
```

Note that the epoch numbers would continue from the last value of the global epoch.

Here are some results when I continued for a total of 50 epochs.

```
 Epoch number : 20

r the first time to the countess was a small conversation with
the state of a strange and the counte

 Epoch number : 30

r the first time the soldiers who were all the same time he had
seen and was about to see the counte

 Epoch number : 40

r the first time the staff officer who had been at the same
time he had seen him to the countess was

 Epoch number : 50

r the first time the streets of the countess was a man of his
soul and the same time he had seen and
```

As you can see, the model's output quality keeps improving on more training.

Some Observations

Just for some experimentation and to reduce the training time, I reduced the number of nodes in each LSTM layer from 800 to 100. This definitely brought down the training time from approximately 10 minutes to 2 minutes. After running the training for 100 epochs, here are the results at a few intervals:

```
Epoch number : 1

 and the same and the same and the same and the same and the
same and the same and the same and the

Epoch number : 25

 and the service and the service and the service and the
service and the service and the service and

Epoch number : 50

 and the same to the same to the same to the same to the same
to the same to the same to the same to

Epoch number : 75

 and the strength to the same and the strength to the same and
the strength to the same and the stre

Epoch number : 100

 and the same and the same and the same and the same and the
same and the same and the same and the
```

As you can see, the model is not learning anymore in spite of a larger number of epochs. We can conclude that understanding the large text requires more memory by way of an increased number of nodes in each LSTM layer.

I ran another experiment by increasing the number of nodes in each LSTM layer to 500. As a result, the training time increased to approximately 11 minutes/epoch. Here are the results:

```
Epoch number : 5

 the same time to the same time to the same time to the same
time to the same time to the same time

Epoch number : 10

 the countess was a strange and the same things and the same
things and the same things and the same

Epoch number : 15

 the countess and the same time the soldiers and the same time
the soldiers and the same time the so

Epoch number : 20

, and the same time the countess was still the staff of the
countess was still the staff of the coun
```

We see some improvement by increasing the number of nodes. In the 20th epoch, it has even generated a comma. Continuing further training may further improve the performance.

Full Source

The full source code for the LargeCorpusTextGeneration is shown in Listing 7-3.

Listing 7-3. LargeCorpusTextGeneration full souce

```
import sys
import requests
import numpy as np
import tensorflow as tf
from tensorflow.keras import Sequential
```

```
from tensorflow.keras.callbacks
                import ModelCheckpoint
from tensorflow.keras.layers
                import Dense, Activation,
                       Dropout, LSTM

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

cd '/content/drive/My Drive/TextGenerationDemo'


r = requests.get("https://cs.stanford.edu/people/karpathy/char-
rnn/warpeace_input.txt")


raw_txt = r.text

chars = sorted(list(set(raw_txt)))
print("Corpus: {}".format(len(raw_txt)))
print("Categories: {}".format(len(chars)))

ix_to_char = {ix:char for ix,
                   char in enumerate(chars)}
char_to_ix = {char:ix for ix,
                   char in enumerate(chars)}

maxlen = 10
x_data = []
y_data = []
for i in range(0, len(raw_txt) - maxlen, 1):
   in_seq  = raw_txt[i: i + maxlen]
   out_seq = raw_txt[i + maxlen]
   x_data.append([char_to_ix[char]
                    for char in in_seq])
   y_data.append([char_to_ix[out_seq]])
```

```
nb_chars = len(x_data)
print('Number of sequences:',
                  int(len(x_data)/maxlen))

# scale and transform data
x = np.reshape(x_data , (nb_chars , maxlen , 1))
n_vocab = len(chars)
x = x/float(n_vocab)

x.shape


y = tf.keras.utils.to_categorical(y_data)

print("The shape of x_training data : " ,x.shape)
print("The shape of y_training data : " ,y.shape)

Model = tf.keras.Sequential([
                  tf.keras.layers.LSTM(800 ,
                  input_shape = (len(x[1]) , 1) ,
                        return_sequences = True),
                  tf.keras.layers.Dropout(0.2),
                        tf.keras.layers.LSTM(800,
                        return_sequences = True),
                  tf.keras.layers.Dropout(0.2),
                  tf.keras.layers.LSTM(800),
                  tf.keras.layers.Dropout(0.2),
                  tf.keras.layers.Dense(len(y[1]),
                        activation = 'softmax')
])

Model.compile(loss = '
                categorical_crossentropy' ,
                optimizer = 'adam')
```

```
filepath = "model_weights_saved.hdf5"
checkpoint = ModelCheckpoint(filepath,
                monitor = 'loss', verbose = 1,
                save_best_only = True, mode = 'min')
model_callbacks = [checkpoint]

epoch_number = 0
filename = 'predictions.txt'
file = open(filename , 'w')
file.truncate()
file.close()
class CustomCallback(tf.keras.callbacks.Callback):

    def on_epoch_end(self , epoch , logs = None):
        global epoch_number
        epoch_number = epoch_number + 1

        filename = 'predictions.txt'
        file = open(filename , 'a')
        seed = "looking fo"

        pattern = []
        for i in seed:
            value = char_to_ix[i]
            pattern.append(value)
        file.seek(0)
        file.write("\n\n Epoch number :
                    {}\n\n".format(epoch_number))
        for i in range(100):
            X = np.reshape(pattern ,
                           (1, len(pattern) , 1))
            X = X/float(n_vocab)
            int_prediction = Model.predict(X ,
```

```
                                    verbose = 0)
            index = np.argmax(int_prediction)
            prediction = ix_to_char[index]
            #sys.stdout.write(prediction)
            file.write(prediction)
            pattern.append(index)
            pattern = pattern[1:len(pattern)]
        file.close()

Model.fit(x, y , batch_size = 2000, epochs = 10 ,
                callbacks = [CustomCallback() ,
                model_callbacks])

try:
    Model.load_weights(filepath)
except Exception as error:
    print("Error loading in model :
                   {}".format(error))

Model.fit(x, y , batch_size = 200, epochs = 25 ,
                callbacks = [CustomCallback() ,
                model_callbacks])
```

Further Work

The work done by Andrej Karpathy on text generation is worth mentioning, which is published in his famous blog *The Unreasonable Effectiveness of Recurrent Neural Networks* (http://karpathy.github.io/2015/05/21/rnn-effectiveness/). For his experiments, he has used Shakespeare literature, Wikipedia articles, LaTeX (Algebraic Geometry language text), and even Linux source code. The results produced after an exhaustive training are fabulous. He is able to generate fantastic mathematical equations which are most of the time syntactically correct. His model is

able to generate computer source code which is almost compilable. This can be considered as a great proof that text generation using LSTMs is practically possible to generate quality text.

From the two examples that I have discussed, you can easily understand that it takes lots of resources and time to train an LSTM for generating quality text. It would also require some experimentation on your side to achieve the best results. Here are a few tips that you will like to consider while fine-tuning your text generation applications:

- To save the training time, reduce the vocabulary size by removing unwanted characters.
- Add more LSTM and Dropout layers with more LSTM units in each layer.
- Try tweaking hyperparameters such as batch size, optimizer, and sequence length and see which works best.
- Try a large number of epochs.
- Use a large text corpus.

Summary

In this chapter, you studied a new neural network architecture and that is RNN. LSTM is a special case of RNN. The traditional DNNs do not have the ability to remember, while LSTM has both long- and short-term memories. Thus, for situations like text generation where the memory is important, LSTMs do a fantastic job. You learned how to create an LSTM-based network for generating baby names and even quality text passages after learning from a novel by a famous author.

In the next chapter, you will learn another language model used for language translation, say from English to French or Spanish to Japanese.

CHAPTER 8

Language Translation

Introduction

The first time I was transiting through Frankfurt International Airport, I had a tough time following the signs at the airport as I do not understand the German language. This was many years ago. Today, you can just point your mobile to these signs, and the app within your phone will provide you with the translation in English or maybe a language of your choice. How are these translations done? There is more than one technology involved behind these translations. The core is a machine learning model that provides a word-to-word translation using a huge vocabulary of the predefined words. Obviously, this kind of word-to-word or in machine learning terms sequence-to-sequence translation works with a great accuracy in the case of the airport and road signs, but it may not produce acceptable translations in natural language sentences. Just to give you an example, the translation of a question like "How are you today?" cannot be simply done by translating each word of the sentence independently of other words. Sophisticated models are made for performing such translations. Google initially used statistical language translation; in 2016, they started using NMT (neural machine translation). You will learn how to develop such a model in this chapter.

P. Sarang, *Artificial Neural Networks with TensorFlow 2*,
https://doi.org/10.1007/978-1-4842-6150-7_8

Sequence-to-Sequence Modeling

In the previous chapter, you learned RNN and LSTMs. We will use these networking models to create a neural machine translation model using the Encoder/Decoder and Attention. As you read along, I will explain to you what Encoder, Decoder, and Attention are. The Encoder/Decoder models are the classes of general sequence-to-sequence modeling, which is used in several applications such as sentiment analysis, neural machine translation, chatbots, name entity recognition, and even text generation. For example, a question like "How are you today?" made to a chatbot may come up with an answer "I am fine, thank you! How are you doing today?" This is certainly not a word-to-word translation and the response thereof. These kinds of translations require model training on a huge repository of words and the use of neural network models which differ considerably from the traditional machine learning models that you have studied so far in the book.

In this chapter, we will just consider neural machine translation. Our model will translate English phrases to the equivalent Spanish words, as shown in the illustration in Figure 8-1.

Figure 8-1. *English to Spanish translation using seq2seq modeling*

The English phrase "How are you?" translates to "¿Cómo estás?" The output sequence need not necessarily be of the same size as its input. To understand how this is done, let me first explain Encoder and Decoder architectures.

Encoder/Decoder

The Encoder and Decoder are the two main components of a sequence-to-sequence model. Both use LSTMs. You will remember that LSTMs are capable of remembering long sequences and also do not suffer from vanishing gradients; thus, they are the ideal candidates for language translation models. The Encoder and Decoder are two different sets of the same LSTM architecture. First, I will describe the Encoder architecture.

Encoder

The encoder architecture is shown in Figure 8-2.

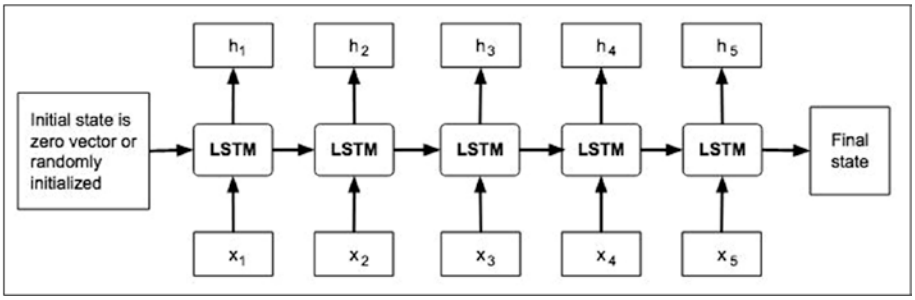


Figure 8-2. *Encoder architecture*

We split the entire input sentence at the word level and input it to our Encoder at each time step. In the case of our example statement "Peter is a good boy," we will have the inputs to LSTM in five time steps as shown here:

X1 = "Peter", X2 = "is", X3 = "a", X4 = "good", X5 = "boy"

The LSTM calculates the hidden state values h_i. These hidden states along with the next word are fed to the decoder in the next time step. This is how the network captures the contextual information of the input sequence. The initial state of the encoder is typically a zero vector. The final state, which is also called a "thought vector" of the encoder, is used as an input to the decoder.

Decoder

The decoder architecture is shown in Figure 8-3 with the timestamps for our example input sentence.

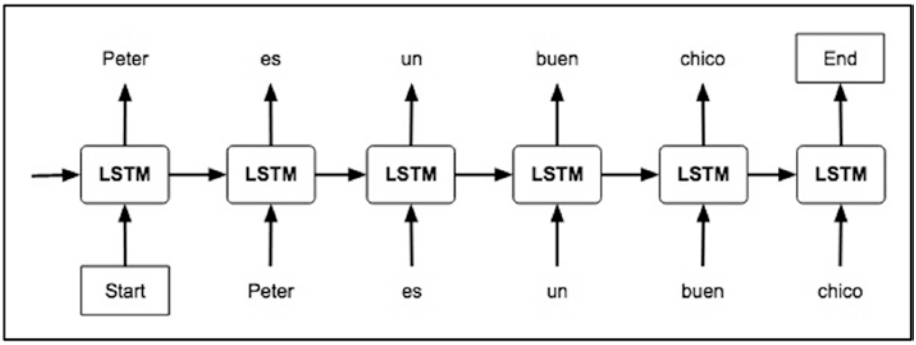


Figure 8-3. *Decoder architecture*

The decoder is trained to predict the next word given the previous word by feeding hidden states of decoder LSTM cells from the previous cell to the next. Before feeding the target sequence to the decoder, special tokens <START> and <END> are added to the beginning and end of the sequence.

The target sequence is unknown while decoding the test sequence. So, we start predicting the target sequence by passing the first word into the decoder which would be always the <start> token. The <end> token signals the end of the sentence.

I will now discuss the decoding steps during inference.

Inference

The time steps during the inference are listed as follows:

- The initial input to the Decoder is the <start> token.
- During inference, the Decoder LSTM is called multiple times in a loop to generate an output word in each time step/iteration.

- The initial states of the decoder equal the final states of the encoder.
- At each time step, the decoder states are preserved and are used as initial states during the next time step/iteration.
- The predicted output of each time step is fed as an input in the next time step/iteration.
- The loop breaks at the <end> tag.

The entire inference process for our sample text is shown in Figure 8-4.

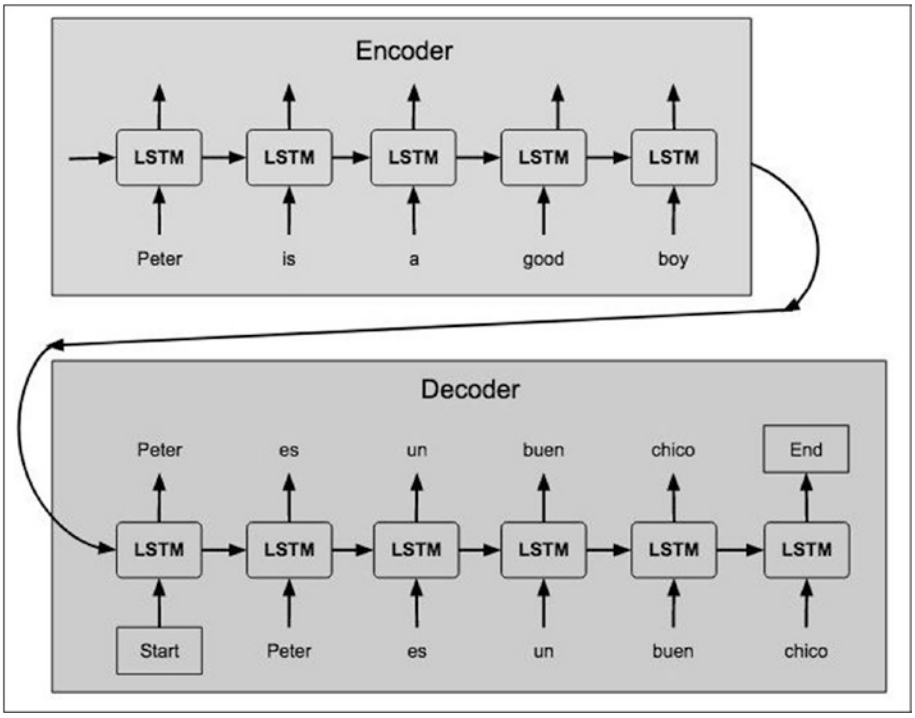


Figure 8-4. *Inference process in the Encoder/Decoder*

This is how the sequence-to-sequence (seq2seq) model works. I will now discuss the shortcomings of the seq2seq model.

Shortcomings of seq2seq Model

The seq2seq model is composed of an encoder-decoder architecture, where the encoder processes the input sequence and encodes the information into a context vector. This context vector is sometimes called a "thought vector" and is of a fixed length. The thought vector is expected to represent a good summary of the entire input sequence. The decoder is then initialized with this thought vector and asked to do the transformation. These fixed-length thought vectors suffer from an apparent disadvantage of being incapable of remembering long sequences. Usually, for long sequences, they forget the earlier part of the sequence by the time they process the entire sequence. To solve this problem, the attention mechanism was proposed.

Attention Model

Humans, when they see something or hear something, consciously pay attention to a portion of a picture or a text. Consider the photo shown in Figure 8-5.

Figure 8-5. *Image that draws attention to a horse and a lady. Image source:* `http://www.clker.com/clipart-man-riding-a-rearing-horse.html`

The moment you see this photo, your attention goes to a horse and a lady. So if you have to generate a caption for the photo, you would say "Lady riding a Horse." Likewise, if somebody raises a question, "Which sport do you like the most?", your immediate attention is on the words "sport" and "you." Our Encoder/Decoder model misses this feature of paying attention to certain keywords in the input sequence. So the intuition behind the attention mechanism is that how much attention do we need to pay to every word while generating a word at timestamp "t"? So, the basic idea behind the attention mechanism is to increase the importance of specific words in the input sequence while generating the target.

The attention model was created to help memorize long sentences in the input source. Instead of building a single context vector from the encoder's last hidden state, the attention model creates a context vector for the entire input sequence. This is depicted in Figure 8-6.

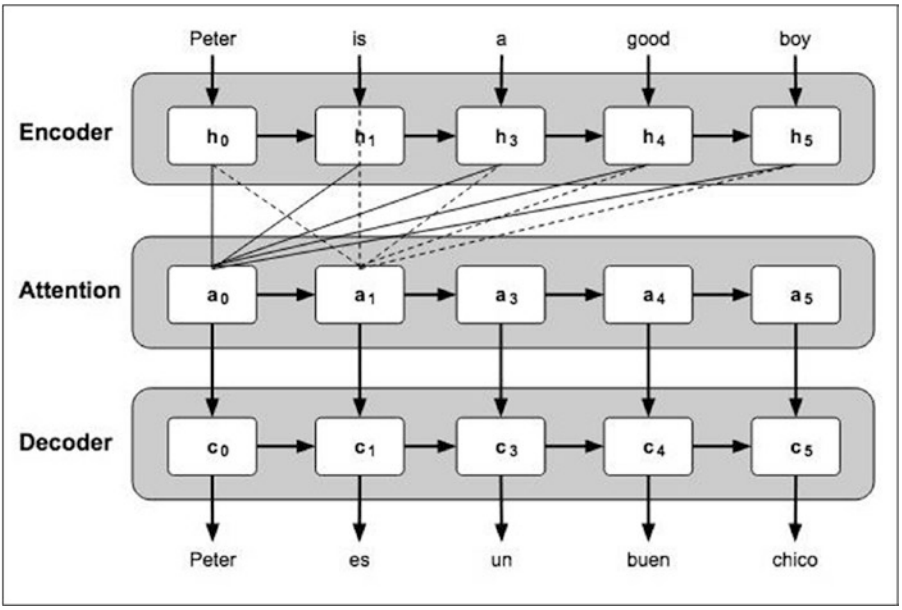


Figure 8-6. *Encoder/Decoder architecture with the Attention module*

Here, the input sequence is "Peter is a good boy." This sequence is split into the individual words. The decoder gets its input from the encoder outputs in time steps. If we use this simple seq2seq modeling, the translation quality will not be good. So, we introduce an Attention layer in between. The inputs to the decoder at each time step will now be its previous state + the attention context on the input sequence. For example, while generating the word "buen," the decoder will pay more attention to the words like Peter, good, and boy. The Attention network assigns different weights to the input words to generate an Attention context that the decoder uses while predicting the next word.

The attention weights (α) to the network at any given time step t_s can be expressed mathematically as follows:

$$\alpha_{t_s} = \frac{\exp\left(score\left(h_t, \bar{h}_s\right)\right)}{\sum_{s'=1}^{s} \exp(score) h_t, \bar{h}_{s'}))}$$

Then the context vector can be expressed as

$$c_t = \sum_s \propto_{ts} \bar{h}_s$$

where c_t represents the context at time step t, and the alpha and h represent the weights and hidden states, respectively. The final Attention vector, which is a function of all c_t and h_t, is computed as follows:

$$a_t = f\left(c_t, h_t\right) = \tanh\left(W_c\left[h_t, h_t\right]\right)$$

The a_t goes as an input to the decoder along with its own previous state. You will be able to get a better understanding of this Attention network when you look at its implementation, which is provided in the project in the next section.

With this introduction to neural machine translation, let us move on to a practical implementation.

English to Spanish Translator

We will create an English to Spanish translation using NMT techniques you learned so far in this chapter. You will be using the Encoder/Decoder model with the Attention module. For training such models, you require a set of sentence mappings between the two languages. Fortunately, somebody has created such mappings. You will find these mappings on this site (www.manythings.org/anki/). These are the tab-delimited bilingual sentence pairs. The number of sets is exhaustive, and you will find mappings between many popular languages in the world. Each line in the file is of the format:

English + TAB + The Other Language + TAB + Attribution

In this project, you are also going to use the Global Vectors for Word Representations created by Jeffrey Pennington et al. It is a huge dataset. I suggest that you download (http://nlp.stanford.edu/data/glove.6B.zip) this set to your local drive before starting on this project. Unzip the downloaded glove*.zip file. It contains the word to vectors in four different dimensions – 50, 100, 200, and 300. The glove files are explained in detail later. These are the text files with names glove.6B.50d.txt, glove.6B.100d.txt, glove.6B.200d.txt, and glove.6B.300d.txt. I have used 200 dimension files in the code. Copy all these files into your Google Drive so that your Colab project can access it.

After completing this download, you are now ready to create your project.

Creating Project

Open the new Colab document and rename it to NMT. Import the following libraries:

```
import tensorflow as tf

from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input,
            Dense, LSTM, Embedding, Bidirectional,
            RepeatVector, Concatenate, Activation,
            Dot, Lambda
from tensorflow.keras.preprocessing.text
            import Tokenizer
from tensorflow.keras.preprocessing.sequence
            import pad_sequences
from keras import preprocessing, utils
import numpy as np
import matplotlib.pyplot as plt
```

Downloading Translation Dataset

Download the Spanish to English translation dataset from the book's repository:

```
!pip install wget
import wget
url = 'https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch08/spa.txt'
wget.download(url,'spa.txt')
```

Creating Datasets

Read the contents of the translation data file using the following statement:

```
# reading data
with open('/content/spa.txt',encoding='utf-8',errors='ignore')
as file:
  text=file.read().split('\n')
```

You can print the first five lines using the following command:

```
text[:5]
```

The output is shown in Figure 8-7.

```
['Go.\tVe.\tCC-BY 2.0 (France) Attribution: tatoeba.org #2877272 (CM) & #4986655 (cueyayotl)',
 'Go.\tVete.\tCC-BY 2.0 (France) Attribution: tatoeba.org #2877272 (CM) & #4986656 (cueyayotl)',
 'Go.\tVaya.\tCC-BY 2.0 (France) Attribution: tatoeba.org #2877272 (CM) & #4986657 (cueyayotl)',
 'Go.\tVáyase.\tCC-BY 2.0 (France) Attribution: tatoeba.org #2877272 (CM) & #6586271 (arh)',
 'Hi.\tHola.\tCC-BY 2.0 (France) Attribution: tatoeba.org #538123 (CM) & #431975 (Leono)']
```

Figure 8-7. *Translation file contents*

Here, the input and target are separated by a tab character.

You may examine a few more records in the range of 100 through 110 with the following loop:

```
for t in text[100:110]:
  print(t)
```

The output is shown in Figure 8-8.

```
Be good.        Sed buenas.     CC-BY 2.0 (France) Attribution: tatoeba.org #7932304 (Seael) & #7932308 (Seael)
Be good.        Sed buenos.     CC-BY 2.0 (France) Attribution: tatoeba.org #7932304 (Seael) & #7932309 (Seael)
Be good.        Sea buena.      CC-BY 2.0 (France) Attribution: tatoeba.org #7932304 (Seael) & #7932310 (Seael)
Be good.        Sea bueno.      CC-BY 2.0 (France) Attribution: tatoeba.org #7932304 (Seael) & #7932311 (Seael)
Be good.        Sean buenas.    CC-BY 2.0 (France) Attribution: tatoeba.org #7932304 (Seael) & #7932313 (Seael)
Be good.        Sean buenos.    CC-BY 2.0 (France) Attribution: tatoeba.org #7932304 (Seael) & #7932315 (Seael)
Be kind.        Sean gentiles.  CC-BY 2.0 (France) Attribution: tatoeba.org #1916315 (CK) & #2092229 (hayastan)
Be nice.        Sé agradable.   CC-BY 2.0 (France) Attribution: tatoeba.org #1916314 (CK) & #5769224 (arh)
Beat it.        Pírate. CC-BY 2.0 (France) Attribution: tatoeba.org #37902 (CM) & #5769215 (arh)
Call me.        Llamame.        CC-BY 2.0 (France) Attribution: tatoeba.org #1553532 (CK) & #1555788 (hayastan)
```

Figure 8-8. *Translation file contents in tabular format*

Notice that the translations are separated by a tab character. The first set of words/sentences is our input dataset, and the second set after the tab is our target dataset. The remaining text is the attributions which we do not use here. For example, as seen in Figure 8-8, the translation of the English text “Be nice.” would be “Sé agradable.” Note that the same input sentences have more than one translation, all of which convey the same meaning.

We will now declare two variables for storing our input and target data.

```
input_texts=[] #encoder input
target_texts=[] # decoder input
```

We will select the first 10,000 words/sentences for our training. There are 122,937 sentences in the document with the last one being blank. There are 2,751,187 words and 18,127,427 characters in the document. If you use the full dataset, it would take a long time to train the model. Using a smaller dataset would also restrict your vocabulary for translation. We will split each line on the tab character to separate the input from the

target. We fill the two declared arrays with the first 10,000 entries in the database using the following code:

```
NUM_SAMPLES = 10000
for line in text[:NUM_SAMPLES]:
  english, spanish  = line.split('\t')[:2]
  target_text = spanish.lower()
  input_texts.append(english.lower())
  target_texts.append(target_text)
```

Print the first five entries from the two arrays for examining its contents:

```
print(input_texts[:5],target_texts[:5])
```

The output is shown here:

```
['go.', 'go.', 'go.', 'go.', 'hi.'] ['ve.', 'vete.', 'vaya.',
'váyase.', 'hola.']
```

As you can see, there is a dot character present in the text. We need to strip it.

Data Preprocessing

We will now do some text processing to make our text ready for machine learning.

Cleaning Up Punctuation

We will first remove all the punctuation characters from both target and input datasets. The punctuation characters are easily listed by calling the punctuation method of the string class.

```
import string
print('Characters to be removed in preprocessing', string.
punctuation)
```

This produces the following output:

```
Characters to be removed in preprocessing !"#$%&'()*+,-
./:;<=>?@[\]^_`{|}~
```

To remove the punctuation characters, we define a small function:

```
def remove_punctuation(s):
  out=s.translate(str.maketrans("","",string.punctuation))
  return out
```

We remove the punctuation from both the input and the target output by calling this function:

```
input_texts = [remove_punctuation(s)
                    for s in input_texts]
target_texts = [remove_punctuation(s)
                    for s in target_texts]
```

We can examine the first five items from both arrays to check that the data is clean:

```
input_texts[:5],target_texts[:5]
```

The output is shown as follows:

```
(['go', 'go', 'go', 'go', 'hi'], ['ve', 'vete', 'vaya',
'váyase', 'hola'])
```

Adding Start/End Tags

For the target text, we need to add the start and end tags. We have seen the purpose of <start> <end> tags while discussing the Encoder/Decoder architecture. We add the tags using the following statement:

```
# adding start and end tags
target_texts=['<start> ' + s + ' <end>'
        for s in target_texts]
```

You can examine one of the target items like this:

```
target_texts[1]
```

You will see the following output:

```
'<start> vete <end>'
```

Tokenizing Input Dataset

Our data is in text format, and we know that neural networks don't understand text data, so we will tokenize each word with some integer value.

We will first tokenize the input text to create the vocabulary. The word_index dictionary provides the mappings such as ('hello':133).

```
tokenizer_in=Tokenizer()
#tokenizing the input texts
tokenizer_in.fit_on_texts(input_texts)
#vocab size of input
input_vocab_size=len(tokenizer_in.word_index) + 1
```

You can check the size of the vocabulary:

```
input_vocab_size
```

The size is 2332. It means we have 2332 tokens in the dictionary, using which we will be able to do the translation. Had you used the full dataset of 122,937 words/sentences, you would have got the vocabulary of 13,731 words/sentences. You can examine the few items from the tokenized dictionary using the following code:

```
# Listing few items
input_tokens = tokenizer_in.index_word
for k,v in sorted(input_tokens.items())[2000:2010]:
   print (k,v)
```

You will see the following output:

```
2001 visa
2002 trains
2003 poems
2004 forgetful
2005 insane
2006 flew
2007 harvard
2008 obvious
2009 lecture
2010 divorce
```

As you can see, each word/sentence gets its unique identity by way of a token.

Tokenizing Output Dataset

Like the input dataset, you will tokenize the output dataset using the following code:

```
#tokenizing output that is spanish translation
tokenizer_out=Tokenizer(filters='')
tokenizer_out.fit_on_texts(target_texts)
```

```
#vocab size of output
output_vocab_size=len(tokenizer_out.word_index) + 1
output_vocab_size
```

As we have used angular brackets in our special tokens such as <start> and <end>, we do not want the tokenizer to filter on these (< >) symbols. So, for the output dataset we set the filter to single quotes (' ') to filter out the output tokens.

This will print the output of 4964, indicating that you have 4964 Spanish words/sentences in your vocabulary. You can check a few items from the dictionary.

```
# Listing few items
output_tokens = tokenizer_out.index_word
for k,v in sorted(output_tokens.items())[2000:2010]:
    print (k,v)
```

You will see the following output:

```
2001 suyos
2002 ley
2003 palabras
2004 ausente
2005 delgaducho
2006 sucio
2007 adoptado
2008 violento
2009 roncando
2010 podríamos
```

As you notice, you will have a list of Spanish words tokenized for our translations.

With these two datasets, you will train your Encoder/Decoder model. The user will be able to use your model to do the translations for these standard phrases appearing in the dataset. If the user inputs a phrase that is not in the standard dataset, the phrase will be split into words, and each word would be translated using these sets. After each translation, the model will try to guess the next word with the help of the Attention module. Note the importance of the attention module here. The decoder takes two inputs – the previous state of the encoder and the context vector that comes out of the Attention module. The decoder then predicts the word with the max probability based on these two inputs.

Creating Input Sequences

You will now convert the input text into sequences by calling the texts_to_sequences method. This is obviously required for feeding the data to our model.

```
#converting tokenized sentence into sequences
tokenized_input = tokenizer_in.texts_to_sequences
                    ( input_texts )
```

The words in our dictionary do not have a fixed length. However, for training purposes, we need to have sequences of fixed length. You will now determine the maximum length of the word in our tokenized input and pad all the sequences to this length with zeros.

```
#max length of the input
maxlen_input = max( [ len(x)
                    for x in tokenized_input ] )

#padding sequence to a maximum fixed length
padded_input = preprocessing.sequence.pad_sequences
        ( tokenized_input , maxlen=maxlen_input ,
                padding='post' )
```

You may examine the few sequences to see the effect of this padding.

```
padded_input[2000:2010]
```

The output is shown as follows:

```
array([[  1, 613, 195,   0,   0],
       [  1,  54, 109,   0,   0],
       [  1,  54, 109,   0,   0],
       [  1,  54, 109,   0,   0],
       [  1,  54, 182,   0,   0],
       [  1,  54,  14,   0,   0],
       [  1,  54,  98,   0,   0],
       [  1,  54,  98,   0,   0],
       [  1,  54,  10,   0,   0],
       [  1,  54,  10,   0,   0]], dtype=int32)
```

Note the presence of zeros at the end of each sequence. In other words, an input sentence like "I voted yes" will be converted to a padded sequence such as [1,613,195,0,0]. Note the last two zeros in the sequence; that's the padding.

So, all our input sequences now have a constant size of 5. Had you taken the entire vocabulary, the maximum length of the word would have been different, and thus the size of each input sequence would have been this maximum value.

We convert the input data into a numpy array:

```
encoder_input_data = np.array( padded_input )
print( encoder_input_data.shape)
```

The program prints the following value:

```
(10000, 5)
```

Thus, there are 10,000 input sequences of fixed width, which is 5.

Creating Output Sequences

Like the input text, you will convert the tokenized output into sequences:

```
#converting tokenized text into sequences
tokenized_output = tokenizer_out.texts_to_sequences
                        (target_texts)
```

You can check the size of the output vocabulary with the following code:

```
output_vocab_size=len(tokenizer_out.word_index) + 1
output_vocab_size
```

The size is 4964, so we have 4964 Spanish words in our vocabulary.

We use teacher forcing on the output so that the training converges faster. In teacher forcing, we are hinting the next word to the decoder, reducing its guessing work and making it learn faster. Note that teacher forcing is used only while training the model and not during testing or in the inference mode.

```
# teacher forcing
for i in range(len(tokenized_output)) :
   tokenized_output[i] = tokenized_output[i][1:]
```

We determine the maximum length for the output and pad all the output tokens with zeros.

```
maxlen_output = max( [ len(x)
                        for x in tokenized_output ] )
padded_output = preprocessing.sequence.pad_sequences
                        ( tokenized_output ,
                            maxlen=maxlen_output ,
                            padding='post' )
```

We convert the output data to a numpy array to make it compatible for machine learning.

```
# converting to numpy
decoder_input_data = np.array( padded_output )
```

Let us print a few elements.

```
decoder_input_data[2000:2010]
```

Printing a few elements of the decoder data produces the following output:

```
array([[  11, 1147,   26,  108,    2,    0,    0,    0,    0],
       [  51,   17,  135,    2,    0,    0,    0,    0,    0],
       [  11,   51,   17,  135,    2,    0,    0,    0,    0],
       [  11,   51,   13,  224,    2,    0,    0,    0,    0],
       [  51,   59,    2,    0,    0,    0,    0,    0,    0],
       [  51,   20,    2,    0,    0,    0,    0,    0,    0],
       [  28,   51,    2,    0,    0,    0,    0,    0,    0],
       [  37,   51,    2,    0,    0,    0,    0,    0,    0],
       [  51,   88,    2,    0,    0,    0,    0,    0,    0],
       [  51,   19,    2,    0,    0,    0,    0,    0,    0]],
      dtype=int32)
```

Note how each output element is padded with zeros to make all of equal length. Here the sequence length is 9 for our chosen dataset. Again, had you used the full dataset, this fixed length would have a different value.

We use one-hot encoding for creating the decoder target output.

```
#decoder target output
decoder_target_one_hot=np.zeros((len(input_texts),
                                 maxlen_output,
                                 output_vocab_size),
                                 dtype='float32')
```

```
for i,d in enumerate(padded_output):
   for t,word in enumerate(d):
       decoder_target_one_hot[i,t,word]=1
```

You may examine the target output by printing one of its elements:

```
decoder_target_one_hot[0]
```

You will see the following output:

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       ...,
       [1., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

Note how the entire output is encoded with one-hot encoding.

Glove Word Embedding

Before starting the project, you downloaded the word to vectors dataset. This dataset gives ratios of word-to-word co-occurrence probabilities. In other words, for any given word, it tells you what the probabilities are of all other words in this world to occur next to this given word. For example, the probability of the word "cream" occurring next to the word "ice" could be the highest. This dataset was literally created on a huge vocabulary. The knowledge of these probabilities can help in encoding some form of meaning to the input statement. We will build a dictionary based on this dataset for our input English language vocabulary. We will feed our input sequence to an embedding layer that converts the input words into word vectors. These word vectors will be passed as an input to the Encoder. We will load this word to vectors dictionary in our program.

Indexing Word Vectors

The glove word embedding gives us four mappings for dimensions: 50, 100, 200, and 300. In our program, we will use a 200-dimension file. The higher the dimensions you use, the better would be the translation, but at the cost of processing time and resources.

A sample line from the 50-dimension file is shown here:

```
more 0.87943 -0.11176 0.4338 -0.42919 0.41989 0.2183 -0.3674
-0.60889 -0.41072 0.4899 -0.4006 -0.50159 0.24187 -0.1564
0.67703 -0.021355 0.33676 0.35209 -0.24232 -1.0745 -0.13775
0.29949 0.44603 -0.14464 0.16625 -1.3699 -0.38233 -0.011387
0.38127 0.038097 4.3657 0.44172 0.34043 -0.35538 0.30073
-0.09223 -0.33221 0.37709 -0.29665 -0.30311 -0.49652 0.34285
0.77089 0.60848 0.15698 0.029356 -0.42687 0.37183 -0.71368 0.30175
' -0.039369 1.2036 0.35401 -0.55999 -0.52078 -0.66988 -0.75417
-0.6534 -0.23246 0.58686 -0.40797 1.2057 -1.11 0.51235 0.1246
0.05306 0.61041 -1.1295 -0.11834 0.26311 -0.72112 -0.079739
0.75497 -0.023356 -0.56079 -2.1037 -1.8793 -0.179 -0.14498
-0.63742 3.181 0.93412 -0.6183 0.58116 0.58956 -0.19806 0.42181
-0.85674 0.33207 0.020538 -0.60141 0.50403 -0.083316 0.20239
0.443 -0.060769 -0.42807 -0.084135 0.49164 0.085654
```

The word index is *more*. The remaining entries in the line are the co-occurrence probabilities of other words. We split each line from the embedding matrix file into word tokens. We will use the first token as the key in our dictionary. The remaining tokens are the co-occurrence probabilities of other words for this target word and are inserted as values in the dictionary.

The code for creating this dictionary is given as follows:

```
#creating dictionary of words corresponding to vectors
print('Indexing word vectors.')

embeddings_index = {}

# Use this open command in case of downloading using wget above
#f = open('glove.6B.200d.txt', encoding='utf-8')

# we can choose any dimensions 50 100 200 300
f = open('drive/My Drive/tfbookdata/glove.6B.200d.txt',
                encoding='utf-8')

for line in f:
   values = line.split()
   word = values[0]
   coefs = np.asarray(values[1:], dtype='float32')
   embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' %
            len(embeddings_index))
```

When you run the code, it will print 400000, indicating that so many embeddings are available in the full dataset. Of course, we will need the embeddings for the words that we have chosen for our dataset of 2332 words. You can try printing the matrix for one of the keywords:

```
embeddings_index["any"]
```

The partial output of the matrix is shown in Figure 8-9.

```
array([ 6.3113e-01,  4.3183e-01,  2.3103e-01, -6.4909e-01,  2.3744e-01,
        4.4619e-01, -8.6148e-01,  2.9341e-01,  8.0033e-02,  8.5633e-03,
        1.0165e-01,  6.2783e-01,  2.5047e-01,  5.2425e-02,  6.3045e-01,
       -4.0008e-02,  2.5212e-01,  6.2147e-01,  6.6967e-02, -7.9787e-02,
       -2.1607e-02,  3.4236e+00, -4.7925e-02,  2.4620e-01, -2.8834e-02,
        4.0330e-02, -2.7858e-01, -2.7939e-01,  3.5606e-01, -5.7373e-01,
       -9.9960e-02, -3.2374e-01,  1.7812e-01,  2.0671e-02,  2.3637e-01,
       -1.7074e-01, -4.9345e-01, -4.0289e-01, -3.4184e-01, -1.9405e-01,
```

Figure 8-9. *The embedding matrix for the keyword "any"*

You will notice that for each word like "any," there are 200 mappings available, which is the size of the dimension that we selected.

Creating Subset

Now, we will extract the entries for our English input vocabulary, using a simple for loop as shown here:

```
#embedding matrix
num_words = len(tokenizer_in.word_index)+1
word2idx_input = tokenizer_in.word_index
embedding_matrix = np.zeros((num_words, 200))
for word,i in word2idx_input.items():
   if i<num_words:
       embedding_vector = embeddings_index.get(word)
   if embedding_vector is not None:
       embedding_matrix[i] = embedding_vector
embedding_matrix.shape
```

After the matrix is created, its shape will be printed on the terminal, which is as follows:

```
(2332, 200)
```

Note that we had earlier seen that there are 2332 tokenized words (English) in our input repository. For each word, we have 200 co-occurrence probabilities. Why 200? This is because we have selected a 200-dimension mapping file.

Defining Embedding Layer

We declare two variables, which we will use while creating LSTM and embedding layers.

```
#lstm hidden dimensions
LATENT_DIM=256
#embeding layer dimensions
EMBEDDING_DIM=200
```

We declare the embedding layer as follows:

```
embedding_layer=Embedding(input_vocab_size,
                          EMBEDDING_DIM,
                          weights=[embedding_matrix],
                          input_length=maxlen_input)
```

Note our input vocab size is 2332, and the dimension is 200 – the size of the matrix which we selected. The weights parameter is set to the embedding matrix we created earlier that provides the word to vectors mappings. The input length is set to the maximum length of our padded sequences, which in our case is 5.

Now, we will start defining models. For this, we will create a few utility methods and layers.

Defining Encoder

We now define the encoder. For the encoder, first we define the input layer.

```
#encoder
encoder_input = Input(shape=(maxlen_input,),
                      name = 'encoderinput')
```

The input to this layer is of size 5, determined by the maxlen_input parameter value. We create an embedding_layer with this input using the following statement:

```
encoder_input = embedding_layer(encoder_input)
```

Next, we define the LSTM layer as follows:

```
encoder = Bidirectional(LSTM(LATENT_DIM,
                             return_sequences=True,
                             dropout = 0.3),
                        name = 'encoder_bidirection')
```

We use a bidirectional LSTM. The number of hidden layers is set to LATENT_DIM, which is 200. The sequences are returned to the next hidden layer at each time step. There is a dropout of 30% at each layer.

Finally, we pass the encoder input layer to this encoder.

```
encoder_outputs = encoder(encoder_input)
```

This generates the encoder outputs. You may check its shape by calling the shape method on it.

```
encoder_outputs.shape
```

It gives the following output:

```
TensorShape([None, 5, 512])
```

As the LSTM is bidirectional, we get the 512 dimension, which is our latent dimension (LATENT_DIM) of 256 multiplied by 2. The latent dimension is the number of units in the LSTM layer.

With this, we will now proceed to define the decoder.

Defining Decoder

We first define the input layer for our decoder model:

```
decoder_inputs = Input(shape=(maxlen_output,),
                       name='decoder_input')
```

where the maxlen_output is 9 for our dataset. The embedding layer is defined using the following statement:

```
decoder_embedding = Embedding(output_vocab_size,
                              EMBEDDING_DIM,
                              name='decoder_embedding')
```

The output_vocab_size is 4964 for our dataset, and the output dimension specified by the value of EMBEDDING_DIM is 200. Finally, we create a decoder input:

```
decoder_input = decoder_embedding(decoder_inputs)
```

Print the shape of this tensor by calling the shape method:

```
decoder_input.shape
```

The output is

```
TensorShape([None, 9, 200])
```

Note that the sequence length for our output vocabulary is 9.

Defining Decoder Layers

We now define decoder layers. We need an LSTM layer and a softmax-activated Dense layer.

The LSTM layer is defined as follows:

```
decoder_lstm = LSTM(LATENT_DIM,
                    return_state = True,
                    name = 'decoder_lstm')
```

The LATENT_DIM parameter has a value of 256, which is the number of hidden layers. At every time step, the state is returned to the next layer.

The softmax Dense layer is defined as follows:

```
decoder_dense = Dense(output_vocab_size,
                      activation='softmax',
                      name='decoder_dense')
```

The dimension of this layer is set to output_vocab_size, which is 4964 in our case. This layer will predict the probability of occurrence of each word from our vocabulary. Had we taken the full dataset, the dimensions of this layer would be very high, resulting in an increase in the decoding time.

We now define the attention layers.

Attention Network

We will now start defining the model for our Attention network.

Defining Softmax

Our Attention context is computed for each time step with a different set of attention weights assigned to the network. The attention weights (α) at any given time step t_s can be expressed mathematically as follows:

$$\alpha_{t_s} = \frac{\exp\left(score\left(h_t, \bar{h}_s\right)\right)}{\sum_{s'=1}^{s} \exp(score) h_t, \bar{h}_{s'}))}$$

We implement the calculation of alphas in the following method called softmax_attention:

```
# Computing alphas
import tensorflow.keras.backend as k
def softmax_attention(x):
   assert(k.ndim(x)>2)

   e=k.exp(x-k.max(x,axis=1,keepdims=True))
   s=k.sum(e,axis=1,keepdims=True)
   return e/s
```

The data is expected to be in the shape NxTxD, where N is the number of samples, T is the sequence length, and D is the vector dimensionality. With softmax over time, we want to ensure all the outputs in time dimension sum to 1. So, we exponentiate all values in the input but subtract the max for numerical stability. Then, we divide the sum of exponentials. The operations are done on axis=1, which is the time dimension.

Attention Layers

We will define a few layers for the use of our Attention model. We declare these layers globally as they will be repeatedly used by the decoder.

We create an attention_repeat that acts as a bridge between the encoder and decoder modules.

```
attention_repeat =  RepeatVector(maxlen_input)
```

The maxlen_input is 5 in our case. We define the concatenation layer as follows:

```
attention_concat = Concatenate(axis=-1)
```

We define a Dense layer with ten nodes and tanh activation.

```
dense1_layer = Dense(10,activation='tanh')
```

We create another Dense layer with a single node. The activation is done by our especially written softmax over time function.

```
dense2_layer = Dense(1,activation = softmax_attention)
```

Finally, we write a Dot layer for performing the weighted sum of $\alpha_t * h_t$.

```
dot_layer = Dot(axes=1)
```

Attention Context

We now write an attention function called context_attention that will be called by the decoder at each time step.

```
def context_attention(h, st_1):
```

The function takes two parameters: h is the current hidden state of the encoder, and s_{t_1} is the previous hidden state of the decoder. So, h will take different values like h_1, h_2, …, h_t at each iteration; similarly, the s will also assume different values at each time step. The shape of this will be LATENT_DIM*2, which equals 512 in our case. We defined LATENT_DIM to be 256. Why is this multiplied by 2? This is because we will use a bidirectional LSTM.

We will now use our repeat layer to copy s_{t-1} t_x times. The shape will be (t_x, LATENT_DIM).

```
st_1=attention_repeat(st_1)
```

We concatenate all h_t with s_{t-1}.

```
x=attention_concat([h,st_1])
```

The shape of x will be (t_x, LATENT_DIM + LATENT_DIM * 2).
We now use our first Dense layer for learning α values.

```
x=dense1_layer(x)
```

We use our second Dense layer with our custom softmax function.

```
alphas=dense2_layer(x)
```

We finally take a dot product of all α and h.

```
context = dot_layer([alphas,h])
```

The final context is returned to the caller.

```
return context
```

The full function code for context_attention is given in Listing 8-1.

Listing 8-1. Function for computing the attention context

```
# computing attention context
def context_attention(h, st_1)
   st_1=attention_repeat(st_1)
   x=attention_concat([h,st_1])
   x=dense1_layer(x)
   alphas=dense2_layer(x)
   context = dot_layer([alphas,h])
   return context
```

Collecting Outputs

We are now going to collect outputs over all time steps. First, we need to create initial states for feeding to our decoder.

```
initial_s = Input(shape=(LATENT_DIM,), name='s0')
initial_c = Input(shape=(LATENT_DIM,), name='c0')
```

We also write a Concatenate function to concatenate outputs along axis 2.

```
context_last_word_concat_layer = Concatenate(axis=2)
```

We will iterate through all time steps to collect the outputs. We copy the initial states to two local variables which are used in the for loop later.

```
s = initial_s
c = initial_c
```

We declare an outputs array for collecting outputs.

```
outputs = []
```

We iterate through all time steps by declaring a for loop:

```
for t in range(maxlen_output): #ty times
```

The maxlen_output for our vocabulary is nine, so the for loop will collect nine outputs.

For the initial state, we create a context by calling our context_attention function.

```
#get the context using attention mechanism
context=context_attention(encoder_outputs,s)
```

Note the parameters to the function; the first parameter is the outputs of the encoder, and the second parameter is the initial state of the decoder.

We need a different layer for each time step:

```
selector=Lambda(lambda x: x[:,t:t+1])
x_t=selector(decoder_input)
```

We call our context_last_word_concat_layer function to create an input for the decoder.

```
decoder_lstm_input=context_last_word_concat_layer
                        ([context,x_t])
```

We pass the combined [context,last word] into the lstm along with s and c to get the new s, c and output.

```
out,s,c=decoder_lstm(decoder_lstm_input, initial_state=[s,c])
```

We call the final Dense layer to get the next word prediction.

```
decoder_outputs = decoder_dense(out)
```

We append the output to our list.

```
outputs.append(decoder_outputs)
```

The complete for loop for collecting outputs is shown in Listing 8-2 for your quick reference.

Listing 8-2. Decoder outputs for time steps

```
s = initial_s
c = initial_c
outputs = []

#collect output in a list at first
for t in range(maxlen_output): #ty times
    #get the context using attention mechanism
    context=context_attention(encoder_outputs,s)
```

```
    #we need a different layer for each time step
    selector=Lambda(lambda x: x[:,t:t+1])
    x_t=selector(decoder_input)

    #combine
    decoder_lstm_input=context_last_word_concat_layer
                          ([context,x_t])

    #pass the combined [context,last word] into lstm
    #along with [s,c]
    #get the new[s,c] and output
    out,s,c=decoder_lstm(decoder_lstm_input,
                           initial_state=[s,c])

    #final dense layer to get next word prediction
    decoder_outputs = decoder_dense(out)
    outputs.append(decoder_outputs)
```

The outputs is now a list of length t_y. Try printing the outputs using the following command:

```
outputs
```

You will get the following output:

```
[<tf.Tensor 'decoder_dense_9/Identity:0' shape=(None, 4964)
dtype=float32>,
 <tf.Tensor 'decoder_dense_10/Identity:0' shape=(None, 4964)
dtype=float32>,
 <tf.Tensor 'decoder_dense_11/Identity:0' shape=(None, 4964)
dtype=float32>,
 <tf.Tensor 'decoder_dense_12/Identity:0' shape=(None, 4964)
dtype=float32>,
 <tf.Tensor 'decoder_dense_13/Identity:0' shape=(None, 4964)
dtype=float32>,
```

```
 <tf.Tensor 'decoder_dense_14/Identity:0' shape=(None, 4964)
 dtype=float32>,
 <tf.Tensor 'decoder_dense_15/Identity:0' shape=(None, 4964)
 dtype=float32>,
 <tf.Tensor 'decoder_dense_16/Identity:0' shape=(None, 4964)
 dtype=float32>,
 <tf.Tensor 'decoder_dense_17/Identity:0' shape=(None, 4964)
 dtype=float32>]
```

Each element is of shape (batch size, output vocab size). If we simply stack all the outputs into a single tensor, it would be of shape T x N x D. We want the N dimension to appear first. So, we define a function to stack and transpose the matrix as follows. The parameter x is a list of length T; each element is a batch_size * output_vocab_size tensor.

```
def stack(x):
   x=k.stack(x)
   x=k.permute_dimensions(x,pattern=(1,0,2))
   return x
```

The return value is a tensor of batch_size x T x output_vocab_size.

We call this function as follows to generate our final outputs:

```
stacker=Lambda(stack)
outputs=stacker(outputs)
outputs
```

This gives the following output:

```
<tf.Tensor 'lambda_18/Identity:0' shape=(None, 9, 4964)
dtype=float32>
```

Note how the shape is changed after transposing the matrix.

Finally, we are now ready to define our model.

Defining Model

Our final model will have an encoder, a decoder, and our initial states for the encoder as inputs, and the outputs would be our final outputs as generated earlier. The model is defined using the following statement:

```
model=Model(inputs=[encoder_inputs,
                    decoder_inputs,
                    initial_s,
                    initial_c],
            outputs=outputs)
```

You can get the model summary by calling its summary method. The output is a long one. I will show you only the first few lines of the summary, which are depicted in Figure 8-10.

```
model.summary()
```

```
Model: "model"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
enocderinput (InputLayer)       [(None, 5)]          0
__________________________________________________________________________________________________
embedding_1 (Embedding)         (None, 5, 200)       466400      enocderinput[0][0]
__________________________________________________________________________________________________
s0 (InputLayer)                 [(None, 256)]        0
__________________________________________________________________________________________________
enocder_bidirection (Bidirectio (None, 5, 512)       935936      embedding_1[1][0]
__________________________________________________________________________________________________
repeat_vector (RepeatVector)    (None, 5, 256)       0           s0[0][0]
                                                                 decoder_lstm[9][1]
                                                                 decoder_lstm[10][1]
                                                                 decoder_lstm[11][1]
                                                                 decoder_lstm[12][1]
                                                                 decoder_lstm[13][1]
                                                                 decoder_lstm[14][1]
                                                                 decoder_lstm[15][1]
                                                                 decoder_lstm[16][1]
__________________________________________________________________________________________________
concatenate (Concatenate)       (None, 5, 768)       0           enocder_bidirection[0][0]
                                                                 repeat_vector[9][0]
                                                                 enocder_bidirection[0][0]
                                                                 repeat_vector[10][0]
```

Figure 8-10. *Model embedding Encoder/Decoder*

The number of trainable parameters for our model is given in the following output:

```
Total params: 4,670,841
Trainable params: 4,670,841
Non-trainable params: 0
```

You can easily judge the complexity of such language models for training. Had we used the full dataset, the number of parameters would have been tremendously large.

Try printing the visual representation of the model using the plot_model function:

```
tf.keras.utils.plot_model (model)
```

The plot is shown in Figure 8-11.

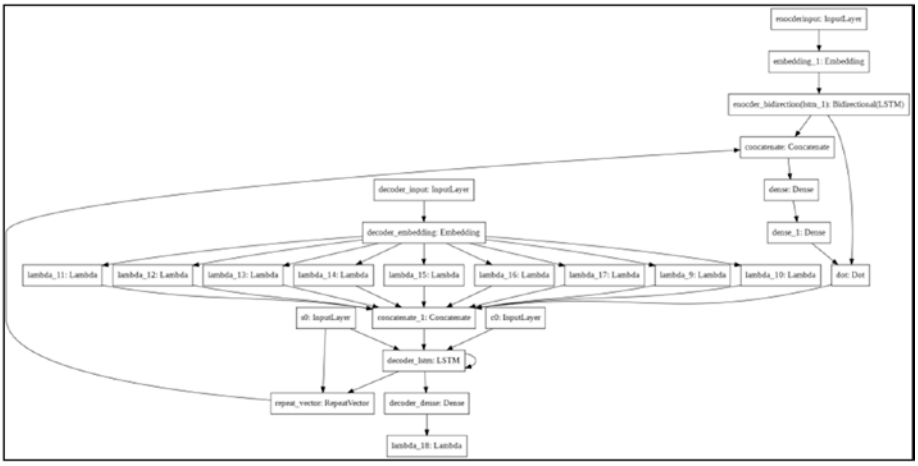


Figure 8-11. *Visual representation of the full model*

From this visual representation, you will be able to see the complete flow of language translation. Verify it for yourself how the input sequence is first embedded using the glove word to vector dictionary (embedding layer at the top-right corner), followed by the bidirectional Encoder and then that entire complex Decoder with the Attention network.

Finally, we compile the model using the categorical cross-entropy and Adam optimizer.

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

Model Training

We create the initial states by taking zero as inputs.

```
initial_s_training=np.zeros((NUM_SAMPLES,LATENT_DIM))
initial_c_training =np.zeros(shape=(NUM_SAMPLES,LATENT_DIM))
```

We train the model using the following statement:

```
R = model.fit([encoder_input_data,
               decoder_input_data,
               initial_s_training,
               initial_c_training],
               decoder_target_one_hot,
               batch_size=100,
               epochs=100,
               validation_split=0.3)
```

I trained the model for 100 epochs; each epoch took about 5 seconds on GPU.

Inference

We are now ready to test our model on some real inference. For this, we will define an Encoder for encoding the user input text and a decoder that does the translations.

Encoding

We create an Encoder model for encoding our input text.

```
encoder_model = Model(encoder_inputs, encoder_outputs)
```

The model summary is shown in Figure 8-12.

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
enocderinput (InputLayer)    [(None, 5)]               0
_________________________________________________________________
embedding_1 (Embedding)      (None, 5, 200)            466400
_________________________________________________________________
enocder_bidirection (Bidirec (None, 5, 512)            935936
=================================================================
Total params: 1,402,336
Trainable params: 1,402,336
Non-trainable params: 0
```

Figure 8-12. *Summary of the Encoder model*

You may also generate the visual representation of this model to find out where it is placed in our full model shown earlier in Figure 8-12.

```
tf.keras.utils.plot_model (encoder_model)
```

The model plot is shown in Figure 8-13.

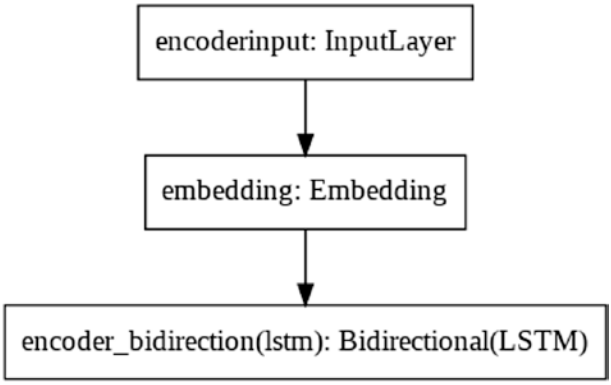


Figure 8-13. *Encoder model visualization*

We will use this model while translating the input text.

Decoding

We will create a few variables for our Decoder model.

```
encoder_outputs_as_input = Input(shape=
                    (maxlen_input, LATENT_DIM* 2,))
```

Note that the length is LATENT_DIM * 2 because we use a bidirectional LSTM.

We are going to predict one word at a time with an input of one word, so we declare another variable as follows:

```
decoder_input_embedding = Input(shape=(1,))
```

We call decoder_embedding to create an input for the decoder with a single word input.

```
decoder_input_ = decoder_embedding
                    (decoder_input_embedding)
```

We calculate the context at each time step by calling our one_step_attention function:

```
context = context_attention
            (encoder_outputs_as_input, initial_s)
```

We compute the decoder input by calling our context_last_word_concat_layer method.

```
decoder_lstm_input = context_last_word_concat_layer(
   [context, decoder_input_])
```

We call the decoder and collect its outputs.

```
out, s, c = decoder_lstm(decoder_lstm_input,
                         initial_state=[initial_s, initial_c])
decoder_outputs = decoder_dense(out)
```

Finally, we define the Decoder model as follows:

```
decoder_model = Model(
  inputs=[
    decoder_input_embedding,
    encoder_outputs_as_input,
    initial_s,
    initial_c
  ],
  outputs=[decoder_outputs, s, c]
)
decoder_model.summary()
```

The model summary is shown in Figure 8-14.

```
Model: "model_2"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
s0 (InputLayer)                 [(None, 256)]        0
__________________________________________________________________________________________________
input_1 (InputLayer)            [(None, 5, 512)]     0
__________________________________________________________________________________________________
repeat_vector (RepeatVector)    (None, 5, 256)       0           s0[0][0]
__________________________________________________________________________________________________
concatenate (Concatenate)       (None, 5, 768)       0           input_1[0][0]
                                                                 repeat_vector[18][0]
__________________________________________________________________________________________________
dense (Dense)                   (None, 5, 10)        7690        concatenate[18][0]
__________________________________________________________________________________________________
dense_1 (Dense)                 (None, 5, 1)         11          dense[18][0]
__________________________________________________________________________________________________
input_2 (InputLayer)            [(None, 1)]          0
__________________________________________________________________________________________________
dot (Dot)                       (None, 1, 512)       0           dense_1[18][0]
                                                                 input_1[0][0]
__________________________________________________________________________________________________
decoder_embedding (Embedding)   multiple             992800      input_2[0][0]
__________________________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 1, 712)       0           dot[18][0]
                                                                 decoder_embedding[1][0]
__________________________________________________________________________________________________
c0 (InputLayer)                 [(None, 256)]        0
__________________________________________________________________________________________________
decoder_lstm (LSTM)             [(None, 256), (None, 992256      concatenate_1[18][0]
                                                                 s0[0][0]
                                                                 c0[0][0]
__________________________________________________________________________________________________
decoder_dense (Dense)           (None, 4964)         1275748     decoder_lstm[18][0]
==================================================================================================
Total params: 3,268,505
Trainable params: 3,268,505
Non-trainable params: 0
```

Figure 8-14. *Summary of the Decoder model*

The number of trainable parameters is a whopping 3 million plus. You may generate a visual representation of the model as follows:

```
tf.keras.utils.plot_model (decoder_model)
```

The model plot is shown in Figure 8-15.

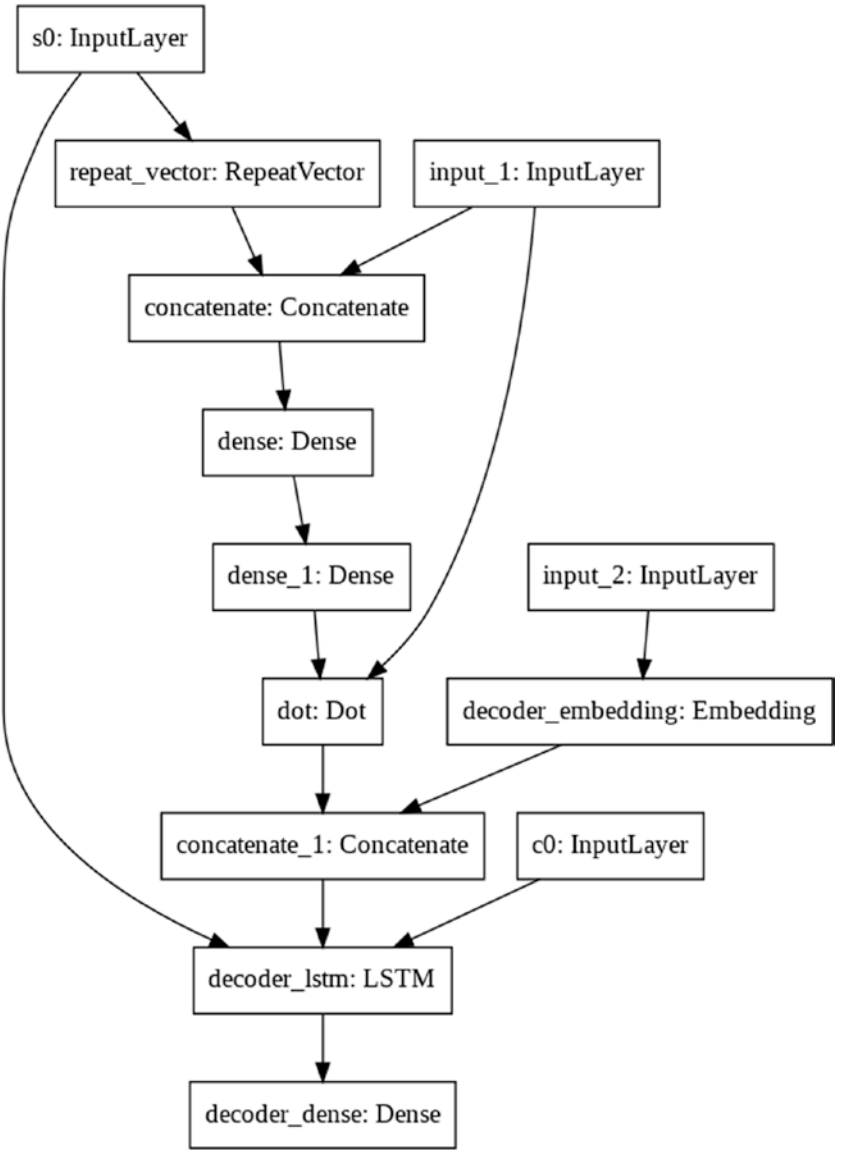


Figure 8-15. *Visual representation of the Decoder model*

We are now ready to try our language translations.

Translating

As the model processes the input text and predicts the output as integers, we need to create a reverse mapping from integers to index. We do this for both the languages using the following code:

```
word2index_input=tokenizer_in.word_index
word2index_output=tokenizer_out.word_index
#reverse mapping integer to words for english
idx2word_eng = {v:k for k, v in
                    word2index_input.items()}
#reverse mapping integer to words for spanish
idx2word_trans = {v:k for k, v in
                    word2index_output.items()}
```

We write a function for decoding a sentence consisting of a few words. The function is defined as follows:

```
def decode_sequence(input_seq):
```

In the function body, we encode the input as state vectors.

```
  enc_out = encoder_model.predict(input_seq)
```

We generate the empty target sequence of length 1.

```
  target_seq = np.zeros((1, 1))
```

We populate the first character of the target sequence with the start character. Note that the tokenizer lowercases all words.

```
  target_seq[0, 0] = word2index_output['<start>']
```

We use the <end> token for breaking the loop:

```
  End_statement = word2index_output['<end>']
```

As the states are updated in each loop, we initialize them on each iteration.

```
s = np.zeros((1,LATENT_DIM))
c = np.zeros((1,LATENT_DIM))
```

For storing the translated sequence, we declare an output array.

```
output_sentence = []
```

We now define a for loop for generating the output.

```
for _ in range(maxlen_output):
```

In each iteration, we call the decoder's predict method.

```
  out, s, c = decoder_model.predict([target_seq,
                                     enc_out, s, c])
```

We pick up the next word based on the prediction probabilities of the decoder output.

```
  index = np.argmax(out.flatten())
```

If we reach the end of the sequence, we terminate the loop.

```
  if End_statement == index:
    break
```

We convert the index of the predicted word to its character format and add it to the output sequence.

```
  word = ''
  if index > 0:
    word = idx2word_trans[index]
    output_sentence.append(word)
```

We update the decoder input, which is just the word last generated.

```
target_seq[0, 0] = index
```

Finally, we return the combined output sequence to the caller.

```
return ' '.join(output_sentence)
```

The entire function is given in Listing 8-3 for your quick reference.

Listing 8-3. Function for decoding an input sequence

```
def decode_sequence(input_seq):
  # Encode the input as state vectors.
  enc_out = encoder_model.predict(input_seq)

  # Generate empty target sequence of length 1.
  target_seq = np.zeros((1, 1))

  # Populate the first character of target sequence with the
    start character.
  # NOTE: tokenizer lower-cases all words
  target_seq[0, 0] = word2index_output['<start>']

  # if we get this we break
  End_statement = word2index_output['<end>']

  # [s, c] will be updated in each loop iteration
  s = np.zeros((1,LATENT_DIM))
  c = np.zeros((1,LATENT_DIM))

  # Create the translation
  output_sentence = []
  for _ in range(maxlen_output):
    out, s, c = decoder_model.predict([target_seq,
                                enc_out, s, c])
```

```
    # Get next word
    index = np.argmax(out.flatten())

    # End sentence
    if End_statement == index:
      break

    word = ''
    if index > 0:
      word = idx2word_trans[index]
      output_sentence.append(word)

    # Update the decoder input
    # which is just the word just generated
    target_seq[0, 0] = index

  return ' '.join(output_sentence)
```

We are now ready to do a few real translations. I have defined a loop for accepting the user input. The user can enter an English statement and will get a translated version in Spanish. The loop code is shown as follows:

```
count=0
while (count<5):

  input_text=[str(input('input the sentence : '))]
  seq=tokenizer_in.texts_to_sequences(input_text)

  input_seq=pad_sequences(seq,maxlen=maxlen_input,
                                padding='post')
  translation=decode_sequence(input_seq)

  print('Predicted translation:', translation)
  count+=1
```

The few translations made by the model are shown in the following output:

```
input the sentence : Hello
Predicted translation: hola
input the sentence : How are you?
Predicted translation: ¿qué están
input the sentence : What are you doing?
Predicted translation: ¿cómo están usted
input the sentence : see you later
Predicted translation: te luego
input the sentence : bye
Predicted translation: ¡ataque
```

Assuming that you understand Spanish, you will see that not all the translations are correct. This is due to low accuracy and high validation loss as we do not have enough data instances. Remember, while creating datasets, we did not use the full vocabulary. Fine-tuning the model by increasing the vocabulary, playing around with embedding layer dimensions, changing the latent dimensions of the Encoder/Decoder, experimenting with different numbers of hidden layers, and having deeper models can certainly improve the translation quality. What you learned so far in language model development for language translation remains the same. It is only your experimentation in fine-tuning the model and having huge resources for training that would create a model for quality translation from one language to any other language in the world, provided, of course, that you have the translation datasets available for those languages.

Full Source

The full source is given in Listing 8-4 for your ready reference.

Listing 8-4. Full source of the NMT project

```
import tensorflow as tf

from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input,
            Dense, LSTM, Embedding, Bidirectional,
            RepeatVector, Concatenate, Activation,
            Dot, Lambda
from tensorflow.keras.preprocessing.text
            import Tokenizer
from tensorflow.keras.preprocessing.sequence
            import pad_sequences
from keras import preprocessing, utils
import numpy as np
import matplotlib.pyplot as plt

!pip install wget
import wget
url = 'https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch08/spa.txt'
wget.download(url,'spa.txt')

# reading data
with open('/content/spa.txt',encoding='utf-8',errors='ignore')
as file:
  text=file.read().split('\n')

len(text)
```

```
# input and target data is separated by tab '\t'
text[:5]

for t in text[100:110]:
  print(t)

input_texts=[] #encoder input
target_texts=[] # decoder input

# we will select subset of the whole data
NUM_SAMPLES = 10000
for line in text[:NUM_SAMPLES]:
  english, spanish  = line.split('\t')[:2]
  target_text = spanish.lower()
  input_texts.append(english.lower())
  target_texts.append(target_text)

print(input_texts[:5],target_texts[:5])

import string
print('Characters to be removed in preprocessing', string.
punctuation)

#removing punctuation from target and input
def remove_punctuation(s):
  out=s.translate(str.maketrans("","",
                    string.punctuation))
  return out

input_texts = [remove_punctuation(s)
                    for s in input_texts]
target_texts = [remove_punctuation(s)
                    for s in target_texts]
```

```
input_texts[:5],target_texts[:5]

# adding start and end tags
target_texts=['<start> ' + s + ' <end>'
        for s in target_texts]

target_texts[1]

tokenizer_in=Tokenizer()
#tokenizing the input texts
tokenizer_in.fit_on_texts(input_texts)
#vocab size of input
input_vocab_size=len(tokenizer_in.word_index) + 1

input_vocab_size

# Listing few items
input_tokens = tokenizer_in.index_word
for k,v in sorted(input_tokens.items())[2000:2010]:
    print (k,v)

#tokenizing output that is spanish translation
tokenizer_out=Tokenizer(filters='')
tokenizer_out.fit_on_texts(target_texts)
#vocab size of output
output_vocab_size=len(tokenizer_out.word_index) + 1
output_vocab_size

# Listing few items
output_tokens = tokenizer_out.index_word
for k,v in sorted(output_tokens.items())[2000:2010]:
    print (k,v)

#converting tokenized sentence into sequences
tokenized_input = tokenizer_in.texts_to_sequences
                    ( input_texts )
```

```
#max length of the input
maxlen_input = max( [ len(x)
                     for x in tokenized_input ] )
#padding sequence to the maximum length
padded_input = preprocessing.sequence.pad_sequences
        ( tokenized_input , maxlen=maxlen_input ,
                padding='post' )

padded_input[2000:2010]

encoder_input_data = np.array( padded_input )
print( encoder_input_data.shape)

#converting tokenized text into sequences
tokenized_output = tokenizer_out.texts_to_sequences
                        (target_texts)

output_vocab_size=len(tokenizer_out.word_index) + 1
output_vocab_size

# teacher forcing
for i in range(len(tokenized_output)) :
    tokenized_output[i] = tokenized_output[i][1:]

# padding
maxlen_output = max( [ len(x)
                        for x in tokenized_output ] )
padded_output = preprocessing.sequence.pad_sequences
                        ( tokenized_output ,
                            maxlen=maxlen_output ,
                            padding='post' )

# converting to numpy
decoder_input_data = np.array( padded_output )
decoder_input_data[2000:2010]
```

```
#decoder target output
decoder_target_one_hot=np.zeros((len(input_texts),
                                 maxlen_output,
                                 output_vocab_size),
                                 dtype='float32')
for i,d in enumerate(padded_output):
    for t,word in enumerate(d):
        decoder_target_one_hot[i,t,word]=1

decoder_target_one_hot[0]

from google.colab import drive
drive.mount('/content/drive', force_remount=True)

#creating dictionary of words corresponding to vectors
print('Indexing word vectors.')

embeddings_index = {}

# Use this open command in case of downloading using wget above
#f = open('glove.6B.200d.txt', encoding='utf-8')

# we can choose any dimensions 50 100 200 300
f = open('drive/My Drive/tfbookdata/glove.6B.200d.txt',
                encoding='utf-8')

for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()
```

```
print('Found %s word vectors.' %
            len(embeddings_index))

embeddings_index["any"]

#embedding matrix
num_words = len(tokenizer_in.word_index)+1
word2idx_input = tokenizer_in.word_index
embedding_matrix = np.zeros((num_words, 200))
for word,i in word2idx_input.items():
    if i<num_words:
        embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
embedding_matrix.shape

#lstm hidden dimensions
LATENT_DIM=256
#embeding layer dimensions
EMBEDDING_DIM=200

embedding_layer=Embedding(input_vocab_size,
                          EMBEDDING_DIM,
                          weights=[embedding_matrix],
                          input_length=maxlen_input)

#encoder
encoder_inputs = Input(shape=(maxlen_input,),
                       name = 'encoderinput')
encoder_input = embedding_layer(encoder_inputs)
encoder = Bidirectional(LSTM(LATENT_DIM,
                             return_sequences=True,
                             dropout = 0.3),
                        name = 'encoder_bidirection')
```

```
encoder_outputs = encoder(encoder_input)

encoder_outputs.shape

decoder_inputs = Input(shape=(maxlen_output,),
                       name='decoder_input')
decoder_embedding = Embedding(output_vocab_size,
                              EMBEDDING_DIM,
                              name='decoder_embedding')
decoder_input=decoder_embedding(decoder_inputs)

decoder_input.shape

#decoder lstm
decoder_lstm = LSTM(LATENT_DIM,
                    return_state = True,
                    name = 'decoder_lstm')
#decoder dense with softmax for predicting each word
decoder_dense = Dense(output_vocab_size,
                      activation='softmax',
                      name='decoder_dense')

# Computing alphas
import tensorflow.keras.backend as k
def softmax_attention(x):
    assert(k.ndim(x)>2)

    e=k.exp(x-k.max(x,axis=1,keepdims=True))
    s=k.sum(e,axis=1,keepdims=True)
    return e/s

# nerual network layers for our repeated use
attention_repeat =  RepeatVector(maxlen_input)
attention_concat = Concatenate(axis=-1)
dense1_layer = Dense(10,activation='tanh')
```

```
dense2_layer = Dense(1,activation = softmax_attention)
dot_layer = Dot(axes=1)

# computing attention context
def context_attention(h, st_1):
    st_1=attention_repeat(st_1)
    x=attention_concat([h,st_1])
    x=dense1_layer(x)
    alphas=dense2_layer(x)
    context = dot_layer([alphas,h])
    return context

#initial states to be fed
initial_s = Input(shape=(LATENT_DIM,), name='s0')
initial_c = Input(shape=(LATENT_DIM,), name='c0')
context_last_word_concat_layer = Concatenate(axis=2)

s = initial_s
c = initial_c
outputs = []

#collect output in a list at first
for t in range(maxlen_output): #ty times
    #get the context using attention mechanism
    context=context_attention(encoder_outputs,s)

    #we need a different layer for each time step
    selector=Lambda(lambda x: x[:,t:t+1])
    x_t=selector(decoder_input)

    #combine
    decoder_lstm_input=context_last_word_concat_layer
                            ([context,x_t])
```

```
    #pass the combined [context,last word] into lstm
    #along with [s,c]
    #get the new[s,c] and output
    out,s,c=decoder_lstm(decoder_lstm_input,
                         initial_state=[s,c])

    #final dense layer to get next word prediction
    decoder_outputs = decoder_dense(out)
    outputs.append(decoder_outputs)

def stack(x):
    x=k.stack(x)
    x=k.permute_dimensions(x,pattern=(1,0,2))
    return x

stacker=Lambda(stack)
outputs=stacker(outputs)
outputs

model=Model(inputs=[encoder_inputs,
                    decoder_inputs,
                    initial_s,
                    initial_c],
            outputs=outputs)

model.summary()

tf.keras.utils.plot_model (model)

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

initial_s_training=np.zeros((NUM_SAMPLES,LATENT_DIM))
#initial s c
```

```
initial_c_training =np.zeros(shape=(NUM_SAMPLES,LATENT_DIM))
r = model.fit([encoder_input_data,
               decoder_input_data,
               initial_s_training,
               initial_c_training],
               decoder_target_one_hot,
               batch_size=100,
               epochs=100,
               validation_split=0.3)

encoder_model = Model(encoder_inputs,
                           encoder_outputs)
encoder_model.summary()
tf.keras.utils.plot_model (encoder_model)

#input will have the length double of latent dimension due to
bidirectional
encoder_outputs_as_input = Input(shape=
                    (maxlen_input, LATENT_DIM* 2,))
#we are going to predict one word at a time with input of one
word
decoder_input_embedding = Input(shape=(1,))
decoder_input_ = decoder_embedding
                    (decoder_input_embedding)

#calculating context
context = context_attention
            (encoder_outputs_as_input, initial_s)

decoder_lstm_input = context_last_word_concat_layer(
    [context, decoder_input_])
```

```
out, s, c = decoder_lstm(decoder_lstm_input,
                         initial_state=[initial_s,initial_c])
decoder_outputs = decoder_dense(out)

decoder_model = Model(
  inputs=[
    decoder_input_embedding,
    encoder_outputs_as_input,
    initial_s,
    initial_c
  ],
  outputs=[decoder_outputs, s, c]
)
decoder_model.summary()

#tf.keras.utils.plot_model (decoder_model,to_file='decoder_
model.jpg')
tf.keras.utils.plot_model (decoder_model)

word2index_input=tokenizer_in.word_index
word2index_output=tokenizer_out.word_index
#reverse mapping integer to words for english
idx2word_eng = {v:k for k, v in
                    word2index_input.items()}
#reverse mapping integer to words for spanish
idx2word_trans = {v:k for k, v in
                    word2index_output.items()}

def decode_sequence(input_seq):
  # Encode the input as state vectors.
  enc_out = encoder_model.predict(input_seq)
```

```
# Generate empty target sequence of length 1.
target_seq = np.zeros((1, 1))

# Populate the first character of target sequence with the
  start character.
# NOTE: tokenizer lower-cases all words
target_seq[0, 0] = word2index_output['<start>']

# if we get this we break
End_statement = word2index_output['<end>']

# [s, c] will be updated in each loop iteration
s = np.zeros((1,LATENT_DIM))
c = np.zeros((1,LATENT_DIM))

# Create the translation
output_sentence = []
for _ in range(maxlen_output):
  out, s, c = decoder_model.predict([target_seq,
                                enc_out, s, c])

  # Get next word
  index = np.argmax(out.flatten())

  # End sentence
  if End_statement == index:
    break
  word = ''
  if index > 0:
    word = idx2word_trans[index]
    output_sentence.append(word)
```

```
    # Update the decoder input
    # which is just the word just generated
    target_seq[0, 0] = index

  return ' '.join(output_sentence)


count=0
while (count<5):

  input_text=[str(input('input the sentence : '))]
  seq=tokenizer_in.texts_to_sequences(input_text)

  input_seq=pad_sequences(seq,maxlen=maxlen_input,
                                padding='post')
  translation=decode_sequence(input_seq)

  print('Predicted translation:', translation)
  count+=1
```

Summary

In this chapter, you studied a complex deep learning application for neural machine translation, also known as NMT. The NMT is used for translating the text from one language to another. You implemented an NMT model for English to Spanish translation. To improve the quality of translation, you also used the word to vectors dictionary. The NMT model is essentially a seq2seq (sequence-to-sequence) model, with an Encoder and a Decoder. The simple Encoder/Decoder model fails to provide a quality translation on large sentences. Thus, we added the Attention network to this model to improve the quality of translation.

In the next chapter, you will study another type of language model for language understanding.

CHAPTER 9

Natural Language Understanding

Introduction

In the last chapter, you used the seq2seq model along with the Attention to perform a language translation. In this chapter, I will show you a more sophisticated technique for Natural Language Processing. You will learn to use the latest innovation in natural language modeling called Transformer. The Transformer model eliminates the need for LSTMs and produces far better results than the seq2seq model that uses LSTMs. So, let us understand what a Transformer model is.

Transformer

In the last chapter, you saw the importance of Attention while doing language translation. The Attention module provided the weightage to the words that are important to the semantics of the sentence. This was then given as an input to the decoder along with the regular translation. Knowing what parts of the sentence are important, the decoder was able to provide a better translation.

The schematic of the Transformer model is shown in Figure 9-1.

P. Sarang, *Artificial Neural Networks with TensorFlow 2*,
https://doi.org/10.1007/978-1-4842-6150-7_9

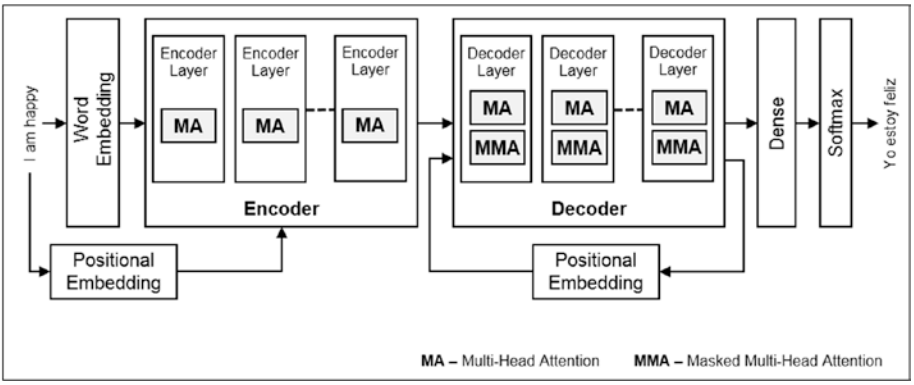


Figure 9-1. *General architecture of Transformer*

As such, the Transformer has an Encoder and a Decoder like the one you saw in the previous chapter. The major difference here is that both Encoders and Decoders do not contain the LSTM layers like in the previous case. Rather, both of them contain multiple layers of specially designed Keras layers called the Encoder layer and Decoder layer. The number of this Encoder Layer and Decoder Layer is user configurable. An Encoder Layer contains a special Attention block called the Multi-Head Attention – marked as MA in the diagram. The Decoder too contains this kind of MA block. Additionally, the decoder contains another Attention block called the Masked Multi-Head Attention – marked as MMA in the diagram. The Encoding/Decoding does not happen sequentially as in the earlier case. Rather, the entire sentence is split into words which are processed in parallel. They are divided into heads, and probably thus the name comes MultiHead. This division also facilitates the distributed training and inference. Thus, the Transformer model outperforms the Encoder/Decoder model with the Attention discussed in Chapter 14.

There is another thing that you may have noticed in the diagram, and that is the use of additional Embedding blocks. Normally, we have Word Embeddings. In Transformer, we have an additional Embedding called Positional Embeddings on a given sentence. The Positional Embedding

specifies the relative position of words in a given sentence. This eliminates the need for a long-term memory which was the feature of Encoder/Decoder models using LSTMs. How the entire Transformer works is in a way complex to digest. For developers, when they look at the runnable code, they get a better understanding of what is happening behind the scenes. And that is the approach I am going to take in this chapter. I will present you the working code of a Transformer, and taking a top-down approach, I will explain to you each and every block relevant to the context.

So, let us get started with the project.

Transformer

Create a new Colab project and rename it to NLP-transformer (NLP stands for Natural Language Processing.) As usual, import the required libraries.

```
import tensorflow as tf

from tensorflow.keras.models import Model
from tensorflow.keras.layers
            import Input,Dense,LSTM,Embedding,
                Bidirectional,RepeatVector,
                Concatenate,Activation,Dot,Lambda
from tensorflow.keras.preprocessing.text
            import Tokenizer
from tensorflow.keras.preprocessing.sequence
            import pad_sequences
from keras import preprocessing,utils
import numpy as np
import matplotlib.pyplot as plt
import tensorflow_datasets as tfds
import os
import re
```

```
import numpy as np
import string
```

Downloading Data

Download the data from the book's site.

```
!pip install wget
import wget
url = 'https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch08/spa.txt'
wget.download(url,'spa.txt')
```

This is the same database that you used in the previous chapter (Chapter 8 on NMT). So, I am not going to describe its structure here.

Creating Datasets

The code for creating datasets is the same as the one you developed in the previous chapter (Chapter 8). I am just going to give you the code here for your quick reference.

```
# reading data
with open('/content/spa.txt',encoding='utf-8',
                errors='ignore') as file:
  text=file.read().split('\n')

input_texts=[] #encoder input
target_texts=[] # decoder input

# we will select subset of the whole data
NUM_SAMPLES = 10000
for line in text[:NUM_SAMPLES]:
  english, spanish  = line.split('\t')[:2]
```

```
  target_text = spanish.lower()
  input_texts.append(english.lower())
  target_texts.append(target_text)
```

As in the earlier case, we are going to use only a subset of the entire corpus. We will use only 10,000 samples from the full corpus.

Data Preprocessing

We remove punctuation characters from the data using the following code:

```
regex = re.compile('[%s]' %re.escape(string.punctuation))
for s in input_texts:
  regex.sub('', s)
for s in target_texts:
  regex.sub('', s)
```

Now, you are ready for tokenizing and padding data.

Tokenizing Data

In this application, we will use the SubWordTextEncoder class to create tokenizers on our data corpus. A tokenizer maps the text to an id. If the specified text is not in the dictionary, it breaks it into subwords.

First, we will build a tokenizer for our input dataset, which is in English.

```
tokenizer_input = tfds.features.text.
                SubwordTextEncoder.build_from_corpus(
    input_texts, target_vocab_size=2**13)
```

The value specified in the second parameter target_vocab_size is an arbitrarily large value.

To test the tokenizer, you can try by inputting a string to it. For example, in the following code segment, we encode the string:

```
# example showing how this tokenizer works
tokenized_string1=tokenizer_input.encode
                    ('hello i am good')
tokenized_string1
```

The output would be as follows:

```
[2269, 1, 41, 89]
```

You can print the same in a tabular format to get a better view of the mapping.

```
for token in tokenized_string1:
  print ('{} ----> {}'.format(token,
                    tokenizer_input.decode([token])))
```

The output is

```
2269 ----> hello
1 ----> i
41 ----> am
89 ----> good
```

As you can see, each word is in the dictionary and is mapped to a unique id. Now, let us try to input something which may not be in the dictionary.

```
# if the word is not in dictionary
tokenized_string2=tokenizer_input.encode
                        ('how is the moon')
```

```
for token in tokenized_string2:
  print ('{} ----> {}'.format(token,
                    tokenizer_input.decode([token])))
```

The output is

```
64 ----> how
4 ----> is
21 ----> the
2827 ----> m
2829 ----> o
75 ----> on
```

Look at how the unknown word *moon* is split into known words.

Likewise, create a mapping for Spanish text, which is our target language for translation.

```
tokenizer_out=tfds.features.text.SubwordTextEncoder.
                                build_from_corpus(
    target_texts, target_vocab_size=2**13)
```

Now, add tokens for start and end tags for both the input and output.

```
START_TOKEN_in=[tokenizer_input.vocab_size]
#input start token
END_TOKEN_in=[tokenizer_input.vocab_size+1]
#input end token
START_TOKEN_out=[tokenizer_out.vocab_size]
#output start token
END_TOKEN_out=[tokenizer_out.vocab_size+1]
#output end token/
```

You can check out their IDs using the following statement:

```
START_TOKEN_in,
        END_TOKEN_in,START_TOKEN_out,END_TOKEN_out
```

The output is

```
([2974], [2975], [5737], [5738])
```

Now, we will write a function for tokenizing and padding out both the datasets. We will take the maximum length for a token to be 10.

```
MAX_LENGTH = 10

# Tokenize, filter and pad sentences
def tokenize_and_padding(inputs, outputs):
  tokenized_inputs, tokenized_outputs = [], []

  for (input_sentence, output_sentence)
                        in zip(inputs, outputs):
    # tokenize sentence
    input_sentence = START_TOKEN_in +
            tokenizer_input.encode(input_sentence) +
                        END_TOKEN_in
    output_sentence = START_TOKEN_out +
             tokenizer_out.encode(output_sentence) +
                        END_TOKEN_out
    # check tokenized sentence max length
    #if len(input_sentence) <= MAX_LENGTH and
                len(output_sentence) <= MAX_LENGTH:
    tokenized_inputs.append(input_sentence)
    tokenized_outputs.append(output_sentence )
  # pad tokenized sentences
  tokenized_inputs =
      tf.keras.preprocessing.sequence.pad_sequences(
      tokenized_inputs, maxlen=MAX_LENGTH,
                                    padding='post')
  tokenized_outputs =
      tf.keras.preprocessing.sequence.pad_sequences(
```

```
      tokenized_outputs, maxlen=MAX_LENGTH,
                                    padding='post')

  return tokenized_inputs, tokenized_outputs

english, spanish = tokenize_and_padding
                        (input_texts,target_texts)
```

You can check the result of this function by printing one of the elements.

```
english[1],spanish[1]
```

The output is

```
(array([2974,   50, 2764, 2975,   0,   0,   0,   0,    0,   0],
       dtype=int32),
 array([5737, 1017, 5527, 5738,   0,   0,   0,   0,   0,    0],
       dtype=int32))
```

See how each sentence is padded up to ten elements. Also, note that the English sentence at index 1 is "go.", which maps to 50 for the word *go* and 2764 for the dot. The 2974 is our start tag and 2975 is our end tag.

Preparing Dataset for Training

We will now prepare the dataset for training. To create the input pipeline, we will use tf.data.Dataset and use features such as cache and prefetch to speed up the process. The teacher forcing is used during training. The following code prepares the dataset:

```
BATCH_SIZE = 32
BUFFER_SIZE = 10000

# decoder inputs use the previous target as input
# remove START_TOKEN from targets
dataset = tf.data.Dataset.from_tensor_slices((
```

```
    {
        'inputs': english,
        'decoder_inputs': spanish[:, :-1]
    },
    {
        'outputs':spanish[:, 1:]
    },
))

dataset = dataset.cache()
dataset = dataset.shuffle(BUFFER_SIZE)
dataset = dataset.batch(BATCH_SIZE)
dataset = dataset.prefetch
                (tf.data.experimental.AUTOTUNE)
```

Prefetching the dataset creates an overlap between the preprocessing and the model execution. While the model is executing a training step n, the input pipeline reads the data for n+1 step. This results in an overall faster training.

Transformer Model

While explaining the implementation of the Transformer model, I am going to take a top-down approach. The model consists of several components, and obviously you cannot construct the model until the components are created. However, for a better understanding of how the model is built, I will first give you an overview of the major components of the model and then go into the finer details of each component and its implementation. The project code is mostly based on the Transformer Chatbot implementation published by TensorFlow authors and licensed under the Apache License, version 2.0. Needless to say, the code is well written and structured and leaves you no scope for any improvements. So, I have picked up their few implementations as is.

So, now let us look at the overall view of the Transformer architecture, which is depicted in Figure 9-2.

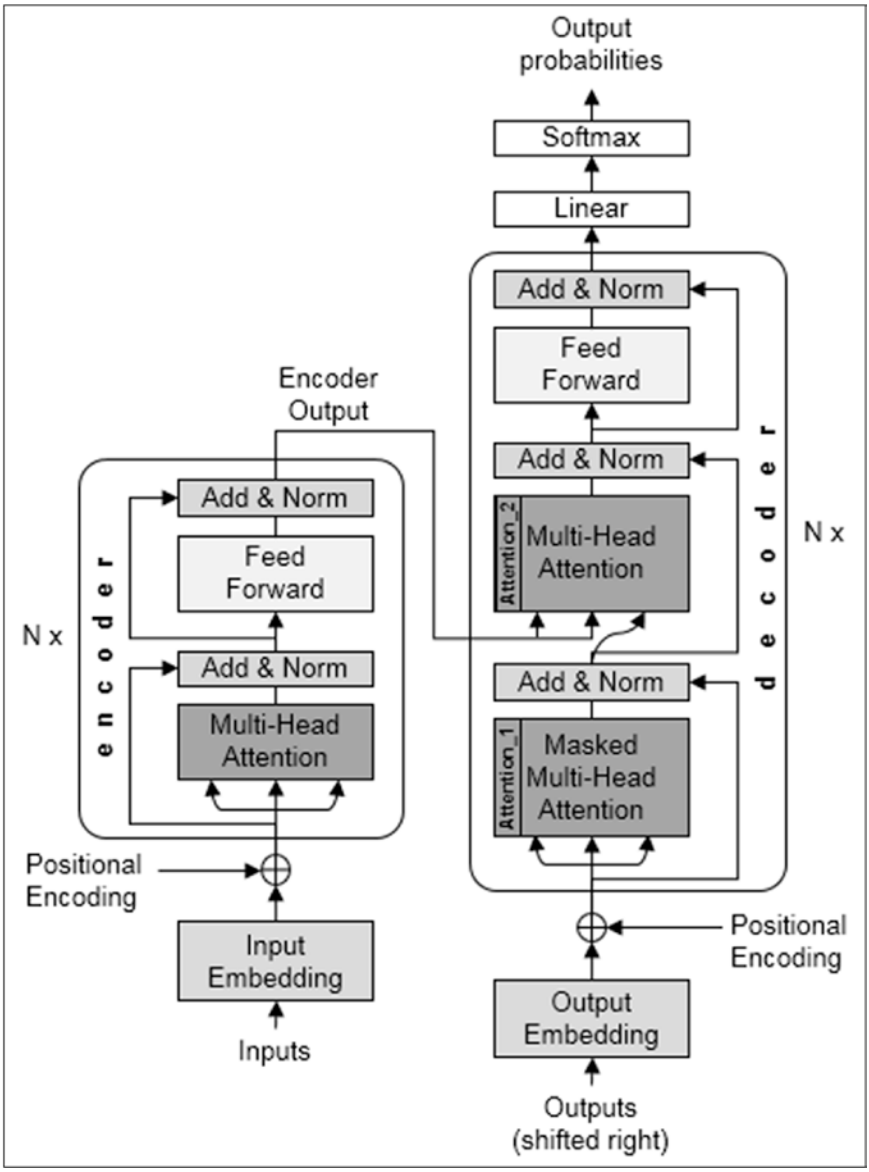


Figure 9-2. *Detailed architecture of Transformer*

As seen, the model has an Encoder, a Decoder, and a Multi-Head Attention module. The heart of the entire architecture is this Multi-Head Attention. I will first discuss its construction and then proceed to explain to you the construction of the Encoder followed by that of the Decoder. Finally, I will show you how to train and use this Transformer model for inference, and that is to convert English to Spanish with a far better Natural Language Understanding than the previous seq2seq models of language translation.

Multi-Head Attention

The Multi-Head Attention is the heart of this project. It is used in several places in our Transformer architecture. It is to be included as a network layer in our definition of the Transformer network. So, we will write a class for it, which is derived from the Keras Layer (tf.keras.layers.Layer) class. The instance of this class that is a Keras Layer will be included in our Encoder and Decoder networks at appropriate places. The schematic components of a Multi-Head Attention are shown in Figure 9-3.

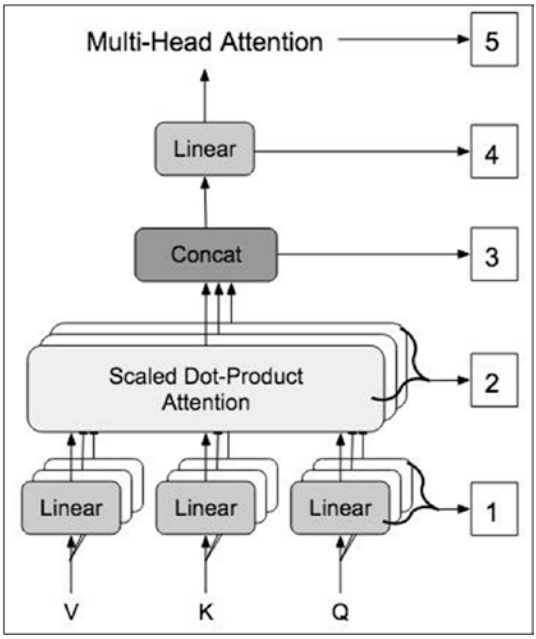


Figure 9-3. *Multi-Head Attention architecture*

One prominent feature that you would notice in this schematic is the use of multiple heads for Q (query), K (key), and V (value) vectors. The three vectors are indeed split into multiple heads and are put through linear (Dense) layers. This allows the model to concurrently process many words in an input sentence and also thereby facilitate the distributed training.

The scaled dot-product attention is applied to each head. Depending on whether this Multi-Head Attention is used in the Encoder or Decoder, we apply an appropriate mask in this attention step. The two types of masks are explained later. The attention outputs for three heads are concatenated by using tf.transpose and tf.reshape and passed through a Dense layer to produce a final attention vector.

Though the schematic may look scary to implement, it is not too complex. The architecture shows that there are five major components (layers), which are numbered 1 to 5. I will give you the full implementation

of the Multi-Head Attention class and then explain to you where those layers 1 to 5 are included in the class definition. Here is the class definition as given in Listing 9-1.

Listing 9-1. MultiHeadAttention class definition

```
class MultiHeadAttention(tf.keras.layers.Layer):

  def __init__(self, d_model, num_heads,
                 name="multi_head_attention"):
    super(MultiHeadAttention, self)
                .__init__(name=name)
    self.num_heads = num_heads
    self.d_model = d_model
    self.depth = d_model // self.num_heads

    self.query_dense =
                tf.keras.layers.Dense(units=d_model)
    self.key_dense =
                tf.keras.layers.Dense(units=d_model)
    self.value_dense =
                tf.keras.layers.Dense(units=d_model)

    self.dense = tf.keras.layers.Dense(units=d_model)

  def split_heads(self, inputs, batch_size):
    inputs = tf.reshape(
        inputs, shape=(batch_size, -1,
                self.num_heads, self.depth))
    return tf.transpose(inputs, perm=[0, 2, 1, 3])

  def call(self, inputs):
    query, key, value, mask = inputs['query'], inputs['key'],
                              inputs['value'], inputs['mask']
```

```
    batch_size = tf.shape(query)[0]

    # linear layers
    query = self.query_dense(query)
    key = self.key_dense(key)
    value = self.value_dense(value)

    # split heads
    query = self.split_heads(query, batch_size)
    key = self.split_heads(key, batch_size)
    value = self.split_heads(value, batch_size)
    # scaled dot-product attention
    scaled_attention = scaled_dot_product_attention
                          (query, key, value, mask)
    scaled_attention = tf.transpose
                          (scaled_attention,
                          perm=[0, 2, 1, 3])

    # concatenation of heads
    concat_attention = tf.reshape(scaled_attention,
                                  (batch_size, -1,
                                   self.d_model))

    # final linear layer
    outputs = self.dense(concat_attention)

    return outputs
```

The MultiHeadAttention class is derived from the Keras Layer class:

```
class MultiHeadAttention(tf.keras.layers.Layer):
```

In the initialization, the class constructor, few parameters are passed which are stored in the class variables for later use.

```
  def __init__(self, d_model, num_heads,
                 name="multi_head_attention"):
    super(MultiHeadAttention, self)
                .__init__(name=name)
    self.num_heads = num_heads
    self.d_model = d_model
    self.depth = d_model // self.num_heads
```

The input to the Attention network is through a Dense linear layer. Note that the network takes three inputs: V, K, and Q (refer to Figure 9-3). Q is a matrix that contains the query, which is a vector representation of one word in a sequence. K (key) is a vector representation of all words in a sequence. V (value) is also a vector representation of all words in a sequence.

These three input layers are defined as follows:

```
self.query_dense = tf.keras.layers.Dense(units=d_model)
self.key_dense = tf.keras.layers.Dense(units=d_model)
self.value_dense = tf.keras.layers.Dense(units=d_model)
```

We also declare one more Dense layer which will be used in the output of our Attention model.

```
self.dense = tf.keras.layers.Dense(units=d_model)
```

This completes our class constructor definition.

Now, let us define a method for splitting the input data into multiple heads, which allows you to do concurrent processing of different words in the input sentence and also facilitates the distributed training.
The function definition is shown as follows that just reshapes the data, transposes a few columns, and returns the processed input to the caller.

```
def split_heads(self, inputs, batch_size):
  inputs = tf.reshape(
      inputs, shape=(batch_size, -1,
              self.num_heads, self.depth))
  return tf.transpose(inputs, perm=[0, 2, 1, 3])
```

Next comes the important function call, which constructs the entire network. We first separate out the four inputs as follows:

```
def call(self, inputs):
  query, key, value, mask = inputs['query'], inputs['key'],
                            inputs['value'], inputs['mask']
  batch_size = tf.shape(query)[0]
```

Next, we create the linear layers for the three inputs as follows. This is part 1 shown in Figure 9-3.

```
# linear layers
query = self.query_dense(query)
key = self.key_dense(key)
value = self.value_dense(value)
```

We split the inputs using our split_heads function. This is part 2 shown in Figure 9-3.

```
# split heads
query = self.split_heads(query, batch_size)
key = self.split_heads(key, batch_size)
value = self.split_heads(value, batch_size)
```

Next, we compute the scaled dot-product attention, shown as part 3 in the diagram in Figure 9-3.

```
# scaled dot-product attention
scaled_attention = scaled_dot_product_attention
                      (query, key, value, mask)
scaled_attention = tf.transpose
                      (scaled_attention,
                      perm=[0, 2, 1, 3])
```

Wait a minute! We have not yet defined this function. As I said, I will take a top-down approach to explain the code. Thus, I will define and explain this function as soon as we are done defining this Multi-Head class.

We now concatenate all heads using the statement. This is part 4 in Figure 9-3.

```
# concatenation of heads
concat_attention = tf.reshape(scaled_attention,
                              (batch_size, -1,
                               self.d_model))
```

Finally, we define our dense output layer – part 5 in Figure 9-3.

```
outputs = self.dense(concat_attention)
```

The output is then returned to the caller. Now, I will define the function for computing the scaled dot-product attention.

Function for Scaled Dot-Product Attention

The scaled dot-product attention function used by the transformer takes three inputs: Q (query), K (key), V (value). The equation used to calculate the attention is

$$Attention(Q,K,V) = softmax_k\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

First, we compute QK^T.

```
QxK_transpose = tf.matmul
                    (query, key, transpose_b=True)
```

Next, we compute the expression QK^T divided by $\sqrt{d_k}$.

```
depth = tf.cast(tf.shape(key)[-1], tf.float32)
logits = QxK_transpose / tf.math.sqrt(depth)
```

If the mask is specified, we add it to zero out the padding tokens.

```
if mask is not None:
  logits += (mask * -1e9)
```

Why multiply the mask by a large negative value? This is because the large negative inputs to softmax are near zero in the output.

We apply the softmax on the resultant:

```
attention_weights = tf.nn.softmax(logits, axis=-1)
```

Finally, we compute the output by performing a matrix multiplication of attention weights and the V input vector.

```
output = tf.matmul(attention_weights, value)
```

The output is then returned to the caller.

```
return output
```

The entire function code is given in Listing 9-2.

Listing 9-2. Function for the scaled dot-product attention

```
def scaled_dot_product_attention
                      (query, key, value, mask):
  QxK_transpose = tf.matmul
                      (query, key, transpose_b=True)

  depth = tf.cast(tf.shape(key)[-1], tf.float32)
  logits = QxK_transpose / tf.math.sqrt(depth)

  if mask is not None:
    logits += (mask * -1e9)

  # softmax is normalized on the last axis (seq_len_k)
  attention_weights = tf.nn.softmax(logits, axis=-1)

  output = tf.matmul(attention_weights, value)


  return output
```

Having defined the main components of the Transformer architecture, now let us define the Encoder and Decoder. First, I will show you how an Encoder is constructed, followed by a Decoder.

Encoder Architecture

The Encoder architecture is depicted in Figure 9-4.

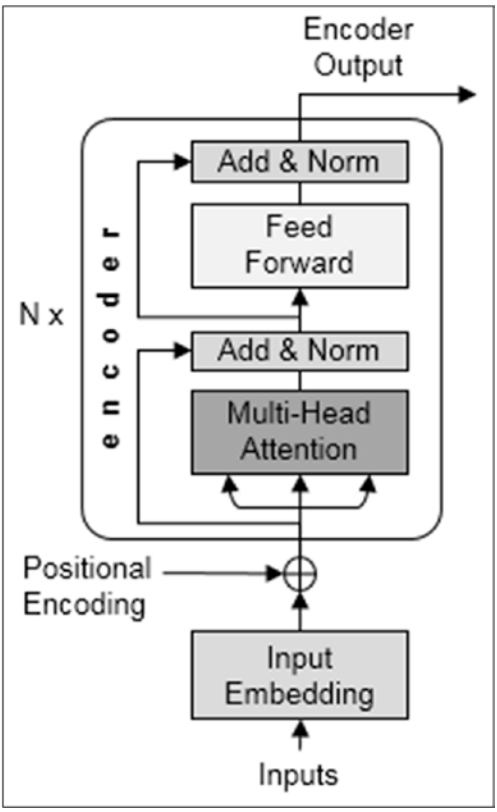


Figure 9-4. *Encoder architecture*

As you can see in Figure 9-4, the Encoder consists of a single Keras layer or a block, let us call it an Encoder Layer, that is repeated N times. In turn, the Encoder Layer itself consists of a few layers, which include our previously defined Multi-Head Attention and a feedforward network with some normalization. The Encoder receives its input as input sentence embedding and also a positional encoding that defines the relative

position of words in a sentence. Before defining the Encoder layer, I will define two helper functions for positional encoding and masking all padded tokens in a batch of sequence.

First, I will describe the masking functions.

Masking Functions

When you input a padded sequence to the network, the network may take those extra tokens (zeros) as an input. For example, our first English sentence "go." was encoded as follows:

```
(array([2974,  50, 2764, 2975,   0,   0,   0,   0,   0,   0],
       dtype=int32),
```

The sequence contains the start and end tags, which are fine for us. However, we do not want the model to treat all trailing zeros as valid input. Thus, we write a masking function to mask out all these extra paddings that we added.

```
def create_padding_mask(x):
  mask = tf.cast(tf.math.equal(x, 0), tf.float32)
  # (batch_size, 1, 1, sequence length)
  return mask[:, tf.newaxis, tf.newaxis, :]
```

You may test the function on the abovementioned valid sequence:

```
x=tf.constant([[2974,   50, 2764, 2975,    0,    0,
                0,    0,    0,    0]])
create_padding_mask(x)
```

This is the output:

```
<tf.Tensor: shape=(1, 1, 1, 10), dtype=float32,
numpy=array([[[[0., 0., 0., 0., 1., 1., 1., 1., 1., 1.]]]],
dtype=float32)>
```

You can clearly see that all the padded zeros are now masked with 1, and the rest of the characters are masked with 0.

Likewise, we write one more padding function to mask the future tokens in a sequence. The look-ahead mask is used to mask the future tokens in a sequence. In other words, the mask indicates which entries should not be used. This means that to predict the third word, only the first and second words will be used. Similarly, to predict the fourth word, only the first, second, and third words will be used and so on.

We define our look-ahead mask function as follows:

```
def create_look_ahead_mask(x):
  seq_len = tf.shape(x)[1]
  look_ahead_mask = 1 - tf.linalg.band_part
                (tf.ones((seq_len, seq_len)), -1, 0)
  padding_mask = create_padding_mask(x)
  return tf.maximum(look_ahead_mask, padding_mask)
```

We will use these helper functions as tf.keras.layers.Lambda layers.

Next, we will make the provision for positional encoding.

PositionalEncoding Class

Like the seq2seq model that you used in the previous chapter, the Transformer model does not have memories through the use of LSTMs. So, to give information about the relative positioning of the words in a given sentence, we feed the network with the additional input called positional encoding. The two vectors – positional encoding and embedding – are then fed together to the network. Embedding brings together the tokens (words) with similar meanings, but they do not specify (encode) the relative position of words in a sentence. Thus, adding both results in bringing words closer together based on the similarity of their meaning and their position in the sentence. This happens in the entire d-dimensional space.

The formula for calculating the positional encoding is given as follows:

$$PE_{(pos,2i)} = \sin\left(pos / 10000^{2i/d_{model}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(pos / 10000^{2i/d_{model}}\right)$$

The proof of this formula (https://arxiv.org/pdf/1706.03762.pdf) is beyond the scope of this book. The implementation of the formula is trivial and can be seen in the class definition given as follows:

```
class PositionalEncoding(tf.keras.layers.Layer):

  def __init__(self, position, d_model):
    super(PositionalEncoding, self).__init__()
    self.pos_encoding = self.positional_encoding
                            (position, d_model)

  def get_angles(self, position, i, d_model):
    angles = 1 / tf.pow(10000, (2 * (i // 2)) /
                  tf.cast(d_model, tf.float32))
    return position * angles

  def positional_encoding(self, position, d_model):
    angle_rads = self.get_angles(
        position=tf.range(position, dtype=tf.float32)
                        [:, tf.newaxis],
        i=tf.range(d_model, dtype=tf.float32)
                        [tf.newaxis, :],
        d_model=d_model)
    # apply sine to even index in the array
    sines = tf.math.sin(angle_rads[:, 0::2])
    # apply cosine to odd index in the array
    cosines = tf.math.cos(angle_rads[:, 1::2])
```

```
    pos_encoding = tf.concat([sines, cosines],
                             axis=-1)
    pos_encoding = pos_encoding[tf.newaxis, ...]
    return tf.cast(pos_encoding, tf.float32)

  def call(self, inputs):
    return inputs + self.pos_encoding
                  [:, :tf.shape(inputs)[1], :]
```

Having defined the helper functions, let us now proceed to define the Encoder Layer, which is repeated multiple times in an Encoder.

Encoder Layer

As seen in the Encoder schematic of Figure 9-4, the Encoder Layer which is repeated N number of times consists of the following:

1. Multi-Head Attention (with padding mask)
2. Two Dense layers followed by dropout

Each of these will also include a normalization that takes care of the vanishing gradient problem in deep neural networks.

This network structure is easily observed in the following definition of the encoder layer:

```
def encoder_layer(units, d_model, num_heads, dropout,
                  name="encoder_layer"):
  inputs = tf.keras.Input(shape=(None, d_model),
                  name="inputs")
  padding_mask = tf.keras.Input(shape=(1, 1, None),
                  name="padding_mask")

  # multi-head attention with padding mask
  attention = MultiHeadAttention(
      d_model, num_heads, name="attention")({
          'query': inputs,
```

```
            'key': inputs,
            'value': inputs,
            'mask': padding_mask
        })
    attention = tf.keras.layers.Dropout
                    (rate=dropout)(attention)
    attention = tf.keras.layers.LayerNormalization(
        epsilon=1e-6)(inputs + attention)

    # two dense layers followed by a dropout
    outputs = tf.keras.layers.Dense(units=units,
                      activation='relu')(attention)
    outputs = tf.keras.layers.Dense(units=d_model)
                      (outputs)
    outputs = tf.keras.layers.Dropout(rate=dropout)
                      (outputs)
    outputs = tf.keras.layers.LayerNormalization(
        epsilon=1e-6)(attention + outputs)

    return tf.keras.Model(
        inputs=[inputs, padding_mask],
               outputs=outputs, name=name)
```

With this setup, we are now ready to define the Encoder itself.

Encoder

As you have seen so far, the Encoder will consist of the following components:

1. Input embedding for a given sentence
2. Positional encoding for the relative position of words in a given sentence
3. Repeating Encoder Layer

The number of repeating Encoder Layers is your choice. We will be using two repetitions in our project. The input to the first Encoding Layer is the sum of input embedding and positional encoding. The output of the second Encoder Layer, which is the final output of the Encoder, becomes the input to the decoder.

```
def encoder(vocab_size,
            num_layers,
            units,
            d_model,
            num_heads,
            dropout,
            name="encoder"):

  inputs = tf.keras.Input(shape=(None,),
                          name="inputs")
  # create padding mask
  padding_mask = tf.keras.Input(shape=(1, 1, None),
                          name="padding_mask")

  # create combination of word embedding + positional encoding
  embeddings = tf.keras.layers.Embedding
                (vocab_size, d_model)(inputs)
  embeddings *= tf.math.sqrt(tf.cast
                (d_model, tf.float32))
  embeddings = PositionalEncoding
                (vocab_size, d_model)(embeddings)

  outputs = tf.keras.layers.Dropout(rate=dropout)
                                    (embeddings)

  # repeat the Encoder Layer two times
  for i in range(num_layers):
    outputs = encoder_layer(
```

```
        units=units,
        d_model=d_model,
        num_heads=num_heads,
        dropout=dropout,
        name="encoder_layer_{}".format(i),
    )([outputs, padding_mask])

  return tf.keras.Model(
      inputs=[inputs, padding_mask], outputs=outputs,
                                     name=name)
```

Notice that between the two embedding layers, we share the same weight matrix, except that we multiply those weights by the square root of d_model before inputting to PositionalEncoding.

To visualize the Encoder architecture, you can generate its model plot with the following code:

```
sample_encoder = encoder(
    vocab_size=8192,
    num_layers=5,
    units=512,
    d_model=128,
    num_heads=4,
    dropout=0.3,
    name="sample_encoder")

tf.keras.utils.plot_model(
   sample_encoder, to_file='encoder.png')
```

Note that I am repeating the Encoder Layer five times here just for the demonstration purpose. In the actual project, I am repeating this layer only twice. The network plot for the Encoder is shown in Figure 9-5.

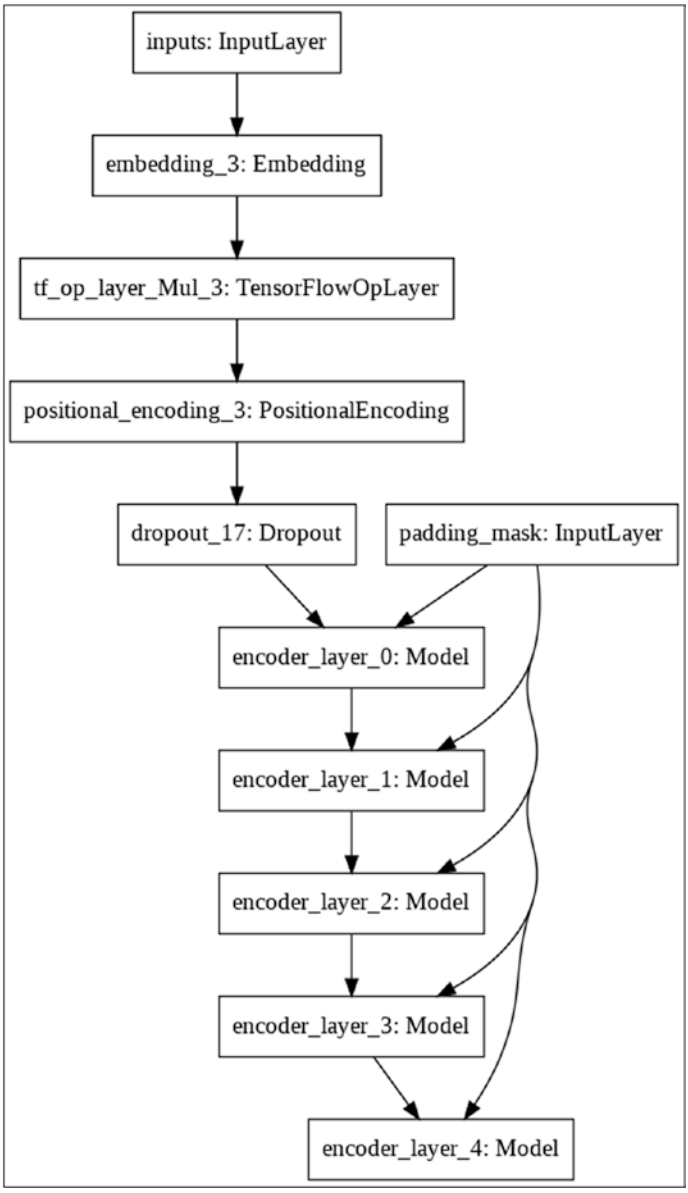


Figure 9-5. *Encoder network plot*

Note how the inputs are word embedded and then combined with positional encoding before feeding to the first encoder layer. The encoder layer itself is repeated five times with a padding mask applied to the input at each layer. The output of the encoder will become the input to the decoder. Though for clarity I have repeated the encoder layer five times, in the project code, it was repeated only two times to save on the training period.

We now proceed to define the Decoder.

Decoder Architecture

The Decoder architecture is shown in Figure 9-6.

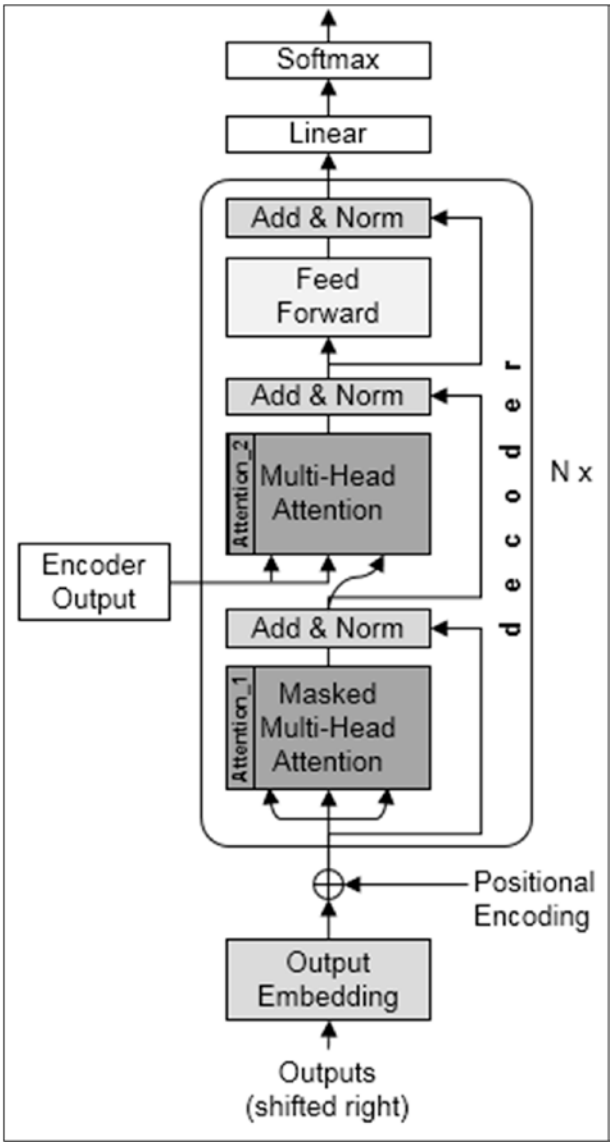


Figure 9-6. *Decoder architecture*

The input to the Decoder is the output of the Encoder. As in the case of the Encoder where we use repetitive Encoder Layers, a Decoder too consists of repetitive Decoder Layers. The first layer receives the input from the Encoder, and the last layer goes through the linear Dense network to the final Softmax layer which spits out the probabilities of the target words. The other inputs to the Decoder are the combination of output word embeddings and positional encoding. I will now describe the construction of the repetitive Decoder Layer.

Decoder Layer

As seen in Figure 9-6, the Decoder Layer actually contains two Multi-Head Attentions:

1. The Multi-Head Attention (Attention_2, as marked in Figure 9-6)
2. The Masked Multi-Head Attention (Attention_1, as marked in Figure 9-6)

The first Multi-Head Attention in the preceding list, Attention_2, is the one which receives its input from the Encoder. This input is the *Value* and *Key* vectors. The other input to this Multi-Head Attention comes from the output of the Masked Multi-Head Attention sublayer. The second Multi-Head Attention in the list (Attention_1), which we prefixed Masked is the one which takes its input as the combination of the previous decoder state and the positional encoding. Both the layers use appropriate padding masks. The output of the Multi-Head Attention Attention_2 passes through two Dense layers followed by dropout. This completes the structure of a Decoder Layer. This layer is multiplied N times in the Decoder implementation.

The Decoder Layer is coded as follows:

```
def decoder_layer(units, d_model, num_heads,
         dropout, name="decoder_layer"):
  inputs = tf.keras.Input(shape=
                (None, d_model), name="inputs")
  enc_outputs = tf.keras.Input(shape=(None, d_model),
                name="encoder_outputs")
  look_ahead_mask = tf.keras.Input(
      shape=(1, None, None), name="look_ahead_mask")
  padding_mask = tf.keras.Input(shape=(1, 1, None),
                name='padding_mask')

  attention1 = MultiHeadAttention(
      d_model, num_heads, name="attention_1")(inputs={
          'query': inputs,
          'key': inputs,
          'value': inputs,
          'mask': look_ahead_mask
      })
  attention1 = tf.keras.layers.LayerNormalization(
      epsilon=1e-6)(attention1 + inputs)

  attention2 = MultiHeadAttention(
      d_model, num_heads, name="attention_2")(inputs={
          'query': attention1,
          'key': enc_outputs,
          'value': enc_outputs,
          'mask': padding_mask
      })
  attention2 = tf.keras.layers.Dropout
                        (rate=dropout)(attention2)
  attention2 = tf.keras.layers.LayerNormalization(
```

```
      epsilon=1e-6)(attention2 + attention1)

  outputs = tf.keras.layers.Dense(units=units,
                 activation='relu')(attention2)
  outputs = tf.keras.layers.Dense(units=d_model)
                (outputs)
  outputs = tf.keras.layers.Dropout(rate=dropout)
                (outputs)
  outputs = tf.keras.layers.LayerNormalization(
      epsilon=1e-6)(outputs + attention2)

  return tf.keras.Model(
      inputs=[inputs, enc_outputs, look_ahead_mask,
                          padding_mask],
      outputs=outputs,
      name=name)
```

Here Attention_1 is our Masked Multi-Head Attention defined as item 2 in the preceding list and Attention_2 is the item 1 defined in the list. The output of Attention_2 goes through two Dense blocks followed by dropout.

You can visualize the network architecture of the decoder layer by executing the following code:

```
sample_decoder_layer = decoder_layer(
    units=512,
    d_model=128,
    num_heads=4,
    dropout=0.3,
    name="sample_decoder_layer")

tf.keras.utils.plot_model(
    sample_decoder_layer,
        to_file='decoder_layer.png')
```

The output which is the architecture of the Decoder Layer is shown in Figure 9-7.

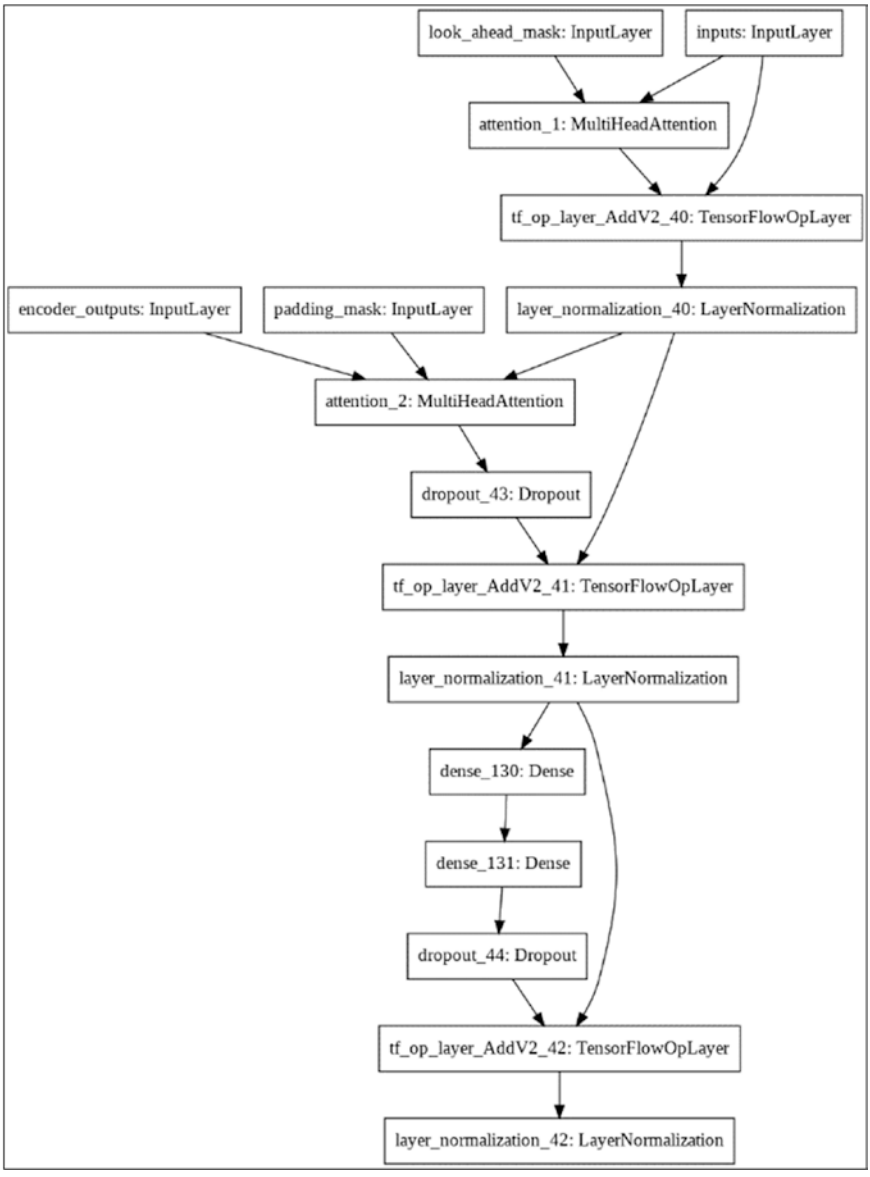


Figure 9-7. *Decoder layer network plot*

Note the position of two Multi-Head Attentions, their inputs, and outputs.

Next, I will define the Decoder block.

Decoder

As in the case of the Encoder, the Decoder network is constructed by multiplying the Decoder Layers N times. We have already seen the various inputs to the Decoder. The Decoder output goes to a linear layer and softmax classification.

The Decoder is defined as follows:

```
def decoder(vocab_size,
            num_layers,
            units,
            d_model,
            num_heads,
            dropout,
            name='decoder'):
  inputs = tf.keras.Input(shape=(None,),
                name='inputs')
  enc_outputs = tf.keras.Input(shape=(None, d_model),
                name='encoder_outputs')
  look_ahead_mask = tf.keras.Input(
      shape=(1, None, None), name='look_ahead_mask')
  padding_mask = tf.keras.Input(shape=(1, 1, None),
                name='padding_mask')

  embeddings = tf.keras.layers.Embedding
                (vocab_size, d_model)(inputs)
  embeddings *= tf.math.sqrt(tf.cast
                (d_model, tf.float32))
```

```
  embeddings = PositionalEncoding
               (vocab_size, d_model)(embeddings)

  outputs = tf.keras.layers.Dropout(rate=dropout)
               (embeddings)

  for i in range(num_layers):
    outputs = decoder_layer(
        units=units,
        d_model=d_model,
        num_heads=num_heads,
        dropout=dropout,
        name='decoder_layer_{}'.format(i),
    )(inputs=[outputs, enc_outputs, look_ahead_mask,
                padding_mask])

  return tf.keras.Model(
      inputs=[inputs, enc_outputs, look_ahead_mask,
                padding_mask],
      outputs=outputs,
      name=name)
```

A sample decoder can be constructed using the following code:

```
sample_decoder = decoder(
    vocab_size=8192,
    num_layers=2,
    units=512,
    d_model=128,
    num_heads=4,
    dropout=0.3,
    name="sample_decoder")

tf.keras.utils.plot_model(
    sample_decoder, to_file='decoder.png')
```

The plot generated by this code is given in Figure 9-8.

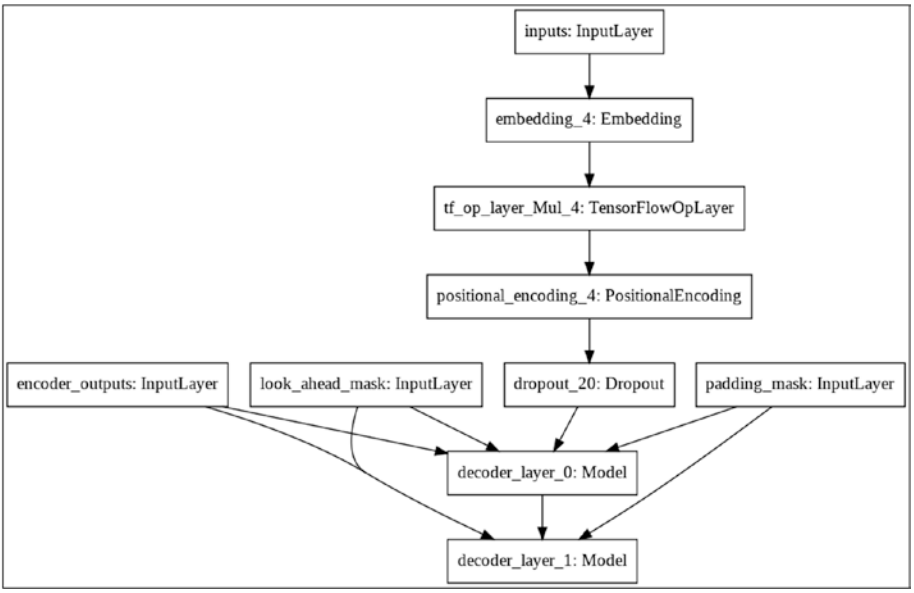


Figure 9-8. *Decoder network plot*

Finally comes our ultimate goal, and that is to define the Transformer.

Transformer Model

The Transformer consists of the encoder, the decoder, and a final linear layer. The output of the decoder is the input to the linear layer.

The Transformer is defined as follows:

```
def transformer(input_vocab_size,
                target_vocab_size,
                num_layers,
                units,
                d_model,
                num_heads,
```

```
                dropout,
                name="transformer"):
  inputs = tf.keras.Input(shape=(None,),
                name="inputs")
  dec_inputs = tf.keras.Input(shape=(None,),
                 name="decoder_inputs")
  enc_padding_mask = tf.keras.layers.Lambda(
      create_padding_mask, output_shape=(1, 1, None),
      name='enc_padding_mask')(inputs)
  # mask the future tokens for decoder inputs at the 1st
attention block
  look_ahead_mask = tf.keras.layers.Lambda(
      create_look_ahead_mask,
      output_shape=(1, None, None),
      name='look_ahead_mask')(dec_inputs)
  # mask the encoder outputs for the 2nd attention block
  dec_padding_mask = tf.keras.layers.Lambda(
      create_padding_mask, output_shape=(1, 1, None),
      name='dec_padding_mask')(inputs)

  enc_outputs = encoder(
      vocab_size=input_vocab_size,
      num_layers=num_layers,
      units=units,
      d_model=d_model,
      num_heads=num_heads,
      dropout=dropout,
  )(inputs=[inputs, enc_padding_mask])

  dec_outputs = decoder(
      vocab_size=target_vocab_size,
      num_layers=num_layers,
```

```
        units=units,
        d_model=d_model,
        num_heads=num_heads,
        dropout=dropout,
    )(inputs=[dec_inputs, enc_outputs, look_ahead_mask,
                  dec_padding_mask])

    outputs = tf.keras.layers.Dense(units=target_vocab_size,
                   name="outputs")(dec_outputs)

    return tf.keras.Model(inputs=[inputs, dec_inputs],
                   outputs=outputs, name=name)
```

You can see the network architecture by running the following code:

```
sample_transformer = transformer(
    input_vocab_size = 100,
    target_vocab_size = 100,
    num_layers=4,
    units=512,
    d_model=128,
    num_heads=4,
    dropout=0.3,
    name="sample_transformer")

tf.keras.utils.plot_model(
    sample_transformer, to_file='transformer.png')
```

The generated plot for the network architecture is shown in Figure 9-9.

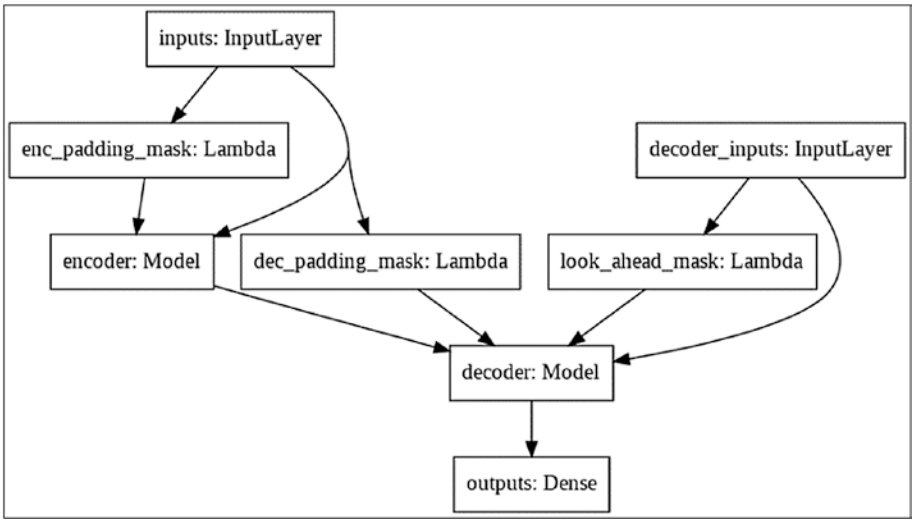


Figure 9-9. *Network plot for sample transformer*

You can see that the entire architecture is simple enough to understand. It mainly contains an encoder and a decoder. Now, as we have the Transformer defined, it is time to create a model on this definition, which can be compiled later.

Creating Model for Training

To create a model for training, we just need to instantiate a transformer class with the desired parameters.

```
D_MODEL = 256
model = transformer(
    tokenizer_input.vocab_size+2,
    tokenizer_out.vocab_size+2,
    num_layers = 2,
    units = 512,
```

```
    d_model = D_MODEL,
    num_heads = 8,
    dropout = 0.1)
```

As mentioned earlier, I have set the number of layers to two, while the referred paper (https://arxiv.org/abs/1706.03762) recommends this value to be six. Likewise, the other parameters are given low values for faster training. For more accurate results, you should experiment with larger values for num_layers, units, and d_model.

To compile this model, we would be required to define the loss function and the optimizer.

Loss Function

The loss is measured between the true and predicted values of the output y by using the sparse categorical entropy function. As all our input sequences are padded, it is important to apply the padding mask while calculating the loss. The loss function is given in the following code:

```
def loss(y_true, y_pred):
  y_true = tf.reshape(y_true, shape=(-1, 10 - 1))

  loss = tf.keras.losses.SparseCategoricalCrossentropy(
      from_logits=True, reduction='none')
                (y_true, y_pred)

  mask = tf.cast(tf.not_equal(y_true, 0), tf.float32)
  loss = tf.multiply(loss, mask)

  return tf.reduce_mean(loss)
```

Optimizer

We will use the Adam optimizer for training the model. The referred paper (https://arxiv.org/pdf/1706.03762.pdf) suggests that you use a custom learning rate for the optimizer. The custom learning rate is specified by the equation:

$$lrate = d_{model}^{-0.5} * \min\left(step\_num^{-0.5}, step\_num * warmup\_steps^{-1.5}\right)$$

As per the equation, the learning rate would initially increase linearly for the number of steps set up by the value of warmup_steps and later decrease proportionally to the inverse square root of the step number. In the paper referred in the previous paragraph, the authors have set warmup_steps to 4000, and we will continue using the same value.

To implement this formula, we create a custom scheduler class as follows:

```
class CustomSchedule
(tf.keras.optimizers.schedules.LearningRateSchedule):

  def __init__(self, d_model, warmup_steps=4000):
    super(CustomSchedule, self).__init__()
    self.d_model = d_model
    self.d_model = tf.cast(self.d_model, tf.float32)
    self.warmup_steps = warmup_steps

  def __call__(self, step):
    arg1 = tf.math.rsqrt(step)
    arg2 = step * (self.warmup_steps**-1.5)

    return tf.math.rsqrt(self.d_model) *
                tf.math.minimum(arg1, arg2)
```

Compiling

We compile our model using the selected optimizer and the loss function:

```
model.compile(optimizer=optimizer, loss=loss)
```

Training

Finally, we start training the model by calling its fit method:

```
EPOCHS = 20
model.fit(dataset, epochs=EPOCHS)
```

It took me approximately 70 seconds per epoch to train the model.

Inference

For inference, which is to translate the given English sentence to the German language, we create a function called translate. We will need to encode the input statement by using our earlier created tokenizer and adding the start and end tokens to it. Our maximum sequence length is 10 defined by the variable MAX_LENGTH. So, we create a loop for iterating through ten input words, translating them one after the other by keeping our attention on the entire input sentence. The function code for translate is given as follows:

```
def translate (input_sentence):

  input_sentence = START_TOKEN_in +
                   tokenizer_input.encode
                   (input_sentence) + END_TOKEN_in
  encoder_input = tf.expand_dims(input_sentence, 0)
```

```
  decoder_input = [tokenizer_out.vocab_size]
  output = tf.expand_dims(decoder_input, 0)

  for i in range(MAX_LENGTH):
    predictions = model(inputs=[encoder_input,
                output], training=False)

    # select the last word
    predictions = predictions[:, -1:, :]
    predicted_id = tf.cast(tf.argmax(predictions,
                   axis=-1), tf.int32)

    # terminate on END_TOKEN
    if tf.equal(predicted_id, END_TOKEN_out[0]):
      break

    # concatenated the predicted_id to the output
    output = tf.concat([output, predicted_id],
                       axis=-1)

  return tf.squeeze(output, axis=0)
```

Testing

It is now time to test our model on some real input sentences. You can set up an array of input sentences and define a loop to perform the translation and print the output on the console. The testing loop code is shown here:

```
test_sentences = ['i am sorry', 'how are you']
for s in test_sentences:
  prediction = translate(s)

  predicted_sentence = tokenizer_out.decode(
      [i for i in prediction if i <
                      qtokenizer_out.vocab_size])
```

```
  print('Input: {}'.format(s))
  print('Output: {}'.format(predicted_sentence))
```

You will see the following output:

```
Input: i am sorry
Output: lo siento.
Input: how are you
Output: cómo estás.
```

Full Source

The full source is given in Listing 9-3 for your reference.

Listing 9-3. NLP_TRANSFORMER

```
import tensorflow as tf

from tensorflow.keras.models import Model
from tensorflow.keras.layers
            import Input,Dense,LSTM,Embedding,
                Bidirectional,RepeatVector,
                Concatenate,Activation,Dot,Lambda
from tensorflow.keras.preprocessing.text
            import Tokenizer
from tensorflow.keras.preprocessing.sequence
            import pad_sequences
from keras import preprocessing,utils
import numpy as np
import matplotlib.pyplot as plt
import tensorflow_datasets as tfds
import os
import re
```

```
import numpy as np
import string

!pip install wget
import wget
url = 'https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch08/spa.txt'
wget.download(url,'spa.txt')

# reading data
with open('/content/spa.txt',encoding='utf-8',
                errors='ignore') as file:
  text=file.read().split('\n')

input_texts=[] #encoder input
target_texts=[] # decoder input

# we will select subset of the whole data
NUM_SAMPLES = 10000
for line in text[:NUM_SAMPLES]:
  english, spanish  = line.split('\t')[:2]
  target_text = spanish.lower()
  input_texts.append(english.lower())
  target_texts.append(target_text)

regex = re.compile('[%s]' %
                re.escape(string.punctuation))
for s in input_texts:
  regex.sub('', s)
for s in target_texts:
  regex.sub('', s)
```

```
input_texts[1],target_texts[1]

tokenizer_input = tfds.features.text.
                SubwordTextEncoder.build_from_corpus(
    input_texts, target_vocab_size=2**13)

# example showing how this tokenizer works
tokenized_string1=tokenizer_input.encode
                    ('hello i am good')
tokenized_string1

for token in tokenized_string1:
  print ('{} ----> {}'.format(token,
                    tokenizer_input.decode([token])))

# if the word is not in dictionary
tokenized_string2=tokenizer_input.encode
                        ('how is the moon')
for token in tokenized_string2:
  print ('{} ----> {}'.format(token,
                    tokenizer_input.decode([token])))

# tokenize Spanish text
tokenizer_out=tfds.features.text.SubwordTextEncoder.
                                build_from_corpus(
    target_texts, target_vocab_size=2**13)

START_TOKEN_in=[tokenizer_input.vocab_size]
#input start token
END_TOKEN_in=[tokenizer_input.vocab_size+1]
#input end token
START_TOKEN_out=[tokenizer_out.vocab_size]
#output start token
END_TOKEN_out=[tokenizer_out.vocab_size+1]
```

```
#output end token/

START_TOKEN_in,
        END_TOKEN_in,START_TOKEN_out,END_TOKEN_out

MAX_LENGTH = 10

# Tokenize, filter and pad sentences
def tokenize_and_padding(inputs, outputs):
  tokenized_inputs, tokenized_outputs = [], []

  for (input_sentence, output_sentence)
                        in zip(inputs, outputs):
    # tokenize sentence
    input_sentence = START_TOKEN_in +
            tokenizer_input.encode(input_sentence) +
                        END_TOKEN_in
    output_sentence = START_TOKEN_out +
             tokenizer_out.encode(output_sentence) +
                        END_TOKEN_out
    # check tokenized sentence max length
    #if len(input_sentence) <= MAX_LENGTH and
                len(output_sentence) <= MAX_LENGTH:
    tokenized_inputs.append(input_sentence)
    tokenized_outputs.append(output_sentence )

  # pad tokenized sentences
  tokenized_inputs =
      tf.keras.preprocessing.sequence.pad_sequences(
      tokenized_inputs, maxlen=MAX_LENGTH,
                                    padding='post')
  tokenized_outputs =
      tf.keras.preprocessing.sequence.pad_sequences(
      tokenized_outputs, maxlen=MAX_LENGTH,
                                    padding='post')
```

```
  return tokenized_inputs, tokenized_outputs

english, spanish = tokenize_and_padding
                        (input_texts,target_texts)

english[1],spanish[1]

BATCH_SIZE = 32
BUFFER_SIZE = 10000

# decoder inputs use the previous target as input
# remove START_TOKEN from targets
dataset = tf.data.Dataset.from_tensor_slices((
    {
        'inputs': english,
        'decoder_inputs': spanish[:, :-1]
    },
    {
        'outputs':spanish[:, 1:]
    },
))

dataset = dataset.cache()
dataset = dataset.shuffle(BUFFER_SIZE)
dataset = dataset.batch(BATCH_SIZE)
dataset = dataset.prefetch
               (tf.data.experimental.AUTOTUNE)

class MultiHeadAttention(tf.keras.layers.Layer):

  def __init__(self, d_model, num_heads,
                  name="multi_head_attention"):
    super(MultiHeadAttention, self)
                .__init__(name=name)
    self.num_heads = num_heads
```

```
    self.d_model = d_model
    self.depth = d_model // self.num_heads

    self.query_dense =
                tf.keras.layers.Dense(units=d_model)
    self.key_dense =
                tf.keras.layers.Dense(units=d_model)
    self.value_dense =
                tf.keras.layers.Dense(units=d_model)

    self.dense = tf.keras.layers.Dense(units=d_model)

  def split_heads(self, inputs, batch_size):
    inputs = tf.reshape(
        inputs, shape=(batch_size, -1,
                self.num_heads, self.depth))
    return tf.transpose(inputs, perm=[0, 2, 1, 3])

  def call(self, inputs):
    query, key, value, mask = inputs['query'], inputs['key'],
                              inputs['value'], inputs['mask']
    batch_size = tf.shape(query)[0]

    # linear layers
    query = self.query_dense(query)
    key = self.key_dense(key)
    value = self.value_dense(value)

    # split heads
    query = self.split_heads(query, batch_size)
    key = self.split_heads(key, batch_size)
    value = self.split_heads(value, batch_size)

    # scaled dot-product attention
```

```
    scaled_attention = scaled_dot_product_attention
                          (query, key, value, mask)
    scaled_attention = tf.transpose
                          (scaled_attention,
                          perm=[0, 2, 1, 3])

    # concatenation of heads
    concat_attention = tf.reshape(scaled_attention,
                                  (batch_size, -1,
                                   self.d_model))

    # final linear layer
    outputs = self.dense(concat_attention)

    return outputs

def scaled_dot_product_attention
                      (query, key, value, mask):
  QxK_transpose = tf.matmul
                      (query, key, transpose_b=True)

  depth = tf.cast(tf.shape(key)[-1], tf.float32)
  logits = QxK_transpose / tf.math.sqrt(depth)

  if mask is not None:
    logits += (mask * -1e9)

  # softmax is normalized on the last axis (seq_len_k)
  attention_weights = tf.nn.softmax(logits, axis=-1)

  output = tf.matmul(attention_weights, value)

  return output

def create_padding_mask(x):
  mask = tf.cast(tf.math.equal(x, 0), tf.float32)
```

```
  # (batch_size, 1, 1, sequence length)
  return mask[:, tf.newaxis, tf.newaxis, :]

# function testing
x=tf.constant([[2974,   50, 2764, 2975,    0,    0,
                0,    0,    0,    0]])
create_padding_mask(x)

def create_look_ahead_mask(x):
  seq_len = tf.shape(x)[1]
  look_ahead_mask = 1 - tf.linalg.band_part
               (tf.ones((seq_len, seq_len)), -1, 0)
  padding_mask = create_padding_mask(x)
  return tf.maximum(look_ahead_mask, padding_mask)

class PositionalEncoding(tf.keras.layers.Layer):

  def __init__(self, position, d_model):
    super(PositionalEncoding, self).__init__()
    self.pos_encoding = self.positional_encoding
                            (position, d_model)

  def get_angles(self, position, i, d_model):
    angles = 1 / tf.pow(10000, (2 * (i // 2)) /
                  tf.cast(d_model, tf.float32))
    return position * angles

  def positional_encoding(self, position, d_model):
    angle_rads = self.get_angles(
        position=tf.range(position, dtype=tf.float32)
                       [:, tf.newaxis],
        i=tf.range(d_model, dtype=tf.float32)
                       [tf.newaxis, :],
        d_model=d_model)
```

```
    # apply sin to even index in the array
    sines = tf.math.sin(angle_rads[:, 0::2])
    # apply cos to odd index in the array
    cosines = tf.math.cos(angle_rads[:, 1::2])

    pos_encoding = tf.concat([sines, cosines],
                             axis=-1)
    pos_encoding = pos_encoding[tf.newaxis, ...]
    return tf.cast(pos_encoding, tf.float32)

  def call(self, inputs):
    return inputs + self.pos_encoding
                  [:, :tf.shape(inputs)[1], :]

def encoder_layer(units, d_model, num_heads, dropout,
                  name="encoder_layer"):
  inputs = tf.keras.Input(shape=(None, d_model),
                  name="inputs")
  padding_mask = tf.keras.Input(shape=(1, 1, None),
                  name="padding_mask")

  # multi-head attention with padding mask
  attention = MultiHeadAttention(
      d_model, num_heads, name="attention")({
          'query': inputs,
          'key': inputs,
          'value': inputs,
          'mask': padding_mask
      })
  attention = tf.keras.layers.Dropout
                (rate=dropout)(attention)
  attention = tf.keras.layers.LayerNormalization(
      epsilon=1e-6)(inputs + attention)
```

```
  # two dense layers followed by a dropout
  outputs = tf.keras.layers.Dense(units=units,
                    activation='relu')(attention)
  outputs = tf.keras.layers.Dense(units=d_model)
                    (outputs)
  outputs = tf.keras.layers.Dropout(rate=dropout)
                    (outputs)
  outputs = tf.keras.layers.LayerNormalization(
      epsilon=1e-6)(attention + outputs)

  return tf.keras.Model(
      inputs=[inputs, padding_mask],
             outputs=outputs, name=name)

def encoder(vocab_size,
            num_layers,
            units,
            d_model,
            num_heads,
            dropout,
            name="encoder"):

  inputs = tf.keras.Input(shape=(None,),
                          name="inputs")
  # create padding mask
  padding_mask = tf.keras.Input(shape=(1, 1, None),
                          name="padding_mask")

  # create combination of word embedding + positional encoding
  embeddings = tf.keras.layers.Embedding
                (vocab_size, d_model)(inputs)
  embeddings *= tf.math.sqrt(tf.cast
                (d_model, tf.float32))
```

```
  embeddings = PositionalEncoding
                (vocab_size, d_model)(embeddings)

  outputs = tf.keras.layers.Dropout(rate=dropout)
                                    (embeddings)
  # repeat the Encoder Layer two times
  for i in range(num_layers):
    outputs = encoder_layer(
        units=units,
        d_model=d_model,
        num_heads=num_heads,
        dropout=dropout,
        name="encoder_layer_{}".format(i),
    )([outputs, padding_mask])

  return tf.keras.Model(
      inputs=[inputs, padding_mask], outputs=outputs,
                                      name=name)

sample_encoder = encoder(
    vocab_size=8192,
    num_layers=5,
    units=512,
    d_model=128,
    num_heads=4,
    dropout=0.3,
    name="sample_encoder")

tf.keras.utils.plot_model(
   sample_encoder, to_file='encoder.png')

def decoder_layer(units, d_model, num_heads,
          dropout, name="decoder_layer"):
```

```
inputs = tf.keras.Input(shape=
              (None, d_model), name="inputs")
enc_outputs = tf.keras.Input(shape=(None, d_model),
              name="encoder_outputs")
look_ahead_mask = tf.keras.Input(
    shape=(1, None, None), name="look_ahead_mask")
padding_mask = tf.keras.Input(shape=(1, 1, None),
              name='padding_mask')

attention1 = MultiHeadAttention(
    d_model, num_heads, name="attention_1")(inputs={
        'query': inputs,
        'key': inputs,
        'value': inputs,
        'mask': look_ahead_mask
    })
attention1 = tf.keras.layers.LayerNormalization(
    epsilon=1e-6)(attention1 + inputs)

attention2 = MultiHeadAttention(
    d_model, num_heads, name="attention_2")(inputs={
        'query': attention1,
        'key': enc_outputs,
        'value': enc_outputs,
        'mask': padding_mask
    })
attention2 = tf.keras.layers.Dropout
                      (rate=dropout)(attention2)
attention2 = tf.keras.layers.LayerNormalization(
    epsilon=1e-6)(attention2 + attention1)

outputs = tf.keras.layers.Dense(units=units,
               activation='relu')(attention2)
```

```
  outputs = tf.keras.layers.Dense(units=d_model)
                (outputs)
  outputs = tf.keras.layers.Dropout(rate=dropout)
                (outputs)
  outputs = tf.keras.layers.LayerNormalization(
      epsilon=1e-6)(outputs + attention2)

  return tf.keras.Model(
      inputs=[inputs, enc_outputs, look_ahead_mask,
                          padding_mask],
      outputs=outputs,
      name=name)

sample_decoder_layer = decoder_layer(
    units=512,
    d_model=128,
    num_heads=4,
    dropout=0.3,
    name="sample_decoder_layer")

tf.keras.utils.plot_model(
    sample_decoder_layer,
        to_file='decoder_layer.png')

def decoder(vocab_size,
            num_layers,
            units,
            d_model,
            num_heads,
            dropout,
            name='decoder'):
  inputs = tf.keras.Input(shape=(None,),
                name='inputs')
```

```
enc_outputs = tf.keras.Input(shape=(None, d_model),
              name='encoder_outputs')
look_ahead_mask = tf.keras.Input(
    shape=(1, None, None), name='look_ahead_mask')
padding_mask = tf.keras.Input(shape=(1, 1, None),
              name='padding_mask')

embeddings = tf.keras.layers.Embedding
              (vocab_size, d_model)(inputs)
embeddings *= tf.math.sqrt(tf.cast
              (d_model, tf.float32))
embeddings = PositionalEncoding
              (vocab_size, d_model)(embeddings)

outputs = tf.keras.layers.Dropout(rate=dropout)
              (embeddings)

for i in range(num_layers):
  outputs = decoder_layer(
      units=units,
      d_model=d_model,
      num_heads=num_heads,
      dropout=dropout,
      name='decoder_layer_{}'.format(i),
  )(inputs=[outputs, enc_outputs, look_ahead_mask,
               padding_mask])

return tf.keras.Model(
    inputs=[inputs, enc_outputs, look_ahead_mask,
               padding_mask],
    outputs=outputs,
    name=name)
```

```
sample_decoder = decoder(
    vocab_size=8192,
    num_layers=2,
    units=512,
    d_model=128,
    num_heads=4,
    dropout=0.3,
    name="sample_decoder")

tf.keras.utils.plot_model(
    sample_decoder, to_file='decoder.png')

def transformer(input_vocab_size,
                target_vocab_size,
                num_layers,
                units,
                d_model,
                num_heads,
                dropout,
                name="transformer"):
  inputs = tf.keras.Input(shape=(None,),
                name="inputs")
  dec_inputs = tf.keras.Input(shape=(None,),
                 name="decoder_inputs")

  enc_padding_mask = tf.keras.layers.Lambda(
      create_padding_mask, output_shape=(1, 1, None),
      name='enc_padding_mask')(inputs)
  # mask the future tokens for decoder inputs at the 1st
attention block
  look_ahead_mask = tf.keras.layers.Lambda(
      create_look_ahead_mask,
      output_shape=(1, None, None),
```

```
        name='look_ahead_mask')(dec_inputs)
    # mask the encoder outputs for the 2nd attention block
    dec_padding_mask = tf.keras.layers.Lambda(
        create_padding_mask, output_shape=(1, 1, None),
        name='dec_padding_mask')(inputs)

    enc_outputs = encoder(
        vocab_size=input_vocab_size,
        num_layers=num_layers,
        units=units,
        d_model=d_model,
        num_heads=num_heads,
        dropout=dropout,
    )(inputs=[inputs, enc_padding_mask])

    dec_outputs = decoder(
        vocab_size=target_vocab_size,
        num_layers=num_layers,
        units=units,
        d_model=d_model,
        num_heads=num_heads,
        dropout=dropout,
    )(inputs=[dec_inputs, enc_outputs, look_ahead_mask,
                  dec_padding_mask])

    outputs = tf.keras.layers.Dense(units=target_vocab_size,
                   name="outputs")(dec_outputs)

    return tf.keras.Model(inputs=[inputs, dec_inputs],
                   outputs=outputs, name=name)

sample_transformer = transformer(
    input_vocab_size = 100,
    target_vocab_size = 100,
```

```
    num_layers=4,
    units=512,
    d_model=128,
    num_heads=4,
    dropout=0.3,
    name="sample_transformer")

tf.keras.utils.plot_model(
    sample_transformer, to_file='transformer.png')

D_MODEL = 256
model = transformer(
    tokenizer_input.vocab_size+2,
    tokenizer_out.vocab_size+2,
    num_layers = 2,
    units = 512,
    d_model = D_MODEL,
    num_heads = 8,
    dropout = 0.1)

def loss(y_true, y_pred):
  y_true = tf.reshape(y_true, shape=(-1, 10 - 1))

  loss =
      tf.keras.losses.SparseCategoricalCrossentropy(
      from_logits=True, reduction='none')
                (y_true, y_pred)

  mask = tf.cast(tf.not_equal(y_true, 0), tf.float32)
  loss = tf.multiply(loss, mask)

  return tf.reduce_mean(loss)
```

```
class CustomSchedule
(tf.keras.optimizers.schedules.LearningRateSchedule):

  def __init__(self, d_model, warmup_steps=4000):
    super(CustomSchedule, self).__init__()
    self.d_model = d_model
    self.d_model = tf.cast(self.d_model, tf.float32)
    self.warmup_steps = warmup_steps

  def __call__(self, step):
    arg1 = tf.math.rsqrt(step)
    arg2 = step * (self.warmup_steps**-1.5)

    return tf.math.rsqrt(self.d_model) *
                tf.math.minimum(arg1, arg2)

learning_rate = CustomSchedule(D_MODEL)

optimizer = tf.keras.optimizers.Adam(
    learning_rate, beta_1=0.9, beta_2=0.98,
                epsilon=1e-9)

model.compile(optimizer=optimizer, loss=loss)

EPOCHS = 20
model.fit(dataset, epochs=EPOCHS)

def translate (input_sentence):

  input_sentence = START_TOKEN_in +
                    tokenizer_input.encode
                    (input_sentence) + END_TOKEN_in
  encoder_input = tf.expand_dims(input_sentence, 0)

  decoder_input = [tokenizer_out.vocab_size]
```

```
  output = tf.expand_dims(decoder_input, 0)

  for i in range(MAX_LENGTH):
    predictions = model(inputs=[encoder_input,
                output], training=False)

    # select the last word
    predictions = predictions[:, -1:, :]
    predicted_id = tf.cast(tf.argmax(predictions,
                   axis=-1), tf.int32)

    # terminate on END_TOKEN
    if tf.equal(predicted_id, END_TOKEN_out[0]):
      break

    # concatenated the predicted_id to the output
    output = tf.concat([output, predicted_id],
                       axis=-1)

  return tf.squeeze(output, axis=0)


test_sentences = ['i am sorry', 'how are you']
for s in test_sentences:
  prediction = translate(s)

  predicted_sentence = tokenizer_out.decode(
      [i for i in prediction if i <
                       qtokenizer_out.vocab_size])

  print('Input: {}'.format(s))
  print('Output: {}'.format(predicted_sentence))
```

What's Next?

In recent days, a new language representation model called BERT has become popular and is widely used. The model was introduced by Jacob Devlin et al. in their paper titled "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" (https://arxiv.org/pdf/1810.04805.pdf) dated Oct 2018. BERT stands for Bidirectional Encoder Representations from Transformers. The beauty of this model is that you can use a pre-trained BERT model and fine-tune it with just one additional layer for creating a wide variety of applications, such as question answering, language translation, and so on. The interested reader is referred to this original paper of Devlin et al. for more information.

Summary

In this chapter, you studied another model for Natural Language Processing, which is called the Transformer. The Transformer model provides better Natural Language Understanding than the previously mentioned seq2seq model with Attention. The Transformer model does not use LSTMs for remembering long input sentences. Rather, it uses Positional Embeddings to get knowledge of the relative positions of the important words in the sentence. The Transformer uses a different Attention model called the Multi-Head Attention that takes inputs from multiple heads created by splitting the input into multiple channels. This facilitates the distributed training and inference. Overall, the Transformer works far better than the other models discussed in this book.

In the next chapter, you will learn about image captioning where you will learn to design a deep neural network that is capable of creating captions on any given image.

CHAPTER 10

Image Captioning

When you are vacationing, you capture several pictures of beautiful landscapes and places, and then you put some captions on these photos and publish them on your social network. Just imagine that you would have a mobile app doing photo captioning for you; would this not be a wonderful thing to achieve? In this chapter, you will learn how to create and train a neural network to create captions for your photos.

Caption generation is a very challenging problem. To solve this problem, you require the knowledge of both computer vision and the natural language processing. The computer vision helps you in understanding the contents of the image that is basically object detection. NLP turns this understanding into words placed in the right order.

A most generalized architecture for solving this problem is shown in Figure 10-1.

P. Sarang, *Artificial Neural Networks with TensorFlow 2*,
https://doi.org/10.1007/978-1-4842-6150-7_10

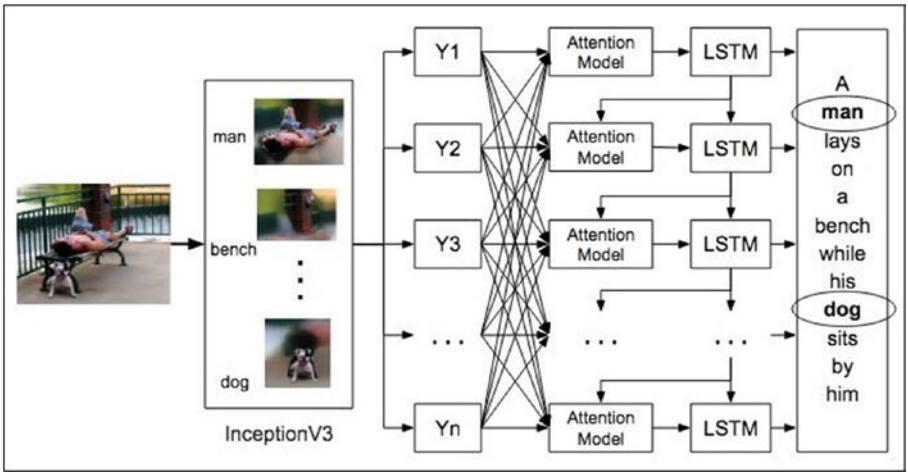


Figure 10-1. *A generalized architecture for image captioning*

We pass the image through a pre-trained network like InceptionV3 or VGG16, which is designed for image classification. However, we will use only the convolutional layers of such network to extract the features in the image and ignore the classification portion of the network. As Figure 10-1 shows, we have a picture of a man lying on the bench with a dog sitting next to it. When you pass this image through an image classification network, you will extract various parts of the image, such as the man, the dog, the bench, and so on. With these objects detected, you need to generate a caption relating all these image components, and this is the most challenging part of this image captioning application. You cannot simply list out the names of the objects. Rather, you need to create an appropriate and a meaningful sentence using these words (name of detected objects). That is where the NLP module comes into the picture, which would consist of LSTMs to generate a sentence. However, to make it more meaningful, you need an attention mechanism similar to the one you used in the previous chapter. To recall, you used a Transformer model in Chapter 9 with Multi-Head Attention to do the translation. In this chapter, for NLP, you are going to use a different Attention module and that

is called Bahdanau Attention. Dzmitry Bahdanau proposed an Additive attention (https://arxiv.org/pdf/1409.0473.pdf) that performs a linear combination of encoder and decoder states. It learns to jointly align and translate words in a given sequence. Originally, this was developed to improve the machine translation based on the Encoder-Decoder RNN (recurrent neural network).

The schematic of the Bahdanau model is shown in Figure 10-2.

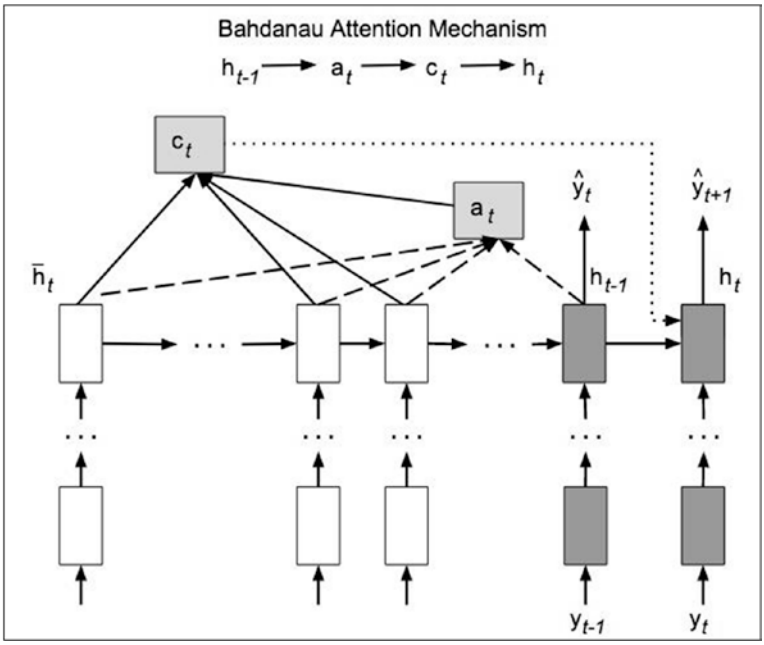


Figure 10-2. *Bahdanau Attention*

This is governed by the following equations:

$$a_t(s) = align(h_{t-1}, \underline{h}_s)$$

$$c_t = \sum a_t h_s$$

$$h_t = RNN\left(h_{t-1}^{l-1}, [c_t; h_{t-1}]\right)$$

Like in the previous chapter, I will explain to you the Bahdanau Attention module through its implementation. So, let us directly jump into developing a project for image captioning.

Project Description

Like any other ML project, what is most important in creating an image captioning network model is the availability of proper training and testing data. Fortunately, Flickr has made such data available publicly. The Flickr8k dataset contains about 8000 images, while the Flickr30k dataset contains about 30,000 images. If you wish to use a larger dataset than these, the MS COCO contains about 180,000 images. All these images are captioned appropriately. Each image may have more than one caption. Using larger datasets would give a better prediction accuracy on an unseen image; however, this would be resource intensive for training. For our learning purpose, Flickr8k data is good enough as we are not much worried about how good the generated caption represents a given image. Each image in this dataset comes with the associated five captions.

Creating Project

Open a new Colab project and rename it to ImageCaptioning. Import the required libraries.

```
import os
import time
import pickle
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
```

```
from tensorflow.keras.applications
        import  InceptionV3
from os import listdir
from tqdm import tqdm
from PIL import Image
```

Downloading Data

For this project, you need to download two types of databases – the images and their corresponding captions. As I mentioned earlier, Flickr has made this database (http://academictorrents.com/details/9dea07ba660a722ae1008c4c8afdd303b6f6e53b) available to the public. The database consists of images and captions. Download the captions data using the following code:

```
!wget --no-check-certificate -r  'https://drive.google.com/uc?
export=download&id=1c7yGTpizf5egVD9dc3Q2lrxS8wtOAV42' -O
Flickr8k_text.zip
```

Unzip the downloaded file:

```
!mkdir captions images
!unzip 'Flickr8k_text.zip' -d '/content/captions'
```

The captions folder in your drive will now contain several files. The training and testing images are maintained in two separate folders. If you open the Flickr_8k.trainImages.txt file, you will find a long list of filenames with extension .jpg. These are your image files. Open Flickr8k.token.txt file, and you will see the captions corresponding to these images. The first few lines of this file are reproduced here for your understanding.

```
1000268201_693b08cb0e.jpg#1    A girl going into a wooden
building .
1000268201_693b08cb0e.jpg#2    A little girl climbing into a
wooden playhouse .
```

```
1000268201_693b08cb0e.jpg#3    A little girl climbing the
stairs to her playhouse .
1000268201_693b08cb0e.jpg#4    A little girl in a pink dress
going into a wooden cabin .
1001773457_577c3a7d70.jpg#0    A black dog and a spotted dog
are fighting
```

Download and unzip the images database using the following code:

```
!wget --no-check-certificate -r 'https://drive.google.com/uc?
export=download&id=1126G_E2OpvULyvTmOKz_oMhOzv8CkiW1' -O
Flickr8k_Dataset.zip
!unzip 'Flickr8k_Dataset.zip' -d '/content/images'
```

The images are unzipped in the images folder of your drive. One such image is shown in Figure 10-3.

Figure 10-3. *A sample image from the dataset*

The captions corresponding to this image can be found in the captions text file (Flickr8k.token.txt) that you downloaded earlier. These are the captions for the preceding image.

```
1003163366_44323f5815.jpg#0    A man lays on a bench while his
dog sits by him .
1003163366_44323f5815.jpg#1    A man lays on the bench to which
a white dog is also tied .
1003163366_44323f5815.jpg#2    a man sleeping on a bench
outside with a white and black dog sitting next to him .
1003163366_44323f5815.jpg#3    A shirtless man lies on a park
bench with his dog .
1003163366_44323f5815.jpg#4    man laying on bench holding
leash of dog sitting on ground
```

You may check how many images are there in the database with a simple code given here:

```
image_dir = '/content/images/Flicker8k_Dataset'
images = listdir(image_dir)
print("The number of jpg flies in Flicker8k: {}"
        .format(len(images)))
```

You will see that the database contains 8091 images – good enough for our experimentation and learnings.

Parsing Token File

We will now parse the token file to create a list of image names and the corresponding captions. You have already seen the structure of the token file – Flickr8k.token.txt. It contains a filename and a tab-separated caption text. We will create two lists, one containing the image path and the other for its respective caption. To keep the training time low, I am going to use only one caption per image. It goes without saying that if you use five captions per image as given in the dataset, the model inference would be better.

First, we will write a function to load the contents of the token file into memory.

Loading Data

The following function loads the contents of the token file into memory and returns a string.

```
# load doc into memory
def load(filename):
  file = open(filename, 'r')
  text = file.read()
  file.close()
  return text
```

```
filename = '/content/captions/Flickr8k.token.txt'
doc = load(filename)
```

When you execute the preceding code, the doc variable will hold the entire file contents. This must be parsed to create two separate lists that we want.

We create an iterator (a list variable) to iterate through the files in the images folder.

```
dirs = listdir('/content/images/Flicker8k_Dataset')
```

You may examine a few entries:

```
dirs[:5]
```

The output should look like this:

```
['3583065748_7d149a865c.jpg',
 '3358621566_12bac2e9d2.jpg',
 '509778093_21236bb64d.jpg',
 '2094323311_27d58b1513.jpg',
 '3314180199_2121e80368.jpg']
```

Creating Lists

We will now write a function to create the two lists:

```
def load_small(doc):
    PATH = '/content/images/Flicker8k_Dataset/'
    img_path = []
    img_id = []
    img_cap = []
    for line in doc.split('\n'):
        tokens = line.split()
        if len(line) < 2:
            continue
        image_id , image_desc = tokens[0] ,
                                tokens[1:]
        image_id = image_id.split('.')[0]
        image_id = image_id + '.jpg'
        image_desc = ' '.join(image_desc)
        if image_id not in img_id:
            if len(img_id) <= 8000:
                img_id.append(image_id)
                image_path = PATH + image_id
                image_desc = '<start> ' + image_desc
                                        + ' <end>'
                if image_id in dirs:
                    img_path.append(image_path)
                    img_cap.append(image_desc)
            else:
                continue
    return img_path , img_cap
```

I am restricting the number of images to 8000 and reading only one caption per image. For each caption, we also add the <start> and <end> tags.

Call this function to create the two desired lists:

```
all_image_path , all_image_captions = load_small(doc)
```

Check the size of the list and a few entries in the images list:

```
print('Number of images: ', len(all_image_path))
all_image_path[:5]
```

The output is shown here:

```
Number of images:  8000
['/content/images/Flicker8k_Dataset/1000268201_693b08cb0e.jpg',
'/content/images/Flicker8k_Dataset/1001773457_577c3a7d70.jpg',
'/content/images/Flicker8k_Dataset/1002674143_1b742ab4b8.jpg',
'/content/images/Flicker8k_Dataset/1003163366_44323f5815.jpg',
'/content/images/Flicker8k_Dataset/1007129816_e794419615.jpg']
```

Also, check the contents of the captions list:

```
print('Number of captions: ',
                len(all_image_captions))
all_image_captions[:5]
```

The output is shown here:

```
Number of captions:  8000
['<start> A child in a pink dress is climbing up a set of
stairs in an entry way . <end>',
 '<start> A black dog and a spotted dog are fighting <end>',
 '<start> A little girl covered in paint sits in front of a
painted rainbow with her hands in a bowl . <end>',
```

```
 '<start> A man lays on a bench while his dog sits by him .
<end>',
 '<start> A man in an orange hat staring at something . <end>']
```

Note that the list contains 8000 captions, one caption per image.

We shuffle the training data:

```
train_captions, img_name_vector =
                shuffle(all_image_captions,
                        all_image_path,
                        random_state=1)
```

Loading InceptionV3 Model

We will use the InceptionV3 model for feature extraction from the image. We load the model using the following statement:

```
image_model = InceptionV3(include_top=False,
                          weights='imagenet')
```

The parameter value specified in weights, which is the *imagenet,* is a dataset of over 15 million labeled high-resolution images with around 22,000 categories. The InceptionV3 model was trained on 1.2 million images with another 50,000 images for validation and 100,000 images for testing. So, we take advantage of this training and use pre-trained weights.

Note that we eliminate the top layers during model extraction. The top layers are used for image classification. As we are interested in only feature extraction, we do not need those top layers. We will now create our own tf.keras model for extracting the features of an image, based on the *image_model*. We first extract the last layer of the InceptionV3 model:

```
new_input = image_model.input
hidden_layer = image_model.layers[-1].output
```

We create our model now by taking the extracted model input architecture and using the hidden_layer as the output layer. This hidden layer is the layer just before the last softmax layer.

```
image_features_extract_model = tf.keras.Model
                (new_input, hidden_layer)
```

The shape of this output layer is 8x8x2048, and this is the last convolutional layer of the InceptionV3 model. You use layers up to the last convolutional layer because you will be using attention on the extracted features from this last layer.

Preparing Dataset

The InceptionV3 model requires images of size 299x299. Also, the image must be normalized so that it contains pixels in the range of –1 to 1.

We write a function for loading and resizing the image:

```
def load_image(image_path):
    img = tf.io.read_file(image_path)
    img = tf.image.decode_jpeg(img, channels=3)
    img = tf.image.resize(img, (299, 299))
    img = tf.keras.applications.inception_v3.preprocess_input
                                    (img)
    return img, image_path
```

We create the image dataset by calling from_tensor_slices and using the preceding load_image function for preprocessing.

```
encode_train = sorted(set(img_name_vector))
image_dataset = tf.data.Dataset.from_tensor_slices
                        (encode_train)
```

```
image_dataset = image_dataset.map(load_image,
                    num_parallel_calls=tf.data.experimental.
                    AUTOTUNE)
                          .batch(16)
```

Extracting Features

For each image in the dataset, we extract the features by calling our previously created model image_features_extract_model. After reshaping the data, we save it to a physical file. We implement this by using the following for loop:

```
for img, path in tqdm(image_dataset):
  batch_features = image_features_extract_model(img)
  batch_features = tf.reshape(batch_features,
                              (batch_features.shape[0],
                               -1,
                               batch_features.shape[3]))

  for bf, p in zip(batch_features, path):
    path_of_feature = p.numpy().decode("utf-8")
    np.save(path_of_feature, bf.numpy())
```

Note that though saving features into memory would be more efficient, it would be resource intensive as each image requires 8x8x2048 floats. Note that as of this writing, the Colab memory limit is 12GB. It took me about 2 minutes to run the preceding loop on a GPU.

Creating Vocabulary

We will now create a vocabulary of all unique words.

```
tokenizer = tf.keras.preprocessing.text.Tokenizer
        (filters='!"#$%&()*+.,-/:;=?@[\]^_`{|}~ ')
tokenizer.fit_on_texts(train_captions)
max_size = len(tokenizer.word_index)
```

Creating Input Sequences

We create input sequences of the tokenized words using the following code:

```
train_seqs = tokenizer.texts_to_sequences
                      (train_captions)
```

Print a few sequences:

```
train_seqs[:5]
```

The output is

```
[[2, 1, 2339, 8, 155, 2340, 1198, 19, 2341, 1390, 24, 480, 554, 3],
 [2, 21, 1714, 7, 1199, 1715, 1, 108, 2342, 19, 5, 173, 3],
 [2, 1, 11, 4, 1, 28, 32, 506, 1, 507, 3],
 [2, 1, 101, 102, 12, 1, 26, 3],
 [2, 63, 34, 4, 1, 272, 3]]
```

As you can see, these sequences are of different lengths. For our model development, we need to have all sequences of the same length. So we do the padding on the sequences.

```
max_length = max(len(t) for t in train_seqs)

cap_vector = tf.keras.preprocessing.
                        sequence.pad_sequences
                                (train_seqs,
                                 padding='post')
```

Print the cap_vector to examine its contents:

```
cap_vector[:5]
```

The output is shown in Figure 10-4.

```
array([[   2,    1, 2339,    8,  155, 2340, 1198,   19, 2341, 1390,   24,
         480,  554,    3,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0],
       [   2,   21, 1714,    7, 1199, 1715,    1,  108, 2342,   19,    5,
         173,    3,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0],
       [   2,    1,   11,    4,    1,   28,   32,  506,    1,  507,    3,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0],
       [   2,    1,  101,  102,   12,    1,   26,    3,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0],
       [   2,   63,   34,    4,    1,  272,    3,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0]], dtype=int32)
```

Figure 10-4. *Sample padded sequences*

You can see that all tokenized words are padded and thus have equal lengths.

Creating Training Datasets

We declare a few variables for creating the datasets:

```
BATCH_SIZE = 64
BUFFER_SIZE = 1000
embedding_dim = 256
units = 512
vocab_size = max_size + 1
num_steps = len(img_name_vector) // BATCH_SIZE
```

The following function loads the previously saved feature vectors of each image into a tensor.

```
def map_func(img_name, cap):
  img_tensor = np.load(img_name.decode
                       ('utf-8')+'.npy')
  return img_tensor, cap
```

We create the dataset using the following function:

```
def create_dataset(img_name_train,caption_train):
  dataset = tf.data.Dataset.from_tensor_slices
               ((img_name_train, caption_train))

  # Use map to load the numpy files in parallel
  dataset = dataset.map(lambda item1, item2:
        tf.numpy_function(map_func, [item1, item2],
        [tf.float32, tf.int32]),num_parallel_calls=
        tf.data.experimental.AUTOTUNE)

  # Shuffle and batch
  dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
 .prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
  return dataset
```

In the preceding code, we use the map function to load the numpy files. We shuffle the data and create data batches.

```
dataset = create_dataset(img_name_vector,cap_vector)
```

Creating Model

We create the sequence-to-sequence model with Bahdanau Attention and Gated Recurrent Unit (GRU). GRUs provide a gating mechanism in RNN. It is like an LSTM that you used in Chapter 9 with a forget gate, but has lesser parameters than LSTM. As it has less training parameters, it trains faster than LSTMs, uses less memory, and executes faster. However, the drawback is that it is less accurate than LSTMs when it comes to long sequences.

The Bahdanau Attention works as follows:

1. Produce the Encoder hidden states for the given input image.
2. Compute alignment scores – the alignment scores are computed between each of the previous Encoder hidden states and the previous decoder hidden state.
3. Softmax the alignment scores.
4. Compute the context vector.
5. Decode the output.
6. Repeat steps 2 through 5 until an end token is encountered.

You will understand these steps better when you look at its implementation, which is discussed in the “Decoder Implementation” section.

Creating Encoder

The Encoder takes the extracted features as input and passes it to a fully connected layer.

The inception encoder is defined as follows:

```
class Inception_Encoder(tf.keras.Model):
    def __init__(self, embedding_dim):
        super(Inception_Encoder, self).__init__()
        # shape after fc = (batch_size, 64,
                            embedding_dim)
        self.fc = tf.keras.layers.Dense
                           (embedding_dim)

    def call(self, x):
        x = self.fc(x)
        x = tf.nn.relu(x)
        return x
```

Creating Decoder

Here comes the most important part of this application. You will now be writing a Decoder with the embedded attention mechanism. You will be using Bahdanau Attention. I will give you a brief introduction to Bahdanau Attention.

Bahdanau Attention

According to the "Show, Attend and Tell" (https://arxiv.org/pdf/1502.03044.pdf) research paper, we extract a set of feature vectors from every image, which is a vector of the N-dimensional representation of the corresponding part of an image. The Encoder passes these feature vectors through a fully connected layer.

In general, there are two types of attention mechanisms:

1. **Bahdanau Attention** – Deterministic "soft" attention
2. **Luong Attention** – Stochastic "hard" attention

We use a deterministic soft attention and that is Bahdanau Attention. This attention mechanism basically computes the attention weights and context vector for every input_vector, which is the extracted features of an image. In simple terms, the context vector is a dynamic representation of the relevant part of the image input at time t. We define an attention mechanism that computes context vectors from the input vectors, which represent features extracted at different image locations. For each location in the image, let us call it loc; the mechanism generates a positive weight, also stated as score in our code. The score can be interpreted either as the probability that location loc is the right place to focus for producing the next word, or whether we should give some relative importance to loc. The attention weights for each input vector are computed by an attention model, and for computing these attention weights, we use a multilayer perceptron layer which takes the previous decoder hidden state and current input vector hidden state as input. This hidden state is the output from the Encoder.

Decoder Functionality

The Decoder functionality can be summarized in these three simple steps.

To predict the target word, the Decoder uses the following:

1. Context vector (the weighted multiplication of attention weights and Encoder output)
2. The Decoder's output from the previous time step
3. The previous Decoder's hidden state

Decoder Initialization

We declare the Decoder class as follows:

```
class RNN_Decoder(tf.keras.Model):
```

In the class initialization, we create a GRU layer:

```
self.gru = tf.keras.layers.GRU(self.units,
                    return_sequences=True,
                    return_state=True,
                    recurrent_initializer=
                   'glorot_uniform')
```

We use batch normalization to accelerate the training process and in general to improve the model's performance. Batch normalization automatically standardizes the inputs to a layer in a deep learning neural network.

```
self.batchnormalization =
     tf.keras.layers.BatchNormalization
               (axis=-1,
                momentum=0.99,
                epsilon=0.001,
                center=True,
                scale=True,
                beta_initializer='zeros',
                gamma_initializer='ones',
                moving_mean_initializer='zeros',
               moving_variance_initializer='ones',
               beta_regularizer=None,
               gamma_regularizer=None,
               beta_constraint=None,
               gamma_constraint=None)
```

For the implementation of the attention mechanism, we declare a few linear layers:

```
self.W1 = tf.keras.layers.Dense(units)
self.W2 = tf.keras.layers.Dense(units)
self.V = tf.keras.layers.Dense(1)
```

Decoder Call Method

The Decoder needs three inputs:

1. The encoder output
2. The hidden state (initialized to 0)
3. The decoder input (which is the start token)

We declare the call method as follows:

```
def call(self, x, features, hidden):
```

where x is the decoder input, *features* represents the Encoder output, and hidden is the decoder hidden state, which is initialized to zero.

First, we change the shape of the previous decoder hidden state by adding 1 to its dimensions.

```
hidden_with_time_axis = tf.expand_dims(hidden, 1)
```

Then, we compute the attention score.

Attention Score

The attention score is specified with this formula:

$$score(h_t, \underline{h}_s) = v_a^T \tanh(W_1 h_t + W_2 \underline{h}_s)$$

The pseudocode for the implementation of the attention code in Bahdanau's style will be as follows:

```
score = FC(tanh(FC(EO) + FC(H)))
```

This is the actual implementation:

```
score = tf.nn.tanh(self.W1(features) +
                   self.W2(hidden_with_time_axis))
```

The previous decoder hidden state and current input vector hidden state are provided as input in the preceding statement.

Next, we compute the attention weights.

Attention Weights

Attention weights are mathematically expressed as

$$\alpha_{ts} = \frac{\exp\left(score\left(h_t, \underline{h}_s\right)\right)}{\Sigma_{s'=1}^{S} \exp\left(score\left(h_t, \underline{h}_{s'}\right)\right)}$$

We implement this as follows:

```
attention_weights = tf.nn.softmax(self.V(score),
                                  axis=1)
```

We apply a softmax activation function to the scores to obtain the attention weights. The softmax activation function will get the probabilities whose sum will be equal to 1. This will help to represent the weight or the influence of each input sequence. The higher the attention weight of the input sequence, the higher will be its influence on predicting the target word.

Then, we proceed to compute the context vector.

Context Vector

A context vector is mathematically expressed as

$$c_t = \sum_s a_{ts} h_s$$

We implement this in two steps. First, we compute the context vector for each input:

```
context_vector = attention_weights * features
```

Note that the context vector is the product of the Encoder hidden states and their respective scores. It is basically a weighted multiplication of attention weights and the current input vector hidden state, which is produced by the Encoder.

Next, we sum up all these products.

```
context_vector = tf.reduce_sum(context_vector, axis=1)
```

This decodes the output – the context vector is concatenated with the previous decoder output and fed into the Decoder RNN for that time step along with the previous decoder hidden state to produce a new output. Basically, we do this by reshaping the context vector in the preceding statement.

This completes the implementation of the Bahdanau Attention model.

We now proceed to the actual decoder implementation.

Decoder Implementation

We first convert the caption index to a vector by passing it through the embedding layer.

```
x = self.embedding(x)
```

Next, map the context vector with the vector of captions (x) and combine them into a single vector.

```
x = tf.concat([tf.expand_dims(context_vector, 1),
                                x], axis=-1)
```

Now, pass this vector through the GRU.

```
output, state = self.gru(x)
```

Pass the output of the GRU layer through a Dense layer.

```
x = self.fc1(output)
```

The shape of x at this stage is (batch_size, max_length, hidden_size). Next, reshape x to (batch_size * max_length, hidden_size).

```
x = tf.reshape(x, (-1, x.shape[2]))
```

Add the Dropout and BatchNorm layers.

```
x = self.dropout(x)
x = self.batchnormalization(x)
```

The output shape at this point is (64 x 512). Pass it through a Dense layer to convert this to (64 x 8329). The 8329 is the vocabulary size in our case.

```
x = self.fc2(x)
```

Finally, return the computed values to the caller.

```
return x, state, attention_weights
```

We define one more function for our decoder class for resetting the initial state of the decoder.

```
  def reset_state(self, batch_size):
    return tf.zeros((batch_size, self.units))
```

The entire code for the Decoder implementation is given in Listing 10-1 for your quick reference.

Listing 10-1. Decoder class with Bahdanau Attention

```
class RNN_Decoder(tf.keras.Model):
  def __init__(self, embedding_dim, units,
               vocab_size):
    super(RNN_Decoder, self).__init__()
    self.units = units

    self.embedding = tf.keras.layers.Embedding
                        (vocab_size, embedding_dim)
    self.gru = tf.keras.layers.GRU(self.units,
                        return_sequences=True,
                        return_state=True,
                        recurrent_initializer=
                       'glorot_uniform')

    self.fc1 = tf.keras.layers.Dense(self.units)

    self.dropout = tf.keras.layers.Dropout
                  (0.5, noise_shape=None, seed=None)
    self.batchnormalization =
         tf.keras.layers.BatchNormalization
                   (axis=-1,
                    momentum=0.99,
                    epsilon=0.001,
                    center=True,
                    scale=True,
                    beta_initializer='zeros',
                    gamma_initializer='ones',
                    moving_mean_initializer='zeros',
                  moving_variance_initializer='ones',
```

```
                beta_regularizer=None,
                gamma_regularizer=None,
                beta_constraint=None,
                gamma_constraint=None)

    self.fc2 = tf.keras.layers.Dense(vocab_size)

    # Implementing Attention Mechanism
    self.W1 = tf.keras.layers.Dense(units)
    self.W2 = tf.keras.layers.Dense(units)
    self.V = tf.keras.layers.Dense(1)

  def call(self, x, features, hidden):

    hidden_with_time_axis = tf.expand_dims(hidden, 1)

    # Attention Function
    # computing scores
    score = tf.nn.tanh(self.W1(features) +
                self.W2(hidden_with_time_axis))

    # Probability using Softmax
    attention_weights = tf.nn.softmax(self.V(score),
                                      axis=1)

    # Compute context vector
    context_vector = attention_weights * features
    context_vector = tf.reduce_sum(context_vector,
                                      axis=1)

    # passing the input caption index(integer) to
    # embedding layer to convert it to vector
    x = self.embedding(x)

    # Map the context vector with the input vector
    # (the vectors of caption) and then concatenate them
```

```
    x = tf.concat([tf.expand_dims(context_vector, 1),
                                  x], axis=-1)

    # Pass concatenated vector to the GRU
    output, state = self.gru(x)

    # shape == (batch_size, max_length, hidden_size)
    # Pass output of GRU layer through a Dense layer
    x = self.fc1(output)

    # x shape == (batch_size * max_length,
                              hidden_size)
    x = tf.reshape(x, (-1, x.shape[2]))

    # Add Dropout and BatchNorm Layers
    x = self.dropout(x)
    x = self.batchnormalization(x)
    # output shape == (64 * 512)
    x = self.fc2(x)
    # shape : (64 * 8329(vocab))
    return x, state, attention_weights

  def reset_state(self, batch_size):
    return tf.zeros((batch_size, self.units))
```

Encoder/Decoder Instantiations

We create the encoder and decoder instances as follows:

```
encoder = Inception_Encoder(embedding_dim)
decoder = RNN_Decoder(embedding_dim,
                      units, vocab_size)
```

Just for curiosity's sake, you may plot the models for both Encoders and Decoders.

```
tf.keras.utils.plot_model (encoder)
tf.keras.utils.plot_model (decoder)
```

The model plots are shown in Figure 10-5.

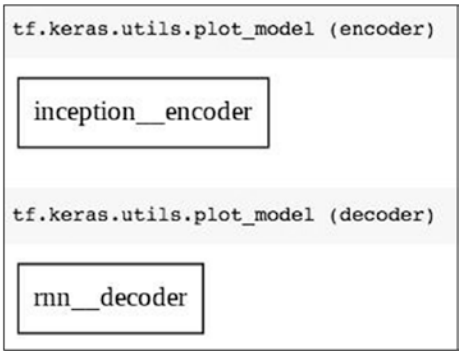


Figure 10-5. *Model plots for the encoder/decoder*

Nothing is really interesting in these model plots as the whole processing is done only in the inner models.

Defining Optimizer and Loss Functions

We use the Adam optimizer and the SparseCategoricalCrossentropy for the loss function.

```
optimizer = tf.keras.optimizers.Adam()
loss_object = tf.keras.losses.
                SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')

def loss_function(real, pred):
  mask = tf.math.logical_not(tf.math.equal(real, 0))
```

```
    loss_ = loss_object(real, pred)
    mask = tf.cast(mask, dtype=loss_.dtype)
    loss_ *= mask

    return tf.reduce_mean(loss_)
```

I will explain to you how the loss is computed with an example. The loss function takes two arguments – the real caption vector and the predicted value. The function computes the loss between these two vectors using the loss_object method of the tf.keras.losses.SparseCategoricalCrossentropy module. Then we use the tf.cast function for converting the mask vector to a float32 data type. It just helps us to convert the vectors into different data types for further manipulation. Finally, we multiply the loss_ with the mask; the reason we are doing this is to map the real values (values of captions present in the dataset) to the computed loss function.

I will illustrate the preceding operations with a realistic example.

Consider the following real caption, which I picked up during training.

```
real(passed as a parameter) : tf.Tensor(
[  0   0   0   0   0   0    0   0    0   0   0   0   0    0
   0   4   0   0   0   0 1760   0  367   0   0   4   0    0
   0   0   0   0   0   0    0   9  453   0   0   0   0    0
   0   0   0   0   0   0  132   0    0   0   0   0   0    0
   0   0   0   4   0   0    0   5], shape=(64,), dtype=int32)
```

Now, apply the following statement on the preceding vector – real. The tf.math.equal just converts the vectors into booleans. It takes two parameters, x and y. If x equals y, it returns true.

```
tf.math.equal(real , 0) tf.Tensor
```

This produces the following Tensor:

```
[True  True  True  True  True  True  True  True  True  True  True  True
 True  True  True False  True  True  True  True False  True False  True
 True False  True  True  True  True  True  True  True  True  True False
False  True  True  True  True  True  True  True  True  True  True  True
False  True  True  True  True  True  True  True  True  True  True False
 True  True  True False], shape=(64,), dtype=bool)
```

Execute the following statement. The tf.math.logical_not takes a boolean parameter and performs a logical NOT operation on it.

```
mask = tf.math.logical_not(tf.math.equal(real, 0))
```

The output would be

```
tf.Tensor(
[False False False False False False False False False False False False
False False False  True False False False False  True False  True False
False  True False False False False False False False False  True
 True False False False False False False False False False False
 True False False False False False False False False False  True
False False False  True], shape=(64,), dtype=bool)
```

Now, look at the loss Tensor produced by this statement:

```
loss_ = loss_object(real, pred)
```

The output is

```
loss_ = loss_object(real, pred)
tf.Tensor(
[13.458616  11.725777  13.339547  13.877813  13.6512375 13.609352
12.680449  13.963526  12.929108  12.504114  12.995626  13.473895
13.966334  13.3766165 13.607654   0.10513641 13.231352  13.313489
13.727711  14.456019  10.560667  13.632038   4.2983437 14.144966
14.331357   0.28515333 13.97144   13.087602  15.597718  13.351999
13.649492  12.489752  12.744471  12.558954  13.255367  1.8581532
```

```
 3.1811125  13.873036   12.329573   12.222642   13.126439   14.233135
12.379726   11.951986   12.869691   13.468082   12.732171   12.240744
 3.8898373  12.682398   13.192276   12.453615   15.758832   14.152502
13.160431   11.863881   12.530688   13.764532   13.640175   0.7283469
14.0648575  12.560375   14.25197     0.53315634], shape=(64,),
                                                 dtype=float32)
```

Create the mask with the following statement:

```
mask = tf.cast(mask, dtype=loss_.dtype)
```

It results in the following Tensor:

```
tf.Tensor(
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 1. 0.
0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 1.], shape=(64,), dtype=float32)
```

Finally, we multiply the loss_ with the mask.

```
loss_ *= mask
```

This will be the final loss Tensor.

```
loss  tf.Tensor(
[ 0.         0.          0.         0.          0.          0.
 0.          0.          0.         0.          0.          0.
 0.          0.          0.         0.10513641  0.          0.
 0.          0.         10.560667   0.          4.2983437   0.
 0.          0.28515333  0.         0.          0.          0.
 0.          0.          0.         0.          0.   1.8581532
 3.1811125   0.          0.         0.          0.          0.
 0.          0.          0.         0.          0.          0.
 3.8898373   0.          0.         0.          0.          0.
 0.          0.          0.         0.          0.   0.7283469
 0.          0.          0.         0.53315634], shape=(64,),
                                                  dtype=float32)
```

Creating Checkpoints

We create a separate folder for saving checkpoints, and we save a maximum of five checkpoints in this folder.

```
checkpoint_path = "./checkpoints/train"
ckpt = tf.train.Checkpoint(encoder=encoder,
                           decoder=decoder,
                           optimizer = optimizer)
ckpt_manager = tf.train.CheckpointManager
            (ckpt, checkpoint_path, max_to_keep=5)
```

We declare a variable called start_epoch in case you want to restart the training from a last known checkpoint.

```
start_epoch = 0
```

Check for the last saved state, if any:

```
if ckpt_manager.latest_checkpoint:
  start_epoch = int
    (ckpt_manager.latest_checkpoint.split('-')[-1])
  # restoring the latest checkpoint in checkpoint_path
  ckpt.restore(ckpt_manager.latest_checkpoint)
```

If checkpoint is not restored, you explicitly do so by calling this code:

```
ckpt.restore(tf.train.latest_checkpoint
               (checkpoint_path))
```

Training Step Function

We now write a function to define a training step. We will call this for our desired number of epochs. The function definition is given as follows:

```
loss_plot = []

def train_step(img_tensor, target):
  loss = 0

  # initialize the hidden state for each batch
  # because the captions are not related from image to image
  hidden = decoder.reset_state
                    (batch_size=target.shape[0])

  dec_input = tf.expand_dims([tokenizer.word_index
                              ['<start>']] *
                              BATCH_SIZE, 1)

  with tf.GradientTape() as tape:
      features = encoder(img_tensor)

      for i in range(1, target.shape[1]):
          # Pass the features through the decoder
          predictions, hidden, _ = decoder(dec_input,
                                    features, hidden)

          loss += loss_function(target[:, i],
                                predictions)

          # Use teacher forcing
          dec_input = tf.expand_dims(target[:, i], 1)

  total_loss = (loss / int(target.shape[1]))
  trainable_variables = encoder.trainable_variables +
                        decoder.trainable_variables
```

```
  gradients = tape.gradient
                  (loss, trainable_variables)
  optimizer.apply_gradients(zip(gradients,
                                trainable_variables))

  return loss, total_loss
```

The function first calls the reset_state defined in the decoder model to initialize the hidden state of the decoder for each batch. We do this because we do not want to use the state/captions of the previous batch. Note the captions won't be the same in every batch. We add the <start> tag in the first batch. We use the gradient tape to iterate through the batches of data and update the gradients at each iteration.

Model Training

We train the model by calling the training step function a desired number of times.

```
for epoch in range(start_epoch, 20):
    start = time.time()

    total_loss_train = 0
    for (batch, (img_tensor, target))
                 in enumerate(dataset):
        batch_loss, t_loss = train_step
                  (img_tensor, target)
        total_loss_train += t_loss

    if epoch % 5 == 0:
      ckpt_manager.save()

    print ('Epoch {} Train-Loss {:.4f}'.format
                (epoch + 1,
```

```
                    (total_loss_train/num_steps)))
    print ('Time taken for this epoch {}
                sec\n'.format(time.time() - start))
```

Model Inference

To generate a caption for an unseen image, we take the same steps as we did for the images during training.

We write the evaluate function as follows:

```
def evaluate(image):
    hidden = decoder.reset_state(batch_size=1)

    temp_input = tf.expand_dims
                    (load_image(image)[0], 0)
    img_tensor_val = image_features_extract_model
                    (temp_input)
    img_tensor_val = tf.reshape(img_tensor_val,
                        (img_tensor_val.shape[0],
                         -1,
                         img_tensor_val.shape[3]))

    features = encoder(img_tensor_val)

    dec_input = tf.expand_dims([tokenizer.word_index
                                  ['<start>']], 0)
    result = []

    for i in range(max_length):
        predictions, hidden, attention_weights =
                        decoder(dec_input,
                        features, hidden)
```

```
        predicted_id = tf.random.categorical
                    (predictions, 1)[0][0].numpy()
        result.append(tokenizer.index_word[predicted_id])

        if tokenizer.index_word[predicted_id] ==
                                        '<end>':
            return result

        dec_input = tf.expand_dims([predicted_id], 0)

    return result
```

The function implementation is straightforward. We create the image tensor as we did for the training images, call the encoder to extract its features, and then call the decoder to predict words for max_length times. Note the max_length was computed earlier and specifies the fixed sequence length.

We write the predict function to accept the image URL and some random name for the image. The downloaded image file is stored in the /root/.keras folder; by appending the user-specified name to the image, each downloaded file will be copied with a distinct name. If you do not do this, the same caption would be generated for all subsequent image downloads.

```
def predict(image_url , random_name):
    image_extension = image_url[-4:]
    image_path = tf.keras.utils.get_file
                  ('image'+ random_name +
                   image_extension,
                   origin=image_url)
    result = evaluate(image_path)
    print ('Prediction Caption:', ' '.join(result))
    Image.open(image_path)
    return image_path
```

We now call the predict function on a test image.

```
image_url = 'https://tensorflow.org/images/surf.jpg'
path = predict(image_url , 'surfee')
Image.open(path)
```

The output is shown in Figure 10-6.

Figure 10-6. *A sample image with a generated caption*

The result on another image is shown in Figure 10-7.

```
image_url = 'https://farm4.staticflickr.com/3296/2765087292_535
6df67ce_z.jpg'
path = predict(image_url , 'baseball')
Image.open(path)
```

Figure 10-7. *Another sample image with a generated caption*

The result on another image is shown in Figure 10-8.

```
image_url = 'https://farm8.staticflickr.com/7139/8156048469_084
7c7ce15_z.jpg'
path = predict(image_url , 'dog')
Image.open(path)
```

Figure 10-8. *Yet another image with a generated caption*

Full Source

The full source is given in Listing 10-2 for your ready reference.

Listing 10-2. Image captioning

```
import os
import time
import pickle
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
from tensorflow.keras.applications
        import  InceptionV3
from os import listdir
from tqdm import tqdm
from PIL import Image

!wget --no-check-certificate -r ‘https://drive.google.com/uc?
export=download&id=1c7yGTpizf5egVD9dc3Q2lrxS8wtOAV42’ -O
Flickr8k_text.zip

!mkdir captions images

!unzip 'Flickr8k_text.zip' -d '/content/captions'

!wget --no-check-certificate -r 'https://drive.google.com/uc?
export=download&id=1126G_E2OpvULyvTmOKz_oMhOzv8CkiW1' -O
Flickr8k_Dataset.zip

!unzip 'Flickr8k_Dataset.zip' -d '/content/images'

## The location of the Flickr8K_ photos
image_dir = '/content/images/Flicker8k_Dataset'
```

```
images = listdir(image_dir)
print("The number of jpg flies in Flicker8k: {}"
        .format(len(images)))

# load doc into memory
def load(filename):
  file = open(filename, 'r')
  text = file.read()
  file.close()
  return text

filename = '/content/captions/Flickr8k.token.txt'
doc = load(filename)

dirs = listdir('/content/images/Flicker8k_Dataset')

dirs[:5]

def load_small(doc):
    PATH = '/content/images/Flicker8k_Dataset/'
    img_path = []
    img_id = []
    img_cap = []
    for line in doc.split('\n'):
        tokens = line.split()
        if len(line) < 2:
            continue
        image_id , image_desc = tokens[0] ,
                                tokens[1:]
        image_id = image_id.split('.')[0]
        image_id = image_id + '.jpg'
        image_desc = ' '.join(image_desc)
        if image_id not in img_id:
            if len(img_id) <= 8000:
```

```
                img_id.append(image_id)
                image_path = PATH + image_id
                image_desc = '<start> ' + image_desc
                                        + ' <end>'
                if image_id in dirs:
                    img_path.append(image_path)
                    img_cap.append(image_desc)
            else:
                continue
    return img_path , img_cap

all_image_path , all_image_captions = load_small(doc)

print('Number of images: ', len(all_image_path))
all_image_path[:5]

print('Number of captions: ',
                len(all_image_captions))
all_image_captions[:5]

train_captions, img_name_vector =
                shuffle(all_image_captions,
                        all_image_path,
                        random_state=1)

image_model = InceptionV3(include_top=False,
                          weights='imagenet')

new_input = image_model.input
hidden_layer = image_model.layers[-1].output
image_features_extract_model = tf.keras.Model
                (new_input, hidden_layer)

def load_image(image_path):
    img = tf.io.read_file(image_path)
```

```
    img = tf.image.decode_jpeg(img, channels=3)
    img = tf.image.resize(img, (299, 299))
    img = tf.keras.applications.inception_v3.preprocess_input
                                    (img)
    return img, image_path

encode_train = sorted(set(img_name_vector))
image_dataset = tf.data.Dataset.from_tensor_slices
                        (encode_train)
image_dataset = image_dataset.map(load_image,
                    num_parallel_calls=tf.data.experimental.
                    AUTOTUNE)
                          .batch(16)

for img, path in tqdm(image_dataset):
  batch_features = image_features_extract_model(img)
  batch_features = tf.reshape(batch_features,
                           (batch_features.shape[0],
                            -1,
                            batch_features.shape[3]))

  for bf, p in zip(batch_features, path):
    path_of_feature = p.numpy().decode("utf-8")
    np.save(path_of_feature, bf.numpy())

tokenizer = tf.keras.preprocessing.text.Tokenizer
        (filters='!"#$%&()*+.,-/:;=?@[\]^_`{|}~ ')
tokenizer.fit_on_texts(train_captions)
max_size = len(tokenizer.word_index)

train_seqs = tokenizer.texts_to_sequences
                      (train_captions)

train_seqs[:5]

max_length = max(len(t) for t in train_seqs)
```

```
cap_vector = tf.keras.preprocessing.
                        sequence.pad_sequences
                                (train_seqs,
                                 padding='post')

cap_vector[:5]

BATCH_SIZE = 64
BUFFER_SIZE = 1000
embedding_dim = 256
units = 512
vocab_size = max_size + 1
num_steps = len(img_name_vector) // BATCH_SIZE

# Loading previously extracted features
def map_func(img_name, cap):
  img_tensor = np.load(img_name.decode
                       ('utf-8')+'.npy')
  return img_tensor, cap

def create_dataset(img_name_train,caption_train):
  dataset = tf.data.Dataset.from_tensor_slices
               ((img_name_train, caption_train))

  # Use map to load the numpy files in parallel
  dataset = dataset.map(lambda item1, item2:
        tf.numpy_function(map_func, [item1, item2],
        [tf.float32, tf.int32]),num_parallel_calls=
        tf.data.experimental.AUTOTUNE)

  # Shuffle and batch
  dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
 .prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
  return dataset
```

```
dataset = create_dataset(img_name_vector,cap_vector)

class Inception_Encoder(tf.keras.Model):
    def __init__(self, embedding_dim):
        super(Inception_Encoder, self).__init__()
        # shape after fc = (batch_size, 64,
                            embedding_dim)
        self.fc = tf.keras.layers.Dense
                           (embedding_dim)

    def call(self, x):
        x = self.fc(x)
        x = tf.nn.relu(x)
        return x

class RNN_Decoder(tf.keras.Model):
  def __init__(self, embedding_dim, units,
               vocab_size):
    super(RNN_Decoder, self).__init__()
    self.units = units

    self.embedding = tf.keras.layers.Embedding
                         (vocab_size, embedding_dim)
    self.gru = tf.keras.layers.GRU(self.units,
                         return_sequences=True,
                         return_state=True,
                         recurrent_initializer=
                        'glorot_uniform')

    self.fc1 = tf.keras.layers.Dense(self.units)

    self.dropout = tf.keras.layers.Dropout
                  (0.5, noise_shape=None, seed=None)
```

```
    self.batchnormalization =
         tf.keras.layers.BatchNormalization
                   (axis=-1,
                    momentum=0.99,
                    epsilon=0.001,
                    center=True,
                    scale=True,
                    beta_initializer='zeros',
                    gamma_initializer='ones',
                    moving_mean_initializer='zeros',
                  moving_variance_initializer='ones',
                  beta_regularizer=None,
                  gamma_regularizer=None,
                  beta_constraint=None,
                  gamma_constraint=None)

    self.fc2 = tf.keras.layers.Dense(vocab_size)

    # Implementing Attention Mechanism
    self.W1 = tf.keras.layers.Dense(units)
    self.W2 = tf.keras.layers.Dense(units)
    self.V = tf.keras.layers.Dense(1)

  def call(self, x, features, hidden):

    hidden_with_time_axis = tf.expand_dims(hidden, 1)

    # Attention Function
    # computing scores
    score = tf.nn.tanh(self.W1(features) +
                self.W2(hidden_with_time_axis))

    # Probability using Softmax
    attention_weights = tf.nn.softmax(self.V(score),
                                      axis=1)
```

```
    # Compute context vector
    context_vector = attention_weights * features
    context_vector = tf.reduce_sum(context_vector,
                                   axis=1)

    # passing the input caption index(integer) to
    # embedding layer to convert it to vector
    x = self.embedding(x)

    # Map the context vector with the input vector
    # (the vectors of caption) and then concatenate them
    x = tf.concat([tf.expand_dims(context_vector, 1),
                                  x], axis=-1)

    # Pass concatenated vector to the GRU
    output, state = self.gru(x)

    # shape == (batch_size, max_length, hidden_size)
    # Pass output of GRU layer through a Dense layer
    x = self.fc1(output)

    # x shape == (batch_size * max_length,
                              hidden_size)
    x = tf.reshape(x, (-1, x.shape[2]))

    # Add Dropout and BatchNorm Layers
    x = self.dropout(x)
    x = self.batchnormalization(x)
    # output shape == (64 * 512)
    x = self.fc2(x)
    # shape : (64 * 8329(vocab))
    return x, state, attention_weights

  def reset_state(self, batch_size):
    return tf.zeros((batch_size, self.units))
```

```
encoder = Inception_Encoder(embedding_dim)
decoder = RNN_Decoder(embedding_dim,
                      units, vocab_size)

tf.keras.utils.plot_model (encoder)


tf.keras.utils.plot_model (decoder)

optimizer = tf.keras.optimizers.Adam()
loss_object = tf.keras.losses.
                 SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')

def loss_function(real, pred):
  mask = tf.math.logical_not(tf.math.equal(real, 0))
  loss_ = loss_object(real, pred)
  mask = tf.cast(mask, dtype=loss_.dtype)
  loss_ *= mask

  return tf.reduce_mean(loss_)


checkpoint_path = "./checkpoints/train"
ckpt = tf.train.Checkpoint(encoder=encoder,
                           decoder=decoder,
                           optimizer = optimizer)
ckpt_manager = tf.train.CheckpointManager
            (ckpt, checkpoint_path, max_to_keep=5)

start_epoch = 0

if ckpt_manager.latest_checkpoint:
  start_epoch = int
    (ckpt_manager.latest_checkpoint.split('-')[-1])
```

```
  # restoring the latest checkpoint in checkpoint_path
  ckpt.restore(ckpt_manager.latest_checkpoint)

# if checkpoint is not restored run this code
ckpt.restore(tf.train.latest_checkpoint
               (checkpoint_path))

loss_plot = []

def train_step(img_tensor, target):
  loss = 0

  # initialize the hidden state for each batch
  # because the captions are not related from image to image
  hidden = decoder.reset_state
                    (batch_size=target.shape[0])

  dec_input = tf.expand_dims([tokenizer.word_index
                              ['<start>']] *
                              BATCH_SIZE, 1)

  with tf.GradientTape() as tape:
      features = encoder(img_tensor)

      for i in range(1, target.shape[1]):
          # Pass the features through the decoder
          predictions, hidden, _ = decoder(dec_input,
                                           features, hidden)

          loss += loss_function(target[:, i],
                                predictions)

          # Use teacher forcing
          dec_input = tf.expand_dims(target[:, i], 1)
```

```
  total_loss = (loss / int(target.shape[1]))
  trainable_variables = encoder.trainable_variables +
                        decoder.trainable_variables
  gradients = tape.gradient
                  (loss, trainable_variables)
  optimizer.apply_gradients(zip(gradients,
                                trainable_variables))

  return loss, total_loss

for epoch in range(start_epoch, 20):
    start = time.time()

    total_loss_train = 0
    for (batch, (img_tensor, target))
                 in enumerate(dataset):
        batch_loss, t_loss = train_step
                 (img_tensor, target)
        total_loss_train += t_loss

    if epoch % 5 == 0:
      ckpt_manager.save()

    print ('Epoch {} Train-Loss {:.4f}'.format
                (epoch + 1,
                    (total_loss_train/num_steps)))
    print ('Time taken for this epoch {}
                sec\n'.format(time.time() - start))

def evaluate(image):
    hidden = decoder.reset_state(batch_size=1)

    temp_input = tf.expand_dims
                    (load_image(image)[0], 0)
    img_tensor_val = image_features_extract_model
                    (temp_input)
```

```
    img_tensor_val = tf.reshape(img_tensor_val,
                        (img_tensor_val.shape[0],
                         -1,
                         img_tensor_val.shape[3]))

    features = encoder(img_tensor_val)

    dec_input = tf.expand_dims([tokenizer.word_index
                                ['<start>']], 0)
    result = []

    for i in range(max_length):
        predictions, hidden, attention_weights =
                        decoder(dec_input,
                        features, hidden)

        predicted_id = tf.random.categorical
                    (predictions, 1)[0][0].numpy()
        result.append(tokenizer.index_word[predicted_id])

        if tokenizer.index_word[predicted_id] ==
                                      '<end>':
            return result

        dec_input = tf.expand_dims([predicted_id], 0)

    return result

def predict(image_url , random_name):
    image_extension = image_url[-4:]
    image_path = tf.keras.utils.get_file
                  ('image'+ random_name +
                   image_extension,
                   origin=image_url)
    result = evaluate(image_path)
```

```
    print ('Prediction Caption:', ' '.join(result))
    Image.open(image_path)
    return image_path

image_url = 'https://tensorflow.org/images/surf.jpg'
path = predict(image_url , 'surfee')
Image.open(path)

image_url = 'https://farm4.staticflickr.com/3296/2765087292_535
6df67ce_z.jpg'
path = predict(image_url , 'baseball')
Image.open(path)

image_url = 'https://farm8.staticflickr.com/7139/8156048469_084
7c7ce15_z.jpg'
path = predict(image_url , 'dog')
Image.open(path)

image_url = 'https://farm5.staticflickr.com/4095/4910762818_
b1e9022005_z.jpg'
path = predict(image_url , 'tennis')
Image.open(path)

image_url = 'https://farm3.staticflickr.com/2690/4179330518_
b82897b153_z.jpg'
path = predict(image_url , 'competition')
Image.open(path)
```

Summary

In this chapter, you learned another important application of natural language processing. You created an application for captioning any given image. Image captioning has two important parts – one to extract the

image features and the other to map these features into a textual caption. Extracting the image features is a trivial job, and there are many pre-trained networks available for this purpose. You used the InceptionV3 network for extracting the features. For translating this to a caption, you used RNN with Bahdanau Attention. The Bahdanau Attention was described in depth alongside its implementation.

In the next chapter, you will learn another important technique and that is time series forecasting.

CHAPTER 11

Time Series Forecasting

Introduction

Forecasting has always been a topic of interest for every human being. What is my future? Will I become a millionaire in the next 5 years? When will I get married? These are the questions raised by several of us. There are people in this world who do forecasting and at least try to provide answers to such questions. Neural networks so far are not successful in doing such kinds of forecasts. But they do certainly forecast the futures where the past data contains some discoverable patterns. The topic of this chapter is to learn how to train neural networks to perform such kinds of forecasts. These are called time series forecasting. Let me first describe what we mean by a time series followed by forecasting the future.

What Is Time Series Forecasting?

Making predictions about the future is called extrapolation in classical statistics. In the modern world, people use the term time series forecasting to mean the same as extrapolation. Forecasting involves creating models

P. Sarang, *Artificial Neural Networks with TensorFlow 2*,
https://doi.org/10.1007/978-1-4842-6150-7_11

that fit on historical data and using them to predict the future – things which have not happened so far. Like what? The answer to the question "Is it going to be hot tomorrow?" is a future prediction. Neural networks can do this kind of prediction, albeit to a greater accuracy than a mere intuitional prediction done by a man. How do they do this? We provide them the temperature data over the last several days, months, and probably years. The network finds the pattern which is essentially a time series pattern and does a prediction based on this pattern. Another example of time series forecasting can be predicting the future price of a stock. The question "What will be the price of IBM stock one month down the line?" can be answered by a trained neural network.

Here are a few more examples of time series:

- Forecasting whether a patient is having a seizure based on their EEG (electroencephalogram)
- Forecasting product sales of Barbie dolls in the month of December
- Forecasting an hourly bandwidth utilization demand on a server
- Forecasting the corp yield next year
- Forecasting the birth rate in a certain city

Like this, you will find many questions which have great interests in the minds of human beings. To solve such problems, many statistical techniques were developed over a period of time. In today's age, neural networks aid in this human effort. Before I discuss how this kind of forecasting is done, let me describe some of the important concerns in such forecasting.

Concerns of Forecasting

Some of the major concerns while forecasting may be listed as follows:

- How much past data do you have?
- Does the user want to forecast short, medium, or long term?
- Are the forecasts static?
- Are periodic forecasts required for a given problem?

The first and foremost important point in time series forecasting is how much past data you have. Note that the neural network is going to analyze the past data to find trends in it. Based only on these trends, future forecasts can be made.

It will be easier to make short-term forecasts, while medium- or long-term forecasts are typically difficult to make due to the fact that there could be lots of uncertainties as the trends in data may change over these time periods.

When a user asks for a forecast, are they looking for a static forecast or are they willing to accept a new forecast when the conditions change? The requirement of a dynamic forecast can be one of the concerns while designing a forecasting model.

Sometimes, the forecasts on the same problem are made at a predecided frequency. The low or high frequency of forecasts becomes a deciding factor while sampling the data, which in turn influences the modeling. Predicting tomorrow's temperature every day is considered a low-frequency forecasting, while high-frequency trading on a stock market is considered a high-frequency forecasting.

I will now describe the various components of a time series.

Components of Time Series

While analyzing a time series for modeling, there are a few points that you will need to observe carefully. These are the components of a time series which may be classified into the following categories:

- General trends
- Seasonal movements
- Cyclical movements
- Noise (irregular fluctuations)

The general or secular trend is the main component of a time series. The trend in a time series which follows a certain observable pattern may show an incline or a decline over a period of time, mainly due to the effects of socioeconomic and political factors. The ice cream manufacturer's increased sales during the summer and a decline during the winter are termed a seasonal movement in the sales trend. You may observe long-time oscillations in a time series assuming you have long-term data, say 5–12 years of data, available for analysis. These are the long-term cycles observed in economic data of a country's economy. Businesses too show long-term oscillations which are well known as business cycles. Lastly, the noise or irregular fluctuations in the data occur when there are unforeseen situations like the COVID-19 epidemic. While developing a model for time series analysis, you must factor out the preceding components in your analysis.

Finally, we come to another important aspect, and that is the classification of time series in its two major components.

Univariate vs. Multivariate

All time series are classified into two main categories:

- Univariate
- Multivariate

A single variable measured over time is called a univariate time series. For example, the consumption of electricity in your home measured over a period of one year is considered a univariate time series. Such time series are best modeled using statistical techniques, such as autoregressive moving average (ARMA). As the name suggests, it combines autoregression techniques with moving average techniques to forecast those future values. It assumes that the previous observations are good predictors of future values.

A multivariate time series has more than one time-dependent variable. Thus, the resultant not merely depends on the past values of the multiple variables but is also a function of interdependencies between them. A well-known statistical technique for modeling a multivariate time series is the vector autoregression (VAR) model. The model assumes that each variable is a function of the past values of itself and also the past values of all correlated variables. An example of a multivariate time series would be air pollution which depends on several interdependent components.

Having said all these, I will now show you how to develop neural network models for forecasting on both univariate and multivariate time series.

Univariate Time Series Analysis

In this project, you will be building a model that predicts the energy consumption. For this purpose, we will use the dataset provided by Kaggle (`www.kaggle.com/robikscube/hourly-energy-consumption`). The dataset provided by PJM Interconnection LLC consists of over 10 years of hourly energy consumption expressed in megawatts.

Creating Project

Create a new Colab project and rename it to Univariate – time series analysis. Import the required libraries:

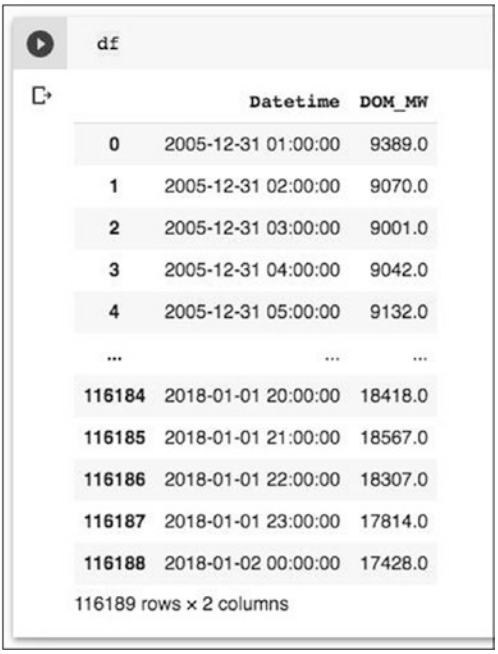
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn.preprocessing
from sklearn.metrics import r2_score
```

Preparing Data

Load the data into the project from the book's download page.

```
url = 'https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch11/DOM_hourly.csv'
df = pd.read_csv(url)
```

Examine the data by dumping it on the screen. Figure 11-1 shows the screen dump.



df

	Datetime	DOM_MW
0	2005-12-31 01:00:00	9389.0
1	2005-12-31 02:00:00	9070.0
2	2005-12-31 03:00:00	9001.0
3	2005-12-31 04:00:00	9042.0
4	2005-12-31 05:00:00	9132.0
...	...	...
116184	2018-01-01 20:00:00	18418.0
116185	2018-01-01 21:00:00	18567.0
116186	2018-01-01 22:00:00	18307.0
116187	2018-01-01 23:00:00	17814.0
116188	2018-01-02 00:00:00	17428.0

116189 rows × 2 columns

Figure 11-1. *Hourly power consumption data for a full period*

The data contains more than 100,000 energy readings. The readings are spaced out every hour. The data starts in 2005 and is recorded up to January 2018. You may generate the daily data from this by using the following statement:

```
# uncomment the following line for generating daily data
# df = df[df['Datetime'].str.contains("00:00:00")]
```

I have commented the code line here and in the downloadable code. Initially, we will run our experiment on an hourly data that is provided in the file. Later on, you will uncomment the preceding line and rerun the project to achieve weekly predictions. The purpose of this is explained toward the end of this project.

Create an index on the datetime field. Creating an index makes the datetime column be treated as date rather than an object.

```
df['Datetime'] = pd.to_datetime(df.Datetime ,
format = '%Y-%m-%d %H:%M:%S')
df.index = df.Datetime
df.drop(['Datetime'], axis = 1,inplace = True)
```

Check the data after indexing with the head() method.

```
df.head()
```

The output is shown in Figure 11-2.

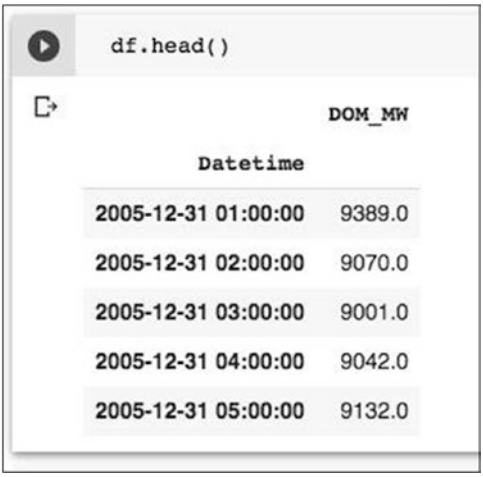


Figure 11-2. *Data indexed on the datetime field*

You may check for the null values in the data for the sake of completeness.

```
#checking missing data
df.isna().sum()
```

There are no null values.

You can see the energy consumption trend by plotting the energy consumption vs. the time. The plot is generated using the following code:

```
#@title Date Range
a = '2005-12-31'#@param {type:"date"}
b = '2018-01-31' #@param {type:"date"}

a = a+" 00:00:00"
b = b+" 00:00:00"
df.loc[a:b].plot(figsize = (16,4),legend = True)

plt.title('DOM hourly power consumption data')
plt.ylabel('Power consumption (MW)')

plt.show()
```

The energy consumption plot is shown in Figure 11-3.

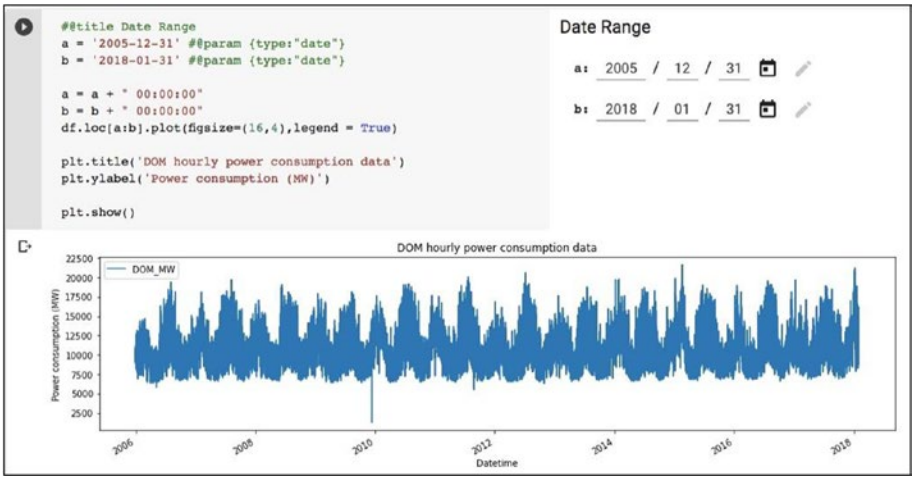


Figure 11-3. *Energy consumption plot for a full period*

As you can see in Figure 11-3, the consumption trend follows almost a uniform pattern over the entire period. In time analysis, it is important to observe seasonal changes in the pattern. Fortunately, Colab allows us to accept the user parameters at runtime. In our plotting code, I am accepting the start and end dates as parameters. You can try changing these dates

to generate a plot for a smaller region. Figure 11-4 shows an energy consumption plot for the month of January 2010.

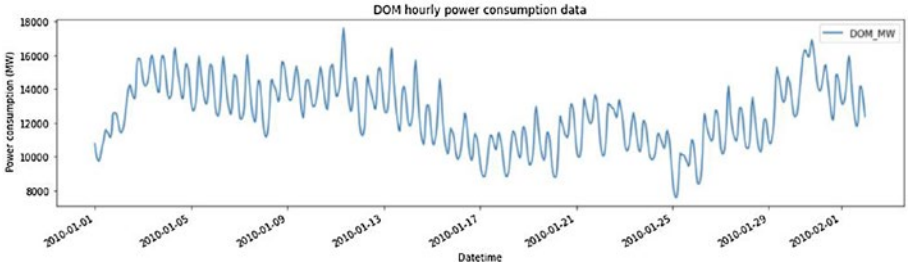


Figure 11-4. *Zooming in on Jan 2010 data*

After observing the trend, say, the aim is to predict energy consumption for the first two weeks of February 2018. So, we need to create a model that is trained on this huge amount of data spanning over 13 years.

Before we input this data to our model for training, the data needs to be normalized in the range 0 to +1. This is done by using the MinMaxScaler function of sklearn.

```
scaler = sklearn.preprocessing.MinMaxScaler()
df['DOM_MW'] = scaler.fit_transform
                  (df['DOM_MW'].values.reshape(-1,1))
```

After normalization, you may plot the chart of the normalized data using the following code snippet:

```
df.plot(figsize = (16,4), legend = True)
plt.title('DOM hourly power consumption data -
                  AFTER NORMALIZATION')
plt.ylabel('Normalized power consumption')
plt.show()
```

The data plot after normalizing is shown in Figure 11-5.

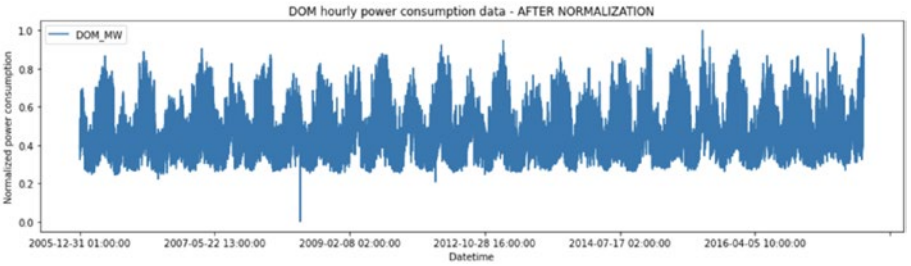


Figure 11-5. *Normalized power consumption plot*

Note that all y values now range in 0 to +1.

At this point, the data preprocessing is complete. You are now ready to create training and testing datasets.

Creating Training/Testing Datasets

To create the training dataset, we need to create sequences in the entire dataset. Suppose we define the sequence length of 20, then the first 20 data points will be our first sequence, and the 21st point will be our target value. The next sequence will be from 1 to 21, and the 22nd will become the target, and so on. This is depicted in Figure 11-6.

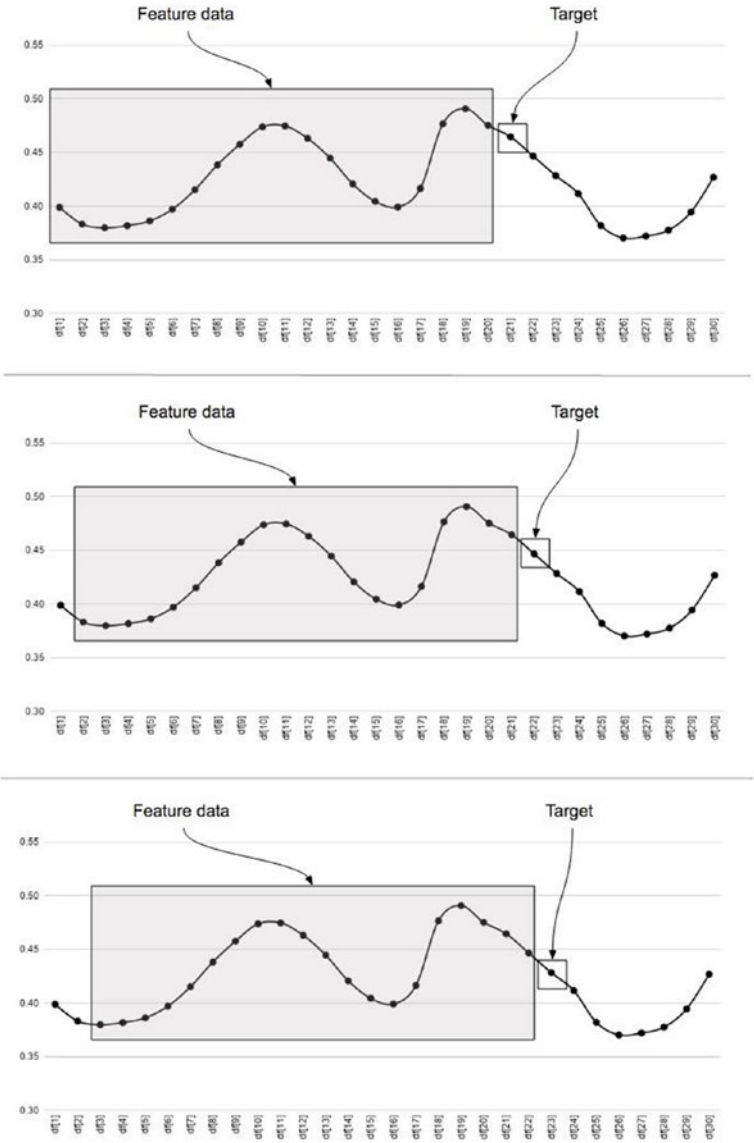


Figure 11-6. *How data is sequenced*

The first chart shows a sequence of 20 data points from df[1] to df[20], 21st being the target. The next sequence includes data points from df[2] to df[21], 22nd being the target. The third sequence includes data from df[3] to df[22], 23rd being the target.

To create the sequencing data like this, we declare a function called load_data as follows:

```
def load_data(stock, seq_len):
```

The stock parameter specifies the dataset which needs to be split in sequences. The seq_len parameter specifies the desired sequence length. I am sending the sequence length as a parameter here so that later on you can experiment with different sequence ranges to predict the short-term/long-time trends. The sequences are created using the following for loop:

```
X_train = []
y_train = []
for i in range(seq_len, len(stock)):
    X_train.append(stock.iloc[i-seq_len : i, 0])
    y_train.append(stock.iloc[i, 0])
```

We have 116,189 data points in our dataset. We will use the first 90% data points for training and the remaining for testing.

```
X_test = X_train[int(0.9*(len(stock))):]
y_test = y_train[int(0.9*(len(stock))):]

X_train = X_train[:int(0.9*(len(stock)))]
y_train = y_train[:int(0.9*(len(stock)))]
```

Next, we convert this data into numpy arrays.

```
# convert to numpy array
X_train = np.array(X_train)
y_train = np.array(y_train)
```

```
X_test = np.array(X_test)
y_test = np.array(y_test)
```

We reshape the numpy array to the desired shape.

```
# reshape data to input into RNN models
X_train = np.reshape(X_train,
           (int(0.9*(len(stock))), seq_len, 1))
X_test = np.reshape(X_test,
           (X_test.shape[0], seq_len, 1))
```

We return the created datasets to the caller:

```
return [X_train, y_train, X_test, y_test]
```

The entire function definition is shown in Listing 11-1.

Listing 11-1. The load_data function code

```
def load_data(stock, seq_len):
    X_train = []
    y_train = []
    for i in range(seq_len, len(stock)):
        X_train.append(stock.iloc[i-seq_len : i, 0])
        y_train.append(stock.iloc[i, 0])

    X_test = X_train[int(0.9*(len(stock))):]
    y_test = y_train[int(0.9*(len(stock))):]

    X_train = X_train[:int(0.9*(len(stock)))]
    y_train = y_train[:int(0.9*(len(stock)))]

    # convert to numpy array
    X_train = np.array(X_train)
    y_train = np.array(y_train)
```

```
    X_test = np.array(X_test)
    y_test = np.array(y_test)

    # reshape data to input into RNN models
    X_train = np.reshape(X_train,
                         (X_train.shape[0], seq_len, 1))
    X_test = np.reshape(X_test,
                         (X_test.shape[0], seq_len, 1))

    return [X_train, y_train, X_test, y_test]
```

Using this function, we now create the training/testing datasets:

```
#create train, test data
seq_len = 20 #choose sequence length
X_train, y_train, X_test, y_test = load_data
                                        (df, seq_len)
```

For your understanding of the dimensions of the datasets, print their shapes using the following code:

```
print('X_train.shape = ',X_train.shape)
print('y_train.shape = ', y_train.shape)
print('X_test.shape = ', X_test.shape)
print('y_test.shape = ',y_test.shape)
```

You will see the following output:

```
X_train.shape =  (104570, 20, 1)
y_train.shape =  (104570,)
X_test.shape =  (11599, 20, 1)
y_test.shape =  (11599,)
```

Note the shape of X_train and X_test. Both contain sequence data of 20 points each.

Creating Input Tensors

We create the tensors for inputting data to our model in batches using the following code:

```
batch_size  = 256
buffer_size = 1000

train_data = tf.data.Dataset.from_tensor_slices
                        ((X_train , y_train))
train_data = train_data.cache().shuffle
             (buffer_size).batch(batch_size).repeat()

test_data = tf.data.Dataset.from_tensor_slices
                    ((X_test , y_test))
test_data = test_data.batch(batch_size).repeat()
```

You would be wondering, how can we shuffle the time series data? What we shuffle is the position of the window and not the data within a single window. Generally, when you shuffle the training data like in this case, you shuffle the order in which these sequences are input to your model. Doing so, you are not shuffling the ordering of data within the individual sequences. For stateless networks, this works as the network's memory does not persist across sequences. For a stateful network where the evaluation of a sequence requires the memory of what happened in a previous sequence, this shuffling will not work. By its nature, all LSTMs are stateful by default. The question of deciding between stateless and stateful comes only when you want to maintain the state from one batch to the next one. Thus, whether to consider a given time series as stateless or stateful while training the network in batches of data is your decision.

Now, we are ready to build our model.

Building Model

We build the model using the following code:

```
rnn_model = tf.keras.models.Sequential([
   tf.keras.layers.LSTM(8, input_shape =
                          X_train.shape[-2:]),
   tf.keras.layers.Dense(1)
])
```

Plot the network model using the following command:

```
tf.keras.utils.plot_model(rnn_model)
```

You will see the output in Figure 11-7.

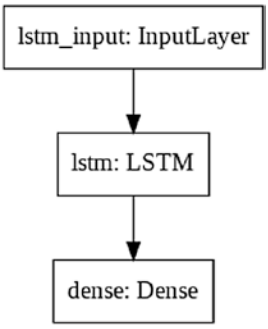


Figure 11-7. *Network model*

Note that the first is an LSTM input layer with 20 nodes, followed by an LSTM layer with 8 nodes, and finally an output layer.

Compiling and Training

We compile the model using its compile method:

```
rnn_model.compile(optimizer = 'adam', loss = 'mae')
```

The model is trained by calling its fit method.

```
EVALUATION_INTERVAL = 200
EPOCHS = 10
rnn_model.fit(train_data, epochs = EPOCHS,
              steps_per_epoch = EVALUATION_INTERVAL,
                     validation_data = test_data,
                     validation_steps = 50)
```

After the model completes the training, we do its performance evaluation.

Evaluation

To evaluate the model's performance, we call its predict method on the test data.

```
rnn_predictions = rnn_model.predict(X_test)
```

```
We call r2_score of sklearn_metrics to check the performance
score:
```

```
rnn_score = r2_score(y_test,rnn_predictions)
print("R2 Score of RNN model =
            "+"{:.4f}".format(rnn_score));
```

You will see the following output:

```
R2 Score of RNN model = 0.9484
```

R^2 - a coefficient of determination is a regression score. The best possible score is 1.0. An R^2 score of 0.0 indicates that the model is constant that always predicts the expected value of y, disregarding the input features. In our case, the value is close to 1.0, and thus we can safely assume that the model is well trained.

We will now plot the chart of actual values vs. the predicted values for the entire testing dataset using the following plotting code:

```
#@title Data Range
a = 0 #@param {type:"slider", min:0,
               max:12000, step:1}
b = 12000 #@param {type:"slider", min:0,
                   max:12000, step:1}
def plot_predictions(test, predicted, title):
  plt.figure(figsize = (16,4))
  plt.plot(test[a:b], color = 'blue',label =
                  'Normalized power consumption')
  plt.plot(predicted[a:b], alpha = 0.7,
           color = 'orange',
           label = 'Predicted power consumption')
  plt.title(title)
  plt.xlabel('Time')
  plt.ylabel('Normalized power consumption')
  plt.legend()
  plt.show()
  plot_predictions(y_test, rnn_predictions,
"Predictions made by simple RNN model")
```

The output is shown in Figure 11-8.

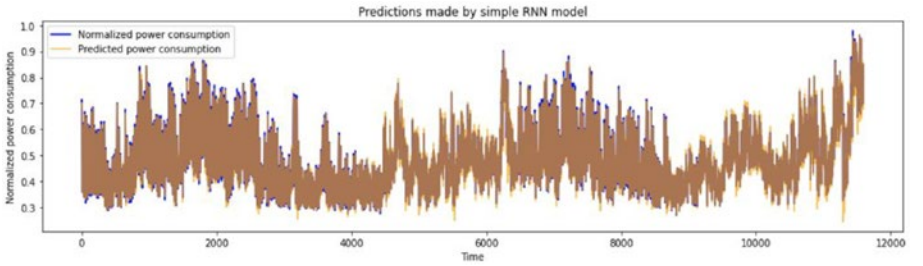


Figure 11-8. *Plot of actual vs. predictions on a normalized scale*

We see that the predicted values are close to the actual values, meaning the RNN model is performing well in predicting the energy consumption.

You may zoom in on the plot by using the provided sliders to select the data range. One such plot for a narrow range is shown in Figure 11-9.

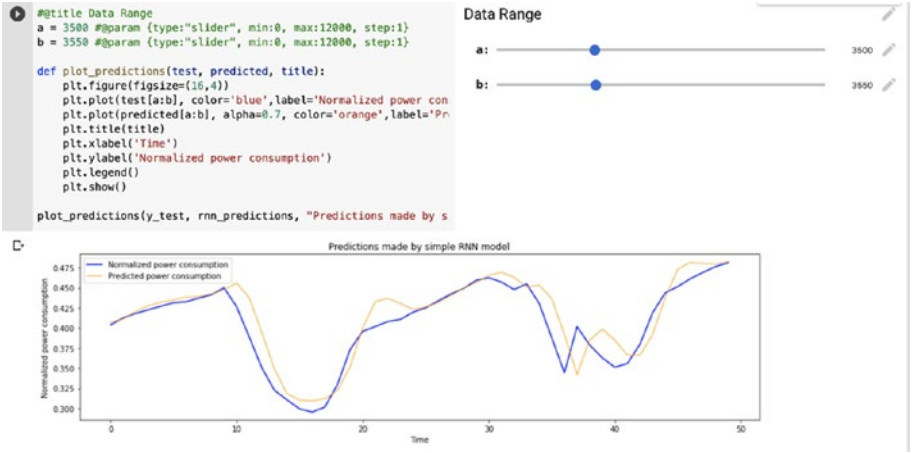


Figure 11-9. *Zooming in on the actual vs. prediction plot*

The prediction curve closely follows the expected target values. Thus, we are assured that the model works even in smaller data ranges.

You may zoom in on the end of the chart to see how the predictions look like using the following code:

```
history_data = list(y_test[-40:])
plottingvalues = list(history_data)+list
                          (rnn_predictions[:50])
plt.figure(figsize = (16,4))
plt.plot(plottingvalues, color = 'orange',
           label = 'forecasted value',marker = 'o')
plt.plot(y_test[-40:], color = 'green',
           label = 'history',marker = 'x')
plt.xlabel('Time')
plt.ylabel('Normalized power consumption scale')
plt.legend()
plt.show()
```

The output is shown in Figure 11-10.

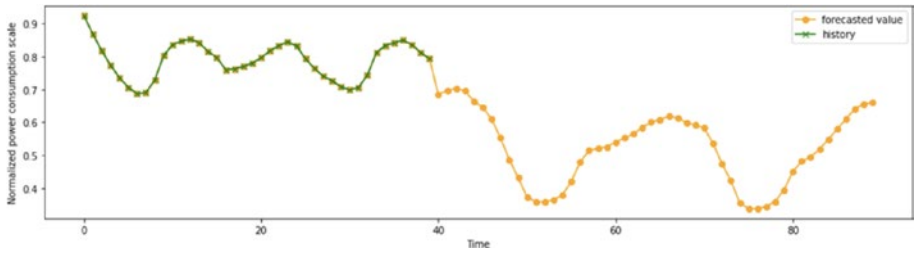


Figure 11-10. *Zooming in on the end of the prediction plot*

Predicting Next Data Point

Now, we will use our model to predict the next data point. For this, we extract the last data point from our test data and apply our predict function on it.

```
X = X_test[-1:]
rnn_predictions1 = rnn_model.predict(X)
```

You may check the predicted value by printing it on the console. The command and its output are shown as follows:

```
rnn_predictions1
```

```
array([[0.798944]], dtype = float32)
```

This is the prediction made by our model for the timestamp 2018:01:02 01:00:00 hours, as the last data point that we have in our dataset is for the timestamp 2018:01:02 00:00:00 hours.

To visualize the result in a better way, I will generate a plot for the last 40 data points along with the predicted value. This is done with the following code snippet:

```
history_data = list(y_test[-40:])
plottingvalues = list(history_data)+list
                          (rnn_predictions1)
plt.figure(figsize = (16,4))
plt.plot(plottingvalues, color = 'orange',
             label = 'forecasted value',marker = 'o')
plt.plot(y_test[-40:], color = 'green',
             label = 'history',marker = 'x')
plt.xlabel('Time')
plt.ylabel('Normalized power consumption scale')
plt.legend()
plt.show()
```

The output of the execution of the preceding code is shown in Figure 11-11.

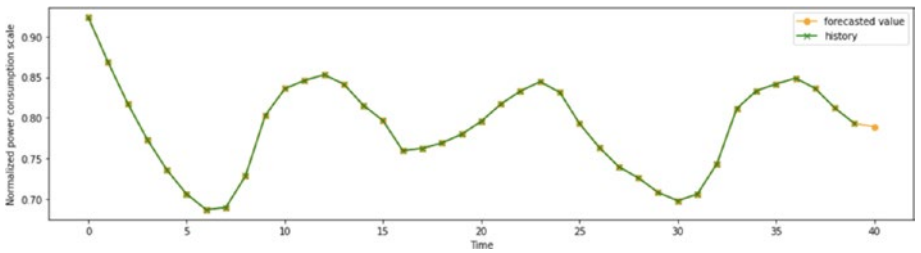


Figure 11-11. *Plot showing the prediction point*

Predicting Range of Data Points

Typically, you would be interested in forecasting the power consumption for a time range beyond the available data. Our last data point is of 2018:01:02 00:00:00 hours. Let us say that you would like to predict the next 25 data points. This is like a multipart regression. The trick for generating these predictions is to use a single prediction as data for the next test set and repeat the prediction for 25 times. I will show you how to do this with the actual code. First, we extract the last 40 data points from the test set.

```
history_data = list(y_test[-40:])
```

Then, we write a function to build a new dataset after adding the last prediction to it.

```
def make_data(X,rnn_predictions1):
 val = list(X[0][1:])+list
              (rnn_predictions1)
 X_new = []
 X_new.append(list(val))
 X_new = np.array(X_new)
 return X_new
```

We will create a list variable to store all our forecasts.

```
forecast = list()
```

We extract our last test data point as before to do the next data point prediction.

```
X = X_test[-1:]
```

We now define a loop to create the test data, make a prediction on it, and then add it to our forecast list.

```
for i in range (25):
 X = make_data(X,rnn_predictions1)
 rnn_predictions1 = rnn_model.predict(X)
 forecast += list(rnn_predictions1)
```

Finally, we plot all data points consisting of our history data and the next 25 predictions.

```
plottingvalues = list(history_data)+list(forecast)
plt.figure(figsize=(16,4))
plt.plot(plottingvalues, color = 'orange',
           label = 'forecasted value',marker = 'o')
plt.plot(y_test[-40:], color = 'green',
           label = 'history',marker = 'x')
plt.xlabel('Time (ticks)')
plt.ylabel('Normalized power consumption scale')
plt.legend()
plt.show()
```

The plot generated by the preceding code is shown in Figure 11-12.

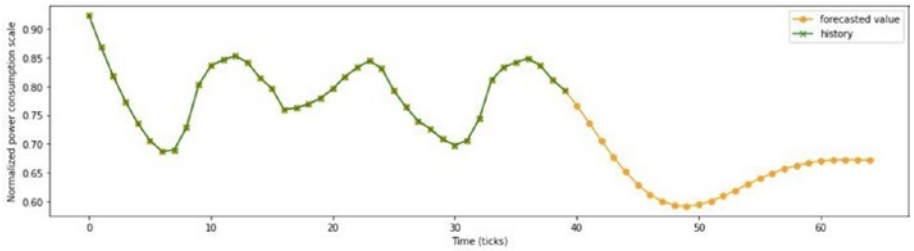


Figure 11-12. *Extended prediction plot*

Okay, you are able to predict the energy consumption for the next few hours. However, when you have the data covering the last 13 years, you would be interested in predicting the consumption for the next one week or next few more weeks. Using the technique of extrapolation that I have shown earlier may not do this job so good. We can now try training the model on the daily data so that we can make predictions for the next 25 days by extrapolation.

Remember, we already have written code for extracting daily consumption data. Just uncomment that code line which is shown again here for your quick reminder.

```
# uncomment the following line for generating daily data
# df = df[df['Datetime'].str.contains("00:00:00")]
```

Now, reset the environment and run the entire project. The prediction chart for the consumption a week after Jan 1, 2018 (our last data point) is shown in Figure 11-13.

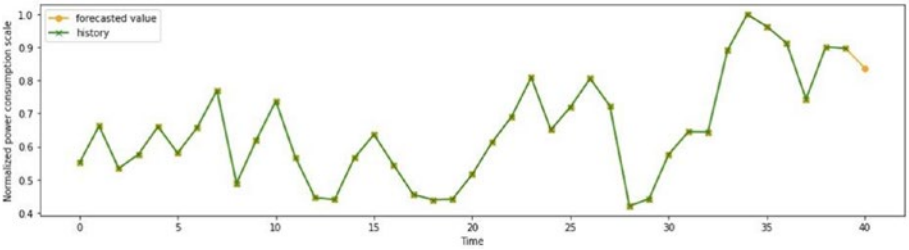


Figure 11-13. *Next week's prediction*

The extrapolated prediction for 25 weeks following Jan 31, 2018, is shown in Figure 11-14.

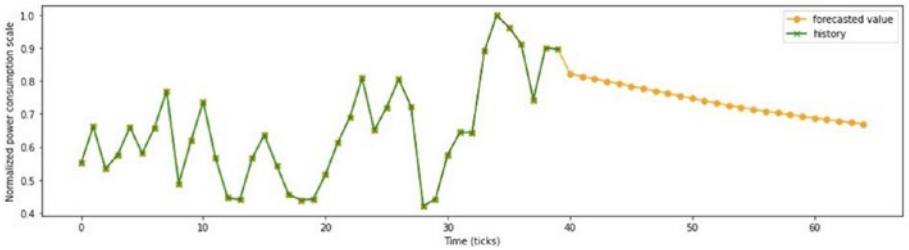


Figure 11-14. *Extrapolated predictions for 25 weeks*

To get a different set of predictions, you will need to carry out several such experiments, including a change of ANN configuration (additional layers, using SimpleRNN, using dropouts, etc.), training for more number of epochs, trying weekly and monthly datasets, and so on. As the future cannot be really predicted with a certain accuracy, you will need to select the most suitable prediction from these experiments.

Full Source

The entire program code is shown in Listing 11-2.

Listing 11-2. Univariate-time.ipynb

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn.preprocessing
from sklearn.metrics import r2_score
```

```
url = 'https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch11/DOM_hourly.csv'
df = pd.read_csv(url)

df

# uncomment the following line for generating daily data
#df = df[df['Datetime'].str.contains("00:00:00")]

df['Datetime'] = pd.to_datetime(df.Datetime ,
                      format = '%Y-%m-%d %H:%M:%S')
df.index = df.Datetime
df.drop(['Datetime'], axis = 1,inplace = True)

df.head()

df

#checking missing data
df.isna().sum()

#@title Date Range
a = '2005-12-31' #@param {type:"date"}
b = '2018-01-31' #@param {type:"date"}

a = a + " 00:00:00"
b = b + " 00:00:00"
df.loc[a:b].plot(figsize = (16,4),legend = True)

plt.title('DOM hourly power consumption data')
plt.ylabel('Power consumption (MW)')

plt.show()

df.shape

scaler = sklearn.preprocessing.MinMaxScaler()
```

```
df['DOM_MW'] = scaler.fit_transform(df['DOM_MW'].
                    values.reshape(-1,1))

df.plot(figsize = (16,4), legend = True)
plt.title('DOM hourly power consumption data -
                    AFTER NORMALIZATION')
plt.ylabel('Normalized power consumption')
plt.show()

def load_data(stock, seq_len):
    X_train = []
    y_train = []
    for i in range(seq_len, len(stock)):
        X_train.append(stock.iloc[i-seq_len : i, 0])
        y_train.append(stock.iloc[i, 0])

    X_test = X_train[int(0.9*(len(stock))):]
    y_test = y_train[int(0.9*(len(stock))):]

    X_train = X_train[:int(0.9*(len(stock)))]
    y_train = y_train[:int(0.9*(len(stock)))]

    # convert to numpy array
    X_train = np.array(X_train)
    y_train = np.array(y_train)

    X_test = np.array(X_test)
    y_test = np.array(y_test)

    # reshape data to input into RNN models
    X_train = np.reshape(X_train,
                    (X_train.shape[0], seq_len, 1))
    X_test = np.reshape(X_test,
                    (X_test.shape[0], seq_len, 1))
```

```
    return [X_train, y_train, X_test, y_test]

#create train, test data
seq_len = 20 #choose sequence length

X_train, y_train, X_test, y_test = load_data
                                    (df, seq_len)

print('X_train.shape = ',X_train.shape)
print('y_train.shape = ', y_train.shape)
print('X_test.shape = ', X_test.shape)
print('y_test.shape = ',y_test.shape)

batch_size  = 256
buffer_size = 1000

train_data = tf.data.Dataset.from_tensor_slices
                     ((X_train , y_train))
train_data = train_data.cache().shuffle(buffer_size).
                     batch(batch_size).repeat()

test_data = tf.data.Dataset.from_tensor_slices
                    ((X_test , y_test))
test_data = test_data.batch(batch_size).repeat()

rnn_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(8, input_shape =
                X_train.shape[-2:]),
    tf.keras.layers.Dense(1)
])

tf.keras.utils.plot_model(rnn_model)

rnn_model.compile(optimizer = 'adam', loss = 'mae')

EVALUATION_INTERVAL = 200
```

```
EPOCHS = 10
rnn_model.fit(train_data, epochs=EPOCHS,
                      steps_per_epoch =
                      EVALUATION_INTERVAL,
                      validation_data = test_data,
                      validation_steps = 50)

rnn_predictions = rnn_model.predict(X_test)
rnn_score = r2_score(y_test,rnn_predictions)
print("R2 Score of RNN model =
                "+"{:.4f}".format(rnn_score));

#@title Data Range
a = 0 #@param {type:"slider", min:0,
                     max:12000, step:1}
b = 12000 #@param {type:"slider", min:0,
                     max:12000, step:1}

def plot_predictions(test, predicted, title):
   plt.figure(figsize = (16,4))
   plt.plot(test[a:b], color = 'blue',
            label = 'Normalized power consumption')
   plt.plot(predicted[a:b], alpha = 0.7,
            color = 'orange',
            label = 'Predicted power consumption')
   plt.title(title)
   plt.xlabel('Time')
   plt.ylabel('Normalized power consumption')
   plt.legend()
   plt.show()

plot_predictions(y_test, rnn_predictions,
            "Predictions made by simple RNN model")
```

```
history_data = list(y_test[-40:])
plottingvalues = list(history_data)+
                      list(rnn_predictions[:50])
plt.figure(figsize = (16,4))
plt.plot(plottingvalues, color = 'orange',
           label = 'forecasted value',marker = 'o')
plt.plot(y_test[-40:], color = 'green',
           label = 'history',marker = 'x')
plt.xlabel('Time')
plt.ylabel('Normalized power consumption scale')
plt.legend()
plt.show()

X = X_test[-1:]

rnn_predictions1 = rnn_model.predict(X)

rnn_predictions1

history_data = list(y_test[-40:])

plottingvalues = list(history_data)+
                      list(rnn_predictions1)
plt.figure(figsize = (16,4))
plt.plot(plottingvalues, color = 'orange',
            label = 'forecasted value',marker = 'o')
plt.plot(y_test[-40:], color = 'green',
            label = 'history',marker = 'x')
plt.xlabel('Time')
plt.ylabel('Normalized power consumption scale')
plt.legend()
plt.show()

history_data = list(y_test[-40:])
```

```
def make_data(X,rnn_predictions1):
  val = list(X[0][1:])+list(rnn_predictions1)
  X_new = []
  X_new.append(list(val))
  X_new = np.array(X_new)
  return X_new

forecast = list()
X = X_test[-1:]

for i in range (25):
  X = make_data(X,rnn_predictions1)
  rnn_predictions1 = rnn_model.predict(X)
  forecast += list(rnn_predictions1)

plottingvalues = list(history_data)+list(forecast)
plt.figure(figsize = (16,4))
plt.plot(plottingvalues, color = 'orange',
             label = 'forecasted value',marker = 'o')
plt.plot(y_test[-40:], color = 'green',
             label = 'history',marker = 'x')
plt.xlabel('Time (ticks)')
plt.ylabel('Normalized power consumption scale')
plt.legend()
plt.show()
```

I will now move on to explain how to develop a model for performing multivariate analysis.

Multivariate Time Series Analysis

In this section, I will describe how to create a multivariate time series analysis machine learning model. You will use the London bike sharing dataset provided by Kaggle (www.kaggle.com/hmavrodiev/london-bike-sharing-dataset) for this purpose. The dataset provides a count of bike shares at a given time along with the weather conditions. It is observed that the demand for bikes is also seasonal. The aim of our model is to take into account all these variables in predicting the future demand for bikes. The various columns in the dataset are described as follows:

1. **timestamp**
2. **cnt** – The count of bike shares
3. **t1** – Real temperature in Celsius
4. **t2** – Temperature in Celsius "feels like"
5. **hum** – Humidity in percentage
6. **wind_speed** – Wind speed in km/h
7. **weather_code** – Weather category
8. **is_holiday** – Boolean, 1-holiday
9. **is_weekend** – Boolean, 1-weekend
10. **season** – Categorical: 0-spring, 1-summer, 2-fall, 3-winter

The cnt column is going to be our target for prediction. All the rest may be used as a feature. So this is a multivariate problem where the target depends on the values of nine other fields.

Creating Project

Create a Colab project and rename it to Multivariate time series analysis. Import the desired libraries as usual:

```
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
import sklearn.preprocessing
import seaborn as sns
```

Preparing Data

You load the data into the project using the following code:

```
url = 'https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch11/london_merged.csv'
df = pd.read_csv(url,parse_dates=['timestamp'],
                                        index_col = "timestamp")
```

Like in the earlier example, we will be using the timestamp column as date rather than an object. Examine the data. The screenshot is shown in Figure 11-15.

```
df
```

timestamp	cnt	t1	t2	hum	wind_speed	weather_code	is_holiday	is_weekend	season
2015-01-04 00:00:00	182	3.0	2.0	93.0	6.0	3.0	0.0	1.0	3.0
2015-01-04 01:00:00	138	3.0	2.5	93.0	5.0	1.0	0.0	1.0	3.0
2015-01-04 02:00:00	134	2.5	2.5	96.5	0.0	1.0	0.0	1.0	3.0
2015-01-04 03:00:00	72	2.0	2.0	100.0	0.0	1.0	0.0	1.0	3.0
2015-01-04 04:00:00	47	2.0	0.0	93.0	6.5	1.0	0.0	1.0	3.0
...	...	...	...	...	...	...	...	...	...
2017-01-03 19:00:00	1042	5.0	1.0	81.0	19.0	3.0	0.0	0.0	3.0
2017-01-03 20:00:00	541	5.0	1.0	81.0	21.0	4.0	0.0	0.0	3.0
2017-01-03 21:00:00	337	5.5	1.5	78.5	24.0	4.0	0.0	0.0	3.0
2017-01-03 22:00:00	224	5.5	1.5	76.0	23.0	4.0	0.0	0.0	3.0
2017-01-03 23:00:00	139	5.0	1.0	76.0	22.0	2.0	0.0	0.0	3.0

17414 rows × 9 columns

Figure 11-15. *Data for the multivariate time series analysis*

As you can see, there are 17,414 records and 9 columns. Note that we have taken out the timestamp column from the dataset and are using it as an index.

We will examine the data types by calling the dftypes method. The output is shown in Figure 11-16.

```
df.dtypes
```

```
cnt              int64
t1             float64
t2             float64
hum            float64
wind_speed     float64
weather_code   float64
is_holiday     float64
is_weekend     float64
season         float64
dtype: object
```

Figure 11-16. *Data types in the bike share database*

All our features columns are numeric type. However, a few columns like weather_code and season used categorical values. The columns is_holiday and is_weekend hold boolean values. We should not be scaling these columns when we scale all our numerical values for standardization.

Checking for Stationarity

We will now check if all columns in our analysis are stationary. A stationary series is one in which the properties – mean, variance, and covariance – do not vary with time. For a series to be stationary, the eigenvalues should be less than one in modulus. The Johansen test can be used to check for cointegration between a maximum of 12 time series. In the current dataset, we have nine time series, so we can apply this test easily without any further provisions in the coding. We conduct the test using the following code:

```
#checking stationarity
from statsmodels.tsa.vector_ar.vecm
                  import coint_johansen
johan_test_temp = df
coint_johansen(johan_test_temp,-1,1).eig
```

When you run the test, it prints the eigenvalues for all nine columns, which are shown as follows:

```
array([2.61219379e-01, 1.31970167e-01, 5.22046139e-02,
4.19830465e-02, 2.10126207e-02, 1.75450605e-02, 1.36518877e-02,
6.26085775e-04, 7.56291478e-05])
```

All eigenvalues are under one, implying that all time series under test are stationary. Had you found a nonstationary series, you will need to examine the list to check which one is causing the problem.

Exploring Data

Now, we will explore a few feature columns and the target value to understand its distribution. Let us start with the cnt column which represents the bike share count and is our target value in the analysis. We plot the cnt distribution using the following lines of code:

```
plt.figure(figsize = (16,4))
plt.plot(df.index, df["cnt"]);
```

The distribution is shown in Figure 11-17.

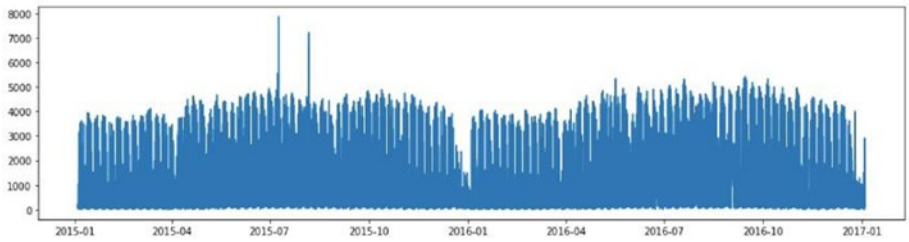


Figure 11-17. *Bike share data distribution for the entire period*

It looks like the demand is evenly distributed throughout the entire period. Now, let us examine if there are any seasonal changes in the demand distribution. We will first create indices for aggregated hourly and monthly data.

```
# create indexes
df['hour'] = df.index.hour
df['month'] = df.index.month
```

We then create three plots using the preceding indices for observing the weekend, holiday, and seasonal demands. We use the following plotting code to generate these plots:

```
fig,(ax1, ax2, ax3)= plt.subplots(nrows = 3)
fig.set_size_inches(16, 10)

sns.pointplot(data = df, x = 'month', y = 'cnt',
                  hue = 'is_weekend', ax = ax1)
sns.pointplot(data = df, x = 'hour', y = 'cnt',
                  hue = 'season', ax = ax2);
sns.pointplot(data = df, x = 'month', y = 'cnt',
                  ax = ax3)
```

The output is shown in Figure 11-18.

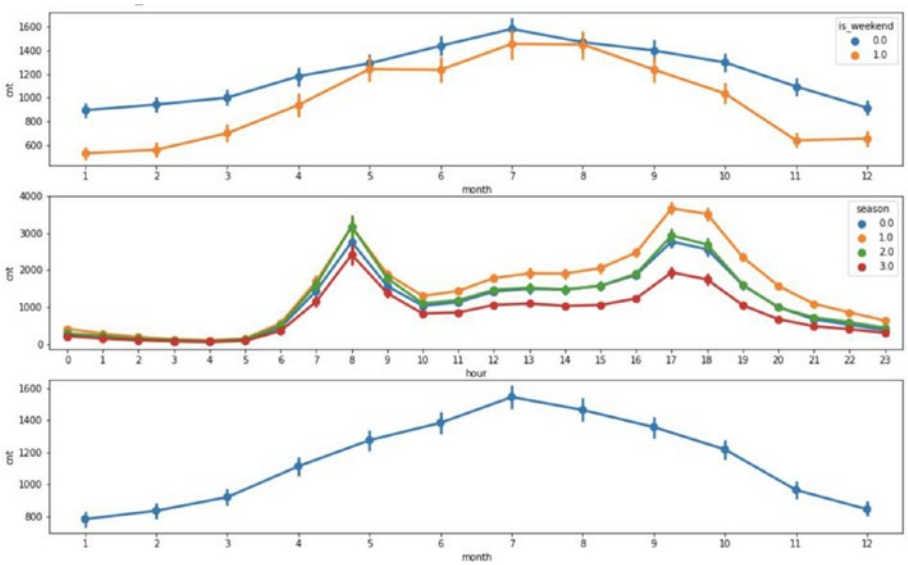


Figure 11-18. *Seasonal dependencies on bike share*

The first chart shows the bike share aggregated over for each month of the year during the weekends and working days. You see clearly that the demand is more during the month of July, every year irrespective of whether it is a weekend or not. The second chart shows demand aggregated for each hour of the day during the four categorical seasons listed in our database (0-spring, 1-summer, 2-fall, 3-winter). During every season, you observe the demand is more during the morning (8 a.m.) and evening (5–6 p.m.) hours. Finally, the last chart shows the aggregated monthly demand irrespective of the weekend, holiday, or seasons. It shows that the demand is more during the month of July and is lowest during the Jan–Dec period.

Having understood the demand patterns, let us now proceed with data cleansing and preparation.

Preparing Data

We scale all numeric columns by using MinMaxScaler:

```
# scaling numeric columns
scaler = sklearn.preprocessing.MinMaxScaler()
df['t1'] = scaler.fit_transform(df['t1'].
                  values.reshape(-1,1))
df['t2'] = scaler.fit_transform(df['t2'].
                  values.reshape(-1,1))
df['hum'] = scaler.fit_transform(df['hum'].
                  values.reshape(-1,1))
df['wind_speed'] = scaler.fit_transform
            (df['wind_speed'].values.reshape(-1,1))
df['cnt'] = scaler.fit_transform
            (df['cnt'].values.reshape(-1,1))
```

Note that we have eliminated the columns holding the categorical and boolean values from this preprocessing.

We will use 90% of data for training and the remaining for testing.

```
# use 90% for training
train_size = int(len(df) * 0.9)
test_size = len(df) - train_size
train, test = df.iloc[0:train_size],
                  df.iloc[train_size:len(df)]
```

We will now create tensors for inputting data to our network. For this, we write a function called create_dataset that we will use for both training and testing datasets.

```
def create_dataset(X, y, time_steps = 1):
   Xs, ys = [], []
   for i in range(len(X) - time_steps):
       v = X.iloc[i:(i + time_steps)].values
       Xs.append(v)
       ys.append(y.iloc[i + time_steps])
   return np.array(Xs), np.array(ys)
```

With the sequence length of 10, we create the training and testing datasets.

```
time_steps = 10
X_train, y_train =
         create_dataset(train, train.cnt, time_steps)
X_test, y_test =
         create_dataset(test, test.cnt, time_steps)
```

We slice datasets and create batches of data for an improved training:

```
batch_size  = 256
buffer_size = 1000

train_data = tf.data.Dataset.from_tensor_slices
                    ((X_train , y_train))
```

```
train_data = train_data.cache().shuffle
           (buffer_size).batch(batch_size).repeat()
test_data = tf.data.Dataset.from_tensor_slices
                     ((X_test , y_test))
test_data = test_data.batch(batch_size).repeat()
```

Creating Model

We define our network model using the following code:

```
simple_lstm_model = tf.keras.models.Sequential([
   tf.keras.layers.LSTM
           (8, input_shape = X_train.shape[-2:]),
   tf.keras.layers.Dense(1)
])
simple_lstm_model.compile
           (optimizer = 'adam', loss = 'mae')
```

The model is similar to the one used in the previous example.

Training

We train the model as usual by calling its fit method.

```
EVALUATION_INTERVAL = 200
EPOCHS = 10

history = simple_lstm_model.fit(
                  train_data,
                  epochs = EPOCHS,
                  steps_per_epoch = EVALUATION_INTERVAL,
                  validation_data = test_data,
                  validation_steps = 50)
```

After the training gets over, we plot the losses using the following code:

```
# plot losses
plt.plot(history.history['loss'],
           label = 'train loss')
plt.plot(history.history['val_loss'],
           label = 'validation loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
```

The output plot is shown in Figure 11-19.

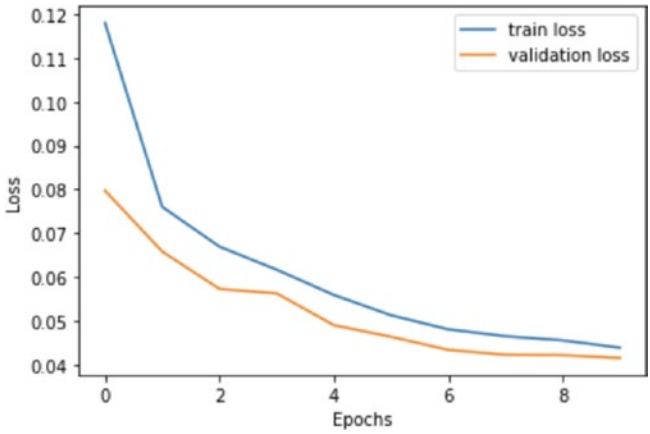


Figure 11-19. *Loss metrics*

The losses start flattening out just after the third epoch. No overfitting is observed. So we are okay with the model's training. Let us now evaluate its performance on the test data.

Evaluation

We evaluate the performance by calling the predict method on the test data.

```
X_test,y_test = create_dataset(df,df.cnt,10)
y_pred = simple_lstm_model.predict(X_test)
```

We will now plot our predictions to get some visualization. Write a small function for creating time steps of a specified length:

```
def create_time_steps(length):
 return list(range(-length, 0))
```

We then do the plotting using the following code:

```
plt.figure(figsize = (16,4))
num_in = create_time_steps(91)
num_out = 28
plt.plot(num_in,y_train[15571:],label = 'history')
plt.plot(np.arange(num_out),
        y_test[15661:15689], 'b',label='Actual ')
plt.plot(np.arange(num_out),
        y_pred[15661:15689], 'r',label = 'Predicted')
plt.xlabel("Time")
plt.ylabel("bike shares (Normalized Value)")
plt.legend()
plt.show()
```

The plot is shown in Figure 11-20.

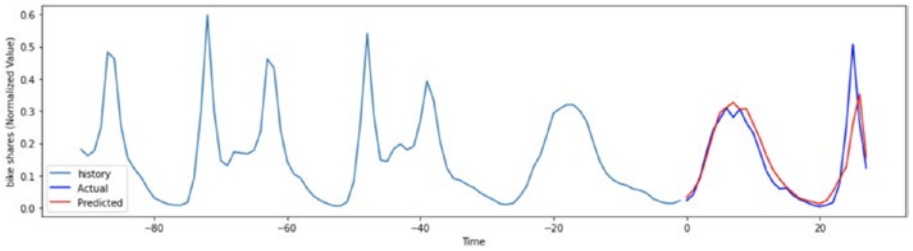


Figure 11-20. *Actual vs. predictions on bike share*

Predicting Future Point

With the model's evaluation completed, we will now predict the next bike share count in the future. We do this prediction with a simple statement:

```
y_pred = simple_lstm_model.predict(X_test[-1:])
```

We print the prediction value:

```
# print value
y_pred
```

The output on your console should be something like this:

```
array([[0.03246454]], dtype = float32)
```

We will now plot this prediction to get some visualization on where this data point fits. We generate the plot using the following code:

```
# plot prediction

# plot prediction
plt.figure(figsize = (16,4))
num_in = create_time_steps(100)
num_out = 1
plt.plot(num_in,y_test[-100:])
plt.plot(np.arange(num_out),y_pred, 'ro',
                    label = 'Predicted')
```

```
plt.xlabel("Time")
plt.ylabel("bike shares (Normalized Value)")
plt.legend()
plt.show()
```

The plot is shown in Figure 11-21.

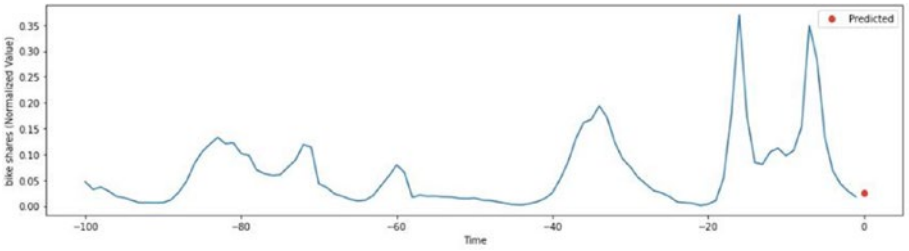


Figure 11-21. *Bike share prediction on the next data point*

Our prediction of the future data point looks okay. Can we do the prediction for more than a single data point? Let us try it.

Predicting Range of Data Points

Our last data point in the test set is for Jan 3, 2017, 23:00:00. We want to predict the demand for Jan 4, 2017, 00:00:00 onward for the next 100 points on an hourly basis. We will need features data for this period. So, what I am going to do is to first pick up a few entries from the test dataset.

```
df2 = df['2017-01-03 14:00:00':'2017-01-03 23:00:00']
```

In the preceding statement, df2 will now contain the test data from Jan 3, 2017, 14:00:00 hours through Jan 3, 2017, 23:00:00 hours. So, essentially, I have picked up the last ten data points of our dataset.

Now, I will pick up the features data from our original dataset during the last year.

```
df1 =  df['2016-01-04 00:00:00':'2016-01-06 23:00:00']
```

Note that I am picking up the data from Jan 4, 2016, 00:00:00 hours, for the next 3 days. We will make the cnt field to zero.

```
df1['cnt'] = 0
```

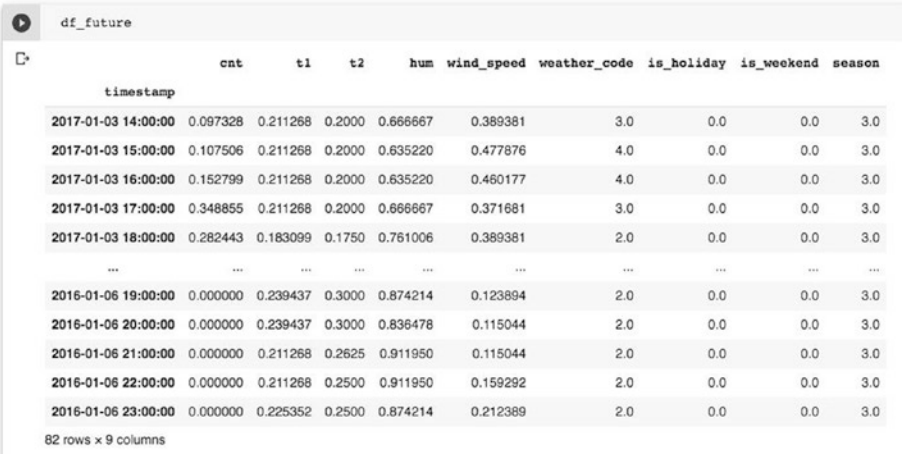
We now create a new dataframe by appending this past data to our current data:

```
df_future = df2.append(df1, sort = False)
```

You may examine the data by printing df_future contents:

```
df_future
```

The output is shown in Figure 11-22.



df_future

timestamp	cnt	t1	t2	hum	wind_speed	weather_code	is_holiday	is_weekend	season
2017-01-03 14:00:00	0.097328	0.211268	0.2000	0.666667	0.389381	3.0	0.0	0.0	3.0
2017-01-03 15:00:00	0.107506	0.211268	0.2000	0.635220	0.477876	4.0	0.0	0.0	3.0
2017-01-03 16:00:00	0.152799	0.211268	0.2000	0.635220	0.460177	4.0	0.0	0.0	3.0
2017-01-03 17:00:00	0.348855	0.211268	0.2000	0.666667	0.371681	3.0	0.0	0.0	3.0
2017-01-03 18:00:00	0.282443	0.183099	0.1750	0.761006	0.389381	2.0	0.0	0.0	3.0
...	...	...	...	...	...	...	...	...	...
2016-01-06 19:00:00	0.000000	0.239437	0.3000	0.874214	0.123894	2.0	0.0	0.0	3.0
2016-01-06 20:00:00	0.000000	0.239437	0.3000	0.836478	0.115044	2.0	0.0	0.0	3.0
2016-01-06 21:00:00	0.000000	0.211268	0.2625	0.911950	0.115044	2.0	0.0	0.0	3.0
2016-01-06 22:00:00	0.000000	0.211268	0.2500	0.911950	0.159292	2.0	0.0	0.0	3.0
2016-01-06 23:00:00	0.000000	0.225352	0.2500	0.874214	0.212389	2.0	0.0	0.0	3.0

82 rows × 9 columns

Figure 11-22. *New dataset for predictions*

As you can see, there is a discontinuity in the timestamp. For our analysis, this does not matter. For the sake of cleanliness, you may simply drop the index with the following statement:

```
#df_future  = df_future.reset_index(drop = True)
```

Now, we do the predictions in a continuous loop every time adding the predicted value to the df_future dataframe. The for loop for doing this is shown here:

```
predictions = []

# make prediction in a loop every time adding the last
prediction
for i in range(50):
 X_f, y_f = create_dataset
             (df_future, df_future.cnt, time_steps)
 y_pred = simple_lstm_model.predict(X_f[i:i+1])
 df_future['cnt'][i+10] = y_pred
 predictions.append(float(y_pred[0][0]))
```

I have used only 50 iterations; you may iterate through the entire data that we have created.

You may print the predictions array to examine all the predicted values. Better yet, we can visualize the predictions by creating a plot using the following code:

```
plt.figure(figsize = (16,4))
num_in = create_time_steps(100)
num_out = 50
plt.plot(num_in,y_test[-100:],label = 'History')
plt.plot(np.arange(num_out),predictions, 'r',
                    label = 'Predicted')
plt.xlabel("Time")
plt.ylabel("bike shares (Normalized Value)")
plt.legend()
plt.show()
```

The generated plot is shown in Figure 11-23.

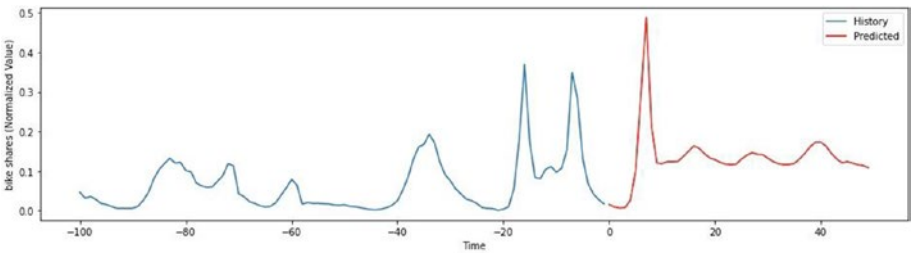


Figure 11-23. *A few weeks' prediction on bike shares*

This completes our discussion on how to create a deep neural network model for a multivariate time series analysis. Note that there are standard implementations available for the statistical models. For example, there is a VAR (vector autoregression) model available in the statsmodels library which can be used as shown in the following code:

```
from statsmodels.tsa.vector_ar.var_model import VAR
model = VAR(endog = train)
result = model.fit()
```

Once the training data is fitted into the model, it can be used for prediction by calling its forecast method.

```
prediction = result.forecast(validation_data.y,
                             steps = len(valid))
```

With the advent of neural networks, you may not find it necessary to understand the mathematics underlying such complex statistical modeling. Let the neural network learn itself and do the predictions for us.

Full Source

The entire program code is shown in Listing 11-3.

Listing 11-3. MultiVariate time.ipynb

```
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
import sklearn.preprocessing
import seaborn as sns

url = 'https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch11/london_merged.csv'
df = pd.read_csv(url,parse_dates=['timestamp'],
                                        index_col="timestamp")

df

df.dtypes

#checking stationarity
from statsmodels.tsa.vector_ar.vecm
                  import coint_johansen
johan_test_temp = df
coint_johansen(johan_test_temp,-1,1).eig

plt.figure(figsize = (16,4))
plt.plot(df.index, df["cnt"]);

# create indexes
df['hour'] = df.index.hour
df['month'] = df.index.month

fig,(ax1, ax2, ax3) = plt.subplots(nrows = 3)
fig.set_size_inches(16, 10)

sns.pointplot(data = df, x = 'month',
              y = 'cnt', hue = 'is_weekend', ax = ax1)
```

```
sns.pointplot(data = df, x = 'hour',
              y = 'cnt', hue = 'season', ax = ax2);
sns.pointplot(data = df, x = 'month',
              y = 'cnt', ax = ax3)

# scaling numeric columns
scaler = sklearn.preprocessing.MinMaxScaler()
df['t1'] = scaler.fit_transform(df['t1'].values.reshape(-1,1))
df['t2'] = scaler.fit_transform
               (df['t2'].values.reshape(-1,1))
df['hum'] = scaler.fit_transform
               (df['hum'].values.reshape(-1,1))
df['wind_speed'] = scaler.fit_transform
               (df['wind_speed'].values.reshape(-1,1))
df['cnt'] = scaler.fit_transform
               (df['cnt'].values.reshape(-1,1))

# use 90% for training
train_size = int(len(df) * 0.9)
test_size = len(df) - train_size
train, test = df.iloc[0:train_size],
               df.iloc[train_size:len(df)]

def create_dataset(X, y, time_steps = 1):
    Xs, ys = [], []
    for i in range(len(X) - time_steps):
        v = X.iloc[i:(i + time_steps)].values
        Xs.append(v)
        ys.append(y.iloc[i + time_steps])
    return np.array(Xs), np.array(ys)

# create input tensors
time_steps = 10
```

```
X_train, y_train = create_dataset
                   (train, train.cnt, time_steps)
X_test, y_test = create_dataset
                   (test, test.cnt, time_steps)

batch_size  = 256
buffer_size = 1000

train_data = tf.data.Dataset.from_tensor_slices
                     ((X_train , y_train))
train_data = train_data.cache().shuffle
             (buffer_size).batch(batch_size).repeat()
test_data = tf.data.Dataset.from_tensor_slices
              ((X_test , y_test))
test_data = test_data.batch(batch_size).repeat()

simple_lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(8, input_shape =
                    X_train.shape[-2:]),
    tf.keras.layers.Dense(1)
])
simple_lstm_model.compile(optimizer = 'adam',
                          loss = 'mae')

EVALUATION_INTERVAL = 200
EPOCHS = 10

history = simple_lstm_model.fit(
                  train_data,
                  epochs = EPOCHS,
                  steps_per_epoch = EVALUATION_INTERVAL,
                  validation_data = test_data,
                  validation_steps = 50)
```

```
# plot losses
plt.plot(history.history['loss'],
                         label = 'train loss')
plt.plot(history.history['val_loss'],
                         label = 'validation loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()

X_test,y_test = create_dataset(df,df.cnt,10)
y_pred = simple_lstm_model.predict(X_test)

def create_time_steps(length):
  return list(range(-length, 0))

plt.figure(figsize = (16,4))
num_in = create_time_steps(91)
num_out = 28
plt.plot(num_in,y_train[15571:],label = 'history')
plt.plot(np.arange(num_out),
         y_test[15661:15689], 'b',label='Actual ')
plt.plot(np.arange(num_out),
       y_pred[15661:15689], 'r',label = 'Predicted')
plt.xlabel("Time")
plt.ylabel("bike shares (Normalized Value)")
plt.legend()
plt.show()

y_pred = simple_lstm_model.predict(X_test[-1:])

# print value
y_pred

# plot prediction
```

```
plt.figure(figsize = (16,4))
num_in = create_time_steps(100)
num_out = 1
plt.plot(num_in,y_test[-100:])
plt.plot(np.arange(num_out),y_pred, 'ro',
         label ='Predicted')
plt.xlabel("Time")
plt.ylabel("bike shares (Normalized Value)")
plt.legend()
plt.show()
```

```
df2 = df['2017-01-03 14:00:00':'2017-01-03 23:00:00']
```

```
df1 =  df
       ['2016-01-04 00:00:00':'2016-01-06 23:00:00']
```

```
df1['cnt'] = 0
```

```
df_future = df2.append(df1, sort = False)
```

```
# dropping index is not truly required.
#df_future  = df_future.reset_index(drop = True)
```

```
df_future
```

```
predictions = []
```

```
# make prediction in a loop every time adding the last
prediction
for i in range(50):
  X_f, y_f = create_dataset
            (df_future, df_future.cnt, time_steps)
  y_pred = simple_lstm_model.predict(X_f[i:i+1])
  df_future['cnt'][i+10] = y_pred
  predictions.append(float(y_pred[0][0]))
```

```
predictions
```

```
plt.figure(figsize = (16,4))
num_in = create_time_steps(100)
num_out = 50
plt.plot(num_in,y_test[-100:],label = 'History')
plt.plot(np.arange(num_out),predictions, 'r',
         label = 'Predicted')
plt.xlabel("Time")
plt.ylabel("bike shares (Normalized Value)")
plt.legend()
plt.show()
```

Summary

In this chapter, you learned how to use neural networks in modeling both univariate and multivariate time series. In a univariate time series, the target depends only on one independent variable, while in the case of multivariate, there are multiple dependent variables on which the target value depends. For both types of analysis, there are widely accepted statistical techniques available for your use. However, with the advent of deep neural networks, it saves you lots of time and efforts in learning the theoretical part of these statistical models and implementing those in the code. Of course, there are standard libraries available for the purpose, but to customize them, you will need to understand the entire mathematics behind them. Neural networks allow you to implement such models with ease and also allow you to play around with different configurations to experiment on your data to achieve an acceptable level of performance.

In the next chapter, you will be studying style transfer.

CHAPTER 12

Style Transfer

Introduction

Ever wish you could paint like Picasso or the famous Indian painter M.F. Husain? It looks like neural networks have made your every wish come true. In this chapter, you would learn one such technique that uses neural networks to compose your own clicked picture in the style of a famous artist or rather in a style of your own choice. The technique is called neural style transfer, which is outlined in Leon A. Gatys' famous paper – "A Neural Algorithm of Artistic Style." Though the paper is a great read, you would not need all those details given in the paper to understand this chapter.

Neural style transfer is an optimization technique that blends the contents of an image in the style of another image. The TensorFlow Hub that you have learned to use in a previous chapter contains a pre-trained model for style transfers. First, I will show you how to use this model to get quickly started on your style transfer learning. This will be followed by a do-it-yourself example that will teach you how to extract the contents and the style from two different images and then perform the transformation on the content image to create another stylized image.

To give you a quick view of what you are going to achieve, look at Table 12-1.

P. Sarang, *Artificial Neural Networks with TensorFlow 2*,
https://doi.org/10.1007/978-1-4842-6150-7_12

Table 12-1. *The content, style, and stylized images*

Original	Image for Styling	Stylized image
		

The first image on the left is the content image, the image in the middle is the style image, and the image on the right is the stylized image. Note how the style of the middle image is applied to the contents of the image on the left to produce a new stylized image.

The theory behind the style transfer is trivial, and I will be covering it in the custom style transfer proGram later in this chapter.

So, let us get started with fast style transfer.

Fast Style Transfer

The TF Hub provides a pre-trained model for doing quick style transfers. The module name is "arbitrary-image-stylization-v1-256/2". This performs a fast artistic style transfer as compared to the original work for artistic style transfer with neural networks. The model may work on any arbitrary painting styles. The model is based on the technique proposed by Golnaz et al. in their famous paper "Exploring the structure of a real-time, arbitrary neural artistic stylization network" (`https://arxiv.org/abs/1705.06830`). The proposed model combines the flexibility of the neural algorithm of artistic style with the speed of fast style transfer

networks. This facilitates the real-time stylization using any content/style image pair.

You will be using this pre-trained model hosted on TensorFlow Hub for getting a quick understanding of the effects of style transfer.

Creating Project

Create a new Colab project and rename it to TFHubStyleTransfer. Import the required libraries.

```
import tensorflow as tf
import re
import urllib
import numpy as np
import matplotlib.pyplot as plt
import PIL.Image
import tensorflow_hub as hub
from tensorflow.keras.preprocessing.image
import load_img, img_to_array
from matplotlib import gridspec
from IPython import display
from PIL import Image
```

Downloading Images

The project requires two images at any given instance – the content image and the style image. The content image will be modified to adapt the style given in the style image. For your testing, I have uploaded a few images on the book's repository. The URL for an image takes the following form:

```
https://raw.githubusercontent.com/Apress/artificial-neural-
networks-with-tensorflow-2/main/ch12/ferns.jpg
```

Write a function to extract the filename, download the file, and set a new path to the image. The function definition is trivial and is shown in Listing 12-1.

Listing 12-1. Function for creating an image URL

```
def download_image_from_URL(imageURL):
  imageName = re.search('[a-z0-9\-]+\.
                        (jpe?g|png|gif|bmp|JPG)',
                        imageURL, re.IGNORECASE)
  imageName = imageName.group(0)
  urllib.request.urlretrieve(imageURL, imageName)
  imagePath = "./" + imageName
  return imagePath
```

Call this function to create a path to the target image.

```
# This is the path to the image you want to transform.
target_url = "https://raw.githubusercontent.com/Apress/
artificial-neural-networks-with-tensorflow-2/main/ch12/ferns.jpg"
target_path = download_image_from_URL(target_url)
```

Likewise, download the image for styling and set its path.

```
# This is the path to the style image.
style_url = "https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch12/on-the-road.jpg"
style_path = download_image_from_URL(style_url)
```

The user interface as seen in the Colab notebook is shown in Figure 12-1.

Figure 12-1. *Colab interface for selecting image files*

Select the desired target and style images from the drop-down list on the right-hand side of the cell.

You can display both the selected images using the matplotlib imshow function with the following code fragment:

```
content = Image.open(target_path)
style = Image.open(style_path)

plt.figure(figsize=(10, 10))

plt.subplot(1, 2, 1)
plt.imshow(content)
plt.title('Content Image')

plt.subplot(1, 2, 2)
plt.imshow(style)
plt.title('Style Image')

plt.tight_layout()
plt.show()
```

The output is shown in Figure 12-2.

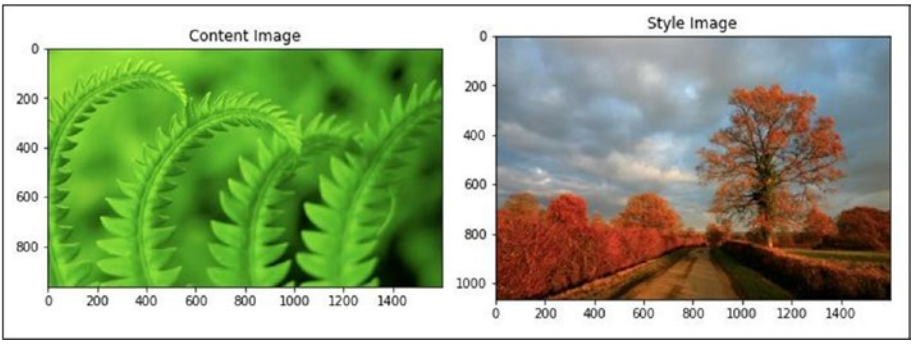


Figure 12-2. *The content and style images*

So, we have two images with different dimensions.

Preparing Images for Model Input

The module in tfhub that does the image transformation requires the images to be in a specific format for good results. First, we transform the style image to a tensor using the following function:

```
def image_to_tensor_style(path_to_img):
    img = tf.io.read_file(path_to_img)
    img = tf.image.decode_image
            (img, channels=3, dtype=tf.float32)
    img = tf.image.resize(img, [256,256])
    img = img[tf.newaxis, :]
    return img
```

The function reads the image data by calling the read_file method. It decodes the image by calling decode_image into three RGB channels. The image is resized to 256x256 as the pre-trained model uses this particular size for styling. Finally, the image data is returned to the caller as a tensor of shape (1, 256, 256, 3). We add a new dimension to the image which is used later while processing a batch of images.

Likewise, we write a function to convert the target image into a tensor as follows:

```
def image_to_tensor_target(path_to_img, image_size):
    img = tf.io.read_file(path_to_img)
    img = tf.image.decode_image
            (img, channels=3, dtype=tf.float32)
    img = tf.image.resize(img, [image_size,image_size],
                          preserve_aspect_ratio=True)
    img = img[tf.newaxis, :]
    return img
```

This function in addition to the image path takes an additional parameter, and that is the user-defined image size. The large images take up a lot of memory during processing, so I have added a parameter so that you can reduce the size of the image while maintaining its aspect ratio. Note the preserve_aspect_ratio parameter in the image resize method call. When the target image that you have loaded earlier is resized to 400, the output tensor will take the shape (1, 1200, 1600, 3). Note that the aspect ratio is maintained. The aspect ratio is the ratio of width to height of an image. If you set the width to 400, the height would be increased/decreased in proportion to this aspect ratio.

You will now convert both the images to tensors by calling the previously defined two methods:

```
output_image_size = 400
target_image = image_to_tensor_target
                    (target_path,output_image_size)
style_image = image_to_tensor_style(style_path)
```

Performing Styling

To apply a new style to the target image, you will need to load the pre-trained module from the tfhub. You do this using the following statement:

```
hub_module = hub.load('https://tfhub.dev/google/magenta/
arbitrary-image-stylization-v1-256/2')
```

The module name is arbitrary-image-stylization-v1-256/2. After the module is loaded, you would perform the transformation by simply sending the two tensors as inputs to the module as follows:

```
outputs = hub_module(tf.constant(target_image),
                     tf.constant(style_image))
stylized_image = outputs[0]
```

The outputs is a tensor of shape (1, 300, 400, 3). This is the data for our transformed image. Note the size 300x400 which is the scaled-down version of our original image of size 1600x1200.

Displaying Output

To display the image, we will need to convert the tensor to the image format using the following code:

```
tensor = stylized_image*256
tensor = np.array(tensor, dtype=np.uint8)
tensor = tensor[0]
PIL.Image.fromarray(tensor)
```

We need to multiply the image data to the scale of 256 as the stylized_ image contains the data that is in the scale of 0 to 1. The output image is shown in Figure 12-3.

Figure 12-3. *Stylized image*

If you want to display a scaled-down image, shown in Figure 12-4, simply call the imshow method as follows:

```
plt.imshow(tensor)
```

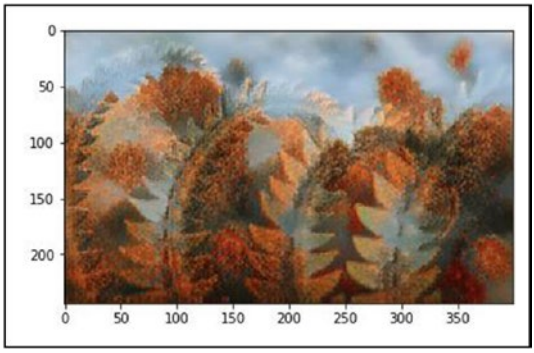


Figure 12-4. *Scaled-down stylized image*

Some More Results

I did a few more transformations on the images loaded in the project. The results are shown in Table 12-2.

Table 12-2. *Model inference on different images*

Original	Image for Styling	Stylized image
		
		
		
		
		

Full Source

The full source code for TFHubStyleTransfer is given in Listing 12-2.

Listing 12-2. TFHubStyleTransfer full source

```
import tensorflow as tf
import re
import urllib
import numpy as np
import matplotlib.pyplot as plt
import PIL.Image
import tensorflow_hub as hub
from tensorflow.keras.preprocessing.image
import load_img, img_to_array
from matplotlib import gridspec
from IPython import display
from PIL import Image

def download_image_from_URL(imageURL):
  imageName = re.search('[a-z0-9\-]+\.
                        (jpe?g|png|gif|bmp|JPG)',
                        imageURL, re.IGNORECASE)
  imageName = imageName.group(0)
  urllib.request.urlretrieve(imageURL, imageName)
  imagePath = "./" + imageName
  return imagePath

# This is the path to the image you want to transform.
target_url = "https://raw.githubusercontent.com/Apress/
artificial-neural-networks-with-tensorflow-2/main/ch12/ferns.jpg"
target_path = download_image_from_URL(target_url)
```

```
# This is the path to the style image.
style_url = "https://raw.githubusercontent.com/Apress/
artificial-neural-networks-with-tensorflow-2/main/ch12/on-the-
road.jpg"
style_path = download_image_from_URL(style_url)

content = Image.open(target_path)
style = Image.open(style_path)

plt.figure(figsize=(10, 10))

plt.subplot(1, 2, 1)
plt.imshow(content)
plt.title('Content Image')

plt.subplot(1, 2, 2)
plt.imshow(style)
plt.title('Style Image')

plt.tight_layout()
plt.show()

def image_to_tensor_style(path_to_img):
    img = tf.io.read_file(path_to_img)
    img = tf.image.decode_image
            (img, channels=3, dtype=tf.float32)
    img = tf.image.resize(img, [256,256])
    img = img[tf.newaxis, :]
    return img

def image_to_tensor_target(path_to_img, image_size):
    img = tf.io.read_file(path_to_img)
    img = tf.image.decode_image
            (img, channels=3, dtype=tf.float32)
    img = tf.image.resize(img,
```

```
                        [image_size,image_size],
                        preserve_aspect_ratio=True)
    img = img[tf.newaxis, :]
    return img

output_image_size = 400

target_image = image_to_tensor_target
                    (target_path,output_image_size)
style_image = image_to_tensor_style(style_path)

hub_module = hub.load('https://tfhub.dev/google/magenta/
arbitrary-image-stylization-v1-256/2')

outputs = hub_module(tf.constant(target_image),
                     tf.constant(style_image))
stylized_image = outputs[0]

tensor = stylized_image*256
tensor = np.array(tensor, dtype=np.uint8)
tensor = tensor[0]
PIL.Image.fromarray(tensor)

plt.imshow(tensor)
```

Do It Yourself

Having learned how to do a quick style transfer, it is time to learn the techniques behind these projects. The principle behind the style transfer is to extract the style of an image, usually a famous painting, and apply it to the contents of an image of your choice. Thus, there are two input images, namely, content image and style image. The newly generated image is generally called a stylized image. The generated image contains

the same contents as the content image, but has acquired the style similar to the *style* image. As you understand, this is obviously not done by just superimposing the images. So, our proGram must be able to separately distinguish between the content and the style of a given image. That is where we will use the VGG16 pre-trained network model to extract this information and build our own network to create a stylized image based on these inputs. The Android apps such as Prisma and Lucid do such style transfers. Though you will not be taught to develop a similar Android application, this project will teach you the internals of such apps.

Let us first look at the VGG16 architecture to understand how to extract the content and style from an image.

VGG16 Architecture

Gatys et al. (2015) came up with the core idea behind the style transfers. The core idea is to say that the CNNs (Convolutional Neural Networks) pre-trained for image classification know how to encode the perceptual and semantic information about an image. There are many such pre-trained CNNs available in the world. We will use VGG16 to extract the features of an image and then work independently on its content and style. The original paper uses the 19-layer VGG network model from Simonyan and Zisserman (2015). The VGG16 model architecture is shown in Figure 12-5.

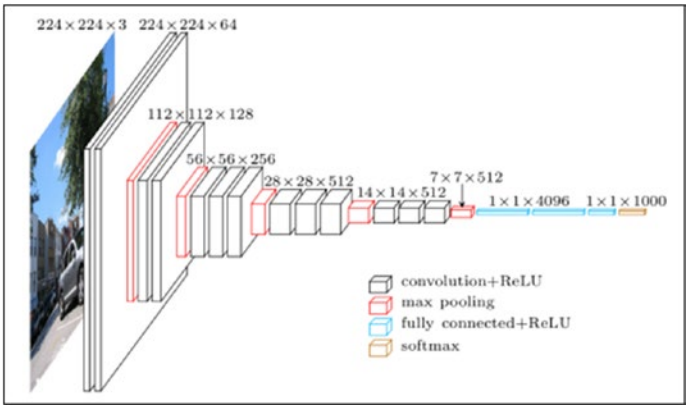


Figure 12-5. *The VGG16 architecture (Image credits: researchgate.net)*

As we are not doing an image classification and are interested in just the feature extraction, we do not need the fully connected layers or the final softmax classifier of the VGG network. We only need the part of the model. So, how do we extract only a certain portion of the model? Fortunately for us, this is a very trivial task as Keras provides a pre-trained VGG16 model in which you can separate out the layers. Keras does provide many other models including the latter VGG19. To remove the top most fully connected layers, you need to set the value of the include_top variable to False while extracting the model layers.

Creating Project

Create a new Colab project and rename it to CustomStyleTransfer. Install the following two packages:

```
!pip install keras==2.3.1
!pip install tensorflow==2.1.0
```

Note At the time of publishing, it was discovered that the pre-trained VGG16 model used in this project runs with the Keras and TensorFlow versions specified above and does not yet support the newer versions as of this writing.

Import the required libraries.

```
import tensorflow as tf
import re
import urllib
from tensorflow.keras.preprocessing.image
import load_img, img_to_array
from matplotlib import pyplot as plt
from IPython import display
from PIL import Image
import numpy as np
from tensorflow.keras.applications import vgg16
from tensorflow.keras import backend as K
from keras import backend as K
from scipy.optimize import fmin_l_bfgs_b
```

Downloading Images

As in the earlier project, you would write a download function and call it to download the two required images for the project. The code is given in Listing 12-3.

Listing 12-3. Function for downloading images

```
def download_image_from_URL(imageURL):
  imageName = re.search
          ('[a-z0-9\-]+\.(jpe?g|png|gif|bmp|JPG)',
                imageURL, re.IGNORECASE)
```

```
  imageName = imageName.group(0)
  urllib.request.urlretrieve(imageURL, imageName)
  imagePath = "./" + imageName
  return imagePath

# This is the path to the image you want to transform.
target_url = "https://raw.githubusercontent.com/Apress/
artificial-neural-networks-with-tensorflow-2/main/ch12/blank-
sign.jpg"
target_path = download_image_from_URL(target_url)
# This is the path to the style image.
style_url = "https://raw.githubusercontent.com/Apress/
artificial-neural-networks-with-tensorflow-2/main/ch12/road.jpg"
```

We would scale the target image to have a height of 400 pixels. To maintain the aspect ratio, we recompute the width as follows:

```
width, height = load_img(target_path).size
img_height = 400
img_width = int(width * img_height / height)
```

Displaying Images

To display the two images, we use a code similar to the one in the previous project. The code is given in Listing 12-4.

Listing 12-4. Displaying content and style images

```
content = Image.open(target_path)
style = Image.open(style_path)

plt.figure(figsize=(10, 10))
```

```
plt.subplot(1, 2, 1)
plt.imshow(content)
plt.title('Content Image')

plt.subplot(1, 2, 2)
plt.imshow(style)
plt.title('Style Image')

plt.tight_layout()
plt.show()
```

The output is shown in Figure 12-6.

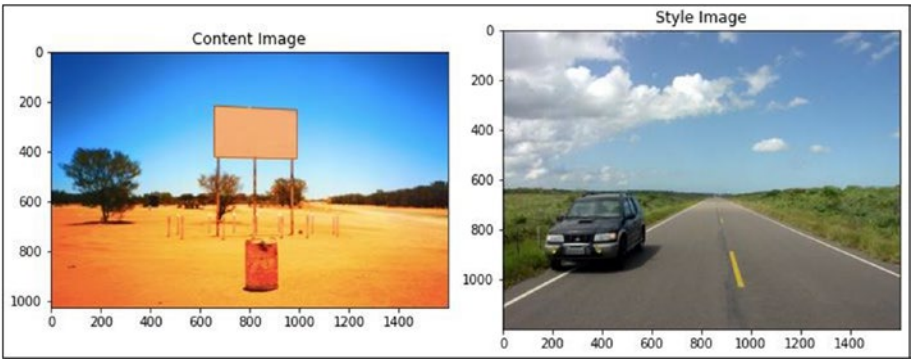


Figure 12-6. *Content and style images*

Preprocessing Images

As said earlier, you will be using the VGG16 model for extracting the features in an image. We need to process our image data as per the VGG training process. Fortunately, Keras provides this preprocessing not just for VGG16, but many other popular models such as ResNet, Inception, DenseNet, and so on. The library provides a function called preprocess_input that takes a tensor or numpy array encoding a batch of images as an input and returns a preprocessed numpy array or a tf.tensor with type float32. The method

converts the image from RGB to BGR and zero-centers each channel. Note that the VGG networks were trained on images with each channel normalized by mean = [103.939, 116.779, 123.68] and having channels BGR (Blue/Green/Red). The code in Listing 12-5 performs this preprocessing on a given image.

Listing 12-5. Preprocessing image for the VGG16 network

```
def preprocess_image(image_path):
    img = load_img(image_path,
            target_size=(img_height, img_width))
    img = img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img =
  tf.keras.applications.vgg16.preprocess_input(img)
    return img
```

If you wish to view the outputs, we need to do the reverse preprocessing. Also, we must clip all the values within the 0–255 range. We do this in the following function definition:

```
def deprocess_image(x):
    # Remove zero-center by mean pixel
    x[:, :, 0] += 103.939
    x[:, :, 1] += 116.779
    x[:, :, 2] += 123.68
    # 'BGR'->'RGB'
    x = x[:, :, ::-1]
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

You will now build the model based on the VGG16 model.

Model Building

To build the model, we feed the VGG16 with our image tensor data and extract the feature maps, the content, and style representations. The model will be loaded with pre-trained ImageNet weights. The model building code is given here:

```
target = K.constant(preprocess_image(target_path))
style = K.constant(preprocess_image(style_path))

# This placeholder will contain our generated image
combination_image = K.placeholder
                      ((1, img_height, img_width, 3))

# We combine the 3 images into a single batch
input_tensor = K.concatenate([target,
                              style,
                              combination_image],
                                  axis=0)

# Build the VGG16 network with our batch of 3 images as input.
model = vgg16.VGG16(input_tensor=input_tensor,
                    weights='imagenet',
                    include_top=False)
```

In this code, we first build the input tensor for our content and style images by calling our earlier defined preprocess method. We create a placeholder for the destination image and then create a tensor for three images by calling the concatenate method. This is then used as a parameter to the VGG16 method to extract our desired model.

You can now examine the model summary.

```
model.summary()
```

The summary output is shown in Figure 12-7.

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(3, 400, 533, 3)]        0
_________________________________________________________________
block1_conv1 (Conv2D)        (3, 400, 533, 64)         1792
_________________________________________________________________
block1_conv2 (Conv2D)        (3, 400, 533, 64)         36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (3, 200, 266, 64)         0
_________________________________________________________________
block2_conv1 (Conv2D)        (3, 200, 266, 128)        73856
_________________________________________________________________
block2_conv2 (Conv2D)        (3, 200, 266, 128)        147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (3, 100, 133, 128)        0
_________________________________________________________________
block3_conv1 (Conv2D)        (3, 100, 133, 256)        295168
_________________________________________________________________
block3_conv2 (Conv2D)        (3, 100, 133, 256)        590080
_________________________________________________________________
block3_conv3 (Conv2D)        (3, 100, 133, 256)        590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (3, 50, 66, 256)          0
_________________________________________________________________
block4_conv1 (Conv2D)        (3, 50, 66, 512)          1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (3, 50, 66, 512)          2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (3, 50, 66, 512)          2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (3, 25, 33, 512)          0
_________________________________________________________________
block5_conv1 (Conv2D)        (3, 25, 33, 512)          2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (3, 25, 33, 512)          2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (3, 25, 33, 512)          2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (3, 12, 16, 512)          0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

Figure 12-7. *Model summary*

Content Loss

We will compute the content loss at each desired layer and add them up. At each iteration, we feed our input image to the model. The model will compute all the content losses properly, and as we have an eager execution, all gradients will also be computed. The content loss is an indication of how similar the randomly generated noisy image (G) is to the content image (C). The content loss is computed as follows.

Assume that we choose a hidden layer (L) in a pre-trained network (VGG network) to compute the loss. Let P and F represent the original image and the image that is generated. Let F[l] and P[l] be the feature representation of the respective images in layer L. Then, the content loss is defined as follows:

$$L_{content}\left(\vec{P},\vec{X},l\right)=\frac{1}{2}\sum_{ij}\left(F_{ij}^{l}-P_{ij}^{l}\right)^{2}$$

We code this formula of the content loss as follows:

```
def content_loss(base, combination):
    return K.sum(K.square(combination - base))
```

Style Loss

To compute the style loss, we first need to compute the Gram matrix. The Gram matrix is an additional preprocessing step that is added to find the correlation between the different channels which will later be used for the measure of the style itself.

We define the Gram matrix as follows:

```
def gram_matrix(x):
    features = K.batch_flatten
                (K.permute_dimensions(x, (2, 0, 1)))
    gram = K.dot(features, K.transpose(features))
    return gram
```

The style loss computes the Gram matrix for both the style and the generated image and then returns the cost to the caller. The cost is the square of the difference between the Gram matrix of the style and the Gram matrix of the generated image. Mathematically, this is expressed as

$$L_{GM}(S,G,l) = \frac{1}{4N_l^2 M_l^2} \sum_{ij} \left(GM[l](S)_{ij} - GM[l](G)_{ij} \right)^2$$

The definition of the style loss function is given here:

```
def style_loss(style, combination):
    S = gram_matrix(style)
    C = gram_matrix(combination)
    channels = 3
    size = img_height * img_width
    return K.sum(K.square(S - C)) /
                (4. * (channels ** 2) * (size ** 2))
```

Total Variation Loss

To regularize the output for smoothness, we define a total variation loss in adjacent pixels as follows:

```
def total_variation_loss(x):
    a = K.square(
        x[:, :img_height - 1, :img_width - 1,
                :] - x[:, 1:, :img_width - 1, :])
    b = K.square(
        x[:, :img_height - 1, :img_width - 1,
                :] - x[:, :img_height - 1, 1:, :])
    return K.sum(K.pow(a + b, 1.25))
```

Computing Losses for Content and Style

We first select the VGG16 content and style layers. I have used the layers defined in Johnson et al. (2016) rather than the ones suggested by Gatys et al. (2015) because this produces a better end result.

First, we map all the layers to a dictionary.

```
# Dict mapping layer names to activation tensors
outputs_dict = dict([(layer.name, layer.output)
                     for layer in model.layers])
```

We extract the content layer:

```
# Name of layer used for content loss
content_layer = 'block5_conv2'
```

We extract the style layers:

```
# Name of layers used for style loss;
style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1']
```

We define a few weight variables to be used in the computation of the weighted average of the loss components. Think of them as the hyperparameters for the style and content layers that decide the weightage given to these layers in the final model.

```
total_variation_weight = 1e-4
style_weight = 10.
content_weight = 0.025
```

We compute the total loss by adding all components.

```
# Define the loss by adding all components to a `loss` variable
loss = K.variable(0.)
layer_features = outputs_dict[content_layer]
target_features = layer_features[0, :, :, :]
combination_features = layer_features[2, :, :, :]
loss = loss + content_weight *
            content_loss(target_features,
                         combination_features)
for layer_name in style_layers:
    layer_features = outputs_dict[layer_name]
    style_reference_features = layer_features
                                [1, :, :, :]
    combination_features = layer_features
                                [2, :, :, :]
    sl = style_loss(style_reference_features,
                            combination_features)
    loss += (style_weight / len(style_layers)) * sl
    loss += total_variation_weight *
             total_variation_loss(combination_image)
```

Evaluator Class

Lastly, we will define a class called Evaluator to compute the loss and the gradients in one pass.

```
grads = K.gradients(loss, combination_image)[0]
# Function to fetch the values of the current loss and the
current gradients
fetch_loss_and_grads =
                 K.function([combination_image],
                       [loss, grads])
```

```
class Evaluator(object):

    def __init__(self):
        self.loss_value = None
        self.grads_values = None

    def loss(self, x):
        assert self.loss_value is None
        x = x.reshape((1, img_height, img_width, 3))
        outs = fetch_loss_and_grads([x])
        loss_value = outs[0]
        grad_values =
                outs[1].flatten().astype('float64')
        self.loss_value = loss_value
        self.grad_values = grad_values
        return self.loss_value

    def grads(self, x):
        assert self.loss_value is not None
        grad_values = np.copy(self.grad_values)
        self.loss_value = None
        self.grad_values = None
        return grad_values

evaluator = Evaluator()
```

Generating Output Image

Now that we have all the utility functions ready, it is time to generate a stylized image. We start with a random collection of pixels (a random image) and use the L-BFGS (Limited Memory Broyden-Fletcher-Goldfarb-Shanno) algorithm for optimization. The algorithm uses the second derivative to minimize or maximize the function and is significantly quicker than the standard gradient descent. The training loop is shown here:

```
iterations = 50

x = preprocess_image(target_path)
x = x.flatten()
for i in range(1, iterations):
    x, min_val, info = fmin_l_bfgs_b
                       (evaluator.loss,
                       x,
                       fprime=evaluator.grads,
                                    maxfun=10)
    print('Iteration %0d, loss: %0.02f' %
                           (i, min_val))
img = x.copy().reshape((img_height, img_width, 3))
img = deprocess_image(img)
```

At the end of the training, we copy the final output image in a variable and reprocess it to make it ready for display.

Displaying Images

We now display all three images using the following code:

```
plt.figure(figsize=(50, 50))

plt.subplot(3,3,1)
plt.imshow(load_img(target_path, target_size=(img_height,
                                              img_width)))

plt.subplot(3,3,2)
plt.imshow(load_img(style_path, target_size=(img_height,
                                             img_width)))
```

```
plt.subplot(3,3,3)
plt.imshow(img)

plt.show()
```

The output image is shown in Figure 12-8.

Figure 12-8. *The content, style, and stylized images*

Full Source

The full source code for CustomStyleTransfer is given in Listing 12-6.

Listing 12-6. CustomStyleTransfer full source

```
!pip install keras==2.3.1
!pip install tensorflow==2.1.0

import tensorflow as tf
import re
import urllib
from tensorflow.keras.preprocessing.image
import load_img, img_to_array
from matplotlib import pyplot as plt
from IPython import display
from PIL import Image
import numpy as np
from tensorflow.keras.applications import vgg16
from tensorflow.keras import backend as K
```

```
from keras import backend as K
from scipy.optimize import fmin_l_bfgs_b

def download_image_from_URL(imageURL):
  imageName = re.search
          ('[a-z0-9\-]+\.(jpe?g|png|gif|bmp|JPG)',
                imageURL, re.IGNORECASE)
  imageName = imageName.group(0)
  urllib.request.urlretrieve(imageURL, imageName)
  imagePath = "./" + imageName
  return imagePath

# This is the path to the image you want to transform.
target_url = "https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch12/blank-sign.jpg"
target_path = download_image_from_URL(target_url)
# This is the path to the style image.
style_url = "https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch12/road.jpg"
style_path = download_image_from_URL(style_url)

# Dimensions for the generated picture.
width, height = load_img(target_path).size
img_height = 400
img_width = int(width * img_height / height)

content = Image.open(target_path)
style = Image.open(style_path)

plt.figure(figsize=(10, 10))

plt.subplot(1, 2, 1)
plt.imshow(content)
plt.title('Content Image')
```

```
plt.subplot(1, 2, 2)
plt.imshow(style)
plt.title('Style Image')

plt.tight_layout()
plt.show()

# Preprocess the data as per VGG16 requirements
def preprocess_image(image_path):
    img = load_img(image_path,
                target_size=(img_height, img_width))
    img = img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img =
  tf.keras.applications.vgg16.preprocess_input(img)
    return img

def deprocess_image(x):
    # Remove zero-center by mean pixel
    x[:, :, 0] += 103.939
    x[:, :, 1] += 116.779
    x[:, :, 2] += 123.68
    # 'BGR'->'RGB'
    x = x[:, :, ::-1]
    x = np.clip(x, 0, 255).astype('uint8')
    return x

target = K.constant(preprocess_image(target_path))
style = K.constant(preprocess_image(style_path))

# This placeholder will contain our generated image
combination_image = K.placeholder
                      ((1, img_height, img_width, 3))
```

```
# We combine the 3 images into a single batch
input_tensor = K.concatenate([target, style, combination_
                              image], axis=0)

# Build the VGG16 network with our batch of 3 images as input.
model = vgg16.VGG16(input_tensor=input_tensor,
                    weights='imagenet',
                    include_top=False)

model.summary()

# compute content loss for the generated image
def content_loss(base, combination):
    return K.sum(K.square(combination - base))

def gram_matrix(x):
    features = K.batch_flatten
                (K.permute_dimensions(x, (2, 0, 1)))
    gram = K.dot(features, K.transpose(features))
    return gram

def style_loss(style, combination):
    S = gram_matrix(style)
    C = gram_matrix(combination)
    channels = 3
    size = img_height * img_width
    return K.sum(K.square(S - C)) /
                (4. * (channels ** 2) * (size ** 2))

def total_variation_loss(x):
    a = K.square(
        x[:, :img_height - 1, :img_width - 1,
                :] - x[:, 1:, :img_width - 1, :])
```

```
    b = K.square(
        x[:, :img_height - 1, :img_width - 1,
                :] - x[:, :img_height - 1, 1:, :])
    return K.sum(K.pow(a + b, 1.25))

# Dict mapping layer names to activation tensors
outputs_dict = dict([(layer.name, layer.output)
                     for layer in model.layers])

# Name of layer used for content loss
content_layer = 'block5_conv2'

# Name of layers used for style loss;
style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1']

# Weights in the weighted average of the loss components
total_variation_weight = 1e-4
style_weight = 10.
content_weight = 0.025

# Define the loss by adding all components to a `loss` variable
loss = K.variable(0.)
layer_features = outputs_dict[content_layer]
target_features = layer_features[0, :, :, :]
combination_features = layer_features[2, :, :, :]
loss = loss + content_weight *
            content_loss(target_features,
                         combination_features)
for layer_name in style_layers:
    layer_features = outputs_dict[layer_name]
```

```
    style_reference_features = layer_features
                                [1, :, :, :]
    combination_features = layer_features
                                [2, :, :, :]
    sl = style_loss(style_reference_features,
                            combination_features)
    loss += (style_weight / len(style_layers)) * sl
    loss += total_variation_weight *
             total_variation_loss(combination_image)

grads = K.gradients(loss, combination_image)[0]

# Function to fetch the values of the current loss and the
current gradients
fetch_loss_and_grads =
                  K.function([combination_image],
                              [loss, grads])

class Evaluator(object):

    def __init__(self):
        self.loss_value = None
        self.grads_values = None

    def loss(self, x):
        assert self.loss_value is None
        x = x.reshape((1, img_height, img_width, 3))
        outs = fetch_loss_and_grads([x])
        loss_value = outs[0]
        grad_values =
                outs[1].flatten().astype('float64')
        self.loss_value = loss_value
        self.grad_values = grad_values
```

```
        return self.loss_value

    def grads(self, x):
        assert self.loss_value is not None
        grad_values = np.copy(self.grad_values)
        self.loss_value = None
        self.grad_values = None
        return grad_values

evaluator = Evaluator()

iterations = 50

x = preprocess_image(target_path)
x = x.flatten()
for i in range(1, iterations):
    x, min_val, info = fmin_l_bfgs_b
                       (evaluator.loss,
                       x,
                       fprime=evaluator.grads,
                                    maxfun=10)
    print('Iteration %0d, loss: %0.02f' %
                           (i, min_val))
img = x.copy().reshape((img_height, img_width, 3))
img = deprocess_image(img)

plt.figure(figsize=(50, 50))

plt.subplot(3,3,1)
plt.imshow(load_img(target_path, target_size=(img_height,
                                              img_width)))

plt.subplot(3,3,2)
```

```
plt.imshow(load_img(style_path, target_size=(img_height,
                                             img_width)))

plt.subplot(3,3,3)
plt.imshow(img)

plt.show()
```

Summary

In this chapter, you learned another important technique in neural networks, and that is neural style transfer. The technique allows you to transform the contents of an image of your choice into a style of another image. You learned how to do this style transfer in two different ways. The first one was to use a pre-trained model provided in tfhub to perform a fast artistic style transfer. The style transfers using this method are quick and do a very good job. The second approach was to build your own network to do the style transfers at the core level. We used a VGG16 pre-trained model for image classification to extract the contents and style of images. You then learned to create a network which learned how to apply the style to the given contents over several iterations. This method allows you to perform your own experiments on style transfer.

In the next chapter, you will learn how to generate images using Generative Adversarial Networks (GANs).

CHAPTER 13

Image Generation

Did you ever imagine that neural networks could be used for generating complex color images? How about Anime? How about the faces of celebrities? How about a bedroom? Doesn't it sound interesting? All these are possible with the most interesting idea in neural networks and that is Generative Adversarial Networks (GANs). The idea was introduced and developed by Ian J. Goodfellow in 2014. The images created by GAN look so real that it becomes practically impossible to differentiate between a fake and a real image. Be warned, to generate complex images of this nature, you would require lots of resources to train the network, but it does certainly work as you would see when you study this chapter. So let us look at what is GAN.

GAN – Generative Adversarial Network

In GAN, there are two neural network models which are trained simultaneously by an adversarial process. One network is called a generator, and the other is called a discriminator. A generator (an artist) learns to create images that look real. A discriminator (the critic) learns to tell real images apart from the fakes. So these are two competing models which try to beat each other. At the end, if you are able to train the generator to outperform the discriminator, you would have achieved your goal.

P. Sarang, *Artificial Neural Networks with TensorFlow 2*,
https://doi.org/10.1007/978-1-4842-6150-7_13

How Does GAN Work?

As I said, the GAN consists of two networks. Training a GAN requires two parts:

1. Keeping the generator idle, train the discriminator. Train the discriminator on real images for a number of epochs and see if it can correctly predict them as real. In the same training phase, train the discriminator on the fake images (generated by the generator) and see if it can predict them as fake.
2. Keeping the discriminator idle, train the generator. Use the prediction results made by the discriminator on the fake images to improve upon those images.

The preceding steps are repeated for a large number of epochs, and the results (fake images) are examined manually to see if they look real. If they do, stop the training. If not, continue the preceding two steps until the fake image looks real.

The whole process is depicted in Figure 13-1.

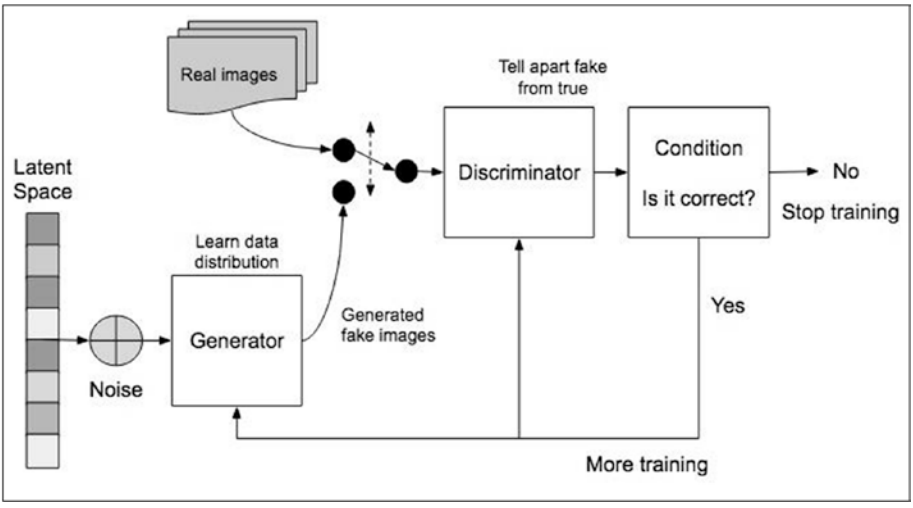


Figure 13-1. *GAN working schematic*

The Generator

The generator schematic is shown in Figure 13-2.

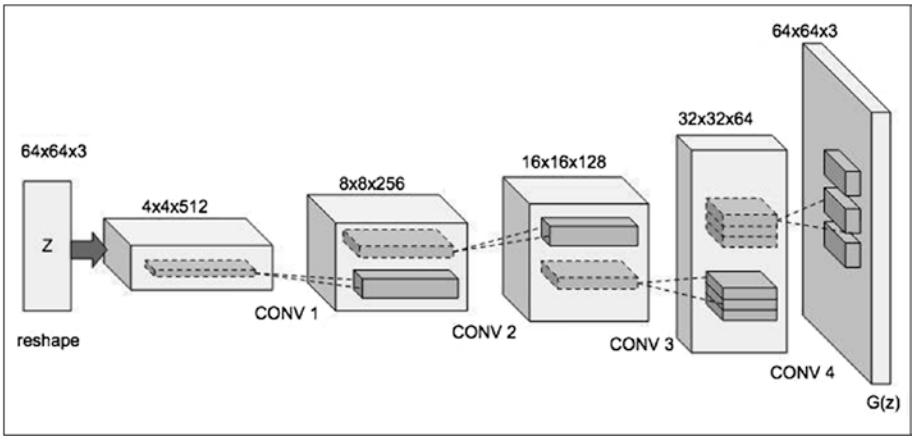


Figure 13-2. *Generator architecture*

The generator takes a random noise vector of some dimension. We will be using a dimension of 100 in our examples. From this random vector, it generates an image of 64x64x3. The image is upscaled by a series of transitions through convolutional layers. Each convolutional transpose layer is followed by a batch normalization and a leaky ReLU. The leaky ReLU has neither vanishing gradient problems nor dying ReLU problems. The leaky ReLU attempts to fix the "dying ReLU" problem. The ReLU will acquire a small value of 0.01 or so instead of dying out to 0. We use strides in each convolutional layer to avoid unstable training.

Note how, at each convolution, the image is upscaled to create a final image of 64x64x3.

The Discriminator

The discriminator schematic is shown in Figure 13-3.

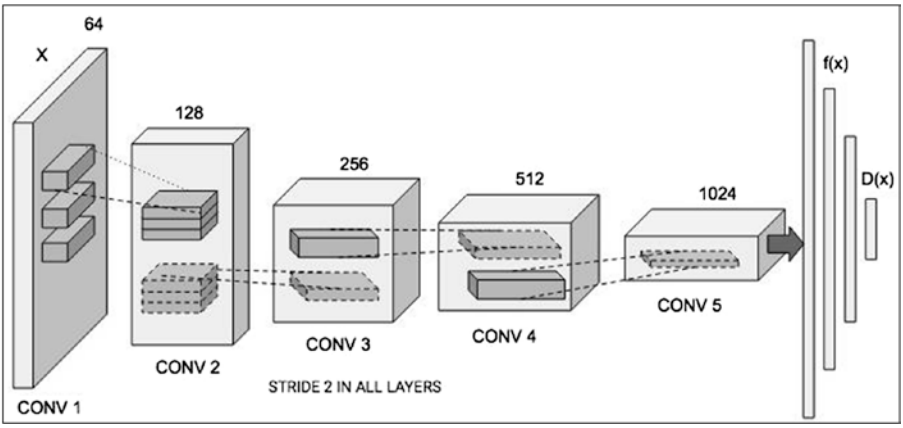


Figure 13-3. *Discriminator architecture*

The discriminator too uses Convolutional layers and downsizes the given image for evaluation.

Mathematical Formulation

The working of GAN can be formulated with the simple mathematical equation given here:

$$\min_G \max_D V(D,G) = \min_G \max_D \left(E_{z \sim P_{data}(x)} \left[logD(x) \right] + E_{z \sim P_z(z)} \left[\log\left(1 - D\left(G(z)\right)\right) \right] \right)$$

where G represents the generator and D the discriminator. The data(x) represents the distribution of real data and $p_z(z)$ the distribution of generated or fake data. The x represents a sample from real data and z

from the generated data. The D(x) represents the discriminator network and G(z) the generator network.

The discriminator loss while training on the real data is expressed as

$$L\big(D(x),1\big)=\log\big(D(x)\big) \tag{1}$$

The discriminator loss while training on the fake data coming from the generator is expressed as

$$L\big(D(G(z)),0\big)=\log\big(1-D(G(z))\big) \tag{2}$$

For real data, the discriminator prediction should be close to 1. Thus, the equation 1 should be maximized in order to get D(x) close to 1. The first equation is the discriminator loss on real data, which should be maximized in order to get D(G(z)) close to 1. Because the second equation is the discriminator loss on the fake data, it should also be maximized. Note that the log is an increasing function.

For the second equation, the discriminator prediction on the generated fake data should be close to zero. In order to maximize the second equation, we have to minimize the value of D(G(z)) to zero. Thus, we need to maximize both losses for the discriminator. The total loss of the discriminator is the addition of two losses given by equations 1 and 2. Thus, the combined total loss will also be maximized.

The generator loss is expressed as

$$L^{(G)}=\min\Big[\log\big(D(x)\big)+\log\big(1-D(G(z))\big)\Big] \tag{3}$$

We need to minimize this loss during the generator training.

Digit Generation

In this project, we will use the popular MNIST dataset provided by Kaggle. As you know, the dataset consists of images of handwritten digits. We will create a GAN model to create additional images looking identical to these images. Maybe this could be helpful to somebody in increasing the training dataset size in their future developments.

Creating Project

Create a Colab project and rename it to DigitGen-GAN. Import the required libraries.

```
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import time
from tensorflow import keras
from tensorflow.keras import layers
import os
```

Loading Dataset

You load the datasets into your code using the following statement:

```
(train_images,train_labels),
      (test_images,test_labels) =
              tf.keras.datasets.mnist.load_data()
```

The training and testing datasets are loaded into separate numpy arrays. As we are interested in generating a single digit, say 9, we will extract all images containing 9 from the train dataset.

```
digit9_images = []
```

```
for i in range(len(train_images)):
    if train_labels[i] == 9:
        digit9_images.append(train_images[i])
train_images = np.array(digit9_images)

train_images.shape
```

The shape is (5949, 28, 28). Thus, we have 5949 images of size 28x28. It is a huge repository. Our model will try to produce images of this size, matching the looks of these images.

You can verify that you have images only for digit 9 by printing a few on the terminal using the following code:

```
n = 10
f = plt.figure()
for i in range(n):
    f.add_subplot(1, n, i + 1)
    plt.subplot(1, n, i+1 ).axis("off")
    plt.imshow(train_images[i])
plt.show()
```

You will see the output in Figure 13-4.

Figure 13-4. *Sample images*

We will now prepare the dataset for training.

Preparing Dataset

First, we reshaped the data using the following statement; note that the image size in our dataset is 28x28 pixels.

```
train_images = train_images.reshape (
    train_images.shape[0], 28, 28, 1).astype
                        ('float32')
```

As each color value in the image is in the range of 0 through 256, we normalize these values to the scale of –1 to 1 for better learning. The mean value is 127.5, and thus the following equation will normalize the values in the range –1 to +1. You could have alternatively chosen 255 to normalize the values between 0 and 1.

```
train_images = (train_images - 127.5) / 127.5
```

We create a batch dataset for training by calling the from_tensor_slices method.

```
train_dataset = tf.data.Dataset.from_tensor_slices(
    train_images).shuffle
            (train_images.shape[0]).batch(32)
```

Next comes the important part of our application, and that is defining the model for our generator.

Defining Generator Model

The generator's purpose is to create images containing digit 9 which look similar to the images in our training dataset.

You will use the Keras sequential model for creating our generator.

```
gen_model = tf.keras.Sequential()
```

You will add the Keras Dense layer as the first layer. Optionally, you could make Conv2D layer as your first layer.

```
gen_model.add(tf.keras.layers.Dense(7*7*256,
                                    use_bias=False,
input_shape=(100,)))
```

The input to this layer is specified as 100 because later on we will use a noise vector of dimension 100 as an input to this GAN model. We will start with an image size of 7x7 and keep upscaling it to a final destination size of 28x28. The z-dimension 256 specifies the filters used for our image, which eventually gets converted to 3 for our final image.

Next, we add a batch normalization layer to the model for providing stability.

```
gen_model.add(tf.keras.layers.BatchNormalization())
gen_model.add(tf.keras.layers.LeakyReLU())
```

We add the activation layer as a leaky ReLU.

We reshape the output to 7x7x256:

```
gen_model.add(tf.keras.layers.Reshape((7, 7, 256)))
```

We now use the Conv2D layer to upscale the generated image.

```
gen_model.add(tf.keras.layers.Conv2DTranspose
                    (128, (5, 5),
                     strides=(1, 1),
                     padding='same',
                     use_bias=False))
```

The first parameter is the dimensionality of the output space, that is, the number of output filters in the convolution. The second parameter is the kernel_size that actually specifies the height and width of the convolution filter. The third parameter specifies the strides of the convolution along the height and width. To understand strides, consider a filter being moved across the image from left to right and top to bottom, 1 pixel at a time. This movement is referred to as the stride. With a stride of (2, 2), the filter is moved 2 pixels on each side, upscaling the image by 2x2.

Since the stride is specified as (1, 1), the output of this layer will be the image of size 7x7 – the same as its input. The padding ensures that the dimensions remain the same. The false value in the use_bias parameter indicates that the layer does not use a bias vector. This layer is then followed by the batch normalization and activation layers as before:

```
gen_model.add(tf.keras.layers.BatchNormalization())
gen_model.add(tf.keras.layers.LeakyReLU())
```

Next, we add the Conv2DTranspose layer with strides set to (2, 2) followed by the batch normalization and activation layers.

```
gen_model.add(tf.keras.layers.Conv2DTranspose
                      (64, (5, 5),
                       strides=(2, 2),
                       padding='same',
                       use_bias=False))
gen_model.add(tf.keras.layers.BatchNormalization())
gen_model.add(tf.keras.layers.LeakyReLU())
```

The output image will now be of size 14x14.

We now add the last Conv2D layer with strides equal to (2, 2), thus further upscaling the image to size 28x28. This is our final desired image size.

```
gen_model.add(tf.keras.layers.Conv2DTranspose
                       (1, (5, 5),
                        strides=(2, 2),
                        padding='same',
                        use_bias=False,
                        activation='tanh'))
```

The last layer uses tanh activation and the map parameter value of 1, thus giving us a single output image.

The model plot as generated by the plot utility is shown in Figure 13-5.

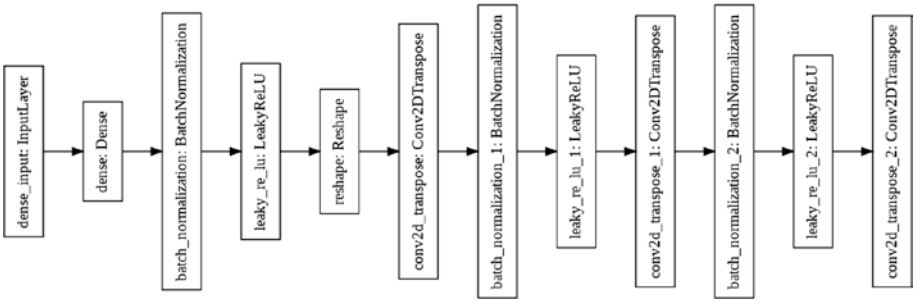


Figure 13-5. *Generator architecture*

The model summary is shown in Figure 13-6.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 12544)             1254400
_________________________________________________________________
batch_normalization (BatchNo (None, 12544)             50176
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 12544)             0
_________________________________________________________________
reshape (Reshape)            (None, 7, 7, 256)         0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 7, 7, 128)         819200
_________________________________________________________________
batch_normalization_1 (Batch (None, 7, 7, 128)         512
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 7, 7, 128)         0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 14, 14, 64)        204800
_________________________________________________________________
batch_normalization_2 (Batch (None, 14, 14, 64)        256
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 28, 28, 1)         1600
=================================================================
Total params: 2,330,944
Trainable params: 2,305,472
Non-trainable params: 25,472
```

Figure 13-6. *Generator model summary*

Testing Generator

You will test the generator with a random input vector and display it on the console using the following code:

```
noise = tf.random.normal([1, 100])
#giving random input vector
generated_image = gen_model(noise, training=False)
plt.imshow(generated_image[0, :, :, 0], cmap='gray')
```

The image will look like the one shown in Figure 13-7.

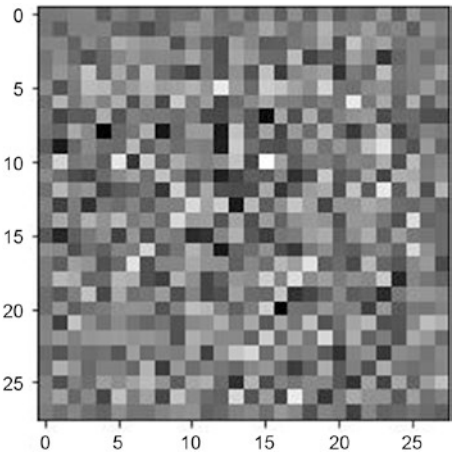


Figure 13-7. *Random image generated by the generator*

Check the shape of the image:

```
generated_image.shape
```

It gives the following output:

```
TensorShape([1, 28, 28, 1])
```

The output indicates that the image has dimensions of 28x28, as desired by us.

Next, we will define the discriminator.

Defining Discriminator Model

We define our discriminator as follows:

```
discri_model = tf.keras.Sequential()

discri_model.add(tf.keras.layers.Conv2D
                        (64, (5, 5),
                         strides=(2, 2),
                         padding='same',
                         input_shape=[28, 28, 1]))
discri_model.add(tf.keras.layers.LeakyReLU())
discri_model.add(tf.keras.layers.Dropout(0.3))

discri_model.add(tf.keras.layers.Conv2D
                        (128, (5, 5),
                         strides=(2, 2),
                         padding='same'))
discri_model.add(tf.keras.layers.LeakyReLU())
discri_model.add(tf.keras.layers.Dropout(0.3))

discri_model.add(tf.keras.layers.Flatten())
discri_model.add(tf.keras.layers.Dense(1))
```

The discriminator uses just two convolutional layers. The output of the last convolutional layer is of type (batch size, height, width, filters). The Flatten layer in our network flattens this output to feed it to our last Dense layer in the network.

The model plot is shown in Figure 13-8.

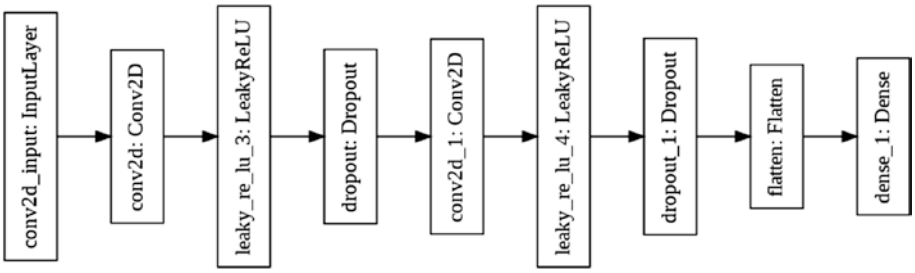


Figure 13-8. *Discriminator architecture*

The model summary is given in Figure 13-9.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 14, 14, 64)        1664
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 14, 14, 64)        0
_________________________________________________________________
dropout (Dropout)            (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 7, 7, 128)         204928
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 7, 7, 128)         0
_________________________________________________________________
dropout_1 (Dropout)          (None, 7, 7, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 6272)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 6273
=================================================================
Total params: 212,865
Trainable params: 212,865
Non-trainable params: 0
```

Figure 13-9. *Discriminator model summary*

Note that the discriminator has only around 200,000 trainable parameters.

Testing Discriminator

You may test the discriminator by feeding it with our earlier generated image.

```
decision = discri_model(generated_image)
```

The discriminator will give a negative value if the image is fake and a positive value if it is real. Print the discriminator's decision for this test image.

```
print (decision)
```

It prints out the following decision:

```
tf.Tensor([[0.0033829]], shape=(1, 1), dtype=float32)
```

The decision value is 0.0033, a positive integer indicating that the image is real. A maximum value of 1 tells us that the model is sure about the image being real. If you generate another image using our generator and test the output of the discriminator, you may get a negative result. This is because we have not yet trained our generator and discriminator on some datasets.

Defining Loss Functions

We will now define the loss function for our generator and discriminator.

```
cross_entropy = tf.keras.losses.BinaryCrossentropy
                                (from_logits=True)
```

We use Keras's binary cross-entropy function for our purpose. Note that we have two classes – one (1) for a real image and the other (0) for a fake one. We compute our loss for both these classes, and thus it makes our problem a binary classification problem. Thus, we use a binary cross-entropy for the loss function.

We define a function to compute the generator loss as follows:

```
def generator_loss(generated_output):
    return cross_entropy(tf.ones_like
                (generated_output),generated_output)
```

The return value of the function is the quantification of how well the generator is able to trick the discriminator. If the generator does its job well, the discriminator will classify the fake image as real, returning a decision of 1. Thus, the function compares the discriminator's decision on the generated image with an array of ones.

We define the discriminator loss function as follows:

```
def discriminator_loss(real_output,
                       generated_output):
# compute loss considering the image is real [1,1,...,1]
    real_loss = cross_entropy(tf.ones_like
                (real_output),real_output)

# compute loss considering the image is fake[0,0,...,0]
    generated_loss = cross_entropy(tf.zeros_like
                (generated_output),
                 generated_output)

    # compute total loss
    total_loss = real_loss + generated_loss

    return total_loss
```

We first let the discriminator consider that the given image is real and then compute the loss with respect to an array of ones. We then let the discriminator consider that the image is fake and then ask it to calculate the loss with respect to an array of zeros. The total loss as determined by the discriminator is the sum of these two losses.

We now define the optimizers for both generator and discriminator, which are set to Adam for both.

```
gen_optimizer = tf.optimizers.Adam(1e-4)
discri_optimizer = tf.optimizers.Adam(1e-4)
```

You will now write a few utility functions, which are used during training.

Defining Few Functions for Training

First, we declare a few variables:

```
epoch_number = 0
EPOCHS = 100
noise_dim = 100
seed = tf.random.normal([1, noise_dim])
```

We will train the model for 100 epochs. You can always change this variable to your choice. A higher number of epochs would generate a better image for the digit 9, which we are trying to generate. The noise dimension is set to 100 which is used while creating the random image for our first input to the generator network. The seed is set to random data for one image.

Checkpoint Setup

As the training may take a long time, we provide the checkpoint facility in the training so that the intermediate states of the generator and discriminator are saved to a local file.

```
checkpoint_dir =
        '/content/drive/My Drive/GAN1/Checkpoint'
checkpoint_prefix =
            os.path.join(checkpoint_dir, "ckpt")
```

```
checkpoint = tf.train.Checkpoint
             (generator_optimizer = gen_optimizer,
discriminator_optimizer=discri_optimizer,
              generator= gen_model,
              discriminator = discri_model)
```

In case of disconnection, you can continue the training from the last checkpoint. I will show you how to do this.

Setting Up Drive

The checkpoint data is saved to a folder called GAN1/Checkpoint in your Google Drive. So, before running the code, make sure that you have created this folder structure in your Google Drive.

Mount the drive in your project using the following code:

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

Change the current folder to the new location so that the checkpoint files are stored at this location.

```
cd '/content/drive/My Drive/GAN1'
```

Next, we write a gradient_tuning function.

Model Training Step

Both our generator and discriminator models will be trained in several steps. We will write a function for these steps.

We will use the gradient tape (tf.GradientTape) for automatic differentiation on both the generator and discriminator. The automatic differentiation computes the gradient of a computation with respect to its input variables. The operations executed in the context of a gradient tape are recorded onto a tape. The reverse mode differentiation is then used

to compute the new gradients. You need not get concerned with these operations for understanding how the models are trained.

At each step, we will give a batch of images to the function as an input. We will ask the discriminator to produce outputs for both the training and the generated images. We will call the training output as real and the generated image output as fake. We will calculate the generator loss on the fake and the discriminator loss on both real and fake. We will use the gradient tape to compute the gradients on both using these losses and then apply the new gradients to the modes. The full function definition along with the comments on each line is given in Listing 13-1.

Listing 13-1. Gradient tuning function

```
def gradient_tuning(images):
    # create a noise vector.
    noise = tf.random.normal([16, noise_dim])

    # Use gradient tapes for automatic
    # differentiation
    with tf.GradientTape()
          as generator_tape, tf.GradientTape()
          as discriminator_tape:

      # ask genertor to generate random images
      generated_images = gen_model(noise, training=True)

      # ask discriminator to evalute the real images and
      generate its output
      real_output = discri_model(images,
                        training = True)
      # ask discriminator to do the evlaution on generated
      (fake) images
      fake_output = discri_model(generated_images,
                        training = True)
```

```
        # calculate generator loss on fake data
        gen_loss = generator_loss(fake_output)

        # calculate discriminator loss as defined earlier
        disc_loss = discriminator_loss(real_output,
                          fake_output)

    # calculate gradients for generator
    gen_gradients = generator_tape.gradient
            (gen_loss, gen_model.trainable_variables)

    # calculate gradients for discriminator
    discri_gradients =
          discriminator_tape.gradient(disc_loss,
                discri_model.trainable_variables)

    # use optimizer to process and apply gradients to variables
    gen_optimizer.apply_gradients(zip(gen_gradients,
                      gen_model.trainable_variables))

    # same as above to discriminator
    discri_optimizer.apply_gradients(
        zip(discri_gradients,
            discri_model.trainable_variables))
```

We now write one more function for generating the image of digit 9, our desired output, and saving it to your Google Drive.

```
def generate_and_save_images(model, epoch,
                                  test_input):
    global epoch_number
    epoch_number = epoch_number + 1

    # set training to false to ensure inference mode
    predictions = model(test_input,
                            training = False)
```

```
# display and save image
fig = plt.figure(figsize=(4,4))
for i in range(predictions.shape[0]):
    plt.imshow(predictions[i, :, :, 0] *
               127.5 + 127.5, cmap='gray')
    plt.axis('off')
plt.savefig('image_at_epoch_{:01d}.png'.format
                      (epoch_number))
plt.show()
```

The function uses a global epoch_number to track the epochs, in case of disconnection and continuing thereof. The test_input to the model will always be our random seed. The inference is done on this seed after setting training to False for batch normalization to be effective. We then display the image on the user's console and also save it to the drive with epoch_ number added as a suffix.

Using these functions, you will now write code to train the models and generate some output.

Model Training

To train the generator and discriminator models, you need to set up a simple for loop as shown in Listing 13-2. The train method accepts the dataset of true images as its first parameter. I have put this as a parameter so that you can experiment with different datasets of images and/or different sizes. The second parameter is the number of epochs for which you want to perform the training. We call the gradient_tuning for batches of data in our dataset. At the end of each epoch, we generate and save the image to the user's drive. Also, the network state is saved as a checkpoint so that we can continue the training in case of disconnection from the last checkpoint. The time taken to execute each epoch is tracked and printed on the user console. The function for the model training is given in Listing 13-2.

Listing 13-2. Model training function

```
def train(dataset, epochs):
  for epoch in range(epochs):
    start = time.time()

    for image_batch in dataset:
      gradient_tuning(image_batch)

    # Produce images as we go
    generate_and_save_images(gen_model,
                             epoch + 1,
                             seed)

    # save checkpoint data
    checkpoint.save(file_prefix = checkpoint_prefix)
    print ('Time for epoch {} is {} sec'.format
                                  (epoch + 1,
                                  time.time()-start))
```

The model training is now started with a call to this train method.

```
train(train_dataset, EPOCHS)
```

As the training progresses, the checkpoints are saved and the images are generated at each epoch. The images are displayed on the console and are also saved to your Google Drive. The output of my run for 100 epochs is shown in Table 13-1.

Table 13-1. *Images for digit 9 generated by the program*

 epoch_1	 epoch_5	 epoch_10	 epoch_20
 epoch_30	 epoch_40	 epoch_50	 epoch_60
 epoch_70	 epoch_80	 epoch_90	 epoch_100

As you can see from the output, the network is able to create an acceptable output just after 20/30 epochs, and at 70 and above, the quality is best.

During training and in case of disconnection, you can restore the network state from a previous known checkpoint as shown in the statement here and continue the training.

```
#run this code only if there is a runtime disconnection
try:
     checkpoint.restore(tf.train.latest_checkpoint
                              (checkpoint_dir))
except Exception as error:
    print("Error loading in model :
                  {}".format(error))
train(train_dataset, EPOCHS)
```

In my run of this application, it took approximately 10 seconds for each epoch on a GPU. Many times, for more complex image generation, it may take several hours to get an acceptable output. In such cases, checkpoints would be useful in restarting the training.

Full Source

The full source code for generating images for handwritten digits is given in Listing 13-3.

Listing 13-3. DigitGen-GAN.ipynb

```
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import time
from tensorflow import keras
from tensorflow.keras import layers
import os

(train_images,train_labels),
      (test_images,test_labels) =
              tf.keras.datasets.mnist.load_data()
```

```
digit9_images = []
for i in range(len(train_images)):
    if train_labels[i] == 9:
        digit9_images.append(train_images[i])
train_images = np.array(digit9_images)

train_images.shape

n = 10
f = plt.figure()
for i in range(n):
    f.add_subplot(1, n, i + 1)
    plt.subplot(1, n, i+1 ).axis("off")
    plt.imshow(train_images[i])
plt.show()

train_images = train_images.reshape (
    train_images.shape[0], 28, 28, 1).astype
                          ('float32')
train_images = (train_images - 127.5) / 127.5
train_dataset = tf.data.Dataset.from_tensor_slices(
    train_images).shuffle
            (train_images.shape[0]).batch(32)

gen_model = tf.keras.Sequential()

# Feed network with a 7x7 random image
gen_model.add(tf.keras.layers.Dense(7*7*256,
                                    use_bias=False,
input_shape=(100,)))
# Add batch normalization for stability
gen_model.add(tf.keras.layers.BatchNormalization())
gen_model.add(tf.keras.layers.LeakyReLU())
```

```
# reshape the output
gen_model.add(tf.keras.layers.Reshape((7, 7, 256)))

# Apply (5x5) filter and shift of (1,1).
# The image output is still 7x7.
gen_model.add(tf.keras.layers.Conv2DTranspose
                    (128, (5, 5),
                     strides=(1, 1),
                     padding='same',
                     use_bias=False))
gen_model.add(tf.keras.layers.BatchNormalization())
gen_model.add(tf.keras.layers.LeakyReLU())

# apply stride of (2,2). The output image is now 14x14.
gen_model.add(tf.keras.layers.Conv2DTranspose
                    (64, (5, 5),
                     strides=(2, 2),
                     padding='same',
                     use_bias=False))
gen_model.add(tf.keras.layers.BatchNormalization())
gen_model.add(tf.keras.layers.LeakyReLU())

# another shift upscales the image to 28x28, whihch is our
final size.
gen_model.add(tf.keras.layers.Conv2DTranspose
                     (1, (5, 5),
                      strides=(2, 2),
                      padding='same',
                      use_bias=False,
                      activation='tanh'))

gen_model.summary()
```

```
tf.keras.utils.plot_model(gen_model)

noise = tf.random.normal([1, 100])
#giving random input vector
generated_image = gen_model(noise, training=False)
plt.imshow(generated_image[0, :, :, 0], cmap='gray')

generated_image.shape

discri_model = tf.keras.Sequential()

discri_model.add(tf.keras.layers.Conv2D
                         (64, (5, 5),
                          strides=(2, 2),
                          padding='same',
                          input_shape=[28, 28, 1]))
discri_model.add(tf.keras.layers.LeakyReLU())
discri_model.add(tf.keras.layers.Dropout(0.3))

discri_model.add(tf.keras.layers.Conv2D
                         (128, (5, 5),
                          strides=(2, 2),
                          padding='same'))
discri_model.add(tf.keras.layers.LeakyReLU())
discri_model.add(tf.keras.layers.Dropout(0.3))

discri_model.add(tf.keras.layers.Flatten())
discri_model.add(tf.keras.layers.Dense(1))
discri_model.summary()

tf.keras.utils.plot_model(discri_model)

decision = discri_model(generated_image)
print (decision)
```

```
cross_entropy = tf.keras.losses.BinaryCrossentropy
                                (from_logits=True)
 #creating loss function

def generator_loss(generated_output):
    return cross_entropy(tf.ones_like
                (generated_output),generated_output)

def discriminator_loss(real_output,
                        generated_output):
# compute loss considering the image is real [1,1,...,1]
    real_loss = cross_entropy(tf.ones_like
                (real_output),real_output)

# compute loss considering the image is fake[0,0,...,0]
    generated_loss = cross_entropy(tf.zeros_like
                (generated_output),
                 generated_output)

    # compute total loss
    total_loss = real_loss + generated_loss

    return total_loss

gen_optimizer = tf.optimizers.Adam(1e-4)
discri_optimizer = tf.optimizers.Adam(1e-4)

epoch_number = 0
EPOCHS = 100
noise_dim = 100
seed = tf.random.normal([1, noise_dim])

checkpoint_dir =
        '/content/drive/My Drive/GAN1/Checkpoint'
checkpoint_prefix =
            os.path.join(checkpoint_dir, "ckpt")
```

```
checkpoint = tf.train.Checkpoint
             (generator_optimizer = gen_optimizer,
                                discriminator_
optimizer=discri_optimizer,
              generator= gen_model,
              discriminator = discri_model)

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

cd '/content/drive/My Drive/GAN1'

def gradient_tuning(images):
    # create a noise vector.
    noise = tf.random.normal([16, noise_dim])

    # Use gradient tapes for automatic differentiation
    with tf.GradientTape()
          as generator_tape, tf.GradientTape()
          as discriminator_tape:

      # ask genertor to generate random images
      generated_images = gen_model(noise,
                           training=True)

      # ask discriminator to evalute the real images and
generate its output
      real_output = discri_model(images,
                         training = True)

      # ask discriminator to do the evlaution on generated
(fake) images
      fake_output = discri_model(generated_images,
                         training = True)
```

```
      # calculate generator loss on fake data
      gen_loss = generator_loss(fake_output)

      # calculate discriminator loss as defined earlier
      disc_loss = discriminator_loss(real_output,
                        fake_output)
    # calculate gradients for generator
    gen_gradients = generator_tape.gradient
            (gen_loss, gen_model.trainable_variables)

    # calculate gradients for discriminator
    discri_gradients =
          discriminator_tape.gradient(disc_loss,
                discri_model.trainable_variables)

    # use optimizer to process and apply gradients to variables
    gen_optimizer.apply_gradients(zip(gen_gradients,
                      gen_model.trainable_variables))

    # same as above to discriminator
    discri_optimizer.apply_gradients(
        zip(discri_gradients,
            discri_model.trainable_variables))

    def generate_and_save_images(model, epoch,
                                 test_input):
        global epoch_number
        epoch_number = epoch_number + 1

        # set training to false to ensure inference mode
        predictions = model(test_input,
                            training = False)
```

```
        # display and save image
        fig = plt.figure(figsize=(4,4))
        for i in range(predictions.shape[0]):
            plt.imshow(predictions[i, :, :, 0] *
                       127.5 + 127.5, cmap='gray')
            plt.axis('off')
        plt.savefig('image_at_epoch_{:01d}.png'.format
                              (epoch_number))
        plt.show()

def train(dataset, epochs):
  for epoch in range(epochs):
    start = time.time()

    for image_batch in dataset:
      gradient_tuning(image_batch)

    # Produce images as we go
    generate_and_save_images(gen_model,
                             epoch + 1,
                             seed)

    # save checkpoint data
    checkpoint.save(file_prefix = checkpoint_prefix)
    print ('Time for epoch {} is {} sec'.format
                                  (epoch + 1,
                                  time.time()-start))

train(train_dataset, EPOCHS)

#run this code only if there is a runtime disconnection
try:
     checkpoint.restore(tf.train.latest_checkpoint
                             (checkpoint_dir))
```

```
except Exception as error:
    print("Error loading in model :
                {}".format(error))
train(train_dataset, EPOCHS)
```

In this example, you trained a model to generate the handwritten digits. In the next example, I will show you how to create handwritten characters.

Alphabet Generation

Just the way Kaggle provides the dataset for digits, the dataset for handwritten alphabets is available in another package called extra-keras-datasets. You will be using this dataset to generate the handwritten alphabets. The generator and discriminator models, their training, and inference all remain the same as in the case of digit generation. So, I am just going to give you the code of how to load the alphabets dataset from the Kaggle site and will give you the output of generated images. The full project source is available in the book's repository under the name emnist-GAN.

Downloading Data

The dataset is available in a separate package that you can install by running the pip utility.

```
pip install extra-keras-datasets
```

Import the package into your project.

```
from extra_keras_datasets import emnist
```

Load the data into your project and display one image and the corresponding label.

```
(train_images,train_labels),
        (test_images,test_labels) =
        emnist.load_data(type='letters')
plt.imshow(train_images[1])
print ("label: ", train_labels[1])
```

The output is shown in Figure 13-10.

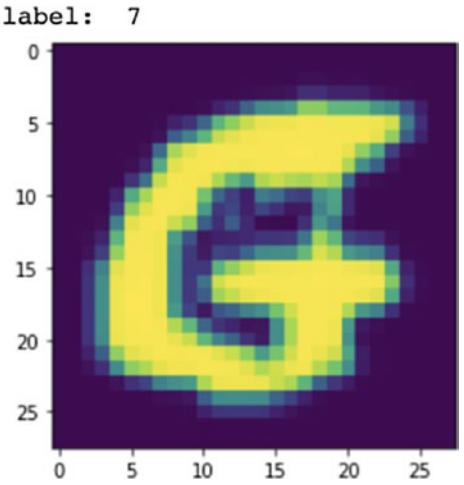


Figure 13-10. *Image for the handwritten G character*

Like in the digits database, each image has a dimension of 28x28. Thus, we will be able to use our previous generator model that produces images of this dimension. One good thing for our experimentation here is that each alphabet label takes the value of its position in the alphabets set. For example, the letter a has a label value of 1, the letter b has a label value of 2, and so on.

Creating Dataset for a Single Alphabet

Like in the digit generation example, we will train the model to produce a single alphabet. Thus, we will need to create a dataset of images containing only the desired alphabet. This is done using the following code:

```
letter_G_images = []
for i in range(len(train_images)):
    if train_labels[i] == 7:
        letter_G_images.append(train_images[i])
train_images = np.array(letter_G_images)
```

You can verify that you have extracted the images of only G alphabet by running a small for loop as follows:

```
n = 10
f = plt.figure()
for i in range(n):
    f.add_subplot(1, n, i + 1)
    plt.subplot(1, n, i+1 ).axis("off")
    plt.imshow(train_images[i])
plt.show()
```

The output is shown in Figure 13-11.

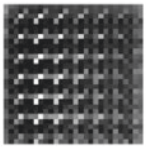
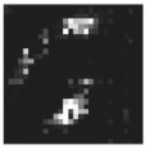
Figure 13-11. *Sample handwritten character images*

Okay, now your dataset is ready. The rest of the code for preprocessing the data, defining models, loss functions, optimizers, training, and so on remains the same as the digit generation application. I will not reproduce the code here, I will simply show you the final output of my run.

Program Output

The program output at various epochs is shown in Table 13-2.

Table 13-2. *Images generated at various epochs*

 eoch_1	 epoch_5	 epoch_10	 epoch_20
 epoch_30	 epoch_40	 epoch_50	 epoch_60
 epoch_70	 epoch_80	 epoch_90	 epoch_100

As in the case of digit generation, you can notice that the model gets quickly trained during the first few epochs, and by the end of 100 epochs, you have a quality output.

Full Source

The full source code for the character image generation is given in Listing 13-4.

Listing 13-4. emnist-GAN.ipynb

```
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import time
from tensorflow import keras
from tensorflow.keras import layers
import os

pip install extra-keras-datasets
from extra_keras_datasets import emnist

(train_images,train_labels),
        (test_images,test_labels) =
        emnist.load_data(type='letters')
plt.imshow(train_images[1])
print ("label: ", train_labels[1])

letter_G_images = []
for i in range(len(train_images)):
    if train_labels[i] == 7:
        letter_G_images.append(train_images[i])
train_images = np.array(letter_G_images)
n = 10
f = plt.figure()
for i in range(n):
    f.add_subplot(1, n, i + 1)
```

```
    plt.subplot(1, n, i+1 ).axis("off")
    plt.imshow(train_images[i])
plt.show()

train_images = train_images.reshape (
    train_images.shape[0], 28, 28, 1).astype('float32')
train_images = (train_images - 127.5) / 127.5
train_dataset = tf.data.Dataset.from_tensor_slices(
    train_images).shuffle
            (train_images.shape[0]).batch(32)

gen_model = tf.keras.Sequential()

# Feed network with a 7x7 random image
gen_model.add(tf.keras.layers.Dense
                            (7*7*256,
                            use_bias=False,
                            input_shape=(100,)))
# Add batch normalization for stability
gen_model.add(tf.keras.layers.BatchNormalization())
gen_model.add(tf.keras.layers.LeakyReLU())

# reshape the output
gen_model.add(tf.keras.layers.Reshape((7, 7, 256)))

# Apply (5x5) filter and shift of (1,1).
# The image output is still 7x7.
gen_model.add(tf.keras.layers.Conv2DTranspose
                                (128, (5, 5),
                                strides=(1, 1),
                                padding='same',
                                use_bias=False))
gen_model.add(tf.keras.layers.BatchNormalization())
gen_model.add(tf.keras.layers.LeakyReLU())
```

```
# apply stride of (2,2). The output image is now 14x14.
gen_model.add(tf.keras.layers.Conv2DTranspose
                                (64, (5, 5),
                                 strides=(2, 2),
                                 padding='same',
                                 use_bias=False))
gen_model.add(tf.keras.layers.BatchNormalization())
gen_model.add(tf.keras.layers.LeakyReLU())

# another shift upscales the image to 28x28, whihch is our
final size.
gen_model.add(tf.keras.layers.Conv2DTranspose
                                 (1, (5, 5),
                                 strides=(2, 2),
                                 padding='same',
                                 use_bias=False,
                                 activation='tanh'))

gen_model.summary()

tf.keras.utils.plot_model(gen_model)

noise = tf.random.normal([1, 100])#giving random input vector
generated_image = gen_model(noise, training=False)
plt.imshow(generated_image[0, :, :, 0], cmap='gray')

generated_image.shape

discri_model = tf.keras.Sequential()

discri_model.add(tf.keras.layers.Conv2D
                            (64, (5, 5),
                             strides=(2, 2),
                             padding='same',
                             input_shape=[28, 28, 1]))
```

```
discri_model.add(tf.keras.layers.LeakyReLU())
discri_model.add(tf.keras.layers.Dropout(0.3))

discri_model.add(tf.keras.layers.Conv2D
                         (128, (5, 5),
                          strides=(2, 2),
                          padding='same'))
discri_model.add(tf.keras.layers.LeakyReLU())
discri_model.add(tf.keras.layers.Dropout(0.3))

discri_model.add(tf.keras.layers.Flatten())
discri_model.add(tf.keras.layers.Dense(1))

discri_model.summary()

tf.keras.utils.plot_model(discri_model)

decision = discri_model(generated_image)
print (decision)

cross_entropy = tf.keras.losses.BinaryCrossentropy(from_
logits=True)
#creating loss function

def generator_loss(generated_output):
    return cross_entropy(tf.ones_like(generated_output),
                         generated_output)

def discriminator_loss(real_output,
                         generated_output):
    # compute loss considering the image is real [1,1,...,1]
    real_loss = cross_entropy(tf.ones_like
                    (real_output),real_output)

    # compute loss considering the image is fake[0,0,...,0]
    generated_loss = cross_entropy
```

```
                    (tf.zeros_like(generated_output),
                                   generated_output)

    # compute total loss
    total_loss = real_loss + generated_loss

    return total_loss

gen_optimizer = tf.optimizers.Adam(1e-4)
discri_optimizer = tf.optimizers.Adam(1e-4)

epoch_number = 0
EPOCHS = 100
noise_dim = 100
seed = tf.random.normal([1, noise_dim])

checkpoint_dir =
      '/content/drive/My Drive/GAN2/Checkpoint'
checkpoint_prefix = os.path.join
                    (checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint
                (generator_optimizer = gen_optimizer,
                discriminator_optimizer =
                           discri_optimizer,
                generator= gen_model,
                discriminator = discri_model)
from google.colab import drive
drive.mount('/content/drive')

cd '/content/drive/My Drive/GAN2'

def gradient_tuning(images):
    # create a noise vector.
    noise = tf.random.normal([16, noise_dim])
```

```
    # Use gradient tapes for automatic differentiation
    with tf.GradientTape()
        as generator_tape, tf.GradientTape()
        as discriminator_tape:

      # ask genertor to generate random images
      generated_images = gen_model
                        (noise, training = True)

      # ask discriminator to evalute the real images and
generate its output
      real_output = discri_model
                        (images, training = True)

      # ask discriminator to do the evlaution on generated
(fake) images
      fake_output = discri_model
                (generated_images, training = True)

      # calculate generator loss on fake data
      gen_loss = generator_loss(fake_output)

      # calculate discriminator loss as defined earlier
      disc_loss = discriminator_loss
                     (real_output, fake_output)
    # calculate gradients for generator
    gen_gradients = generator_tape.gradient
                    (gen_loss,
                    gen_model.trainable_variables)

    # calculate gradients for discriminator
    discri_gradients = discriminator_tape.gradient
                        (disc_loss,
                   discri_model.trainable_variables)
```

```
    # use optimizer to process and apply gradients to variables
    gen_optimizer.apply_gradients(zip(gen_gradients,
                  gen_model.trainable_variables))

    # same as above to discriminator
    discri_optimizer.apply_gradients(
        zip(discri_gradients,
            discri_model.trainable_variables))
    def generate_and_save_images
                (model, epoch, test_input):
        global epoch_number
        epoch_number = epoch_number + 1

        # set training to false to ensure inference mode
        predictions = model(test_input,
                            training = False)
        # display and save image
        fig = plt.figure(figsize=(4,4))
        for i in range(predictions.shape[0]):
            plt.imshow(predictions
                 [i, :, :, 0] * 127.5 + 127.5,
                                cmap='gray')
            plt.axis('off')
        plt.savefig('image_at_epoch_
                  {:01d}.png'.format(epoch_number))
        plt.show()

def train(dataset, epochs):
  for epoch in range(epochs):
    start = time.time()
```

```
    for image_batch in dataset:
      gradient_tuning(image_batch)

    # Produce images as we go
    generate_and_save_images(gen_model,
                             epoch + 1,
                             seed)

    # save checkpoint data
    checkpoint.save(file_prefix =
                            checkpoint_prefix)
    print ('Time for epoch {} is {} sec'.format
                           (epoch + 1,
                           time.time()-start))
train(train_dataset, EPOCHS)

#run this code only if there is a runtime disconnection
try:
     checkpoint.restore(tf.train.latest_checkpoint
                             (checkpoint_dir))
except Exception as error:
    print("Error loading in model :
                             {}".format(error))
train(train_dataset, 100)
```

Printed to Handwritten Text

You may train the preceding network for your own handwriting by feeding your handwritten images for a–z, A–Z, and digits 0–9. With this trained model, anybody who has access to the model will be able to convert any printed text to a personalized handwritten text authored by you. I used a few characters from such a trained network to personalize the writing of the word "tensor," which is shown in Figure 13-12.

Figure 13-12. *Sample text created by combining images*

Next comes the creation of a more complex image.

Color Cartoons

So far, you have created images of handwritten digits and alphabets. What about creating complex color images like cartoons? The techniques that you have learned so far can be applied to create complex color images. And that is what I am going to demonstrate in this project.

Downloading Data

There are a good deal of anime character datasets available on Kaggle site. We have kept the dataset for this project ready for your use on the book's download site. Download the data into your project using the wget utility.

```
! wget --no-check-certificate -r 'https://drive.google.com/uc?exp
ort=download&id=1z7rXRIFtRBFZHt-Mmti4HxrxHqUfG3Y8' -O tf-book.zip
```

Unzip the contents of the downloaded file.

```
!unzip tf-book.zip
```

Creating Dataset

Write a function to create a dataset:

```
def load_dataset(batch_size, img_shape,
                        data_dir = None):
    # Create a tuple of size(30000,64,64,3)
    sample_dim = (batch_size,) + img_shape
```

```
    # Create an uninitialized array of shape (30000,64,64,3)
    sample = np.empty(sample_dim, dtype=np.float32)
    # Extract all images from our file
    all_data_dirlist = list(glob.glob(data_dir))

    # Randomly select an image file from our data list
    sample_imgs_paths = np.random.choice
                   (all_data_dirlist,batch_size)

    for index,img_filename in enumerate
                          (sample_imgs_paths):
        # Open the image
        image = Image.open(img_filename)
        # Resize the image
        image = image.resize(img_shape[:-1])
        # Convert the input into an array
        image = np.asarray(image)
        # Normalize data
        image = (image/127.5) -1
        # Assign the preprocessed image to our sample
        sample[index,...] = image
    print("Data loaded")
    return sample
```

The download code is self-explanatory and fully commented for your ease of understanding. You now call this function to create a dataset:

```
x_train=load_dataset(30000,(64,64,3),
      "/content/tf-book/chapter13/anime/data/*.png")
BUFFER_SIZE = 30000
BATCH_SIZE = 256
train_dataset = tf.data.Dataset.from_tensor_slices
              (x_train).shuffle(BUFFER_SIZE).batch
                      (BATCH_SIZE)
```

Displaying Images

You can check that the dataset is correctly loaded by printing a few images from the set.

```
n = 10
f = plt.figure(figsize=(15,15))
for i in range(n):
    f.add_subplot(1, n, i + 1)
    plt.subplot(1, n, i+1 ).axis("off")
    plt.imshow(x_train[i])
plt.show()
```

The output is shown in Figure 13-13.

Figure 13-13. *Sample anime images*

Check the shape of your training data.

```
x_train.shape
```

The shape will be printed as follows:

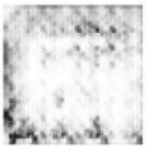
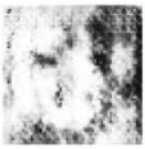
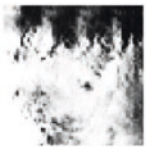
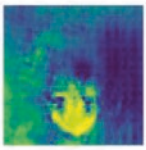
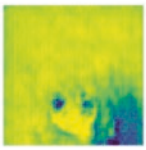
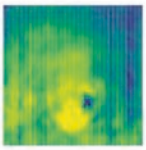
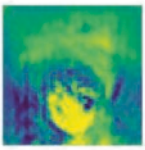
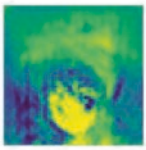
```
(30000, 64, 64, 3)
```

There are 30,000 RGB images, each of size 64x64. At this point, you are ready for defining your models, training them, and doing inference. The rest of the code that follows here is exactly identical to the earlier two projects, and thus I am not reproducing the code here. You may look up the entire source code in the project download. I will present only the output at different epochs.

Output

The generated images at various epochs are shown in Table 13-3.

Table 13-3. *Sample images generated at various epochs*

 Epoch 1	 Epoch 50	 Epoch 100	 Epoch 200
 Epoch 300	 Epoch 400	 Epoch 500	 Epoch 600
 Epoch 700	 Epoch 800	 Epoch 900	 Epoch 1000

You can see that by about 1000 epochs, the network learns quite a bit to reproduce the original cartoon image. To train the model to generate a real-like image, you would need to run the code for 10,000 epochs or more. Each epoch took me about 16 seconds to run on a GPU. Basically, what I

wanted to show you here is that the GAN technique that we develop for creating trivial handwritten digits can be applied as is to generate complex large images too.

Full source

The full source code for generating anime images is given in Listing 13-5.

Listing 13-5. CS-Anime.ipynb

```
import tensorflow as tf
import numpy as np
import sys
import os
import cv2
import glob
from PIL import Image
import matplotlib.pyplot as plt
import time
from tensorflow import keras
from tensorflow.keras import layers
from keras.layers import UpSampling2D, Conv2D

! wget --no-check-certificate -r 'https://drive.google.com/uc?exp
ort=download&id=1z7rXRIFtRBFZHt-Mmti4HxrxHqUfG3Y8' -O tf-book.zip

!unzip tf-book.zip

def load_dataset(batch_size, img_shape,
                         data_dir = None):
    # Create a tuple of size(30000,64,64,3)
    sample_dim = (batch_size,) + img_shape
    # Create an uninitialized array of shape (30000,64,64,3)
    sample = np.empty(sample_dim, dtype=np.float32)
```

```
    # Extract all images from our file
    all_data_dirlist = list(glob.glob(data_dir))

    # Randomly select an image file from our data list
    sample_imgs_paths = np.random.choice
                   (all_data_dirlist,batch_size)

    for index,img_filename in enumerate
                           (sample_imgs_paths):
        # Open the image
        image = Image.open(img_filename)
        # Resize the image
        image = image.resize(img_shape[:-1])
        # Convert the input into an array
        image = np.asarray(image)
        # Normalize data
        image = (image/127.5) -1
        # Assign the preprocessed image to our sample
        sample[index,...] = image
    print("Data loaded")
    return sample

x_train=load_dataset(30000,(64,64,3),
      "/content/tf-book/chapter13/anime/data/*.png")
BUFFER_SIZE = 30000
BATCH_SIZE = 256
train_dataset = tf.data.Dataset.from_tensor_slices
              (x_train).shuffle(BUFFER_SIZE).batch
                      (BATCH_SIZE)

n = 10
f = plt.figure(figsize=(15,15))
```

```
for i in range(n):
    f.add_subplot(1, n, i + 1)
    plt.subplot(1, n, i+1 ).axis("off")
    plt.imshow(x_train[i])
plt.show()

x_train.shape

gen_model = tf.keras.Sequential()

# seed image of size 4x4
gen_model.add(tf.keras.layers.Dense
                        (64*4*4,
                        use_bias=False,
                        input_shape=(100,)))
gen_model.add(tf.keras.layers.BatchNormalization())
gen_model.add(tf.keras.layers.LeakyReLU())

gen_model.add(tf.keras.layers.Reshape((4,4,64)))


# size of output image is still 4x4
gen_model.add(tf.keras.layers.Conv2DTranspose
                       (256, (5, 5),
                        strides=(1, 1),
                        padding='same',
                        use_bias=False))
gen_model.add(tf.keras.layers.BatchNormalization())
gen_model.add(tf.keras.layers.LeakyReLU())
# size of output image is 8x8
gen_model.add(tf.keras.layers.Conv2DTranspose
                       (128, (5, 5),
                       strides=(2, 2),
                       padding='same',
                       use_bias=False))
```

```
gen_model.add(tf.keras.layers.BatchNormalization())
gen_model.add(tf.keras.layers.LeakyReLU())

# size of output image is 16x16
gen_model.add(tf.keras.layers.Conv2DTranspose
                        (64, (5, 5),
                        strides=(2, 2),
                        padding='same',
                        use_bias=False))
gen_model.add(tf.keras.layers.BatchNormalization())
gen_model.add(tf.keras.layers.LeakyReLU())

# size of output image is 32x32
gen_model.add(tf.keras.layers.Conv2DTranspose
                        (32, (5, 5),
                        strides=(2, 2),
                        padding='same',
                        use_bias=False))
gen_model.add(tf.keras.layers.BatchNormalization())
gen_model.add(tf.keras.layers.LeakyReLU())

# size of output image is 64x64
gen_model.add(tf.keras.layers.Conv2DTranspose
                        (3, (5, 5),
                        strides=(2, 2),
                        padding='same',
                        use_bias=False,
                        activation='tanh'))

gen_model.summary()
noise = tf.random.normal([1, 100])
generated_image = gen_model(noise, training=False)
plt.imshow(generated_image[0, :, :, 0] )
```

```
discri_model = tf.keras.Sequential()
discri_model.add(tf.keras.layers.Conv2D
                    (128, (5, 5), strides=(2, 2),
                    padding='same',
                    input_shape=[64,64,3]))
discri_model.add(tf.keras.layers.LeakyReLU())
discri_model.add(tf.keras.layers.Dropout(0.3))

discri_model.add(tf.keras.layers.Conv2D(
                    256, (5, 5), strides=(2, 2),
                    padding='same'))
discri_model.add(tf.keras.layers.LeakyReLU())
discri_model.add(tf.keras.layers.Dropout(0.3))

discri_model.add(tf.keras.layers.Flatten())
discri_model.add(tf.keras.layers.Dense(1))
discri_model.summary()

tf.keras.utils.plot_model(discri_model)

decision = discri_model(generated_image)
#giving the generated image to discriminator,the discriminator
will give negative value if it is fake,while if it is real then
it will give positive value.
print (decision)

cross_entropy = tf.keras.losses.BinaryCrossentropy
                        (from_logits=True)

def generator_loss(generated_output):
    return cross_entropy(tf.ones_like(generated_output),
                            generated_output)

def discriminator_loss(real_output,
                        generated_output):
```

```
    # compute loss considering the image is real [1,1,...,1]
    real_loss = cross_entropy
                   (tf.ones_like(real_output),
                        real_output)
    # compute loss considering the image is fake[0,0,...,0]
    generated_loss = cross_entropy
                      (tf.zeros_like
                      (generated_output),
                       generated_output)

    # compute total loss
    total_loss = real_loss + generated_loss

    return total_loss

gen_optimizer = tf.optimizers.Adam(1e-4)
discri_optimizer = tf.optimizers.Adam(1e-4)

epoch_number = 0
EPOCHS = 10000
noise_dim = 100
seed = tf.random.normal([1, noise_dim])

checkpoint_dir =
       '/content/drive/My Drive/GAN3/Checkpoint'
checkpoint_prefix = os.path.join
                        (checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint
            (generator_optimizer=gen_optimizer,
            discriminator_optimizer=discri_optimizer,
            generator= gen_model,
            discriminator = discri_model)
```

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

cd '/content/drive/My Drive/GAN3'

def gradient_tuning(images):
    # create a noise vector.
    noise = tf.random.normal([16, noise_dim])

    # Use gradient tapes for automatic differentiation
    with tf.GradientTape()
            as generator_tape, tf.GradientTape()
            as discriminator_tape:

      # ask genertor to generate random images
      generated_images = gen_model
                         (noise, training=True)
      # ask discriminator to evalute the real images and
      generate its output
      real_output = discri_model(images,
                                 training=True)

      # ask discriminator to do the evlaution on generated
      (fake) images
      fake_output = discri_model(generated_images,
                                  training=True)

      # calculate generator loss on fake data
      gen_loss = generator_loss(fake_output)

      # calculate discriminator loss as defined earlier
      disc_loss = discriminator_loss(real_output,
                                     fake_output)
```

```
# calculate gradients for generator
gen_gradients = generator_tape.gradient
                (gen_loss,
                 gen_model.trainable_variables)

# calculate gradients for discriminator
discri_gradients = discriminator_tape.gradient
               (disc_loss,
               discri_model.trainable_variables)
# use optimizer to process and apply gradients to variables
gen_optimizer.apply_gradients(zip(gen_gradients,
                gen_model.trainable_variables))

# same as above to discriminator
discri_optimizer.apply_gradients(
    zip(discri_gradients,
        discri_model.trainable_variables))

def generate_and_save_images(model, epoch,
                             test_input):
    global epoch_number
    epoch_number = epoch_number + 1

    # set training to false to ensure inference mode
    predictions = model(test_input,
                             training=False)

    # display and save image
    fig = plt.figure(figsize=(4,4))
    for i in range(predictions.shape[0]):
        plt.imshow(predictions[i, :, :, 0]
                   * 127.5 + 127.5, cmap='gray')
        plt.axis('off')
```

```
        plt.savefig('image_at_epoch_
                    {:01d}.png'.format(epoch_number))
        plt.show()
def train(dataset, epochs):
  for epoch in range(epochs):
    start = time.time()

    for image_batch in dataset:
      gradient_tuning(image_batch)

    # Produce images as we go
    generate_and_save_images(gen_model,
                             epoch + 1,
                             seed)

    # save checkpoint data
    checkpoint.save(file_prefix = checkpoint_prefix)
    print ('Time for epoch {} is {} sec'.format
                             (epoch + 1,
                             time.time()-start))

train(train_dataset, EPOCHS)

#run this code only if there is a runtime disconnection
try:
     checkpoint.restore(tf.train.latest_checkpoint
                                (checkpoint_dir))
except Exception as error:
    print("Error loading in model :
                                {}".format(error))
train(train_dataset, EPOCHS)
```

Summary

The Generative Adversarial Network (GAN) provides a novel idea of mimicking any given image. The GAN consists of two networks – generator and discriminator. Both models are trained simultaneously by an adversarial process. In this chapter, you studied to construct a GAN network, which was used for creating handwritten digits, alphabets, and even animes. To train a GAN requires huge processing resources. Yet, it produces very satisfactory results. Today, GANs have been successfully used in many applications. For example, GANs are used for creating large image datasets like the handwritten digits provided on the Kaggle site that you used in the first example in this chapter. It is successful in creating human faces of celebrities. It can be used for generating cartoon characters like the anime example in this chapter. It has also been applied in the areas of image-to-image and text-to-image translations. It can be used for creating emojis from photos. It can also be used for aging faces in the photos. The possibilities are endless; you should explore further to see for yourself how people have used GANs in creating many interesting applications. With the techniques that you have learned in this chapter, you would be able to implement your own ideas to add this repository of endless GAN applications.

CHAPTER 14

Image Translation

Have you ever thought of colorizing the old B&W photograph of your granny? You would probably approach a Photoshop artist to do this job for you, paying them hefty fees and awaiting a couple of days/weeks for them to finish the job. If I tell you that you could do this with a deep neural network, would you not get excited about learning how to do it? Well, this chapter teaches you the technique of converting your B&W images to a colorized image almost instantaneously. The technique is simple and uses a network architecture known as AutoEncoders. So, let us first look at AutoEncoders.

AutoEncoders

AutoEncoders consist of two parts – an Encoder and a Decoder. It may be schematically represented as shown in Figure 14-1.

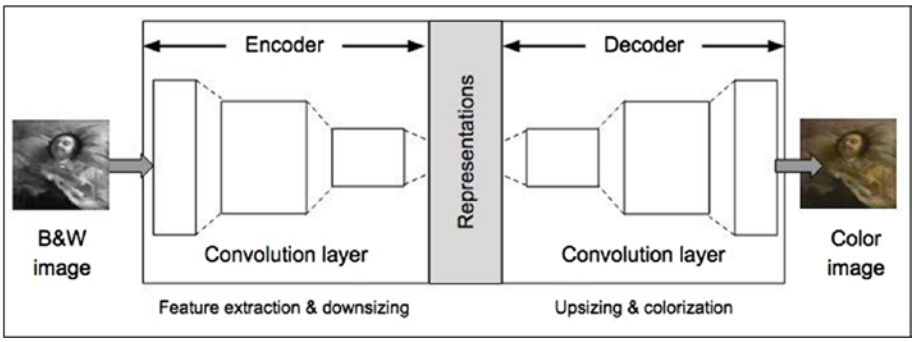


Figure 14-1. *AutoEncoder architecture*

P. Sarang, *Artificial Neural Networks with TensorFlow 2*,
https://doi.org/10.1007/978-1-4842-6150-7_14

On the left, we have a B&W image fed to our network. On the right, where we have the network output, we have a colorized image of the same input content. What goes in between can be described like this. The Encoder processes the image through a series of Convolutional layers and downsizes the image to learn the reduced dimensional representation of the input image. The decoder then attempts to regenerate the image by passing it through another series of Convolutional layers, upsizing and adding colors in the process.

Now, to understand how to colorize an image, you must first understand the color spaces.

Color Spaces

A color image consists of the colors in a given color space and the luminous intensity. A range of colors are created by using the primary colors such as red, green, blue. This entire range of colors is then called a color space, for example, RGB. In mathematical terms, a color space is an abstract mathematical model that simply describes the range of colors as tuples of numbers. Each color is represented by a single dot.

I will describe the three most popular color spaces:

- RGB
- YCbCr
- Lab

RGB is the most commonly used color space. It contains three channels – red (R), green (G), and blue (B). Each channel is represented by 8 bits and can take a maximum value of 256. Combined together, they can represent over 16 million colors.

The JPEG and MPEG formats use the YCbCr color space. It is more efficient for digital transmission and storage as compared to RGB. The Y channel represents the luminosity of a grayscale image. The Cb and Cr

represent the blue and red difference chroma components. The Y channel takes values from 16 through 235. The Cb and Cr values range from 16 to 240. Be careful, the combined value of all these channels may not represent a valid color. In our application of colorization, we don't use this color space.

The Lab color space was designed by the International Commission on Illumination (CIE). A visual representation of this color space is shown in Figure 14-2.

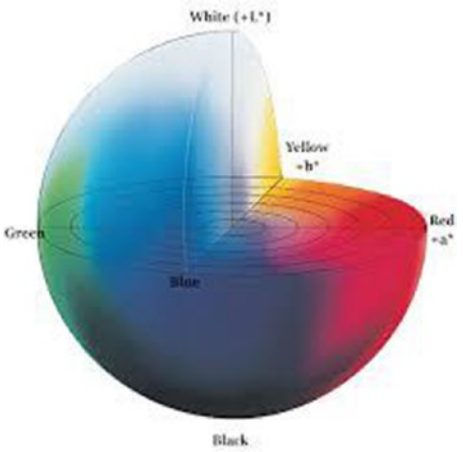


Figure 14-2. *The Lab color space*

The Lab color space is larger than the gamut of computer displays and printers. Then, a bitmap image represented as a Lab requires more data per pixel to obtain the same precision as an RGB or CMYK. Thus, the Lab color space is typically used as an intermediary rather than an end color space.

The L channel represents the luminosity and takes values in the range 0 to 100. The “a” channel codes from green (-) to red (+), and the “b” channel codes from blue (-) to yellow (+). For an 8-bit implementation, both take values in the range -127 to +127. The Lab color space approximates the human vision. The amount of numerical change in these component values corresponds to roughly the same amount of visually perceived change.

We use the Lab color space in our project. By separating the grayscale component which represents the luminosity, the network has to learn only two remaining channels for colorization. This helps in reducing the network size and results in faster convergence.

I will now discuss the different network topologies for our AutoEncoders.

Network Configurations

The AutoEncoder network may be configured in three different ways:

- Vanilla
- Merged
- Merged model using pre-trained network

I will now discuss all three models.

Vanilla Model

The vanilla model has the configuration shown in Figure 14-1, where the Encoder has a series of Convolutional layers with strides for downsizing the image and extracting features. The Decoder too has Convolutional layers which are used for upsizing and colorization. In such autoencoders, the encoders are not deep enough to extract the global features of an image. The global features help us in determining how to colorize certain regions of the image. If we make the encoder network deep, the dimensions of the representation would be too small for the decoder to faithfully reproduce the original image. Thus, we need two paths in an Encoder – one to obtain the global features and the other one to obtain a rich representation of the image. This is what is done in the next two models. You will be constructing a vanilla network for the first project in this chapter.

Merged Model

This model was proposed by Lizuka et al. in their paper "Let there be Color!" (http://iizuka.cs.tsukuba.ac.jp/projects/colorization/data/colorization_sig2016.pdf). The model architecture is shown in Figure 14-3.

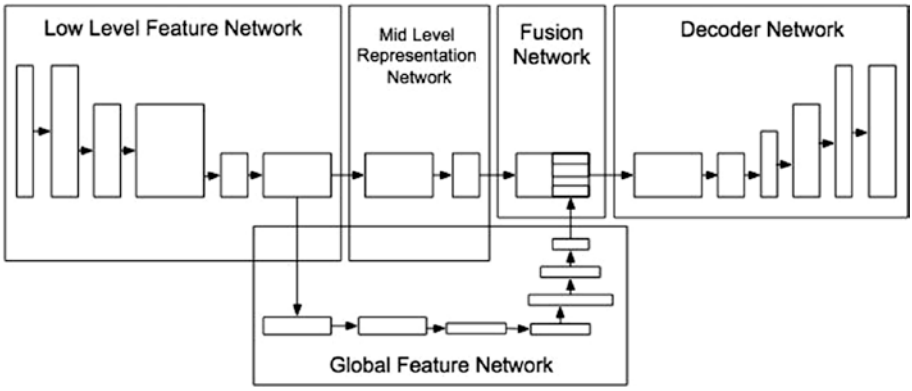


Figure 14-3. *Merged model architecture*

An eight-layer Encoder was used to extract the mid-level representations. The output of the sixth layer is forked and fed through another seven-layer network to extract the global features. Another Fusion network then concatenates the two outputs and feeds them to the decoder.

Merged Model Using Pre-trained Network

This was proposed by Baldassarre et al. in their paper "Deep Koalarization: Image Colorization using CNNs and Inception-Resnet-v2" (https://arxiv.org/pdf/1712.03400.pdf). The schematic diagram of the model architecture is shown in Figure 14-4.

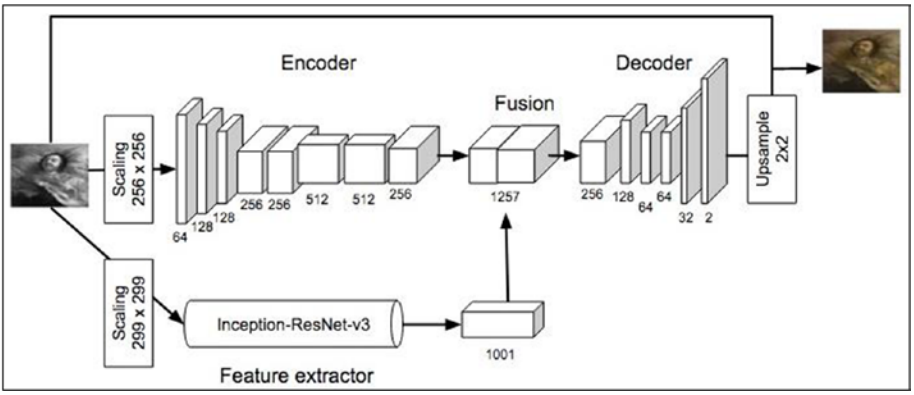


Figure 14-4. *Model using a pre-trained network*

The feature extraction is done by a pre-trained ResNet.

In the second project in this chapter, I will show you how to use a pre-trained model for feature extraction, though you will not be constructing as complicated a model as shown in this schematic.

With this introduction to AutoEncoders and their configurations, let us start with some practical implementations of them.

AutoEncoder

In this project, you will be using the vanilla autoencoder.

Open a new Colab notebook and rename it to AutoEncoder – Custom. Add the following imports:

```
import numpy as np
import pandas as pd
import os

import matplotlib.pyplot as plt

from tqdm import tqdm
from itertools import chain
```

```
import skimage
from skimage.io import imread, imshow
from skimage.transform import resize
from skimage.util import crop, pad
from skimage.morphology import label
from skimage.color import rgb2gray, gray2rgb,
                          rgb2lab, lab2rgb
from sklearn.model_selection import train_test_split

import tensorflow as tf
from tensorflow.keras.models
            import Model, load_model,Sequential
from tensorflow.keras.preprocessing.image
            import ImageDataGenerator
from tensorflow.keras.layers import Input, Dense,
            UpSampling2D, RepeatVector, Reshape
from tensorflow.keras.layers import Dropout, Lambda
from tensorflow.keras.layers import Conv2D,
                                    Conv2DTranspose
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras import backend as K
```

Loading Data

You will be using the dataset provided on the Kaggle site for this project. The site (www.kaggle.com/thedownhill/art-images-drawings-painting-sculpture-engraving) provides a dataset of about 9000 images containing five types of arts. If you have a Kaggle account, you may download the dataset using your credentials with the following code:

```
#!pip install -q kaggle
#!mkdir ~/.kaggle
#!touch ~/.kaggle/kaggle.json
```

```
#api_token = {"username":"Your UserName",
                                  "key":"Your key"}

#import json

#with open('/root/.kaggle/kaggle.json', 'w') as file:
#    json.dump(api_token, file)

#!chmod 600 ~/.kaggle/kaggle.json
#!kaggle datasets download -d thedownhill/art-images-drawings-
painting-sculpture-engraving
```

Optionally, the data is also available on the book's download site and can be downloaded into your project using wget as in the following code fragment:

```
!wget --no-check-certificate -r 'https://drive.google.com/
uc?export=download&id=1CKs7s_MZMuZFBXDchcL_AgmCxgPBTJXK' -O
art-images-drawings-painting-sculpture-engraving.zip
```

After the data file is downloaded, unzip it to your drive using the unzip utility:

```
!unzip art-images-drawings-painting-sculpture-engraving.zip
```

When the file is unzipped, you will have lots of images stored on your drive arranged in a specific folder structure. The images have varied sizes. We will convert all our training images to a fixed size of 256x256. We define a few variables for creating our training dataset as follows:

```
IMG_WIDTH = 256
IMG_HEIGHT = 256
TRAIN_PATH =
'/content/dataset/dataset_updated/training_set/painting/'
train_ids = next(os.walk(TRAIN_PATH))[2]
```

The os.walk gets all the filenames present in the folder.

We will first check if there are any bad images (unreadable) in the code and remove those from our dataset, though this step is not truly required for our purpose.

```
missing_count = 0
for n, id_ in tqdm(enumerate(train_ids),
                        total=len(train_ids)):
    path = TRAIN_PATH + id_+''
    try:
        img = imread(path)
    except:
        missing_count += 1

print("\n\nTotal missing: "+ str(missing_count))
```

When you run this code, you will discover that there are 86 bad images in the set.

Now, we will create the training set taking care to remove the bad images.

```
X_train = np.zeros((len(train_ids)-missing_count,
          IMG_HEIGHT, IMG_WIDTH, 3), dtype=np.uint8)
missing_images = 0
for n, id_ in tqdm(enumerate(train_ids),
                          total=len(train_ids)):
    path = TRAIN_PATH + id_+''
    try:
        img = imread(path)
        img = resize(img, (IMG_HEIGHT, IMG_WIDTH),
                 mode='constant', preserve_range=True)
        X_train[n-missing_images] = img
    except:
        missing_images += 1

X_train = X_train.astype('float32') / 255.
```

You may now examine how the image looks like using the following statement:

```
plt.imshow(X_train[5])
```

The output is shown in Figure 14-5.

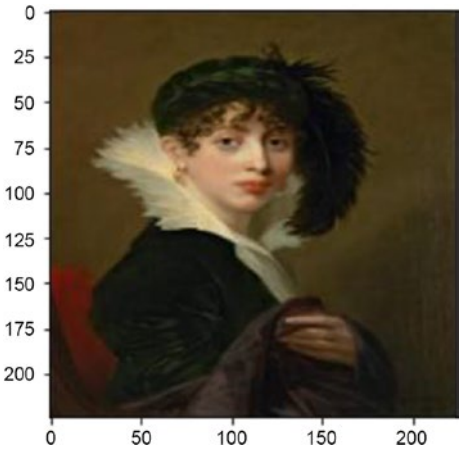


Figure 14-5. *A sample image*

Creating Training/Testing Datasets

We will just reserve a few images for testing from the dataset that we have created.

```
x_train, x_test = train_test_split(X_train,
                                   test_size=20)
```

The train_test_split method as specified in the test_size parameter reserves 20 images for testing.

Preparing Training Dataset

For training the model, we will convert the images from RGB to Lab format. As said earlier, the L channel is the grayscale. It represents the luminance of the image. The "a" is the color balance between green and red, and the "b" is the color balance between blue and yellow.

First, we will create an instance of ImageDataGenerator from the Keras library to convert the images into an array of pixels and finally combine them into a giant vector.

```
datagen = ImageDataGenerator(
        shear_range=0.2,
        zoom_range=0.2,
        rotation_range=20,
        horizontal_flip=True)
```

If each image is skewed, the model will learn better. The shear_range tilts the image to the left or right, and the other parameters zoom, rotation, and horizontal flip have their respective meanings.

Now, we will write a function for creating batches of data for training. The function definition is given as follows:

```
def create_training_batches(dataset=X_train,
                                batch_size = 20):
    # iteration for every image
    for batch in datagen.flow(dataset, batch_size=batch_size):
        # convert from rgb to grayscale
        X_batch = rgb2gray(batch)
        # convert rgb to Lab format
        lab_batch = rgb2lab(batch)
        # extract L component
        X_batch = lab_batch[:,:,:,0]
        # reshape
```

```
        X_batch = X_batch.reshape(X_batch.shape+(1,))
        # extract a and b features of the image
        Y_batch = lab_batch[:,:,:,1:] / 128
        yield X_batch, Y_batch
```

The function first converts the given image from RGB to grayscale by calling the rgb2gray method. The image is then converted to Lab format by calling the rgb2lab method. If we take Lab color space, we need to predict only two components as compared to other color spaces where we would need to predict three or four components. As said earlier, this helps in reducing the network size and results in a faster convergence. Finally, we extract the L, a, and b components from the image.

Defining Model

Now, we will define our Autoencoder model. The model configuration is based on the suggestions made in the paper "Let there be Color!" (http://iizuka.cs.tsukuba.ac.jp/projects/colorization/data/colorization_sig2016.pdf).

```
# the input for the encoder layer
inputs1 = Input(shape=(IMG_WIDTH, IMG_HEIGHT, 1,))

# encoder

# Using Conv2d to reduce the size of feature maps and image
size
# convert image to 128x128
encoder_output = Conv2D(64, (3,3), activation='relu',
                  padding='same', strides=2)(inputs1)
encoder_output = Conv2D(128, (3,3),
                  activation='relu',
                  padding='same')(encoder_output)
```

```
# convert image to 64x64
encoder_output = Conv2D(128, (3,3),
                  activation='relu', padding='same',
                  strides=2)(encoder_output)
encoder_output = Conv2D(256, (3,3),
                  activation='relu',
                  padding='same')(encoder_output)
# convert image to 32x32
encoder_output = Conv2D(256, (3,3),
                  activation='relu', padding='same',
                  strides=2)(encoder_output)
encoder_output = Conv2D(512, (3,3),
                  activation='relu', padding='same')
                  (encoder_output)

# mid-level feature extractions
encoder_output = Conv2D(512, (3,3),
                  activation='relu',
                  padding='same')(encoder_output)
encoder_output = Conv2D(256, (3,3),
                  activation='relu',
                  padding='same')(encoder_output)

# decoder

# Adding colors to the grayscale image and upsizing it
decoder_output = Conv2D(128, (3,3),
                  activation='relu',
                  padding='same')(encoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
# image size 64x64
decoder_output = Conv2D(64, (3,3), activation='relu',
```

```
                  padding='same')(decoder_output)
decoder_output = Conv2D(64, (3,3), activation='relu',
                  padding='same')(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
# image size 128x128
decoder_output = Conv2D(32, (3,3), activation='relu',
                  padding='same')(decoder_output)
decoder_output = Conv2D(2, (3, 3), activation='tanh',
                  padding='same')(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
# image size 256x256
```

Both the Encoder and Decoder contain few Conv2D layers. The Encoder through a series of layers downsamples the image to extract its features, and the Decoder through its own set of layers attempts to regenerate the original image using upsampling at various points and adding colors to the grayscale image to create a final image of size 256x256. The last decoder layer uses tanh activation for squashing the values between –1 and +1. Remember that we had earlier normalized the a and b values in the range –1 through +1.

After the Encoder and Decoder layers are defined, construct the model and compile it using its compile method. We use mse for the loss function and Adam optimizer.

```
model = Model(inputs=inputs1, outputs=decoder_output)
model.compile(loss='mse', optimizer='adam',
                          metrics=['accuracy'])
print(model.summary())
```

The model summary is shown in Figure 14-6.

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 256, 256, 1)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 128, 128, 64)      640
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 128, 128, 128)     73856
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 64, 64, 128)       147584
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 64, 64, 256)       295168
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 32, 32, 256)       590080
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 32, 32, 512)       1180160
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 32, 32, 512)       2359808
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 32, 32, 256)       1179904
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 32, 32, 128)       295040
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 64, 64, 128)       0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 64, 64, 64)        73792
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 64, 64, 64)        36928
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, 128, 128, 64)      0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 128, 128, 32)      18464
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 128, 128, 2)       578
_________________________________________________________________
up_sampling2d_3 (UpSampling2 (None, 256, 256, 2)       0
=================================================================
Total params: 6,252,002
Trainable params: 6,252,002
Non-trainable params: 0
_________________________________________________________________
None
```

Figure 14-6. *Autoencoder model summary*

You can get the visualization by plotting the model:

```
tf.keras.utils.plot_model(model)
```

The output is shown in Figure 14-7.

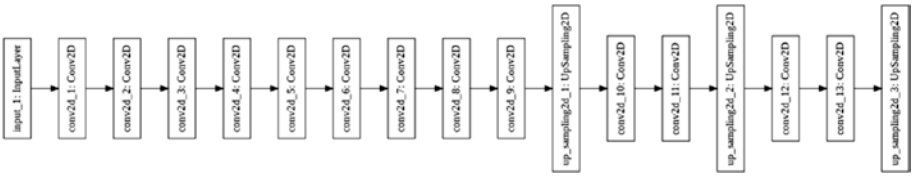


Figure 14-7. *Model plot*

Model Training

We train the model by calling its fit method.

```
BATCH_SIZE = 20
model.fit_generator(create_training_batches
                        (X_train,BATCH_SIZE),
            epochs= 100,
            verbose=1,
            steps_per_epoch=X_train.shape[0]/BATCH_SIZE)
```

It took me slightly over a minute per epoch to train the model on a GPU. By using the pre-trained model, this training time came down to about a second per epoch as you would see when you run the second project in this chapter.

Testing

Now, you can check the model performance on the test dataset that we have created earlier. Note that for the test images, we do not skew them as we did for the training. We simply convert the images to Lab format and do the prediction. Here is the code for model predictions on test images.

```
test_image = rgb2lab(x_test)[:,:,:,0]
test_image = test_image.reshape
                (test_image.shape+(1,))
output = model.predict(test_image)
output = output * 128

# making the output image array
generated_images = np.zeros
                        ((len(output),256, 256, 3))

for i in range(len(output)):
#iterating for the output
    cur = np.zeros((256, 256, 3))
# dummy array
    cur[:,:,0] = test_image[i][:,:,0]
#assigning the gray scale component
    cur[:,:,1:] = output[i]
#assigning the a and b component
#converting from lab to rgb format as plt only work for rgb
mode
    generated_images[i] = lab2rgb(cur)
```

Display the generated images along with the originals using the following code fragment:

```
plt.figure(figsize=(20, 6))
for i in range(10):
    # grayscale
    plt.subplot(3, 10, i + 1)
    plt.imshow(rgb2gray(x_test)[i].reshape(256, 256))
    plt.gray()
    plt.axis('off')
```

```
    # recolorization
    plt.subplot(3, 10, i + 1 +10)
    plt.imshow(generated_images[i].reshape
                                    (256, 256,3))
    plt.axis('off')

    # original
    plt.subplot(3, 10, i + 1 + 20)
    plt.imshow(x_test[i].reshape(256, 256,3))
    plt.axis('off')

plt.tight_layout()
plt.show()
```

The output is shown in Figure 14-8.

Figure 14-8. *Model inference*

The first row is the set of grayscale images created from the original color images given in the third row. The middle row shows the images generated by the model. As you can see, the model is able to generate the images close enough to the original images.

Now, I will show you how to use this model on an unseen image of a different size.

Inference on an Unseen Image

You can test the model's performance on an unseen image of your choice. A sample image is available on the book's site, which can be downloaded using wget.

```
!wget https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch14/mountain.jpg
```

Display the original image:

```
img = imread("mountain.jpg")
plt.imshow(img)
```

The image is shown in Figure 14-9.

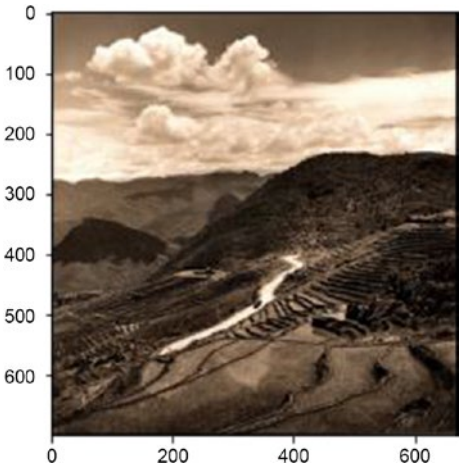


Figure 14-9. *Sample image with different dimensions*

Now, run the inference using the following code. Note that we need to change the image size before inputting the image to the network.

```
img = resize(img, (IMG_HEIGHT, IMG_WIDTH),
             mode='constant', preserve_range=True)
img = img.astype('float32') / 255.

test_image = rgb2lab(img)[:,:,0]
test_image = test_image.reshape
                    ((1,)+test_image.shape+(1,))
output = model.predict(test_image)
output = output * 128

plt.imshow(img)
plt.axis('off')
```

The generated image is shown in Figure 14-10.

Figure 14-10. *A colorized image generated by the custom autoencoder model*

Full Source

The full source is given in Listing 14-1 for your reference.

Listing 14-1. AutoEncoder_Custom

```
import numpy as np
import pandas as pd
import os

import matplotlib.pyplot as plt

from tqdm import tqdm
from itertools import chain
import skimage
from skimage.io import imread, imshow
from skimage.transform import resize
from skimage.util import crop, pad
from skimage.morphology import label
from skimage.color import rgb2gray, gray2rgb,
                          rgb2lab, lab2rgb
from sklearn.model_selection import train_test_split

import tensorflow as tf
from tensorflow.keras.models
            import Model, load_model,Sequential
from tensorflow.keras.preprocessing.image
            import ImageDataGenerator
from tensorflow.keras.layers import Input, Dense,
            UpSampling2D, RepeatVector, Reshape
from tensorflow.keras.layers import Dropout, Lambda
from tensorflow.keras.layers import Conv2D,
                                    Conv2DTranspose
```

```
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras import backend as K

#!pip install -q kaggle
#!mkdir ~/.kaggle
#!touch ~/.kaggle/kaggle.json

#api_token = {"username":"Your UserName",
                                  "key":"Your key"}

#import json

#with open('/root/.kaggle/kaggle.json', 'w') as file:
#    json.dump(api_token, file)

#!chmod 600 ~/.kaggle/kaggle.json
#!kaggle datasets download -d thedownhill/art-images-drawings-
painting-sculpture-engraving

!wget --no-check-certificate -r 'https://drive.google.com/
uc?export=download&id=1CKs7s_MZMuZFBXDchcL_AgmCxgPBTJXK' -O
art-images-drawings-painting-sculpture-engraving.zip

!unzip art-images-drawings-painting-sculpture-engraving.zip

IMG_WIDTH = 256
IMG_HEIGHT = 256
TRAIN_PATH =
'/content/dataset/dataset_updated/training_set/painting/'
train_ids = next(os.walk(TRAIN_PATH))[2]

missing_count = 0
for n, id_ in tqdm(enumerate(train_ids),
                         total=len(train_ids)):
    path = TRAIN_PATH + id_+''
```

```
    try:
        img = imread(path)
    except:
        missing_count += 1

print("\n\nTotal missing: "+ str(missing_count))

X_train = np.zeros((len(train_ids)-missing_count,
          IMG_HEIGHT, IMG_WIDTH, 3), dtype=np.uint8)
missing_images = 0
for n, id_ in tqdm(enumerate(train_ids),
                         total=len(train_ids)):
    path = TRAIN_PATH + id_+''
    try:
        img = imread(path)
        img = resize(img, (IMG_HEIGHT, IMG_WIDTH),
                mode='constant', preserve_range=True)
        X_train[n-missing_images] = img
    except:
        missing_images += 1

X_train = X_train.astype('float32') / 255.

plt.imshow(X_train[5])

x_train, x_test = train_test_split(X_train,
                                   test_size=20)

datagen = ImageDataGenerator(
        shear_range=0.2,
        zoom_range=0.2,
        rotation_range=20,
        horizontal_flip=True)
```

```
def create_training_batches(dataset=X_train,
                                    batch_size = 20):
    # iteration for every image
    for batch in datagen.flow(dataset, batch_size=batch_size):
        # convert from rgb to grayscale
        X_batch = rgb2gray(batch)
        # convert rgb to Lab format
        lab_batch = rgb2lab(batch)
        # extract L component
        X_batch = lab_batch[:,:,:,0]
        # reshape
        X_batch = X_batch.reshape(X_batch.shape+(1,))
        # extract a and b features of the image
        Y_batch = lab_batch[:,:,:,1:] / 128
        yield X_batch, Y_batch

# the input for the encoder layer
inputs1 = Input(shape=(IMG_WIDTH, IMG_HEIGHT, 1,))

# encoder

# Using Conv2d to reduce the size of feature maps and image
size
# convert image to 128x128
encoder_output = Conv2D(64, (3,3), activation='relu',
                  padding='same', strides=2)(inputs1)
encoder_output = Conv2D(128, (3,3),
                  activation='relu',
                  padding='same')(encoder_output)
# convert image to 64x64
encoder_output = Conv2D(128, (3,3),
                  activation='relu', padding='same',
                  strides=2)(encoder_output)
```

```
encoder_output = Conv2D(256, (3,3),
                  activation='relu',
                  padding='same')(encoder_output)
# convert image to 32x32
encoder_output = Conv2D(256, (3,3),
                  activation='relu', padding='same',
                  strides=2)(encoder_output)
encoder_output = Conv2D(512, (3,3),
                  activation='relu', padding='same')
                  (encoder_output)

# mid-level feature extractions
encoder_output = Conv2D(512, (3,3),
                  activation='relu',
                  padding='same')(encoder_output)
encoder_output = Conv2D(256, (3,3),
                  activation='relu',
                  padding='same')(encoder_output)

# decoder

# Adding colors to the grayscale image and upsizing it
decoder_output = Conv2D(128, (3,3),
                  activation='relu',
                  padding='same')(encoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
# image size 64x64
decoder_output = Conv2D(64, (3,3), activation='relu',
                  padding='same')(decoder_output)
decoder_output = Conv2D(64, (3,3), activation='relu',
                  padding='same')(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
```

```
# image size 128x128
decoder_output = Conv2D(32, (3,3), activation='relu',
                  padding='same')(decoder_output)
decoder_output = Conv2D(2, (3, 3), activation='tanh',
                  padding='same')(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
# image size 256x256

# compiling  model
model = Model(inputs=inputs1, outputs=decoder_output)
model.compile(loss='mse', optimizer='adam',
                          metrics=['accuracy'])
print(model.summary())

tf.keras.utils.plot_model(model)


BATCH_SIZE = 20
model.fit_generator(create_training_batches
                        (X_train,BATCH_SIZE),
            epochs= 100,
            verbose=1,
            steps_per_epoch=X_train.shape[0]/BATCH_SIZE)

test_image = rgb2lab(x_test)[:,:,:,0]
test_image = test_image.reshape
               (test_image.shape+(1,))
output = model.predict(test_image)
output = output * 128

# making the output image array
generated_images = np.zeros
                        ((len(output),256, 256, 3))
```

```
for i in range(len(output)):
#iterating for the output
    cur = np.zeros((256, 256, 3))
# dummy array
    cur[:,:,0] = test_image[i][:,:,0]
#assigning the gray scale component
    cur[:,:,1:] = output[i]
#assigning the a and b component
#converting from lab to rgb format as plt only work for rgb
mode
    generated_images[i] = lab2rgb(cur)

plt.figure(figsize=(20, 6))
for i in range(10):
    # grayscale
    plt.subplot(3, 10, i + 1)
    plt.imshow(rgb2gray(x_test)[i].reshape(256, 256))
    plt.gray()
    plt.axis('off')

    # recolorization
    plt.subplot(3, 10, i + 1 +10)
    plt.imshow(generated_images[i].reshape
                                    (256, 256,3))
    plt.axis('off')


    # original
    plt.subplot(3, 10, i + 1 + 20)
    plt.imshow(x_test[i].reshape(256, 256,3))
    plt.axis('off')

plt.tight_layout()
plt.show()
```

```
!wget https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch14/mountain.jpg

img = imread("mountain.jpg")
plt.imshow(img)

img = resize(img, (IMG_HEIGHT, IMG_WIDTH),
             mode='constant', preserve_range=True)
img = img.astype('float32') / 255.

test_image = rgb2lab(img)[:,:,0]
test_image = test_image.reshape
                    ((1,)+test_image.shape+(1,))
output = model.predict(test_image)
output = output * 128
plt.imshow(img)
plt.axis('off')
```

Now, I will show you how to use a pre-trained model for features extraction, thereby saving you a lot of training time and giving better feature extraction.

Pre-trained Model as Encoder

There are several pre-trained models available for image processing. You have used one such VGG16 model in Chapter 12. The use of this model allows you to extract the image features, and that is what we did in our previous program by creating our own encoder. So why not use the transfer learning by using a VGG16 pre-trained model in place of an encoder? And that is what I am going to demonstrate in this application. The use of a pre-trained model would certainly provide better results as compared to your own defined encoder and a faster training too.

Project Description

You will be using the same image dataset as in the previous project. Thus, the data loading and preprocessing code would remain the same. What changes is the model definition and the inference. So, I will describe only the relevant changes. The entire project source is available in the book's download site and also given at the end of this section for your quick reference. The project is named AutoEncoder-TransferLearning.

As the VGG16 was trained on images of size 224x224, you will need to change those two constant values to the following:

```
IMG_WIDTH = 224
IMG_HEIGHT = 224
```

Defining Model

You have already seen the VGG16 architecture in Chapter 12 (Figure 12-5). The first 18 layers of the VGG model extract the image features. So, we will use these layers and discard all subsequent layers. We create a new sequential model using the following code snippet:

```
vggmodel = tf.keras.applications.vgg16.VGG16()
newmodel = Sequential()
num = 0
for i, layer in enumerate(vggmodel.layers):
    if i<19:
      newmodel.add(layer)
newmodel.summary()
for layer in newmodel.layers:
  layer.trainable=False
```

We set the trainable parameter for all these layers to false as we intend to use a pre-trained model for feature extraction. The model summary is shown in Figure 14-11.

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

Figure 14-11. *Pre-trained encoder model summary*

Extracting Features

You will extract the features in the training dataset images by using the newmodel that you have created. We just pass each training image through the network and collect the predictions at layer 19.

```
vggfeatures = []
for sample in x_train:
  sample = gray2rgb(sample)
  sample = sample.reshape((1,224,224,3))
  prediction = newmodel.predict(sample)
  prediction = prediction.reshape((7,7,512))
  vggfeatures.append(prediction)
vggfeatures = np.array(vggfeatures)
```

Defining Network

Now, you will define our encoder/decoder architecture as follows:

```
#Encoder
encoder_input = Input(shape=(7, 7, 512,))
#Decoder
decoder_output = Conv2D(256, (3,3),
                  activation='relu', padding='same')
                        (encoder_input)
decoder_output = Conv2D(128, (3,3),
                  activation='relu', padding='same')
                        (decoder_output)
```

```
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(64, (3,3), activation='relu',
                  padding='same')(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(32, (3,3), activation='relu',
                  padding='same')(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(16, (3,3), activation='relu',
                  padding='same')(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(2, (3, 3), activation='tanh',
                  padding='same')(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
model = Model(inputs=encoder_input,
                  outputs=decoder_output)
model.summary()
```

For the encoder, we just specify the input, and the decoder architecture is the same as in the previous example where we keep on upsizing the image and adding colors to it.

The model summary is shown in Figure 14-12.

```
Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         (None, 7, 7, 512)         0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 7, 7, 256)         1179904
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 7, 7, 128)         295040
_________________________________________________________________
up_sampling2d_6 (UpSampling2 (None, 14, 14, 128)       0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 14, 14, 64)        73792
_________________________________________________________________
up_sampling2d_7 (UpSampling2 (None, 28, 28, 64)        0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 28, 28, 32)        18464
_________________________________________________________________
up_sampling2d_8 (UpSampling2 (None, 56, 56, 32)        0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 56, 56, 16)        4624
_________________________________________________________________
up_sampling2d_9 (UpSampling2 (None, 112, 112, 16)      0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 112, 112, 2)       290
_________________________________________________________________
up_sampling2d_10 (UpSampling (None, 224, 224, 2)       0
=================================================================
Total params: 1,572,114
Trainable params: 1,572,114
Non-trainable params: 0
```

Figure 14-12. *Encoder decoder model summary*

Model Training

Compile and train the model using the following two statements:

```
model.compile(optimizer='Adam', loss='mse')
model.fit(vggfeatures, image_a_b_gen(x_train),
          verbose=1, epochs=100, batch_size=128)
```

We use the Adam optimizer and the mse loss for training. Training the network on a GPU, the epoch time was about a second – a considerable improvement over our earlier network. As we have used a pre-trained encoder, only the decoder parameters need to be trained.

Inference

Now, run the following code to generate the images from the test dataset. The code is trivial enough to understand.

```
sample = x_test[1:6]
for image in sample:
  lab = rgb2lab(image)
  l = lab[:,:,0]
  L = gray2rgb(l)
  L = L.reshape((1,224,224,3))
  vggpred = newmodel.predict(L)
  ab = model.predict(vggpred)
  ab = ab*128
  cur = np.zeros((224, 224, 3))
  cur[:,:,0] = l
  cur[:,:,1:] = ab
  plt.subplot(1,2,1)
  plt.title("Generated Image")
  plt.imshow( lab2rgb(cur))
  plt.axis('off')
  plt.subplot(1,2,2)
  plt.title("Original Image")
  plt.imshow(image)
  plt.axis('off')
  plt.show()
```

The output of the preceding code is shown in Figure 14-13.

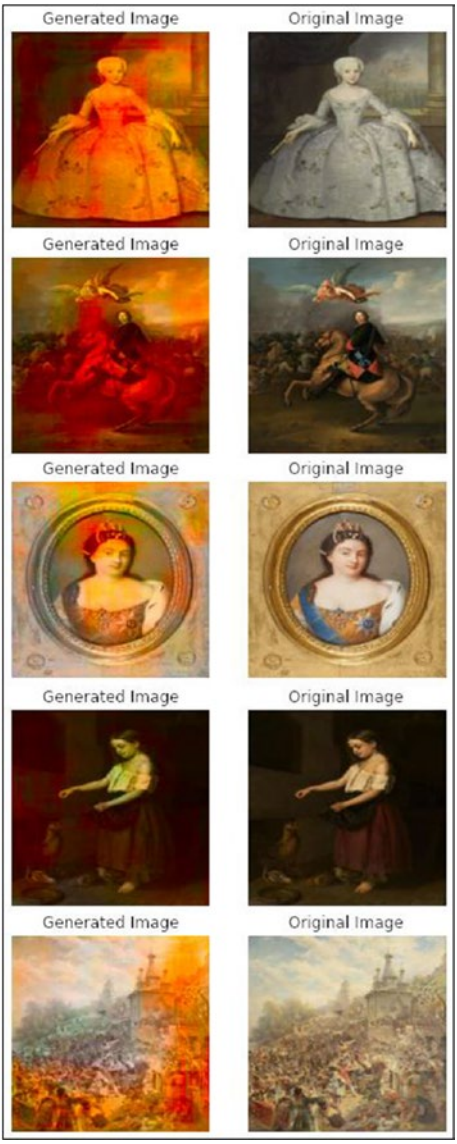


Figure 14-13. *Model inference on test images*

Inference on an Unseen Image

Like the earlier example, test the model's performance on an unseen image of your choice. We will use the same image as in the earlier example.

```
!wget https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch14/mountain.jpg
```

Display the original image if you wish to see it again.

```
img = imread("mountain.jpg")
plt.imshow(img)
```

Now, run the inference using the following code. Note that we need to change the image size to 224x224 before inputting the image to the network.

```
test = img_to_array(load_img("mountain.jpg"))
test = resize(test, (224,224), anti_aliasing=True)
test*= 1.0/255
lab = rgb2lab(test)
l = lab[:,:,0]
L = gray2rgb(l)
L = L.reshape((1,224,224,3))
vggpred = newmodel.predict(L)
ab = model.predict(vggpred)
ab = ab*128
cur = np.zeros((224, 224, 3))
cur[:,:,0] = l
cur[:,:,1:] = ab
plt.imshow( lab2rgb(cur))
plt.axis('off')
```

The program output is shown in Figure 14-14.

Figure 14-14. *A colorized image generated by the autoencoder transfer learning model*

Full Source

The full source is given in Listing 14-2 for your reference.

Listing 14-2. AutoEncoder_TransferLearning

```
import numpy as np
import pandas as pd
import cv2
import os
import sys

import matplotlib.pyplot as plt

from tqdm import tqdm
from itertools import chain
import skimage
from PIL import Image
from skimage.io import imread, imshow,
        imread_collection, concatenate_images
from skimage.transform import resize
from skimage.util import crop, pad
```

```
from skimage.morphology import label
from skimage.color import rgb2gray, gray2rgb,
        rgb2lab, lab2rgb
from sklearn.model_selection import train_test_split

from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.preprocessing.image
        import load_img
from tensorflow.keras.preprocessing.image
        import img_to_array
from tensorflow.keras.applications.vgg16
        import preprocess_input

import tensorflow as tf
from tensorflow.keras.models
        import Model, load_model,Sequential
from tensorflow.keras.preprocessing.image
        import ImageDataGenerator
from tensorflow.keras.layers import Input, Dense,
        UpSampling2D, RepeatVector, Reshape
from tensorflow.keras.layers import Dropout, Lambda
from tensorflow.keras.layers
        import Conv2D, Conv2DTranspose
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.layers import concatenate
from tensorflow.keras import backend as K
#!pip install -q kaggle
#!mkdir ~/.kaggle
#!touch ~/.kaggle/kaggle.json

#api_token = {"username":"","key":""}

#import json
```

```
#with open('/root/.kaggle/kaggle.json', 'w') as file:
#    json.dump(api_token, file)

#!chmod 600 ~/.kaggle/kaggle.json
#!kaggle datasets download -d thedownhill/art-images-drawings-
painting-sculpture-engraving

!wget --no-check-certificate -r 'https://drive.google.com/
uc?export=download&id=1CKs7s_MZMuZFBXDchcL_AgmCxgPBTJXK' -O
art-images-drawings-painting-sculpture-engraving.zip
!unzip art-images-drawings-painting-sculpture-engraving.zip

IMG_WIDTH = 224
IMG_HEIGHT = 224
TRAIN_PATH =
'/content/dataset/dataset_updated/training_set/painting/'
train_ids = next(os.walk(TRAIN_PATH))[2]

missing_count = 0
for n, id_ in tqdm(enumerate(train_ids),
                        total=len(train_ids)):
    path = TRAIN_PATH + id_+''
    try:
        img = imread(path)
    except:
        missing_count += 1

print("\n\nTotal missing: "+ str(missing_count))
X_train = np.zeros((len(train_ids)-missing_count,
          IMG_HEIGHT, IMG_WIDTH, 3), dtype=np.uint8)
missing_images = 0
for n, id_ in tqdm(enumerate(train_ids),
                        total=len(train_ids)):
    path = TRAIN_PATH + id_+''
```

```
    try:
        img = imread(path)
        img = resize(img, (IMG_HEIGHT, IMG_WIDTH),
                             mode='constant',
                             preserve_range=True)
        X_train[n-missing_images] = img
    except:
        missing_images += 1

X_train = X_train.astype('float32') / 255.

plt.imshow(X_train[5])

x_train, x_test = train_test_split
                         (X_train, test_size=1500)

datagen = ImageDataGenerator(
        shear_range=0.2,
        zoom_range=0.2,
        rotation_range=20,
        horizontal_flip=True)
def image_a_b_gen(dataset=X_train):
    # iteration for every image
    for batch in datagen.flow(dataset, batch_size=542):
        # convert from rgb to grayscale
        X_batch = rgb2gray(batch)
        # convert the rgb to Lab format
        lab_batch = rgb2lab(batch)

        X_batch = lab_batch[:,:,:,1:] /128

        return X_batch
```

```
vggmodel = tf.keras.applications.vgg16.VGG16()
newmodel = Sequential()
num = 0
for i, layer in enumerate(vggmodel.layers):
    if i<19:
      newmodel.add(layer)
newmodel.summary()
for layer in newmodel.layers:
  layer.trainable=False

vggfeatures = []
for sample in x_train:
  sample = gray2rgb(sample)
  sample = sample.reshape((1,224,224,3))
  prediction = newmodel.predict(sample)
  prediction = prediction.reshape((7,7,512))
  vggfeatures.append(prediction)
vggfeatures = np.array(vggfeatures)
#Encoder
encoder_input = Input(shape=(7, 7, 512,))
#Decoder
decoder_output = Conv2D(256, (3,3),
                  activation='relu', padding='same')
                        (encoder_input)
decoder_output = Conv2D(128, (3,3),
                  activation='relu', padding='same')
                        (decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(64, (3,3), activation='relu',
                  padding='same')(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(32, (3,3), activation='relu',
```

```
                    padding='same')(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(16, (3,3), activation='relu',
                    padding='same')(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(2, (3, 3), activation='tanh',
                    padding='same')(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
model = Model(inputs=encoder_input,
                    outputs=decoder_output)
model.summary()
model.compile(optimizer='Adam', loss='mse')
model.fit(vggfeatures, image_a_b_gen(x_train),
            verbose=1, epochs=100, batch_size=128)

sample = x_test[1:6]
for image in sample:
  lab = rgb2lab(image)
  l = lab[:,:,0]
  L = gray2rgb(l)
  L = L.reshape((1,224,224,3))
  vggpred = newmodel.predict(L)
  ab = model.predict(vggpred)
  ab = ab*128
  cur = np.zeros((224, 224, 3))
  cur[:,:,0] = l
  cur[:,:,1:] = ab
  plt.subplot(1,2,1)
  plt.title("Generated Image")
  plt.imshow( lab2rgb(cur))
  plt.axis('off')
  plt.subplot(1,2,2)
```

```
  plt.title("Original Image")
  plt.imshow(image)
  plt.axis('off')
  plt.show()
!wget https://raw.githubusercontent.com/Apress/artificial-
neural-networks-with-tensorflow-2/main/ch14/mountain.jpg

img = imread("mountain.jpg")
plt.imshow(img)

test = img_to_array(load_img("mountain.jpg"))
test = resize(test, (224,224), anti_aliasing=True)
test*= 1.0/255
lab = rgb2lab(test)
l = lab[:,:,0]
L = gray2rgb(l)
L = L.reshape((1,224,224,3))
vggpred = newmodel.predict(L)
ab = model.predict(vggpred)
ab = ab*128
cur = np.zeros((224, 224, 3))
cur[:,:,0] = l
cur[:,:,1:] = ab
plt.imshow( lab2rgb(cur))
plt.axis('off')
```

Summary

The use of deep neural networks makes it possible to add colors to a B&W image. You learned to create AutoEncoders, which we used for colorizing B&W images. The AutoEncoder contains an Encoder to extract the image features, and the Decoder recreates the image using the representations extracted from the encoder. You also learned how to use a pre-trained image classifier to extract the image features for using it as a part of the Encoder.

Index

A

P. Sarang, *Artificial Neural Networks with TensorFlow 2*,
https://doi.org/10.1007/978-1-4842-6150-7

B

C

D

E

F

G

H

I, J

K

L

M

N

O

P, Q

R

S

T

U

V

W, X, Y, Z