DEEP LEARNING WITH PYTHON

Advanced and Effective Strategies of Using Deep Learning with Python Theories

ETHAN WILLIAMS

Book Description

This book discusses the intricacies of the internal workings of a deep learning model. It addresses the techniques and methods that can not only boost the productivity of your machine learning architectural skills, but also introduces new concepts. Implemented correctly, these can set your deep learning model a league apart from all other models.

This book not only focuses on theoretical and conceptual realms of such knowledge, but also gives equal importance to putting this information to the test. We do this by including some common practical examples and demonstrations that you would normally build deep learning for, hence giving you the best of both worlds. The main features of this book include:

- Refreshing the fundamentals of a deep learning model and neural networks and connecting them with the advanced knowledge laid out in this book, reinforcing the reader's prior knowledge and transforming it into an expert-level understanding.
- Emphasizing those tasks that are commonly demanded from deep learning models and breathing new life into them by introducing new techniques, methods, and elements that enable the model to drastically improve the performance of deep learning models on such tasks.
- Discussing experimentally tested methods that are known to have a positive impact on the model's effectiveness.
- No usage of mathematical notations in the examples detailed in this book so that the concepts can be readily assimilated and mastered by programmers that do not have a mathematical background, hence prioritizing clarity of concepts.
- Keeping this requirement in mind, the examples use Numpy code throughout as it best represents what the code actually means and its purpose.

If you want to learn advanced strategies for Python this is the book for you. Click the Buy Now button to get started today!

Deep Learning With Python

Advanced and Effective Strategies of Using Deep Learning with Python Theories

Table of Contents

Introduction

In this book, we will explore the advanced theories relating to Deep learning in Python and reinforce these theoretical concepts on some fundamental experiments and practices to build a better understanding of the intricate workings of these theories. As such, we will not only learn about useful tools, functions, techniques, and methods that can be used in deep learning models. We will also put this knowledge into a practical demonstration as it would clarify arbitrary concepts that would otherwise be impossible to convey through the use of words. We will start our exploration of these concepts in an orderly fashion and subtly explain new concepts and link them with our existing background knowledge regarding models, networks, and such to build a steady and clear comprehension. The initial part of this book focuses more on theoretical discourse. As we move on, we will begin immersing ourselves in solving some common real-world problems that use deep learning models to resolve, strengthening not only the knowledge of theories but also giving due importance to the practical implementation of this information.

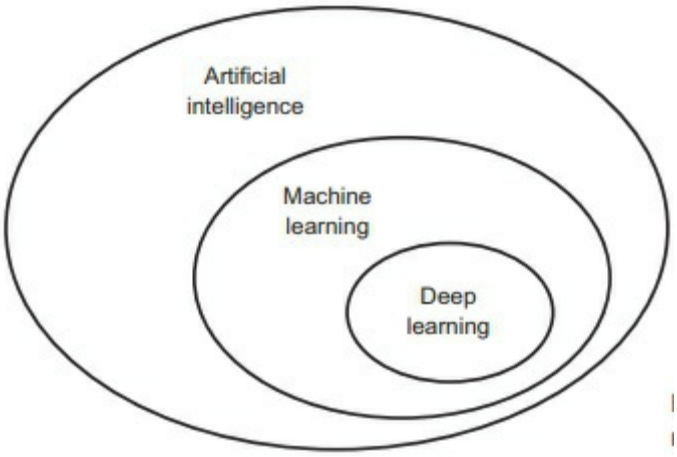
The goal of each chapter is to take the concepts that the readers might be familiar with and, after briefly explaining them, introduce new concepts that offer advanced functionality, enabling the readers to understand what exactly has changed and how it impacts the network’s and the model’s working. By building up a strong baseline in the beginning chapters, we pick up the pace and learn more than before by the ending chapter.

Chapter 1: A General Overview of Deep Learning

The main focus of this chapter is to provide the reader with a general outline of the roadmap of Deep Learning through an informative discussion while journeying through the first chapter of the book. Moreover, the primary concern of this chapter is to address some of the most common misconceptions about what Deep Learning really is, along with refreshing the reader's concept as we proceed.

Artificial Intelligence, Machine Learning, and Deep Learning

When talking about Artificial Learning and Deep Learning, it becomes compulsory to include Machine Learning into the discussion because these three are related to each other. As such, we will steer the direction of our discussion towards discerning whether the difference between these three types of learning is evidently pronounced or just subtle. The diagram below depicts the relationship between Artificial Intelligence (Artificial Learning), Machine Learning, and Deep Learning.



Artificial Intelligence

Let's discuss Artificial Intelligence first. Artificial Intelligence is basically referred to as the development of an Artificial Intelligence System all by itself, imitating the learning pattern of human beings. In other words, A.I is a computer system that can harness the human learning prowess and perform tasks without any output from human operators. Artificial Intelligence is actually a scientific endeavor that was pioneered by a handful of computer scientists back in the 1950s, and the main goal of this endeavor is to enable computers to automate all those tasks which are by nature, intellectual, and

performed by humans. Hence, due to the nature of Artificial Intelligence, it encompasses Machine Learning as well as Deep Learning, along with many other techniques and approaches which don't involve any learning whatsoever. The field and concept of A.I is still largely unexplored even after so many years of its introduction because of modern hardware limitations. However, this does not insinuate that Artificial Intelligence is just pure science fiction; on the contrary, A.I has many promising prospects and is currently a field of extensive research and experimentation.

Machine Learning

The concept of machine learning goes far back to the early 19th century when the advent of the first general-purpose computer, the "Analytical Engine," was realized. Although this machine wasn't meant as a "general-purpose computer" because the very concept of general computation was not present at the time. Hence, the major purpose of this machine was to be a tool that could automate certain computations in mathematical analysis. Due to this, the machine was given the name of "Analytical Engine." Now the reason we are discussing such an early invention is because of a remark made by a certain lady on this very machine. The name of this woman was Lady Ada Lovelace, and she was an acquaintance and a collaborator of the inventor of the Analytical Engine, *Charles Babbage*. The main idea of the remark was that the Analytical Engine was just a piece of machinery that could only assist us in matters which we already know of and just automate some mathematical functions and calculations through our input.

The reason why this remark is so important in the history of Machine Learning is that the pioneer of Artificial Intelligence, Alan Turing quoted Lady Ada Lovelace's remark as a "Lady Ada Lovelace's objection," in his paper when he introduced the Turing Test for the first time. He concluded that general-purpose machines (computers) did have the potential for originality and learning, much like human beings.

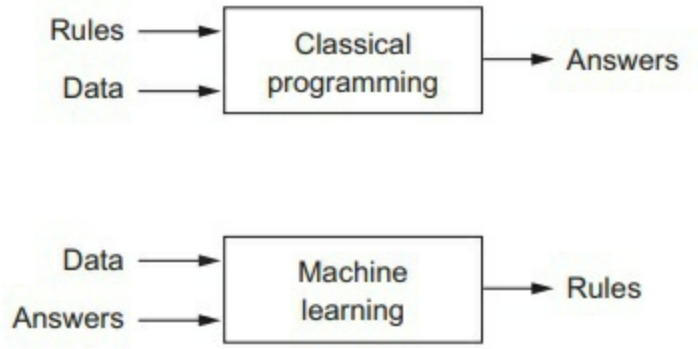
We have briefly discussed some boring and seemingly irrelevant history of a general-purpose computer. Still, it is quite the contrary, the very concept of Machine Learning arises from this question that we have just outlined and that question is this:

"Is it possible for a computer to perform a task or solve a problem in a way that is beyond what we know and also learn to perform certain specific tasks

independently without any human input?

Is it possible for a computer, a machine made by humans, and programmed by humans with functioning that is pre-determined and predicted to surprise us? Instead of being inputted with rules of data-processing by programmers, is a computer capable of just looking at the sample data and learning the rules by itself?

These very questions have brought into existence an entirely new and ambitiously unique programming paradigm. To get a better understanding of the potentials of Machine Learning, let's make a simple contrast between the working of an A.I system and a Machine Learning system on data-processing. In a system using Artificial Intelligence (symbolic A.I), the user basically inputs a program that contains specific rules for data processing. Then data is introduced, which is to be processed according to the specified rules. Hence we obtain a result showing us the answers. However, in a Machine Learning system, we do not provide the rules. Instead, the system is given the data as well as a result (answers), which are expected from this sample data. The Machine Learning system then proceeds to infer and learn the rules through which the corresponding result can be obtained. Once the system has learned the rules, it can be applied to new data and produce entirely original results.



To sum it up, the major feature which distinguishes Machine Learning from Artificial Intelligence is that the former is systematically trained rather than being explicitly programmed, as is the case in Artificial Intelligence. So, a system using Machine Learning is simply given a bundle of examples that are relevant to the primary task. The system then figures out the underlying statistical structure of these examples and then figures out the rules governing

the end results of these examples. Once the system has learned the rules, corresponding tasks can now be automated by the system.

In conclusion, Machine Learning is a sub-field of Artificial Intelligence that gained popularity and flourished ever since the 1990s to this day. As new computer hardware started to be developed and existing technologies saw major upgrades and improvements, a trend has been set for faster hardware and larger datasets paving the way for exploring more possibilities in Machine Learning. Machine Learning, along with Deep Learning, is more focused on being practical rather than theoretical as very little mathematical theory is involved. Hence, this discipline is geared towards proving ideas empirically rather than theoretically.

Deep Learning

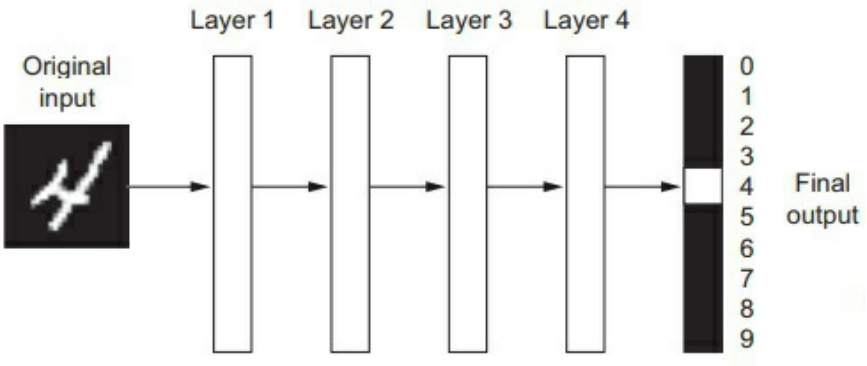
Just as how Machine Learning is a sub-field of Artificial Intelligence, similarly, Deep Learning is a sub-field of Machine Learning. The primary aspect which sets the two apart is that Deep Learning employs an entirely new learning representation from data. Deep Learning majorly emphasizes the idea of successive layers in a data sample, which gives a model of increasingly meaningful representations. In other words, this learning model takes into account the multiple layers of data representation while analyzing the sample. In addition, there is another terminology used for the data sample in Deep Learning, known as the “depth of the model,” this basically refers to the number of layers that are contributing to a model of the data.

Today, the Deep Learning system uses models of data which consist of tens to thousands of successive layers of representation. This widens the scope of the system as it can handle the learning process involving so many layers. On the other hand, Machine Learning often uses learning approaches which only involve one or two layer of representations of data (due to which they are termed as *shallow learning*).

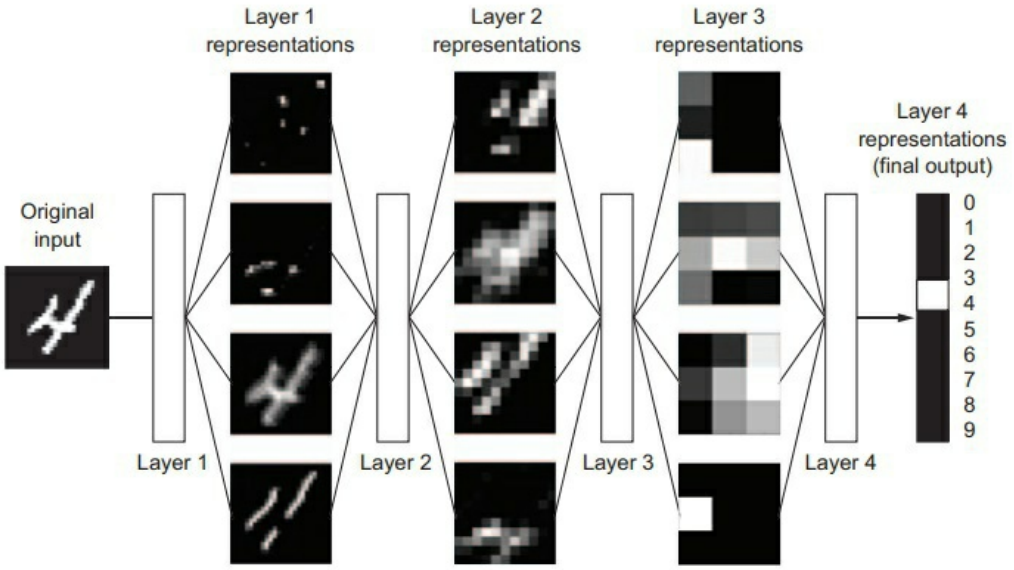
To further understand Deep Learning, we need to understand the mechanism through which the system actually “*learns*,” and this learning is done through networks that are structured in such a way that they have layers piled upon one another. This network is known as a *neural network*. It is also important to clear a major misunderstanding that Deep Learning imitates the working of our brain. While it is true that the neural network used in Deep Learning does draw some inspiration from the human brain, but that is the extent of it. On

the contrary, Deep Learning is essentially a mathematical framework that is designed to learn representations from data. Neural Networks have no relevance to the concepts described in Neurobiology, and hence, associating Deep Learning to it would mislead the general masses.

We have been discussing the layers of representations from data, however, to reinforce our understanding of this concept, we will proceed to examine how a neural network of a Deep Learning system learns to recognize and distinguish a digit. An image first represents the digit, and this image is then transformed into a different form of representation when passed through each successive layer as shown below:



In other words, the original image has been converted into representations that are entirely different from the original image. Hence, we can consider the Neural Network of a Deep Learning system as a distilling apparatus that distills information through a multi-stage process to produce a concentrated representation of the sample data, as shown below in detail.



Modeling of Machine Learning

In this section, we will discuss the classical machine learning approaches that hold conceptual and contextual importance when understanding deep learning, however, we will not include the details of such topics as they are an entirely separate topic of discussion.

Probabilistic Modeling

Probabilistic modeling is basically defined as employing the fundamental principles of statistics to a Machine learning system's method of data analysis. Probabilistic modeling gave birth to one of the most premature forms of Machine learning, but due to its practicality, it is still viable to this day and age. For instance, the Naive Bayes Algorithm is a prominent algorithm that stands out within the category of Probabilistic modeling.

To understand why this algorithm holds prominence and importance within Probabilistic modeling, let's discuss some of its features. This algorithm utilizes Bayes' theorem, and the basic assumption of this theorem is that the input data's basic characteristic upholds that its features are all independent. Hence, to use the Naive Bayes machine-learning classifier, this characteristic is all you need to understand.

Logistic Regression

Despite its name, Logistic Regression is essentially a classification algorithm, while many would-be misled towards thinking that it is a regression algorithm. Logistic Regression is closely related to the Probabilistic model in the sense that Logistic Regression also predates the advent of computers by quite a mile. Because of its simplistic nature and versatility, it has found its way to be useful to machine learning to this day. Moreover, Logistic Regression is the go-to model for a data scientist to try on a given dataset to understand and get a grasp of the classification task in front of him.

Early Neural Networks

Although newer and more efficient modern versions have replaced the early versions of the Neural networks, understanding their origin will also provide us with a better understanding of the advent of Deep Learning. The core concept of Neural networks had already surfaced as early as the 1950s. However, it was not until a decade later that the concept was adopted, and the approach was given serious attention. At the beginning of the life-cycle of Neural networks, the major problem which puzzled computer scientists was the method through which they could efficiently train large Neural networks. The missing piece of the puzzle was discovered as the resurfacing of the Backpropagation algorithm, which gave wonderous results as soon as it was implemented in the Neural networks.

The very first successful and practical application of a Neural network was seen in the 1990s, developed by Bell Labs. They used a combination of convolutional neural networks and backpropagation to achieve this.

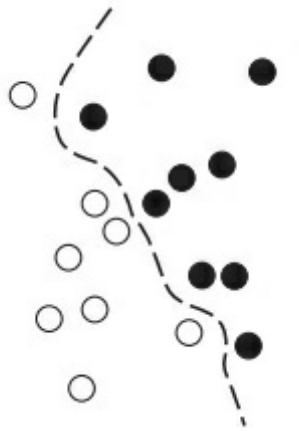
It was designed primarily to address the problem of classifying handwritten digits. It was this characteristic that led to the United States Postal Service using it to enable computers to easily read and identify the ZIP codes on mail envelopes, making their work much easier.

Kernel Methods

Due to the first successful application of Neural networks, it gained recognition among the research community, but its fame was overshadowed by another approach known as Kernel methods.

Kernel methods are essentially a group made up of classification algorithms, and the most commonly referred to Kernel method is the *“SVM”* or, in other words, the Support Vector Machine.

Support Vector Machine algorithms are mainly used to solve classification problems. They work by focusing their goal towards formulating unique decision boundaries in between the two different points and elements within a data sample. These decision boundaries can be interpreted as basically just a line that separates the data sample into different corresponding categories. A visual representation of this would look something like this:



(A Decision Boundary)

An SVM formulates a typical data boundary by following these two steps:

1. The SVM analyzes the data sample and maps it to an entirely new and unique representation of a higher dimension. In this representation, we can describe the decision boundary as a hyperplane. Moreover, the hyperplane is expressed as a straight line if the SVM is forming a representation of a two-dimensional data sample.
2. Maximizing the distance of the data points from the decision boundary. The further data points are from the hyperplane (decision boundary), the better the quality of the decision boundary. This step is also commonly known as “maximizing the margin,” and it enables the decision boundary to effectively generalize data samples that are not included in the training data set.

In addition to forming a data boundary and mapping data points onto a high-dimensional representation, the essence of the Kernel method is incorporating a key idea known as the “kernel trick” (on which the whole name of the

method is based on).

Here’s a brief explanation of how this “kernel trick” actually works. Let’s consider a scenario where we want to know about the good decision hyperplanes in any new representation space. There are two ways to go about this - we can compute and explicitly specify the coordinates of the data points within this representation space, or we could simply just compute how far the pairs of data points are from each other by using a kernel function. Hence, this is why this trick is known as the kernel trick.

Foreseeable Benefits of Deep Learning

Although the key concepts that laid a strong foundation for deep learning were already present and well understood in 90s such as:

- The Convolutional Neural Networks
- Backpropagation

And not only these important concepts, but the LTSM algorithm, also known as the Long Short-Term Memory algorithm, which is present even today, was developed for the first time in the year 1997, and this algorithm has seen minor changes over the coming decades. With such mathematical and functional resources at hand, Deep Learning took flight only after two decades in 2012. The reason for this delayed awakening is because of the rapid advancement in technologies such as:

- Hardware
- Datasets and benchmarks
- Algorithmic optimization

Another reason for Deep Learnings' rapid development in this day and age is because this field is focused primarily on practical findings rather than theoretical conjecture. Hence, finding new algorithms or optimizing existing ones can only be done when the appropriate hardware and datasets are accessible for experimentation. In short, with technological advancement, the prospects of Deep learning are not only unlimited, but the scale of improvement and development of this field is also inevitably profound because Deep learning is an engineering science. Like all engineering sciences, the prospects of this discipline grow vastly as technology keeps removing bottlenecks and paves the way for innumerable possibilities.

Chapter 2: The Arithmetic Foundations of a Neural Network

Before we can explore advanced concepts and practical examples of Deep learning, we must first familiarize ourselves with the mathematical ideas that are the foundations of the Neural networks making up the Deep learning system. These mathematical concepts include:

- Tensors
- Tensor operations
- Differentiation
- Gradient descent, and many more.

The main focus of this chapter will be to structure our discussion to encompass the core ideas and workings of these mathematical concepts, all while keeping our discussion from becoming too technical. In other words, this chapter will cater to the needs of the readers by building up an intuitive understanding of these important concepts and refrain from using mathematical notations. This way, the readers who do not have over the top mathematical prowess can also benefit and build a good understanding relating to the topic.

This chapter is essential to understand as it is the blueprint for understanding the practical examples, which will be detailed in the later sections of this book. As such, the chapter will take off by introducing a practical example of a Neural network, and then we will work our way through it to understand the corresponding concepts.

A Peek into a Neural Network

For our example, we will discuss a Neural network that utilizes the “Keras” library (a Python library to be exact). The main concern of this Neural network is learning on how to classify handwritten digits. For now, it is not necessary to be familiar with the Keras library as we will discuss the details of every element of this example in the following chapters. Right now, it is necessary to understand the fundamentals of what makes up this Neural network.

In this example of a Neural network, we are concerned with classifying handwritten digits. These handwritten digits are in the form of grayscale

images with a resolution of 28x28 pixels. The job of the Neural network is to classify these images of handwritten digits into their respective categories of 0 to 9 (in short, classify them into 10 categories). The dataset which will be used in this example is none other than the classic MNIST dataset because this specific dataset has been intensively studied in the Machine-learning community, making it a suitable choice for our example. Hence, solving this classic dataset can also be considered as making your very first program (Hello World!). Below are sample digits that are taken from the MNIST dataset.

One more reason why we have used the Keras Python library in our demonstrative Neural network is that the MNIST dataset (set of four NumPy Arrays) comes preloaded with this library.

The following lines of code shown below depict how you can load the MNIST dataset into the Keras library:

```
from keras.datasets import mnist

(train_images, train_labels), (test _images, test_labels) = mnist.load_data()
```

The train_images and train_labels are basically the resources which the model will use as learning material, and later onwards, it will be tested on the test_images and test_labels. In other words, the MNIST dataset includes a training set and a test set.

The learning process works in this way that the model establishes a mirrored correspondence to the images (encoded as NumPy arrays) and the labels (array of digits).

The resulting training data obtained would look like this:

```
>>> train_images.shape
```

```
(60000, 28, 28)

>>> len(train_labels)

60000

>>> train_labels

array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

Similarly, the corresponding test data is:

```
>>> test_images.shape

(10000, 28, 28)

>>> len(test_labels)

10000

>>> test_labels

array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

Now proceeding to explain the workflow of the model as it trains itself using the training set and later tests itself through the test set. As it is obvious, we are providing the Neural network with data to train it by using the train_images and train_labels from the training set of the MNIST dataset. After the Neural network has learned how to identify and associate the corresponding images and labels, we move on to testing its capability by exposing it to the data from test_images and ask it to predict what labels do these images correspond to. After this, the predictions produced by the Neural network are matched with the test_labels to check whether they are accurate or completely off the mark.

Now, we will proceed to build the Neural network even further. The architecture of the network will look like this:

```
from keras import models

from keras import layers

network = models.Sequential()
```

```
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

From the above Neural network architecture, we get to see the following important terms:

- **Layers**: The layer is considered as the most important building block of any Neural network. Layers are basically modules that are concerned with data processing, and they act as filters for data, in the sense that when a layer processes data, the resulting data is in a form that is more useful and effective than it previously was. In other words, the specific function of a layer is to take the sample data and extract the representations from it. In short, layers are essentially just data filters.
- **Data Distillation**: Data distillation is a progressive process that takes shape when multiple simple layers are chained together.
- **Dense Layers**: Dense layers are basically neural layers that are completely connected (densely connected).
- **Softmax Layer**: The network sequence shown above details a 10-way softmax layer, which is actually a probability layer, i.e., it functions to return an array that consists of 10 probability scores. Each of these 1o scores gives us a probability of how the active (current) digit is actually part of one of the 10 digit classes.

Even now, our network is still not ready for training. We still need three vital elements that will essentially make up the compilation step:

- **Loss Function**: This details the way through which the network will be able to gain an idea of its performance, and in this way, it will be able to shift the direction of its work towards the right direction.
- **Optimizer**: This is a mechanism that allows the network to update itself corresponding to two elements: the analyzed data and the loss function.
- **Metrics**: These metrics are to be used for monitoring the network during its training and testing stages, and the main concern of this step is the accuracy of the Neural network by which it can

correctly classify the images.

Although the purpose of the loss function and optimizer may seem unclear right now, however, the practical examples and explanations in the coming chapters will address their workings in more detail.

Now, we will implement the compilation step into our Neural network:

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

However, our Neural network is still not ready for training. There are still quite a few steps required before we can proceed to train the method in Keras by using a method known as “fit” (which will be explained shortly). Hence, the first thing we need to do here is to pre-process the data. This is done by reshaping the data into a form that is expected by the network. In addition, we also need to scale the data to bring all the corresponding values into an interval of [0, 1]. For example, consider the very first example we discussed in this chapter, if you look at it closely, you will notice that the training images are a unit8 type and not only that, but they are also stored in an array of (6000, 28, 28). Moreover, the values were unscaled and in an interval of [0, 255]. Hence, we will transform this example in the following parameters:

- Array type from unit8 to float32
- Array shape from (6000, 28, 28) to (6000, 28 * 28)
- Values in [0, 255] interval to [0, 1] interval

The lines of code to do this are:

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

The last step left to do is that the labels are needed to be encoded categorically, which can be done as shown below:

```
from keras.utils import to_categorical


train_labels = to_categorical(train_labels)

test_labels = to_categorical(test_labels)
```

Finally, we are now all set up to proceed with training the Neural network by using the Keras library. To do this, we will call to a method known as the "network.fit". This basically fits the network's model to the corresponding training data, as shown below:

```
>>> network.fit(train_images, train_labels, epochs=5, batch_size=128)

Epoch 1/5

60000/60000 [==============================] - 9s - loss: 0.2524 - acc: 0.9273

Epoch 2/5

51328/60000 [========================>.....] - ETA: 1s - loss: 0.1035 - acc: 0.9692
```

Now, if we take a closer look at the result, we can see that two quantities are being displayed in the training session. These quantities of the network over the training data respectively are:

- **The Loss**
- **The Accuracy**

In the training session, we observe an accuracy of 98.9%. We will now proceed to determine whether the model performs similarly on the test dataset.

```
>>> test_loss, test_acc = network.evaluate(test_images, test_labels)

>>> print('test_acc:', test_acc)

test_acc: 0.9785
```

The accuracy shown in the test session is actually lower than the accuracy

shown in the training session, i.e., 97.8% and 98.9%, respectively. This gap of accuracy is termed as “overfitting.” This reinforces the fact that the Machine-learning models do not perform at the same level when exposed to new data as compared to their performance with the training dataset.

Data Attributes of Neural Networks

In this section, we will be focusing on the attributes of the most common data structure, **tensors**. Tensors are basically data storage containers as the very NumPy arrays in which multi-dimensional data is stored itself called a tensor. The type of data stored in tensors is usually numerical data. Hence it is also technically plausible to consider tensors as storage containers for numbers. Also, it is important to remember that in the context of a tensor, a dimension is basically referring to an axis.

Based on the dimensions, there are four common types of tensors used in Machine learning, namely:

1. Scalars (0D tensors)
2. Vectors (1D tensors)
3. Matrices (2D tensors)
4. 3D tensors and higher dimensional tensors

Scalars (0D tensors)

Scalars are basically those tensors that only have one number stored. Other names for scalars are scalar tensors, 0-dimensional tensor, or 0D tensor. Scalar tensors are represented by both floath32 and float64 numbers in NumPy. Moreover, a “rank” basically refers to the number of axes that are within a tensor. We can also find out the number of axes within a NumPy tensor by using an attribute known as the **ndim** attribute. (Remember that as Scalars are zero-dimensional, their ndim attribute will always be 0). Below is an example of a NumPy scalar:

```
>>> import numpy as np

>>> x = np.array(12)

>>> x

array(12)
```

```
>>> x.ndim

0
```

Vectors (1D tensors)

Vectors are those tensors that contain an array of numbers while having only one axis, hence the name “1D tensor”. A NumPy vector is shown below:

```
>>> x = np.array([18, 5, 9, 11])

>>> x

array([18, 5, 9, 11])

>>> x.ndim

1
```

Upon careful inspection, we can see that a vector consists of 5 entries and because of this we can call such a vector a 5-dimensional vector. It is important to note that a 5-dimensional vector is not the same as a 5D tensor, as a dimension can both refer to the number of entries and the number of axes in a tensor. As such, it is when faced with such confusing scenarios, it is better to refer to a 5D tensor as a tensor of rank 5).

Matrices (2D tensors)

Matrices are those tensors that contain an array of vectors. A matrix is also known as a 2D tensor because it consists of two axes (the rows and columns). A NumPy matrix is shown below:

```
>>> x = np.array([[2, 52, 8, 22, 9],

                  [4, 2 12, 23, 5],

                  [8, 11, 33, 98, 5]])

>>> x.ndim

2
```

In this example, the horizontal entries (the x-axis) are considered as the rows, for instance, in the above example, the entry [2, 52, 8, 22, 9] is the first axis

(the row) and the vertical entry [2, 4, 8] is the second axis (the column).

3D Tensors and Higher-Dimensional Tensors

3D tensors are basically a bunch of matrices packed in such a way that they are visually interpreted as a cube of numbers. For example, a typical 3D tensor looks like this:

```
>>> x = np.array([[[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]],
                  [[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]],
                  [[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                  [7, 80, 4, 36, 2]]])
>>> x.ndim
3
```

Just as how we created a 3D tensors by packing multiple matrices in an array, similarly, we can obtain a 4D tensor by packing multiple 3D tensors in an array and the same process holds for obtaining 5D tensors and so on. Generally we are only required to manipulate tensors from 0D to 4D when using them in deep learning, however, we may also need to go to 5D tensors if we are dealing with the processing of video data.

The Key Attributes of Tensors

Generally, a tensor has three defining key attributes. These attributes are:

1. **The Rank**: Rank refers to the number of axes a tensor has. For example, we discussed a Matrix that has two axes; hence it has a

rank of 2. Similarly, a 3D tensor has three axes, so it has a rank of 1. Ranks are referred to as **ndim** in the NumPy library.

2. **Shape**: A shape is the computing structure that defines the number of dimensions that a tensor has along each axis. For instance, consider the above examples of different types of tensors. A scalar tensor has no dimension; hence its shape is (), a vector has only one dimension, so its shape is (3,) and a matrix has two dimensions, so its shape is (2, 6). A 3D tensor has three dimensions; hence, it has a shape of (1, 2, 3) and so on.
3. **Data Type**: Data type is commonly referred to as "dtype," specifically in the Python libraries. As the name suggests, this refers to the very type of data that is being stored in the tensor itself, for example, unit8, float32, and float64 are all data types.

Let's take this discussion a bit further and explain these attributes more clearly by looking at a demonstration of the data we processed previously in our MNIST example. To start, we will first load up the MNIST data set by:

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Now, by using the **ndim** attribute, we will proceed to display the number of axes that the tensor (train_images) currently has:

```
>>> print(train_images.ndim)

3
```

Now to display the shape attribute of the tensor:

```
>>> print(train_images.shape)

(30000, 18, 18)
```

And finally, the **dtype** attribute or the data type of the tensor:

```
>>> print(train_images.dtype)

uint8
```

From the above results, we come to know that the tensor is, in fact, a 3D tensor, specifically of 8-bit integers. Furthermore, the shape of this tensor is

that of an array of 30000 matrices with 18x8 integers.

We can also display any digit within this 3D tensor. This can be done by utilizing the Matplotlib library. For instance, the example below depicts the lines through which we are displaying the fourth digit in our 3D tensor:

```
digit = train_images[4]


import matplotlib.pyplot as plt

plt.imshow(digit, cmap=plt.cm.binary)

plt.show()
```

Gearing the Neural Network through Tensor Operations

Technically, if we begin to break down any computer program to its fundamentals, then we will ultimately come down to a mere set of binary operations on binary inputs. This includes the AND, OR, NOR logic inputs, etc.. Similarly, we can do the same thing to the transformations that have been learned by the Neural networks. The difference over here is that instead of binary operations, we come across a bunch of “tensor operations.” These tensor operations are applied to tensors of numeric data. Hence, the network can perform arithmetic functions on it by multiplying tensors or even adding tensors with each other. In other words, tensors are literally the cogs and gears of any Neural network.

Looking back to our very first example, we notice that the way we have been building a Neural network is by assembling “Dense” layers in such a way that they pile up on each other. An instance of the resulting Keras layer would look like this:

```
keras.layers.Dense(512, activation='relu')
```

Now, an interpretation of such a layer results in revealing it as a function, basically taking input and output both as a 2D tensor (takes in 2D tensor, gives out 2D tensor), providing us with an entirely new representation of the original inputted tensor. This function has also been depicted below for better understanding:

```
output = relu(dot(W, input) + b)
```

Upon careful inspection, we come to know that this function itself has three tensor operations namely:

- **dot** (product of the input tensor and the "W" tensor)
- **addition** (addition of a 2D tensor that resulted from the product and the vector "b")
- **relu operation**

We will now proceed towards discussing the details and some advanced concepts of these three outline tensor operations.

Element-Wise Operations

Element-wise operations basically refer to the relu and addition operations. The reason as to why they are termed as "Element-wise operations" is because of their character according to which we observe that to each entry in the tensor which we are analyzing, these operations are always applied independently to the entries mentioned above. Due to this characteristic, element-wise operations are incredibly responsive to huge implementations that are vectorized (parallel implementations). If we wish to create an implementation for an element-wise operation, especially a naive Python implementation, then we can do so by using a "for" loop. For example:

```
def naive_relu(x):

        assert len(x.shape) == 2                    (over here, x is a 2D NumPy
tensor)


                        x = x.copy()                (always refrain from
overwriting input tensor)
```

```
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] = max(x[i, j], 0)
    return x
```

The same can be done for the addition tensor operation as shown below:

```
def naive_add(x, y):
    assert len(x.shape) == 2                (over here, “x” and “y” are 2D NumPy tensors)
    assert x.shape == y.shape

    x = x.copy()                            (Avoid overwriting the input tensor)
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[i, j]
    return x
```

By following the same principles, we can also perform other element-wise operations such as multiplication and subtraction.

In practical implementations, when we are working with NumPy arrays, we can increase the speed of the NumPy arrays by an incredible amount by using these very operations. The reason for this is that the element-wise operations are actually preloaded as optimized NumPy functions. These functions shift all of the heavy workloads onto the BLAS implementation as these implementations are characterized as:

- Low-level
- Highly parallel
- Efficient tensor manipulation routines

Moreover, BLAS (Basic Linear Algebra Subprograms) are commonly used and implemented in Fortran or C.

Coming back to the main topic, in a NumPy array, by using the element-wise operations shown below, we can optimize the array to a great extent:

```
import numpy as np

z=x+y                          (element-wise addition operation)

z = np.maximum(z, 0.)          (element-wsie relu operation)
```

Broadcasting

Broadcasting basically refers to the phenomenon of tensor addition, specifically in the case where the two tensors being added are different from each other with regards to their shape. Obviously, a conflict arises during the addition operation. Hence, this is remedied by the phenomenon of broadcasting, which involves broadcasting the shape of the smaller tensor to match with the shape of the bigger and larger tensor.

Broadcasting is essentially done in two steps:

1. The first step is adding axes to the tensor being broadcasted. These axes are known as broadcast axes, and by adding them, the smaller tensor can match the **ndim** of the bigger tensor.
2. The second step is to repeat the smaller tensor beside these new axes such that the shape of the tensor matches with the entire shape of the big tensor.

To understand the concept of broadcasting even better, let's consider an example. We are working with two tensors, "x" and "y." The shape of the "x" tensor is as follows: (22, 5), and the shape of the "y" tensor is (5). The shapes of these two tensors differ from each other clearly as the "x" tensor is bigger. We cannot add these two tensors. Hence to make the shapes match with each other, the "y" tensor is broadcasted as follows:

1. An empty first axis is added, and the resulting shape of the "y" tensor becomes (1, 5)
2. The original y tensor is repeated for a total of 22 times besides this

new axis, and the resulting shape of the “y” tensor afterward would become (22, 5)

Where y[i, :] == y for i in range (0, 22). Now we can proceed with adding the “x” and “y” tensors as they now have identical shapes.

Furthermore, it is also very important to clarify that this procedure does not result in the making of a new 2D tensor as if this were the case, then it would be horridly inefficient. To make things more clear, note that this outlined operation of repetition is, in fact, all virtual, meaning that it actually takes place on an algorithmic level instead of a memory level. Still, there is a benefit for conceptual clarity and understanding if we consider the repetition of a vector for a total of 15 times beside a new axis. A naive implementation of such a situation would be like this:

```
def naive_add_matrix_and_vector(x, y):
        assert len(x.shape) == 2                    (x is a 2D NumPy tensor)
        assert len(y.shape) == 1                    (y is a NumPy vector)
        assert x.shape[1] == y.shape[0]
        x = x.copy()                                (avoid overwriting the
input tensor)
        for i in range(x.shape[0]):
            for j in range(x.shape[1]):
                x[i, j] += y[j]
return x
```

Here’s a cool trick, it is possible to apply two-tensor element-wise operations along with broadcasting. The only pre-requisite for this to work is that if one of the two tensors has a shape of (a, b, … n, n+1, … m) and the shape of the second tensor is (n, n+1, … m), if this condition is fulfilled then for axes “a” through “n-1”, they will be broadcasted automatically. For instance, the demonstration shown below is applicable for two tensors that have different shapes, in addition, we will see the element-wise “maximum” operation

being applied on the tensors above:

```
import numpy as np

x = np.random.random((60, 4, 37, 12))

y = np.random.random((27, 15))

z = np.maximum(x, y)
```

In the above example, “x” is a random tensor with shape (60, 4, 37, 12) while “y” is also a random tensor but with a different shape (27, 15). Finally, the “z” is an output tensor that has the same shape as the “x” tensor.

Tensor Dot

The tensor dot or more commonly known as “tensor product” is arguably the most used tensor operation. However, it is important to clarify that a tensor product and an element-wise product are not the same. A tensor product is a tensor operation, while the latter is an element-wise operation. In addition, the tensor product also works contrary to the element-wise product in the sense that the tensor product combines the entries that we find in the input tensors.

Besides the difference in function, the syntax of an element-wise product is also different than a tensor product. The syntax of an element-wise product in different libraries (NumPy, Keras, Theano, and TensorFlow) is the same, i.e., the “*” operator. While the tensor product has a different syntax in TensorFlow, its syntax is the same for NumPy and Keras, i.e., the “dot” operator, as shown below:

```
import numpy as np

z = np.dot(x, y)
```

If we want to denote the dot operation mathematically, then we would do so by using a dot (.):

```
z=x.y
```

Let’s proceed to discuss the mathematical functions of a dot operation by computing the dot product of two vectors, namely “x” and “y”:

```
def naive_vector_dot(x, y):
    assert len(x.shape) == 1
    assert len(y.shape) == 1
    assert x.shape[0] == y.shape[0]
  z = 0.
    for i in range(x.shape[0]):
        z += x[i] * y[i]
    return z
```

Upon analyzing this example, we conclude that the compatibility for a dot product largely depends on the aspect that vectors are similar in terms of elements. Moreover, the dot product of two vectors always results in a scalar.

Let's discuss a dot product between two matrices. These matrices are labeled "x" and "y" respectively. If we apply a dot product between these two matrices, then we will be given a vector and the coefficients of this retrieved vector as basically the result of a dot product of "y" and the rows of "x." This will be implemented as follows:

```
import numpy as np
def naive_matrix_vector_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            z[i] += x[i, j] * y[j]
```

```
        return z
```

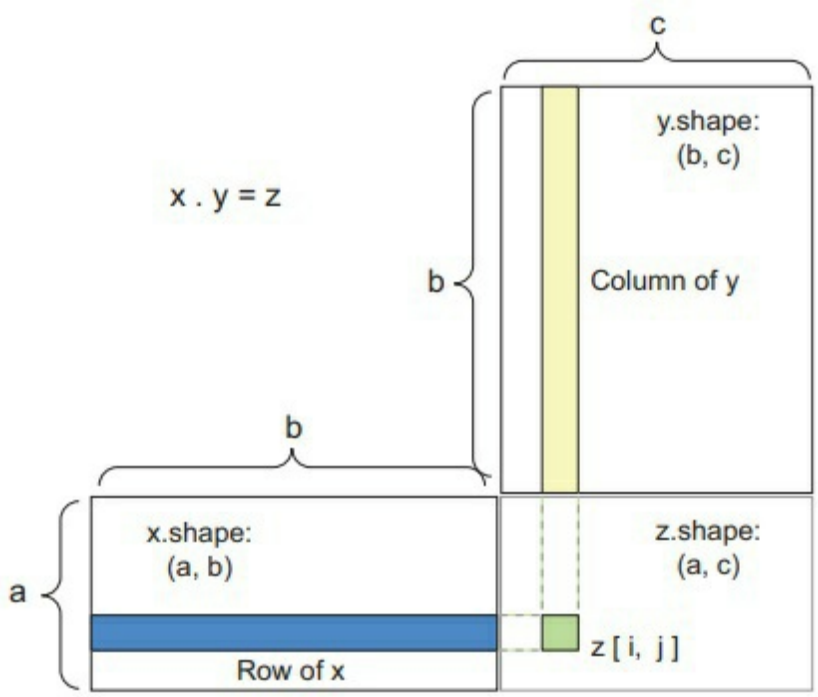
In the above demonstration, there is a very key concept which we need to keep in mind whenever working with the dot tensor operation is that the first dimension of the tensor x must be the same as the 0^{th} dimension of the tensor y.

Furthermore, in the previous sections of this chapter, we have discussed an example regarding naive implementations of tensors. We can use the codes written in these examples to specifically bring the relationship of matrix-vector product and vector product to the spotlight, as shown below:

```
def naive_matrix_vector_dot(x, y):
        z = np.zeros(x.shape[0])
        for i in range(x.shape[0]):
                z[i] = naive_vector_dot(x[i, :], y)
        return z
```

If we analyze this block of code carefully, we conclude that the dot operation is not commutative, i.e., dot(x, y) is not the same as the dot(y, x). This is because the dot operation loses its symmetry as soon as the **ndim** of any of the two tensors becomes greater than 1.

The shape compatibility between tensors for dot product operations can be very confusing and misleading at times. Below is a box diagram that is aimed at improving the reader's visualization of the input and output tensors.



In this diagram, the tensors x, y, and z have been depicted as rectangles. For a dot product, the rows of the tensor “x” must be matching in size with the columns of the tensor “y,” hence this would insinuate that in the above box diagram, the width of the “x” rectangle should match the height of the “y” rectangle.

In the case of tensors that are of higher-dimensions, the dot product between these tensors would follow the shape compatibility rules as in the case for 2D tensors:

```
(a, b, c, d) . (d,) -> (a, b, c)

(a, b, c, d) . (d, e) -> (a, b, c, e)
```

Tensor Reshaping

The third tensor operation which we will be discussing is the tensor reshaping. We did not use tensor reshaping in the example of the Neural network, which was using dense layers. However, the tensor reshaping operation was used for preprocessing data pertaining to the digits prior to giving it to the Neural network, this example was:

```
train_images = train_images.reshape((60000, 28 * 28))
```

The term reshaping is self-explanatory. The tensor on which this operation is applied is basically reshaped (basically rearrangement of the rows and columns) to match the desired shape, which is detailed beside the operation. Furthermore, it is important to keep in mind that tensor reshaping only rearranges the rows and columns, not change them. Hence, the total number of coefficients (of the original tensor) remains the same for the reshaped tensor. Here's a simple example to better understand reshaping:

```
>>> x = np.array([[0., 1.],
                  [2., 3.],
                  [4., 5.]])
>>> print(x.shape)
(3, 2)
>>> x = x.reshape((6, 1))
>>> x
array([[ 0.],
       [ 1.],
       [ 2.],
       [ 3.],
       [ 4.],
       [ 5.]])
>>> x = x.reshape((2, 3))
>>> x
array([[ 0., 1., 2.],
       [ 3., 4., 5.]])
```

Aside from simply rearranging the rows and columns of a tensor in reshaping, another type of reshaping is also very common, known as transposition. Unlike reshaping where the shape of a tensor is changed by rearrangement, transposition basically exchanges the position of the rows with the columns, hence turning rows into columns and columns into rows. In transposition, a tensor

y[j, :] becomes

y[:, j]

```
>>> x = np.zeros((150, 10))

>>> x = np.transpose(x)

>>> print(x.shape)

(10, 150)
```

Gradient-Based Optimization in Neural Networks

In the preceding sections, specifically in the element-wise operation section, the data inputted in a Neural network is transformed by each layer of the network as shown below:

```
output = relu(dot(W, input) + b)
```

We will now analyze the elements of this expression. Two attributes are very interesting. These attributes are “W” and “b” respectively. They are known as the “weight” and “trainable parameters” of any layer, and in more technical terms, they are referred to as the **kernel** and **bias** attributes of a layer, respectively. The weights essentially store the data, which is learned by the system during training.

In the beginning, the weight matrices are observed to have a bunch of random values filling them. These random values are generated through a step known as random initialization. Although the above expression will not yield any useful results as the parameters of the attributes generated are random. However, everything needs to have a starting point, and for machine learning, this is the starting point because of the fact that the system will respond to the feedback signal and adjust these weights accordingly. Hence, we have now

discussed what training is, the gradual adjustment of weights according to a feedback signal is called training, and this is the essence of machine learning.

The training of a network is done in **training loops**. The following steps outlined below explain a typical training loop routine, and they are repeated as long as necessary:

1. Take a collection of training samples labeled “x” along with the corresponding targets of these samples labeled “y.”
2. This step is known as the **forward pass**. Once the batch of the samples and targets have been drawn, proceed to get the predictions **y_pred** by running the Neural network on the training sample “x.”
3. Measure the results with the prediction and note any mismatch and discrepancies between the two values. This is basically computing the loss of the network.
4. Update the weights of the Neural network. Keep in mind that the weights should be updated in a way to lower the loss on this collection or batch.

Through this training loop, we will eventually obtain a network that exhibits considerably low levels of discrepancies and mismatching between the predictions and targets, i.e., a mismatch between y_pred and y, respectively. Hence, the network has essentially learned how to map the data inputs to the corresponding correct targets correctly.

Chapter 3: Starting Our Tasks with Neural Networks

In this chapter, we will discuss the practical applications of Neural networks in more detail and incorporate the concepts we went through in the previous chapters. Moreover, the main focus of this chapter will be a chance for the reader to reinforce his knowledge of deep learning and neural network while going through problems that address the most common practical uses of Neural networks which are:

- Binary classification
- Multiclass classification
- Scalar regression

In addition, this chapter will give an introduction to the deep learning libraries we will be using throughout this book, namely the Python and Keras deep learning libraries. The topics will include a closer and detailed inspection of some of the core components discussed in the previous chapter such as:

- Layers, networks, objective functions and optimizers

Before we proceed with the chapter, here are the practical examples which we will be a demonstration on how we can use Neural networks to solve real-world issues:

1. Identifying if the movie reviews are either positive or negative (Category: of binary classification)
2. Cataloging the new wires according to the topic (Category: Multiclass classification)
3. By inputting real-estate data to the network, we procure price estimations for houses (Category: Scalar regression)

After concluding this chapter, the reader will be able to make use of Neural networks and implement it to solve machine problems of simple nature; for instance, the classification and regression over vector data.

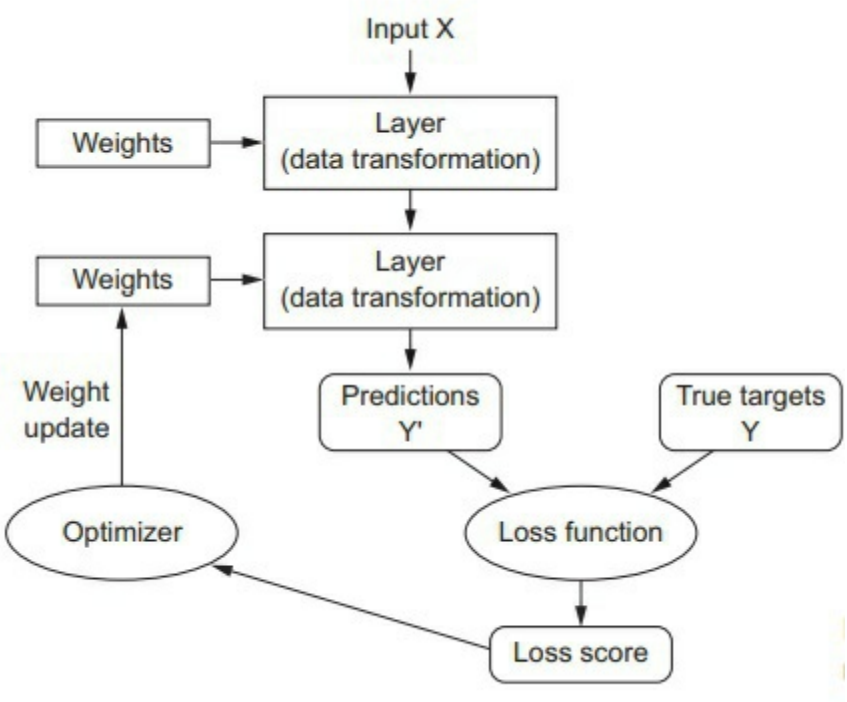
Inspection of a Neural Network

From our preceding discussions we have concluded that the Neural network's

training essentially depends upon the following objects:

- Layers that have been merged together forming a network (or in other words, a model)
- The data which has been inputted into the Neural network and the targets which correspond to this input data
- The loss function (this function essentially the defining factor for the feedback signal that is purposed for learning)
- The optimizer (the determinant of the learning procedure)

Below is a figure which emphasizes the relationship between these objects (the network, the layers, the loss function, and the optimizer):



Before moving further, let's discuss these anatomical elements of a Neural network in more detail.

Layers

By now, we have become familiar with the importance of layers as the building blocks of deep learning. To reiterate the concept in brief and simple terms, a data-processing module that receives an input in the form of a tensor (can be one or two tensors) and gives an output in the form of a tensor as well

can be one or two tensors) is known as a layer. In addition to this, we discussed the attributes of a layer, such as the "weight briefly." The weight of a layer refers to its state. While some layers may be stateless, most of the layers possess a state (several tensors that are learned with stochastic gradient descent, these tensors as a group form what we term as, the network's **knowledge**).

Aside from this, different types of layers are suitable for different purposes (different tensor formats and different types of data processing needs). The types of layers we have discussed so far include:

- Densely connected layers
- Recurrent layers
- Convolution layers

To elaborate further, we can consider an example of a simple vector data which is contained within a 2D tensor of shape (**samples, features**). Dense layers will process such a tensor. Similarly, a 3D tensor with a shape of (**samples, timestamps, features**) storing a type of data known as a "sequence data," will be processed by a recurrent layer (for instance, an LTSM layer). Furthermore, if we are dealing with image data, then the corresponding container will be a 4D tensor, and such a tensor is typically processed by a layer known as CONV2D, which is basically a 2D convolution layer.

In the Keras framework, building a deep-learning model is just like joining pieces of a puzzle together. To be more specific, deep-learning models are built upon data-transformation pipelines that are actually made by putting together compatible layers. In this context, compatibility refers to the ability of a specific type of layer to be able to accept tensors that have a specific shape as input and return corresponding tensors of a specific shape. For example:

```
from keras import layers

layer = layers.Dense(32, input_shape=(770,))
```

In this example, we have created a dense layer that is specified only to accept 2D tensor inputs in which the very first dimension is 770. Moreover, we have left the batch dimension as unspecified and because of this the layer will accept any value. Hence, the input tensor will be a 2D tensor with a shape of

(770,) and the output tensor will have its first dimension changed to 32. Meaning that the layer we have built above is only able to be connected to a downstream layer, which is specified to accept vector inputs that are 32 dimensional. In Keras, the hassle of considering each layer's compatibility becomes non-existent because of the fact that each layer added to the model is built dynamically so that it can match with the shape of the next layer. For example, we have added another layer on top of an existing one while using Keras:

```
from keras import models

from keras import layers

model = models.Sequential()

model.add(layers.Dense(32, input_shape=(770,)))

model.add(layers.Dense(32))
```

We can see that the layer succeeding the last one is devoid of any input shape argument. Regardless of not receiving an input shape argument, the second layer has inferred that its input shape should be the output shape of the preceding layer.

Models

Deep learning models are essentially a network of layers directed into an acyclic graphical form. The simplest form of a deep learning model would be one that is made up of a linear network of layers. Such a model is capable of only single mapping tensors, i.e., one input to one corresponding output.

However, as we progress further understanding the anatomical features of Neural networks, we come to know that there is a large variety of topologies for Neural networks, which in turn, form a variety of deep learning models. Following are the most common:

- Two-branch networks
- Multihead networks
- Inception blocks

Now let's elaborate on the concept of Neural network topologies. Basically, a Neural network works in a pre-defined space of possibilities, and it is within

this space that the network looks for viable representations of data. Hence, a topology essentially defines a constrained and pre-defined space for a Neural network, and because of this, a Neural network topology is also termed as a "hypothesis space." Hence, selecting a specific topology will essentially limit the network's hypothesis space to only a particular set of tensor operations. Hence, the main focus of your job would be to simply look for the optimal set of values for the specific topology's corresponding weight tensors.

As such, selecting the most optimal network architecture for your deep learning model is closer to being an artistic choice rather than a logically sound decision. This is because no defined parameter dictates whether a network topology is superior to the other or if it is the right one for your model. This is why a good neural-network architect can only develop a good intuition for choosing network architectures through repeated practice.

Loss Functions and Optimizers

Now, after deciding on network architecture and successfully defining it for the deep learning model, there are still two things that are missing and require our immediate attention, namely:

1. **The Loss Function**: also known as the objective function. This represents the minimized quantity in the training session of a model. In other words, the success rate of the task which the network is working on.
2. **Optimizer**: It works according to the results of the loss function in the sense that the Neural network's data update is done according to the loss function, hence controlled by the optimizer. The optimizer uses an implementation of a variant of an SGD (stochastic gradient descent).

One interesting point to note about the loss functions and the gradient descent process is that within a Neural network that is architecture as a multi loss network, there can be multiple loss functions to accommodate the multiple outputs of the network. However, the same thing does not apply to a gradient descent process. Instead, the gradient descent process is always based upon one singular scalar loss value, therefore in multi loss networks, a technique known as "averaging" is used to merge the multiple losses into a singular scalar value.

A very important point to always keep in mind when working with Neural networks is always to try to select the most optimal and correct objective function for a given problem. The reason for this is because a Neural network will always look for shortcuts and choose that path to keep the loss at a minimum. Hence, if we do not make sure of the clarity of the objective and its correlation to the success of the given task, the Neural network will perform actions that are either unnecessary or unwanted. So choosing a correct objective will save one from facing these unpredictable side-effects.

What is Keras?

In the previous sections of this book, we have studied demonstrations and examples of code that are using Keras. In this section, we will discuss what Keras actually is in detail. To define Keras, it is simply a framework for Python, which is specialized for deep-learning projects. Due to this, Keras provides programmers with a simple and convenient route towards being able to define and train deep-learning models of almost any kind. Although this framework was originally developed for people that were researchers to facilitate them with fast-experimentation, its functionality made it suitable for being used in deep-learning models as well.

The core features of the Keras framework include:

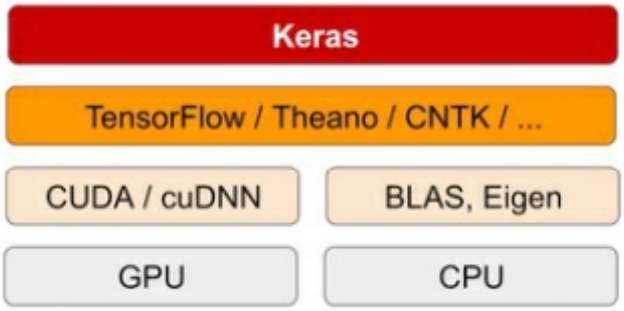
- Allows code to be cross-run between the GPU and CPU. Meaning that the source code is compatible with both of the components without needing to make any changes to it.
- By featuring a user-friendly API, Keras makes it easier for users to prototype deep-learning models with incredible convenience.
- Keras also conveniently features native support for both convolutional and recurrent networks or any combination of these two types of networks. By having built-in support for these networks, Keras indirectly provides native support for computer vision and sequence processing.
- The reason why Keras can be used for almost any type of Neural network is because it supports arbitrary network architectures. In other words, if you are looking to build a generative adversarial network to even a Turing machine, Keras remains relevant and appropriate for the task.

Keras and the Backend Engines

A distinctive feature of Keras is that being a model-level library in Python, it is associated with the supply of high-level basic elements and operations to develop a deep learning model. On the other hand, Keras does not associate itself with low tier operations (for example, tensor manipulation and differentiation). To make up for this, Keras includes a well-optimized tensor library specialized just for this purpose, and such libraries serve the purpose of being the backend engines of Keras. Moreover, Keras does not include a single exclusive backend engine. Instead, Keras supports multiple different backend engines with seamless support. Following are the three most popular backend engines supported by Keras:

1. TensorFlow backend
2. Theano backend
3. Microsoft Cognitive Toolkit (CTNK) backend

The modular architecture of Keras is shown in a visual representation below:



In addition, the code written on Keras is capable of being executed by these backends without any changes made to the source code, and the different backends can be swapped out if a specific backend proves to be more efficient and faster for a specific task. The TensorFlow backend enables the Keras framework to run on both CPUs and GPUs efficiently. This is possible as TensorFlow uses further smaller libraries as its own backend, and these backends are different for when running code on a CPU and for a GPU. For a CPU, TensorFlow uses a low tier library known as “Eigen” for performing tensor operations. At the same time, in a GPU, TensorFlow utilizes the popular cuDNN (Nvidia Cuda Deep Neural Network) library developed and optimized by NVIDIA.

A Short Overview of Deep Learning Development with Keras

If we look back, we will see that in the MNIST example demonstrated in the early chapters of this book, are also an example of a Keras model. Keeping this example in mind, a standard Keras workflow is exactly the same as shown in that demonstration. Here's a quick and brief overview of the elements of a Keras workflow:

1. Specifying the input tensors and the corresponding target tensors (collectively known as the training data).
2. Build a deep learning model that optimally maps the input tensors to the target tensors. Remember that a model is just a combination of layers in a network.
3. Select the elements (loss function, optimizer, and metrics to monitor) through which the network's process of learning can be configured and customized.
4. Keep repeating the fit() method of the deep learning model on the training data.

A deep learning model can be defined and specified by using either of the two methods (depending on how many layers a model has):

- Using the Sequential class. This method is viable only for models that are built up from linear stacks of layers.
- Using the functional API. This method is viable for models that are made up of an acyclic graph of layers supporting arbitrary model architectures.

To refresh our memory, below is an example of a model being defined by the Sequential class. Note that the deep learning model has a linear stack of two layers:

```
from keras import models

from keras import layers


model = models.Sequential()

model.add(layers.Dense(32, activation='relu', input_shape=(770,)))
```

```
model.add(layers.Dense(10, activation='softmax'))
```

Similarly, an example is shown below that shows a deep learning model being defined by a functional API:

```
input_tensor = layers.Input(shape=(770,))

x = layers.Dense(32, activation='relu')(input_tensor)

output_tensor = layers.Dense(10, activation='softmax')(x)


model = models.Model(inputs=input_tensor, outputs=output_tensor)
```

If we examine this demonstration, then we can see that the functional API is essentially concerned with manipulating the data tensors. Now, these data tensors are the ones that the deep learning model processes, and as such, the functional API manipulates them and applies layers to these tensors, treating them as functions rather than tensors.

Once we have specified and defined the architecture of our deep learning model, the notion of having used either the sequential class or the functional API no longer holds any importance. This is due to the reason that the steps which come after this will be the same regardless of the method used for defining the model.

In the compilation step, we proceed to choose the loss function and optimizer and hence, configuring the learning process of the deep learning model. In addition, we can also select some metrics that we want to monitor in the training session of the model. Now let’s see a simple and common example of a model in the compilation step being specified one loss function:

```
from keras import optimizers


model.compile(optimizer=optimizers.RMSprop(lr=0.001),

                    loss='mse',

                    metrics=['accuracy'])
```

Now, the last thing left to do is using the fit() method to pass the NumPy

arrays of training data (the input data and the corresponding target data) to the deep learning model, as shown below:

```
model.fit(input_tensor, target_tensor, batch_size=128, epochs=10)
```

The Pre-requisites for a Deep Learning Workstation

In this section, we will discuss what you need to set up your deep learning workstation. We will look at what is necessary, what is optional and what can speed up your work and what can slow it down, all of the aspects you would want to know about when putting together your workstation for deep learning.

The first point of discussion is the importance of a GPU in a deep learning workstation. While you may think that a deep learning code run on multicore and fast CPU seems adequate, that is not the case. A GPU is highly recommended for running deep learning code because it not only increases the speed factor by a whopping five or even 10. Moreover, deep learning code for applications, such as image processing (made up of a convolutional Neural network,) will be immensely slow and time-consuming. In such cases, a GPU proves to be the ideal contender as it not only increases the speed of the entire work process, but modern NVIDIA GPUs also feature well-optimized and dedicated “Tensor” cores developed for such tasks. In short, using a NIVIDA GPU is highly recommended for your deep learning workstation. Although there are cloud solutions, such as Google’s cloud platform, it can prove to be very expensive and inefficient in the long run.

Furthermore, the recommended operating system for a deep learning workstation, regardless of the GPU you are using (Nvidia or a Cloud solution), the recommended OS is Unix. Although the Keras backends do support Windows, it is still desirable if you choose Unix, or just install an Ubuntu OS as a dual boot alongside the Windows OS in your machine - Ubuntu will save a huge chunk of your time and resource later as compared to Windows so keep this in mind. In addition, before you can use Keras, you will also need to install its backend engines, i.e., TensorFlow, Theano, CNTK, or even all of them (if you intend to switch between the backend engines more often). However, be mindful that this book will focus primarily on TensorFlow and occasionally discuss Theano; however, we will discuss CNTK.

If you have the budget, go for the latest Nvidia Quadro GPUs as they have the most vRAM in the current GPU industry and dedicated tensor cores for such experiments. Moreover, Nvidia is the only company that has heavily invested in the deep learning market, hence the support and compatibility it provides for deep learning projects are outstanding and the best at the moment.

Jupyter Notebooks

Arguably the most efficient way to run any deep learning experiment is by using a Jupyter notebook. The Jupyter notebook is an application that generates a file known as a "notebook." This file can easily be edited in the browser, giving the best of both worlds, being able to execute Python code while also having the ability to annotate (what you are doing in) the code with rich text-editing features of the Jupyter notebook. This is why Jupyter Notebooks is so popular among the communities of data science and machine learning niche.

You can easily turn long experiments into smaller components with the ability to execute each small component independently, meaning that if you come across a problem in the later parts of your code, you can just execute the smaller components to find where the problem is instead of executing the entire code from the beginning to the end.

Jupyter Notebooks is not a necessary requirement for setting up your deep learning workstation as you can just execute the deep learning code from the standalone Python scripts within the IDE. Still, it is recommended to use this application for the sake of your convenience.

Deep Learning Binary Classification Example

We will now discuss the examples which we briefly discussed in the introduction of this chapter. The first example will be a machine learning problem that is most common in the real world, binary classification. The situation we will consider for the demonstration is that we need to set up a deep learning model to classify movie reviews as either positive or negative by giving it input data of the text content of the reviews being classified.

Dataset to be Used

The dataset we will be using for our deep learning model is the Internet Movie Database (IMDB), which consists of a set of fifty thousand (50,000)

reviews that are highly polarized. Moreover, this set of reviews is split into two equal parts. One part consists of 25,000 reviews for training the deep learning Neural network and the other part consists of 25,000 reviews for testing the deep learning network; each of these parts has a percentage of 50% negative and 50% positive reviews.

The reason we always use a separate test for testing the deep learning model is that using the same set it has been trained on is practically useless as the machine, as well as the user already knows the labels of the training set. Moreover, our main concern is to use this deep learning model for helping us with data that neither the machine has seen nor the user has sorted before.

Similar to when we used the MNIST dataset, it came preloaded with the Keras framework. The IMDB dataset also comes packaged with the same Keras framework. Hence we do not need to load it up separately. Moreover, the IMBD dataset already has preprocessed data, i.e., the sequence of words in the reviews has already been transformed into the corresponding sequence of integers (each of these integers corresponds to a word in the system's defined dictionary).

We will now proceed to load the IMDB dataset into our workstation. To do this, we will use the following lines of code:

```
from keras.datasets import imdb

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(

        num_words=10000)
```

If you look at the above lines of code, you'll see that there is an argument **num_words=10000**. This argument tells the system only to retain 10000 words that are the most frequently occurring in the dataset. By doing this, we discard the rare words used in the dataset and keep the size of vector data within a manageable range.

In the dataset, the labels (train_labels and test_labels) are basically the preprocessed integers 0 and 1 that indicate the corresponding annotation of the words. A word with a label integer "0" is a word with negative annotation, and a word with a label integer "1" is a word with positive annotation.

```
>>> train_data[0]

[1, 14, 22, 16, ... 178, 32]

>>> train_labels[0]

1
```

As we have already restricted the system to the top 10,000 most frequently used words, hence it is a given that no word index will exceed past this limit.

```
>>> max([max(sequence) for sequence in train_data])

9999
```

To decode the preprocessed reviews back into English, use the following lines of code (this is just for informative purposes, it's not necessary to perform this step in the usual sequence of things):

```
word_index = imdb.get_word_index()

reverse_word_index = dict(

        [(value, key) for (key, value) in word_index.items()])

decoded_review = ' '.join(

        [reverse_word_index.get(i - 3, '?') for i in train_data[0]])
```

In the first line, the argument word_index is the dictionary responsible for mapping the corresponding words to integers.

Preparing to Feed Data into the Neural Network

The data we have so far is still not ready to be fed into the Neural network yet. What we have is data in the form of integers, and what we need is data in the form of tensors. Hence the next plan of action is to convert this list of integers into tensors. This can be done in two ways which are:

1. First, we convert the data lists such that all of the lists are of the same length. This is done by padding them. After this, we proceed to convert the same length integer lists into integer tensors. Be mindful that the shape of these tensors should be (samples,

word_indices). We will now use the embedding layer as the first layer in the Neural network because it can handle these integer tensors.

2. The second method is to turn the data lists which we are working with into vectors. These vectors would be 0s and 1s, and this conversion can is done by one-hot encoding the data lists. The practical meaning of what this insinuates is that a sequence let's say [2, 4] is converted into a vector that is 10,000 dimensional. This vector would be largely 0s, with the exception of the two indices, 2 and 4. These indices would be vector 1s, and the rest of the indices would be vector 0s. To handle such type of floating data, we will use a dense layer as the first layer in the Neural network.

In this example, we will use the second method to convert the list of integers into tensors (vectorizing the source data). This will be done manually as follows:

```
import numpy as np


def vectorize_sequences(sequences, dimension=10000):
        results = np.zeros((len(sequences), dimension))
        for i, sequence in enumerate(sequences):
                results[i, sequence] = 1.
        return results


x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

After performing this step, the data samples would now look like this:

```
>>> x_train[0]
array([ 0., 1., 1., ..., 0., 0., 0.])
```

Apart from vectorizing the train_data and test_data, we should also vectorize the train_labels and test_labels. This is not hard, on the contrary, it is actually pretty straightforward as you can see from the two lines of code below:

```
y_train = np.asarray(train_labels).astype('float32')

y_test = np.asarray(test_labels).astype('float32')
```

Now, our data is ready to be inputted into the Neural network of the deep learning model.

Establishing the Neural Network

Here's a summary of the ingredients we are working with so far to build our Neural network. Our training data is made up of vectors (the input data) and scalars (the labels).

Now, we have to choose a type of network that works best with our ingredients, and so far, the choice is very straightforward and simple because we know that with such type of data, a **dense** layer will work best. Hence, our Neural network will be one that has completely connected stacks of layers featuring **relu** activations. Hence the argument to define such a network would be:

```
Dense(16, activation='relu').
```

The above argument is specifying the number of hidden units (16) of layers that are to be passed to each dense layer. To refresh our memory below is the chain of tensor operation that is typically implemented by a dense layer with a relu activation:

```
output = relu(dot(W, input) + b)
```

Since we have a total of 16 hidden units, the above matrix W (also known as the weight matrix) will be shaped as **(input_dimension, 16).** Similarly, if we consider a dot product of this weight matrix W, the input data will end up being projected onto representation space that is actually 16-dimension.

It is also important to understand the meaning of a representation's space dimensionality. In essence, the dimensionality of a representation space defines the freedom with which you allow the Neural network to learn internal representations. So, if we have a greater number of hidden units, this

means that we are allowing for a bigger high-dimensional space of representation. Thus, the Neural network has more freedom in regards to being able to learn representations that are even more complex; however, with more freedom to learn comes greater risks. It becomes harder to control what the network learns, and may lead to the Neural network learning patterns that are useless or may even harm the success potential, so be wise when making such a network.

Regarding the network architecture, you should always ask yourself these two questions when implementing a stack of dense layers into the Neural network:

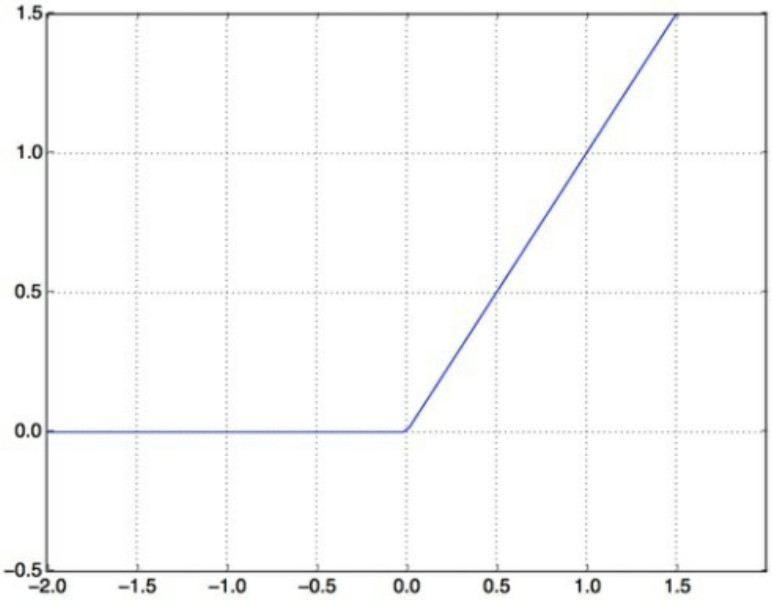
1. How many layers should I use?
2. What's the optimal number of hidden units that I should appoint to each layer?

In this example, we sought the following answers to the questions above:

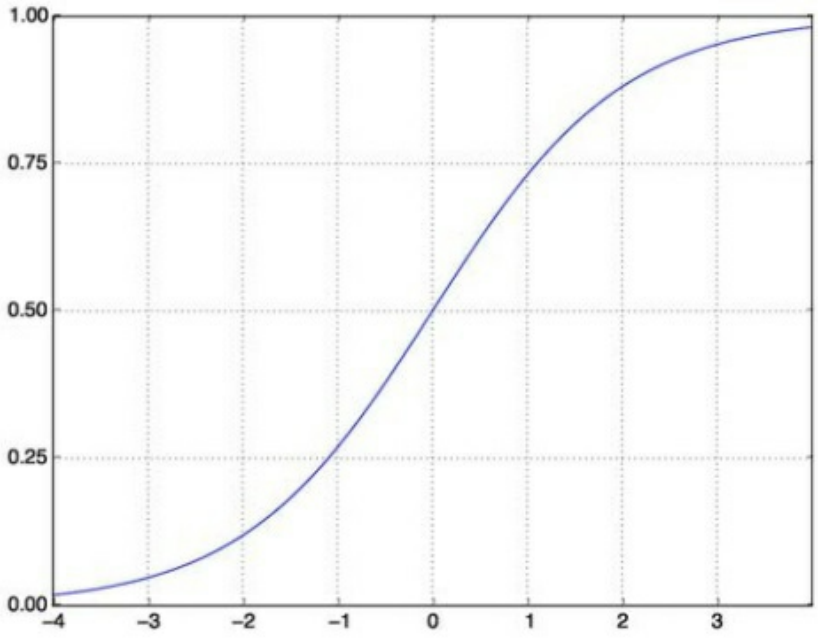
1. About two intermediate layers, each having a total of 16 hidden units.
2. A third layer, whose job will be to take the scalar predictions as its output. The prediction will be the annotation of the current review being analyzed (if the review is positive or negative).

In addition, the activation functions for the intermediate and third (final) layer are also different. The intermediate layer uses **relu** as the activation function while the third layer is using the sigmoid as the activation function. This is down to the nature of its job (outputting a probability value of either 0 or 1, for instance, a probability value of 1 depicts the likelihood of a review of being positive and vice versa).

The job of the relu function is to take the negative values and convert them into zeroes. On the other hand, a sigmoid function basically forces all arbitrary values into an interval of the shape [0, 1], giving us an output which can then be interpreted as a probability.



A Rectified Linear Unit Function (relu)



A Sigmoid Function

The Keras implementation of the building this network is shown below:

```
from keras import models

from keras import layers
```

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

The final step to complete the Neural network is to select a suitable loss function and an optimizer. Before making your choice, analyze the network you have built so far and ponder on what type of loss function would be the most optimal considering the type of problem we are tackling. In this case, as we are dealing with a problem relating to binary classification and the output of our network is in the form of a probability value, then the ideal choice would be to go for a **binary_crossentropy** loss function. While there are other choices available such as the **mean_squred_error** loss function, we will still go for the former as we are working with a network that gives an output of probability values. Cross-entropy is the most suitable for such situations. The reason why cross-entropy is a good contender is because of its function - it basically measures the distance between the probability distributions and the predictions.

So we are going with the cross-entropy loss function, and the optimizer we are choosing is the **rmsprop** optimizer. The metric which we will monitor during the training is the “accuracy.” To implement these elements into the network, we will use the following lines of code:

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

We can pass off these elements as strings due to the fact all of them (binary_crossentropy, rmsprop, and accuracy) are basically part of the Keras framework.

We can also configure our optimizer (its parameters) according to our needs by simply passing the optimizer class as an argument. Pretty simple as shown below:

```
from keras import optimizers


model.compile(optimizer=optimizers.RMSprop(lr=0.001),
        loss='binary_crossentropy',
        metrics=['accuracy'])
```

We can also use custom loss functions and metrics as shown below:

```
from keras import losses
from keras import metrics


model.compile(optimizer=optimizers.RMSprop(lr=0.001),
        loss=losses.binary_crossentropy,
        metrics=[metrics.binary_accuracy])
```

Validating the Approach

To monitor the accuracy of the deep learning model during its training, we need to create a "validation set" by setting aside a portion of samples from the training dataset. Let's set aside 10,000 samples for our validation dataset as shown below:

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

Repeating the training process for a dataset is known as an epoch. Once we have created a validation set, we will proceed with training the deep learning model for a total of 15 epochs on the samples x_train and y_train. This training will be done in batches with a size of 256 samples per batch. Moreover, the accuracy and loss of the samples we previously set aside (the

10,000 samples) will also be monitored. To do this, we will simply pass the set as an argument, i.e., validation_data as shown below:

```
model.compile(optimizer='rmsprop',
                    loss='binary_crossentropy',
                    metrics=['acc'])
history = model.fit(partial_x_train,
                        partial_y_train,
                        epochs=15,
                        batch_size=256,
                        validation_data=(x_val, y_val))
```

It takes for an epoch to complete on a CPU is roughly 2 seconds or even less than 2 seconds. The entire process of training is completed in a timeframe of 15 seconds, and when an epoch is completed, there is a brief pause before the next epoch starts. This pause happens so that the model can easily compute the loss and accuracy (in our case, it is computed on the validation dataset, which consists of ten thousand samples).

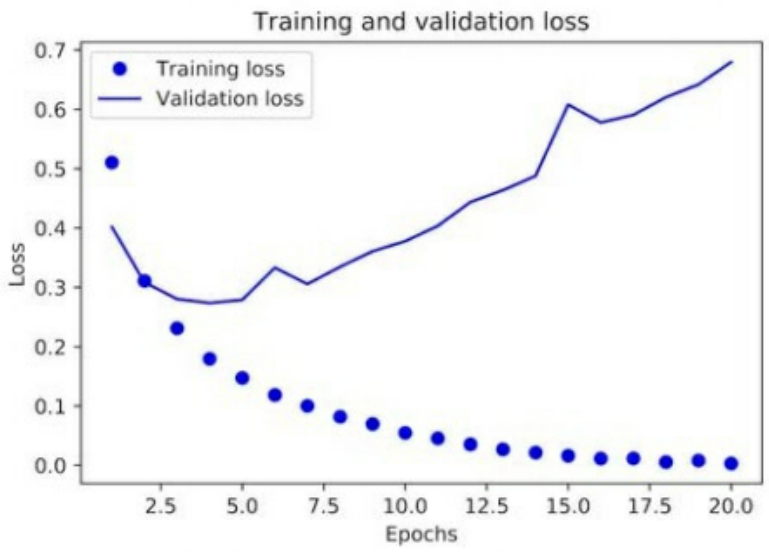
One more noticeable aspect of the model's working is that when we use the function model.fit(), we get a **history** object. The specialty of this "history" object is that it holds the data pertaining to everything that happened in the training process in the form of a dictionary.

```
>>> history_dict = history.history
>>> history_dict.keys()
[u'acc', u'loss', u'val_acc', u'val_loss']
```

Upon further analysis, we come to know that there are four distinct entries within the dictionary, and each entry is a per metric record of their values in the training session and validation session. We will now proceed to use the Matplotlib library to plot the losses and accuracy of both training and validation datasets beside each other.

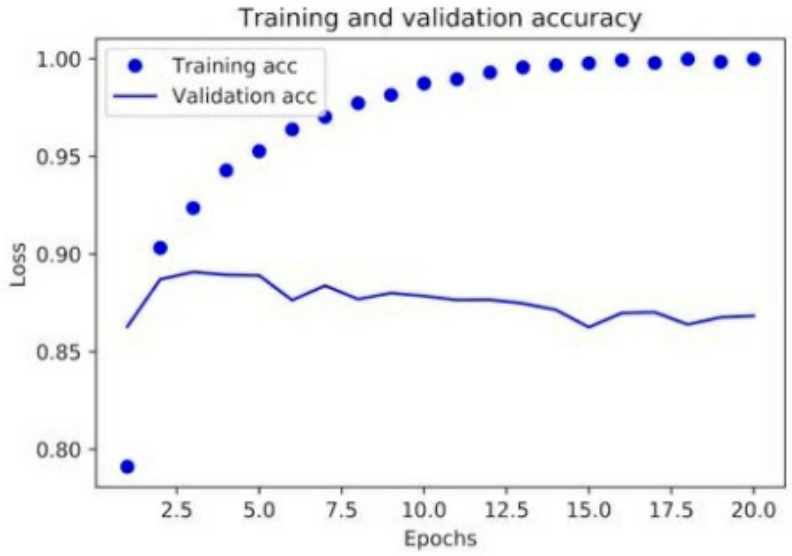
```
import matplotlib.pyplot as plt

history_dict = history.history

loss_values = history_dict['loss']

val_loss_values = history_dict['val_loss']

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

plt.plot(epochs, loss_values, 'bo', label='Training loss')

plt.plot(epochs, val_loss_values, 'b', label='Validation loss')

plt.title('Training and validation loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()
```

In the above demonstration, the label “bo” basically refers to a blue dot in the resulting graphical representation, and the label “b” refers to a static blue line in the same graph as shown below:



Now for plotting the accuracy of the training and validation datasets:

```
plt.clf()                                          ‘clearing the figure’

acc_values = history_dict['acc']

val_acc_values = history_dict['val_acc']


plt.plot(epochs, acc, 'bo', label='Training acc')

plt.plot(epochs, val_acc, 'b', label='Validation acc')

plt.title('Training and validation accuracy')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()


plt.show()
```



From these two graphs, we can clearly see the trend that with every training epoch, there is a visible decrease in loss and an increase in accuracy as we move through epochs. This is the essence of gradient descent optimization, which basically decreases the quantity which we want to minimize every epoch or iteration. However, the trend seems to take the opposite turn in the cases of validation loss and accuracy.

Both the loss and accuracy are seen to peak out at the 4^{th} epoch. This is an accurate representation of the warnings given to learning programmers that a deep learning model performing outstandingly on a training dataset is not sure to have to the same success on a new dataset, such as the validation dataset in our case. The exact phenomenon that we are dealing with is basically "overfitting," in other words, the training has become over-optimized, and the representations being learned by the model are too specific. Hence, there is no generalization to account for data that is outside the training dataset.

There is a range of techniques through which overfitting can be mitigated, which we will discuss in the coming chapters. For now, we could tackle the issue of overfitting by just stopping the training session just before it peaks, i.e., after three epochs.

Now let's train a new Neural network for four epochs and observe the results when testing it on new data:

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

```
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

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

model.fit(x_train, y_train, epochs=4, batch_size=256)
results = model.evaluate(x_test, y_test)
```

The end results are as shown below:

```
>>> results
[0.2929924130630493, 0.88327999999999995]
```

With such an ordinary approach, we scored an accuracy of 88%. Hence, if we use top of the line approaches, scoring accuracy of 95% would be fairly easy.

Using this Trained Network for New Data

After we have trained the Neural network, we can now use it for our own practical purposes, which was to classify the reviews as positive or negative. To do this, we will be using the **predict** method as shown below:

```
>>> model.predict(x_test)
array([[ 0.98006207]
[ 0.99758697]
[ 0.99975556]
...,
```

```
[ 0.82167041]

[ 0.02885115]

[ 0.65371346]], dtype=float32)
```

This shows the predictive results of the likelihood of a review to be positive or negative.

We cannot go into the very fundamental details of the following examples, as you should already have an idea of what's happening in the lines of code. We will be giving brief explanations and go through these examples to save our resources for more important discussions in the following chapters.

Deep Learning Multiclass Classification Example

In this example, the task faced by the Neural network will be to classify the newswires of Reuter into 46 mutually exclusive topics. The classes in this example are 46, making it a multiclass classification problem, more specifically a single-label multiclass classification problem because each class is exclusive from one another.

The Reuters Dataset

For this example, we will be working with the Reuters dataset. It comes preloaded with the Keras library, so it is easy to use as shown below:

```
from keras.datasets import reuters


(train_data, train_labels), (test_data, test_labels) = reuters.load_data(

        num_words=10000)
```

The number for samples for train_data and test_data are as follows:

```
>>> len(train_data)

8982

>>> len(test_data)

2246
```

The word indices are in the form of a list of integers, just like the IMDB reviews in the previous example.

```
>>> train_data[10]

[1, 245, 273, 207, 156, 53, 74, 160, 26, 14, 46, 296, 26, 39, 74, 2979,

3554, 14, 46, 4689, 4329, 86, 61, 3499, 4795, 14, 61, 451, 4329, 17, 12]
```

These list of integers can be decoded as follows:

```
word_index = reuters.get_word_index()

reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])

decoded_newswire = ' '.join([reverse_word_index.get(i - 3, '?') for i in

        train_data[0]])
```

Preparing the Data

We will now proceed to vectorize the data by using the same code we outlined in the previous example:

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):

        results = np.zeros((len(sequences), dimension))

        for i, sequence in enumerate(sequences):

                results[i, sequence] = 1.

        return results

x_train = vectorize_sequences(train_data)

x_test = vectorize_sequences(test_data)
```

As discussed earlier, data can be vectorized either by label listing it as an integer tensor or by one-hot encoding (mainly used for categorical encoding). The method of one-hot encoding is shown below:

```
def to_one_hot(labels, dimension=46):
        results = np.zeros((len(labels), dimension))
        for i, label in enumerate(labels):
            results[i, label] = 1.
        return results

one_hot_train_labels = to_one_hot(train_labels)
one_hot_test_labels = to_one_hot(test_labels)
```

This can also be done natively in Keras, as we did in the MNIST example in the beginning:

```
from keras.utils.np_utils import to_categorical

one_hot_train_labels = to_categorical(train_labels)
one_hot_test_labels = to_categorical(test_labels)
```

Building the Neural Network

This case is similar to the binary classification problem; however, the methods and parameters used will be completely different because adopting the same procedure, and choosing the same attributes would lead to big information bottlenecks.

For this scenario, we will be using a larger layer with 64 unit

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
```

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

model.add(layers.Dense(46, activation='softmax'))
```

Since the output of the layers is in the form of a probability, hence we will be using the **categorical_crossentropy** as our loss function. We can improve the result by minimizing the distance between the probability distribution output of the network and the true distribution of the labels.

```
model.compile(optimizer='rmsprop',
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])
```

Validating the Approach

The validation dataset for this network will consist of 1,000 samples.

```
x_val = x_train[:1000]

partial_x_train = x_train[1000:]


y_val = one_hot_train_labels[:1000]

partial_y_train = one_hot_train_labels[1000:]
```

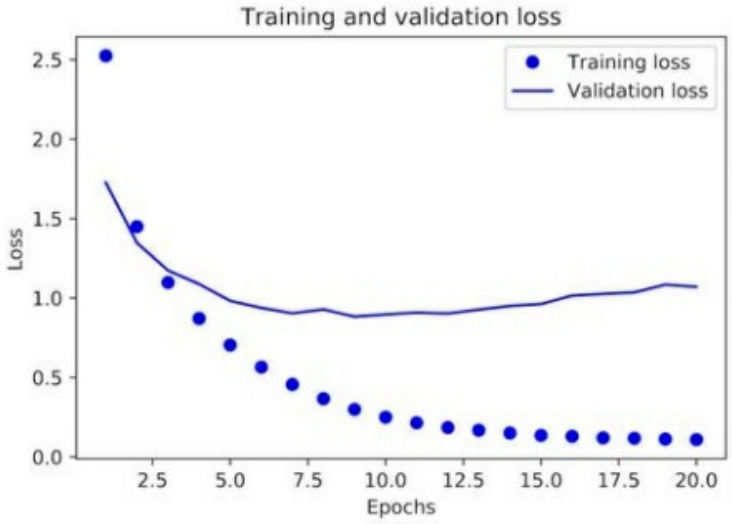
We will train the network for 15 epochs (be mindful that the graphical representation will show upto 20 epochs so don’t be bothered by that):

```
history = model.fit(partial_x_train,
                        partial_y_train,
                        epochs=15,
                        batch_size=512,
                        validation_data=(x_val, y_val))
```

We will now move towards showcasing the graphical representation of the loss and accuracy metrics.

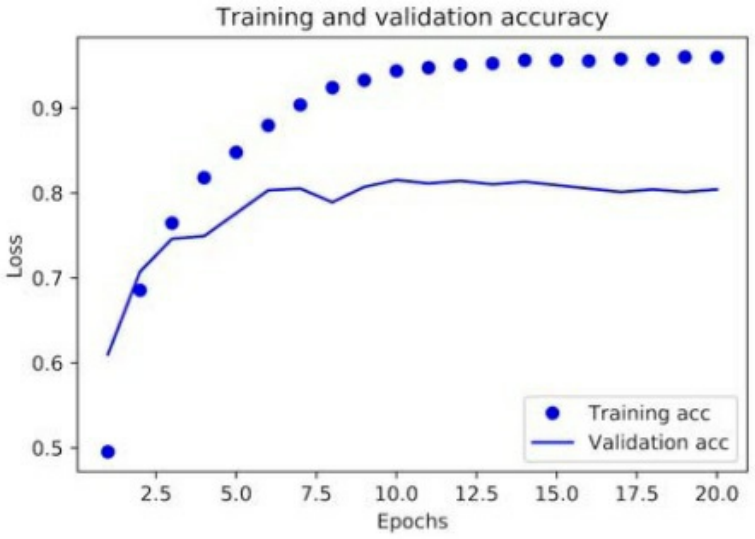
For the loss of the training and validation sets:

```
import matplotlib.pyplot as plt

loss = history.history['loss']
val_loss = history.history['val_loss']

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

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

plt.show()
```



For the accuracy of the training and validation sets:

```
plt.clf()

acc = history.history['acc']
val_acc = history.history['val_acc']

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



From the graphical representation, we can see that the overfitting phenomenon comes into play after 9 epochs. So we will train a new network for only 9 epochs and then test it on the testing dataset:

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

```
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(partial_x_train,
          partial_y_train,
          epochs=9,
          batch_size=512,
          validation_data=(x_val, y_val))
results = model.evaluate(x_test, one_hot_test_labels)
```

After training, the results we obtain are shown below:

```
>>> results
[0.9565213431445807, 0.79697239536954589]
```

We obtain an accuracy of approximately 80% by using this approach. When compared to a random baseline, these results are good. The results shown by a random baseline would be as follows:

```
>>> import copy
>>> test_labels_copy = copy.copy(test_labels)
>>> np.random.shuffle(test_labels_copy)
>>> hits_array = np.array(test_labels) == np.array(test_labels_copy)
```

```
>>> float(np.sum(hits_array)) / len(test_labels)

0.18655387355298308
```

Hence, 19% would be the accuracy.

Generating Predictions on New Data

We will now take all of the data from the testing set and generate topic predictions for it. Each entry in the **predictions** method will be a 46 length vector with coefficients summing to 1 along with the last line of code showcasing the class with the highest probability:

```
predictions = model.predict(x_test)

>>> predictions[0].shape

(46)

>>> np.sum(predictions[0])

1.0

>>> np.argmax(predictions[0])

4
```

Deep Learning Regression Example

In the previous examples, we dealt with problems relating to binary classification and multiclass classification, which was solved by training a Neural network such that when it would be given an input data point, it would be able to predict its discrete label.

This example relates to an entirely different machine learning problem known as regression. Instead of predicting a discrete label, the Neural network is required to predict a continuous value (for instance, meteorological predictions).

This example will focus on predicting house prices.

The Boston Housing Price Dataset

The dataset we will be using is the Boston Housing Price dataset, which

features data points of the Boston suburb in the era of the mid-1970s. The goal is to predict the median house prices while taking into consideration other factors such as crime rates and tax rates. Moreover, unlike the datasets used in the previous example, this one has a relatively small pool of data points, i.e., a total of 506 data points split into 404 training samples and 102 test samples.

We will now proceed to load the Boston Housing Price dataset into Keras:

```
from keras.datasets import boston_housing

(train_data, train_targets), (test_data, test_targets) =

➥ boston_housing.load_data()
```

After loading, let's take a glance at the data:

```
>>> train_data.shape

(404, 13)

>>> test_data.shape

(102, 13)
```

Each of the data samples comes with a total of 13 numerical features, such as crime rate, average rooms, and accessibility, etc.

The targets data points are in reality the median prices of the homes whose tenants are the owners themselves:

```
>>> train_targets

[ 15.2, 42.3, 50. ... 19.4, 19.4, 29.1]
```

The average price is seen to be anywhere from $10,000 and $50,000.

Preparing the Data

This time, the data we are dealing with features values that have different ranges. This makes learning very hard for the network even if it manages to adapt to the heterogeneous data. To prepare it for feeding into the network, we will perform a feature-wise normalization. This process basically takes each feature of the input data and performs a series of

arithmetic functions, specifically subtracting the feature's mean value and then dividing it by the standard deviation, as shown below:

```
mean = train_data.mean(axis=0)

train_data -= mean

std = train_data.std(axis=0)

train_data /= std


test_data -= mean

test_data /= std
```

Always remember that the quantities which are used for feature-wise normalizing the testing dataset are the ones that have already been computed from the training dataset.

Building the Neural Network

As we are working with a smaller sample size this time, i.e., a total of 506 samples, a small network will suffice. The makeup of this network will be a total of two hidden layers, and each of these hidden layers will have 64 units. Moreover, by using a small network, we will automatically lessen the overfitting in our Neural network. The deep learning Neural network will be built as follows:

```
from keras import models

from keras import layers


def build_model():

        model = models.Sequential()

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

                                input_shape=(train_data.shape[1],)))

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

```
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    return model
```

The above setup is a typical build for dealing with scalar regression problems. Our Neural network will end up a linear layer (single unit without any activation function). The reason why we did not add any activation function is because it would lead to only restraining the range that the output can take. As we have set the last layer of the network to be linear, in return, the network has more freedom in regard to predicting values in any range, instead of a constrained range with an activation sequence.

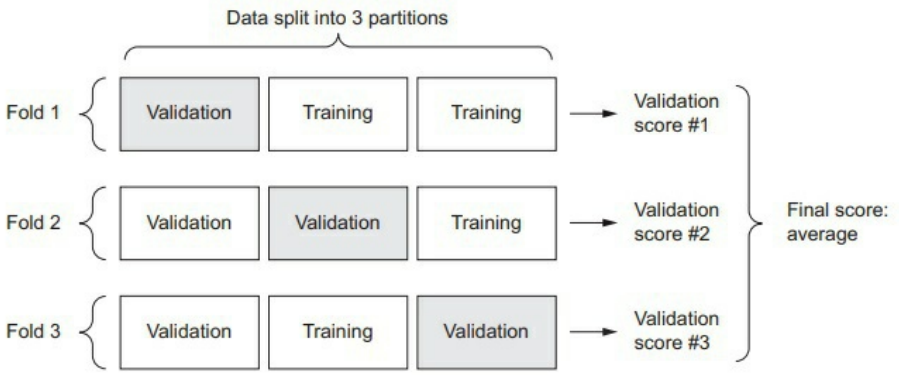
The loss function we are using in this network is the **mse** function or the mean squared error function. This function is related to the compilation of the network by taking the difference between the predictions and the target and squaring it.

Apart from this, we are now monitoring an entirely new metric during the training process, which is the “Mean Absolute Error” (MAE). This takes the difference between the predictions and the target and provides a corresponding absolute value, for instance, an MAE of 0.5 in this example would insinuate that the predictions of the house prices are deviating by an amount of $500.

Validating the Approach

The method of validation we will be using in this example is the K-Fold validation. First things first, we will set aside a portion of samples as our validation dataset. While we are already dealing with a small number of data samples, hence the validation dataset will also be considerably smaller, i.e., a dataset of only 100 samples. In turn, our validation scores will be more prone to changes depending on the data points being used for validation and training. Hence we will be using the K-Fold validation to cover this discrepancy. In K-Fold validation, we basically split the data, which is currently available into partitions known as K-Partitions (K = 4 or 5). After partitioning, we make identical K-models and train the Neural Network on parameters such as K-1 partitions and the evaluations are done on the rest of the partitions, to understand this better, take a look at the demonstration

below:



We have coded a K-Fold validation below:

```
import numpy as np

k=4

num_val_samples = len(train_data) // k

num_epochs = 100

all_scores = []

for i in range(k):

        print('processing fold #', i)

        val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]

        val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]


        partial_train_data = np.concatenate(

                [train_data[:i * num_val_samples],

                 train_data[(i + 1) * num_val_samples:]],

                axis=0)
```

```
        partial_train_targets = np.concatenate(
                [train_targets[:i * num_val_samples],
                 train_targets[(i + 1) * num_val_samples:]],
                axis=0)
        model = build_model()
        model.fit(partial_train_data, partial_train_targets,
                  epochs=num_epochs, batch_size=1, verbose=0)
        val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
        all_scores.append(val_mae)
```

We will specify the number of epochs as 100 by the argument **num_epochs = 100**. The corresponding results are shown below:

```
>>> all_scores
[2.588258957792037, 3.1289568449719116, 3.1856116051248984, 3.0763342615401386]
>>> np.mean(all_scores)
2.9947904173572462
```

Different number of epochs are giving different results, in our case, ranging from 2.6 to 3.2. The entire purpose of the K-Fold validation is to give a mean of these different scores, which is 3.0 in our case. However, we are still deviating by an average of $3,000 and this is very significant.

We will now try training the network longer, this time for 500 epochs. In addition, we will modify the training session loops such that the performance of the model one each epoch is recorded in a validation score log.

The code to save the validation logs a each fold is as shown below:

```
num_epochs = 500
```

```
all_mae_histories = []
for i in range(k):
    print('processing fold #', i)
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]

    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)
    model = build_model()
    history = model.fit(partial_train_data, partial_train_targets,
                        validation_data=(val_data, val_targets),
                        epochs=num_epochs, batch_size=1, verbose=0)
    mae_history = history.history['val_mean_absolute_error']
    all_mae_histories.append(mae_history)
```

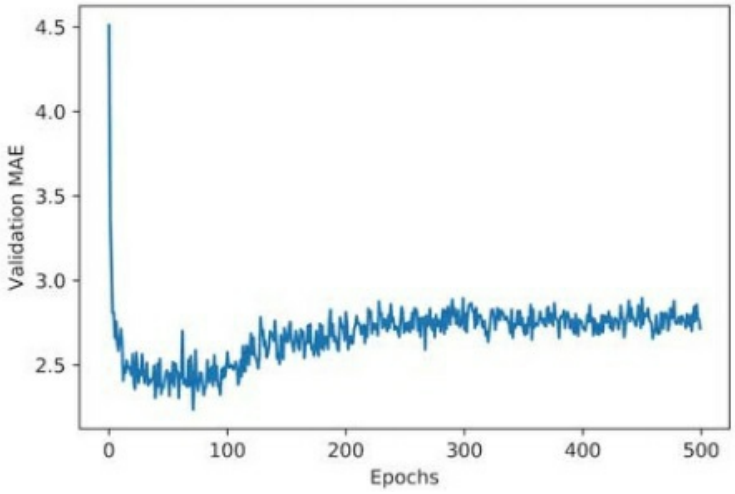
To compute and find out the average MAE scores of all the K-Folds, we will

use the following lines of code:

```
average_mae_history = [

        np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)]
```

We will now plot the validation scores we have obtained from the Neural network's training so far and plot it into a graph to analyze the results with more clarity:

```
import matplotlib.pyplot as plt


plt.plot(range(1, len(average_mae_history) + 1), average_mae_history)

plt.xlabel('Epochs')

plt.ylabel('Validation MAE')

plt.show()
```



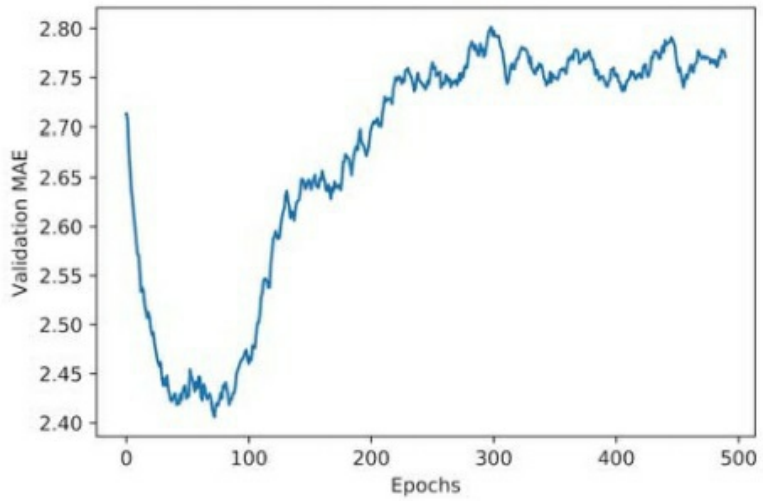
The graphical representation is still unclear because of the issues relating to scaling and frequent variance. To remedy this, we will do the following:

1. We will take the initial data points and omit the first 10 that are already on a different scale as compared to the rest of the curve
2. We will replace each point in the curve with an exponential

moving average. This average is taken from the previous points and will give us a smooth curve.

Now, we will use the following lines of code to plot the validation scores by omitting the initial 10 data points:

```
def smooth_curve(points, factor=0.9):
    smoothed_points = []
    for point in points:
        if smoothed_points:
            previous = smoothed_points[-1]
            smoothed_points.append(previous * factor + point * (1 - factor))
        else:
            smoothed_points.append(point)
    return smoothed_points
smooth_mae_history = smooth_curve(average_mae_history[10:])

plt.plot(range(1, len(smooth_mae_history) + 1), smooth_mae_history)
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
```



In this graphical representation, we can see that after 80 epochs, the improvement of the validation MAE comes to a halt, and past that, the network experiences overfitting.

Once we have tuned the network on our desired parameters, we can begin testing it with these optimal parameters and check the performance on the testing dataset.

```
model = build_model()

model.fit(train_data, train_targets,

          epochs=80, batch_size=16, verbose=0)

test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)
```

The final result obtained is:

```
>>> test_mae_score

2.5532484335057877
```

We can see that even after all this, the Neural network’s predictions are still off by a value of $2,550.

Chapter 4: Using Deep Learning for Computer Vision

In this chapter, our discussion will revolve around the understanding and visualization of convnets, which are known as convolutional neural networks. The importance of convnets can be estimated from the fact that its use stretches out universally to every computer vision application. After understanding the conceptual realms surrounding convnets, we will proceed to use them practically in problems relating to image-classification. The dataset we will be using in these practical demonstrations will be small, and almost anyone can practice light on computational resources. As such, an example, without needing to arrange resources that are typically available in a big tech company.

What is Convnet? Working with Convolution Operations

Before we begin discussing the details of a convnet, we will first look at a practical demonstration of a deep learning model using a convnet. We will solve the problem of classifying digits in the MNIST dataset (the same task which we demonstrated previously in the second chapter and received an accuracy of 97.8% by using a densely connected Neural network). Compared to the previous method we used for performing this task, implementing a convnet is relatively simple. We will also see how the results of a convnet hold up to a densely connected Neural network.

The following lines of code show how we can instantiate a small convnet in our deep learning model:

```
from keras import layers

from keras import models


model = models.Sequential()

model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))

model.add(layers.MaxPooling2D((2, 2)))
```

```
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

From the above lines of code, we can see that a typical small convnet is just a stack of two types of layers, namely the Conv2D and MaxPooling2D layers. Regarding the input data, a convnet only accepts tensors that are of the following shape: (image_height, image_width, image_channels). Now we know the shape of the input tensors required, we will configure the convnet accordingly to make it process tensor inputs corresponding to the format of the MNIST images dataset, of the size (28, 28, 1). This can be easily done by using an argument on the first layer. This argument is input_shape (28, 28, 1).

The Network's architecture will look like this according to the changes we just mentioned:

```
>>> model.summary()

_________________________________________________________________

Layer (type) Output Shape Param #

=================================================================

conv2d_1 (Conv2D) (None, 26, 26, 32) 320

_________________________________________________________________

maxpooling2d_1 (MaxPooling2D) (None, 13, 13, 32) 0

_________________________________________________________________

conv2d_2 (Conv2D) (None, 11, 11, 64) 18496

_________________________________________________________________

maxpooling2d_2 (MaxPooling2D) (None, 5, 5, 64) 0

_________________________________________________________________
```

```
conv2d_3 (Conv2D) (None, 3, 3, 64) 36928

=================================================================

Total params: 55,744

Trainable params: 55,744

Non-trainable params: 0
```

From the network's architectrure displayed above, we can see that the Conv2D and MaxPooling2D layers output a 3D tensor and the shape of this tensor is (height, width, channels). One more thing to point out is that as we move deeper into the Neural network, the dimensions of width and height will consequently shrink, in addition, the very first argument which we passed on to the layer Conv2D is responsible for controlling the amount of channels in the network.

The next plan of action is to take the output tensor of the last layer and feed it into a classifier network. This classifier network is made up of a stack of dense layers and what they essentially do is take the 1D vectors and process them even though the network is outputting 3D tensors. So, we will proceed to convert the 3D tensors into a 1D tensor and after doing this, we will also throw in a bunch of dense layers, as shown below:

```
model.add(layers.Flatten())

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

model.add(layers.Dense(10, activation='softmax'))
```

In the above lines of code, we are performing a 10-way classification with a softmax activation. Moreover, the classification is being done with the last layer giving 10 outputs. The Neural network thus far looks like this:

```
>>> model.summary()

Layer (type) Output Shape Param #

=================================================================

conv2d_1 (Conv2D) (None, 26, 26, 32) 320
```

```
_________________________________________________________________
maxpooling2d_1 (MaxPooling2D) (None, 13, 13, 32) 0
_________________________________________________________________
conv2d_2 (Conv2D) (None, 11, 11, 64) 18496
_________________________________________________________________
maxpooling2d_2 (MaxPooling2D) (None, 5, 5, 64) 0
_________________________________________________________________
conv2d_3 (Conv2D) (None, 3, 3, 64) 36928
_________________________________________________________________
flatten_1 (Flatten) (None, 576) 0
_________________________________________________________________
dense_1 (Dense) (None, 64) 36928
_________________________________________________________________
dense_2 (Dense) (None, 10) 650
=================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
```

We will now start training of the convnet Neural network on the MNIST training set. You will notice most of the code being repeated from the example in chapter 2.

```
from keras.datasets import mnist
from keras.utils import to_categorical
```

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

Now to evaluate the deep learning model after training it:

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> test_acc
0.99080000000000001
```

So from the above results, we can see that by using a convnet on the same MNIST dataset, we get an accuracy of 99.3% as compared to the 97.8% accuracy by using a densely connected network.

The reason for this improvement in accuracy is because of the Conv2D and MaxPooling2D layers, and to understand this better, we will discuss these two layers in detail.

The Convolution Operation (CONV2D)

Let's talk about what makes a convolution layer different from a densely connected layer. The most fundamental difference between these two comes in their learning patterns. While a dense layer takes the approach of learning a global pattern within the input space, on the other hand, a convolution layer does the opposite, i.e., it learns a local pattern.

Due to this characteristic of convnets, they have developed the following two important properties:

1. The patterns learned by convnets are translation invariant. This means that when a convnet is finished with learning a particular pattern, let's say, in the upper-left corner of a drawing or a picture, it can then detect and recognize this particular pattern in any location of the picture. This is where a densely connected network falls behind as for it to recognize the same pattern in a new location, it will have to learn it from scratch. Hence, in terms of image processing, convnets are not only data efficient, but they are also resource-efficient as well as they only require a small number of training samples to learn those representations that have a much-needed generalization power.
2. Convnets can learn the spatial hierarchies of patterns. The visual world is in nature, spatially hierarchical, hence making this property of convnets very important. The learning process works as follows: in a convnet, there are several convolution layers. Based on the hierarchical position of these layers, they learn different patterns.

   The first convolutional layer will only learn small local patterns, while the second convolutional layer will learn bigger local patterns that have the same features as the smaller ones. In this way, a convnet is capable of not only learning increasingly complex patterns, but it can also learn abstract concepts.

Convolutional operations work on 3D tensors. This tensor is known as a "feature map." This tensor has two spatial axes and a depth axis. In other words, a convolution operation takes out a patch from this inputted feature map and applies this transformation on all of the other patches. In this way,

the network obtains an output feature map from an input feature map.

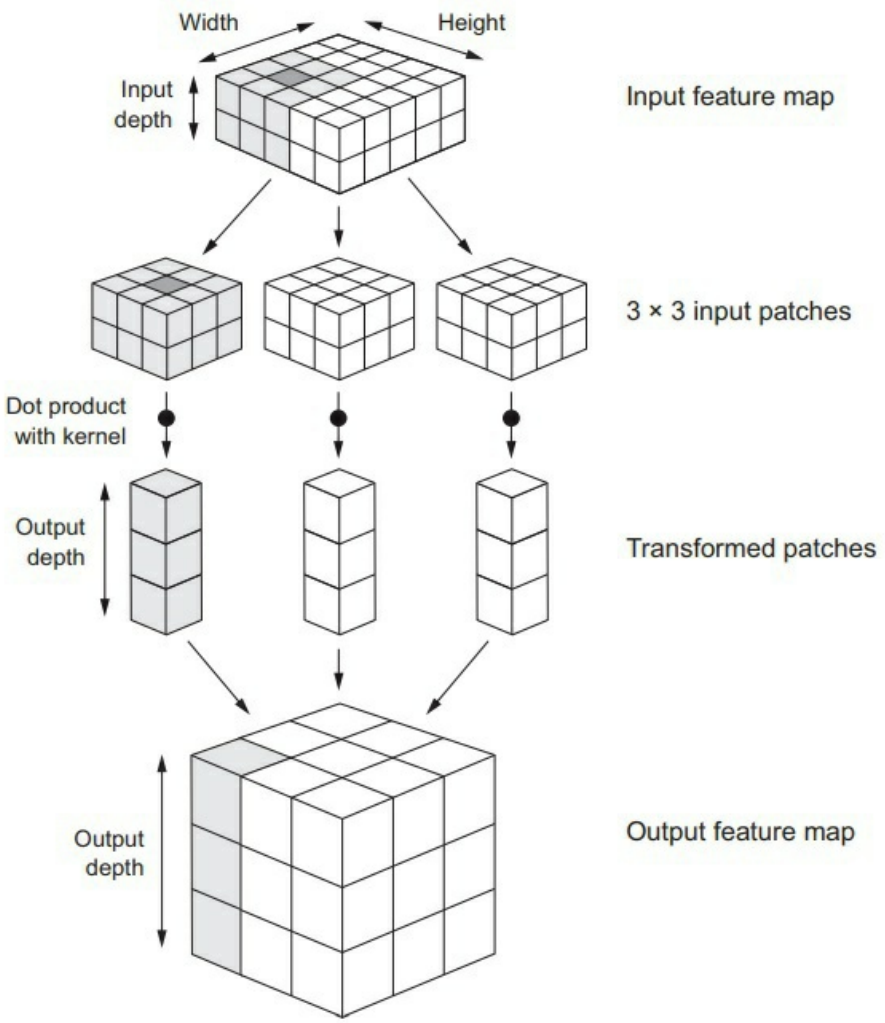
Furthermore, the following two parameters define convolutions:

1. Size of the patches taken from the input feature map. This size is either 3x3 or 5x5.
2. The output feature map's depth refers to the number of filters that have been computed by the convolution operation.

You can see these parameters in Keras as well so much so that they are the very first arguments which are passed to the convolution layers as:

```
Conv2D(output_depth, (window_height, window_width))
```

The size of the windows, as shown in the argument above, can be either 3x3 or 5x5. What the convolution operation does in this argument is that it "slides" the windows mentioned above over a given 3D input feature map. During the sliding, the operation searches for 3D patches that have the corresponding features (shape (window_height, window_width, input_depth)), and wherever it finds such a 3D patch, it stops and extracts this 3D patch. Once a 3D patch has been extracted, it is converted into a 1D vector with shape (output_depth). Once enough 3D patches have been extracted and converted into 1D vectors, the next plan of action is for the convolution is to reassemble them into a 3D feature map spatially. This is the output feature map. The shape of this 3D feature map would be (height, width, output_depth). To understand this concept better, take a look at the visualization of how convolutions work.



(The Working of a Convolution)

Some characteristics of the output feature map may be different from the input feature map, such as the height and width, this is because of two main reasons:

1. Border effects (This can be remedied by padding the input feature map).
2. Using strides.

The MaxPooling2D Operation

The main job of the MaxPooling2D operation is to downsample the feature maps. In the previous examples of convolution operations, we can see that initially, the size of the feature map was 26x26, but as soon as it passed through the maxpooling2D layer, the size of the feature map was

downsampled to 13x13.

Conceptually, the max-pooling operation is similar to the convolution operation, the point where this similarity comes to an end is when both of the operations need to convert the local patches. A convolution operation does this by using a convolution kernel (basically an already learned linear transformation) while, on the other hand, a max-pooling operation performs this transformation by using a **max** tensor operation, which is hardcoded. One more big difference between the two operations is that max-pooling is most of the time, done with windows of the size 2x2 and stride 2 while a convolution is done with a window that has a size of 3x3. Moreover, no stride is used (stride 1).

You might be wondering that what is the point of even downsampling the feature maps in the first place. To answer this, let's look at a convolutional base model without a downsized feature map:

```
model_no_max_pool = models.Sequential()

model_no_max_pool.add(layers.Conv2D(32, (3, 3), activation='relu',

                                    input_shape=(28, 28, 1)))

model_no_max_pool.add(layers.Conv2D(64, (3, 3), activation='relu'))

model_no_max_pool.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

The summary of this model is as follows:

```
>>> model_no_max_pool.summary()

Layer (type) Output Shape Param #

=================================================================

conv2d_4 (Conv2D) (None, 26, 26, 32) 320

_________________________________________________________________

conv2d_5 (Conv2D) (None, 24, 24, 64) 18496

_________________________________________________________________
```

conv2d_6 (Conv2D) (None, 22, 22, 64) 36928

==========

Total params: 55,744

Trainable params: 55,744

Non-trainable params: 0

After analyzing this model, we come to know that two major things are wrong with this particular setup:

1. This setup is not optimal in regards to learning a feature's spatial hierarchy. This is because the window in the network's third layer is of the size 3x3, and it will only contain information that is coming from the initial input, and the size of this window is 7x7. Hence, we require the features of the convolution layer to store the information pertaining to the input's totality.
2. The setup is unnecessarily large for our purposes of a small deep learning model. This setup's final feature map has an enormous number of coefficients, i.e., 22 x 22 x 64 which per sample, equals to a total of 30,976 coefficients. Now, if we want to add a dense layer on this model, then we would first need to flatten the feature map. So let's say we do flatten it out and put a size 512 dense layer on top of it, the resulting parameters of the layer would be around 15.8 million. This would result in overfitting of the model.

So, the essence of downsampling is just to cut down the number of coefficients our feature map has, besides, downsampling also creates a kind of filter for spatial hierarchies. This is done by arranging successive convolutional layers in such a way that they progressively deal with larger windows.

Training a Convnet

In computer vision, the most common task for which deep learning models are used for is image-classification. Hence, we will be training our deep learning model using a convnet on a dataset consisting of 4,000 picture samples of cats and dogs. The task which is to be performed by the network

is to learn to recognize dogs and cats separately in the picture samples. As always, we divide the 4,000 samples between training, validation, and testing. 2,000 samples will be used for training the network, 1,000 samples will be used as the validation set, and the remaining 1,000 samples will be used as the testing set.

As we are working with a considerably small dataset, we will need to strategize the training of the network accordingly. At first, we will not consider any regularization while training the convnet with the 2,000 sample training dataset. By doing so, we will essentially establish a baseline detailing the limits of what is the network is capable of. The classification accuracy thus obtained will be a measly 71%, with the main point of concern being overfitting. We will then proceed to diminish this overfitting as much as possible by using a rather robust and potent technique known as "data augmentation." This will bring the classification accuracy of the convnet up to 81%, a sizeable difference as compared to before.

This is all which will be covered in this section. In the following sections of this chapter, we will discuss another pair of techniques that help us implement deep learning models on small datasets which are:

1. Feature extraction with a pertained network. (Accuracy of 90-96%)
2. Fine-tuning a pertained network. (Accuracy of 97%).

By combining these two techniques with the one we have just discussed, you can develop a very powerful toolbox for handling image-classification problems anytime and anywhere.

Downloading the Data

The very first thing to do is to obtain the data on which we will train and build our deep learning model's network. We will be using a dataset released by Kaggle, and this dataset is known as the Dogs vs. Cats dataset. This dataset does not come packaged in Keras. Hence we will need to download it from Kaggle's website. Also, before you can download the dataset, you will need to create a Kaggle account if you don't have one.

www.kaggle.com/c/dogs-vs-cats/data

The pictures included in this dataset are in a JPEG format, and they are of

medium resolution. Some examples of the pictures included in this dataset are shown below (the pictures shown are not modified or edited, they differ in sizes to make the dataset heterogenous):

Although the dataset consists of a total of 25,000 pictures (12,500 dogs and 12,500 cats), we will be only using a total of 4,000 samples in accordance to the training, validation and testing sets we discussed beforehand. If we work with a large dataset, then it would defeat the purpose of using deep learning for small datasets.

Hence, we will divide the dataset accordingly: 1,000 samples of each class (dogs and cats) for our training dataset, 500 samples of each class for our validation dataset, and 500 samples of each class for the testing dataset.

To copy these samples to their corresponding training, validation and testing directories, we will use the following lines of code:

```
import os, shutil


original_dataset_dir = '/Users/fchollet/Downloads/kaggle_original_data'


base_dir = '/Users/fchollet/Downloads/cats_and_dogs_small'

os.mkdir(base_dir)
```

```
train_dir = os.path.join(base_dir, 'train')
os.mkdir(train_dir)
validation_dir = os.path.join(base_dir, 'validation')
os.mkdir(validation_dir)

test_dir = os.path.join(base_dir, 'test')
os.mkdir(test_dir)

train_cats_dir = os.path.join(train_dir, 'cats')
os.mkdir(train_cats_dir)

train_dogs_dir = os.path.join(train_dir, 'dogs')
os.mkdir(train_dogs_dir)

validation_cats_dir = os.path.join(validation_dir, 'cats')
os.mkdir(validation_cats_dir)

validation_dogs_dir = os.path.join(validation_dir, 'dogs')
os.mkdir(validation_dogs_dir)

test_cats_dir = os.path.join(test_dir, 'cats')
os.mkdir(test_cats_dir)

test_dogs_dir = os.path.join(test_dir, 'dogs')
os.mkdir(test_dogs_dir)
```

```
fnames = ['cat.{}.jpg'.format(i) for i in range(1000)]
for fname in fnames:
        src = os.path.join(original_dataset_dir, fname)
        dst = os.path.join(train_cats_dir, fname)
        shutil.copyfile(src, dst)

fnames = ['cat.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
        src = os.path.join(original_dataset_dir, fname)
        dst = os.path.join(validation_cats_dir, fname)
        shutil.copyfile(src, dst)

fnames = ['cat.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
        src = os.path.join(original_dataset_dir, fname)
        dst = os.path.join(test_cats_dir, fname)
        shutil.copyfile(src, dst)
fnames = ['dog.{}.jpg'.format(i) for i in range(1000)]
for fname in fnames:
        src = os.path.join(original_dataset_dir, fname)
        dst = os.path.join(train_dogs_dir, fname)
        shutil.copyfile(src, dst)
```

```
fnames = ['dog.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
        src = os.path.join(original_dataset_dir, fname)
        dst = os.path.join(validation_dogs_dir, fname)
        shutil.copyfile(src, dst)

fnames = ['dog.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
        src = os.path.join(original_dataset_dir, fname)
        dst = os.path.join(test_dogs_dir, fname)
        shutil.copyfile(src, dst)
```

We will now double check number of pictures in each directory:

```
>>> print('total training cat images:', len(os.listdir(train_cats_dir)))
total training cat images: 1000
>>> print('total training dog images:', len(os.listdir(train_dogs_dir)))
total training dog images: 1000
>>> print('total validation cat images:', len(os.listdir(validation_cats_dir)))
total validation cat images: 500
>>> print('total validation dog images:', len(os.listdir(validation_dogs_dir)))
total validation dog images: 500
```

```
>>> print('total test cat images:', len(os.listdir(test_cats_dir)))

total test cat images: 500

>>> print('total test dog images:', len(os.listdir(test_dogs_dir)))

total test dog images: 500
```

So far, everything looks good and in place. We will now proceed to build the network as we have obtained the necessary data.

Building the Network

As we have already built a small yet simple convnet for an MNIST dataset, we will be using the same general structure, i.e., the stack of layers being used will be an alternating arrangement of **Conv2D** and **MaxPooling2D** layers. The convolution operation will be using a relu activation function. However, we will be adding another Conv2D and MaxPooling2D layer to make the network larger because, unlike before, we are now dealing with a more complex problem and larger images. By doing so, we will not only be augmenting the capacity of the network we are building, but we will also be able to reduce the size of the feature maps even further before we reach the point of the **flatten** layer. Following this concept, we will be starting from inputs with a size of 150x150 and ending up feature maps of the size 7x7 right before the **flatten** layer.

By using common sense, we can see that the nature of the problem we are tackling is, in essence, a binary classification problem. Just as how we dealt with such problems in the previous chapters, we will be using a size 1 dense layer with a sigmoid activation function to end our network, hence encoding the probability of whether the network is looking at a picture of a dog or a cat.

The network will be built as follows:

```
from keras import layers

from keras import models


model = models.Sequential()
```

```
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

While we are at it, let’s take a sneak peek at how the dimensions of the feature maps are changing through every succeeding layer they go through:

```
>>> model.summary()
Layer (type) Output Shape Param #
=================================================================
conv2d_1 (Conv2D) (None, 148, 148, 32) 896
_________________________________________________________________
maxpooling2d_1 (MaxPooling2D) (None, 74, 74, 32) 0
_________________________________________________________________
conv2d_2 (Conv2D) (None, 72, 72, 64) 18496
```

maxpooling2d_2 (MaxPooling2D) (None, 36, 36, 64) 0
conv2d_3 (Conv2D) (None, 34, 34, 128) 73856
maxpooling2d_3 (MaxPooling2D) (None, 17, 17, 128) 0
conv2d_4 (Conv2D) (None, 15, 15, 128) 147584
maxpooling2d_4 (MaxPooling2D) (None, 7, 7, 128) 0
flatten_1 (Flatten) (None, 6272) 0
dense_1 (Dense) (None, 512) 3211776
dense_2 (Dense) (None, 1) 513
===
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0

We are using a network that ends with a singular sigmoid unit, hence the loss function which we will be using for the network is crossentropy. The optimizer used will be rmsprop.

We will configure the network accordingly:

```
from keras import optimizers

model.compile(loss='binary_crossentropy',

optimizer=optimizers.RMSprop(lr=1e-4),

metrics=['acc'])
```

Preprocessing the Data

As is the case with building Neural networks, we will now proceed to convert the data into the appropriate preprocessed floating-point tensors. This will be done as follows:

1. Go through the pictures in the sample data
2. Decode these pictures which are in a JPEG format into pixels on RGB grids
3. Convert these pixels into floating-point tensors
4. Rescale the values of the resulting pixels into the (0, 1) interval

Fortunately, we will not have to perform these steps entirely manually. Keras has the necessary utilities to help us perform these steps as shown in the code below:

```
from keras.preprocessing.image import ImageDataGenerator


train_datagen = ImageDataGenerator(rescale=1./255)

test_datagen = ImageDataGenerator(rescale=1./255)


train_generator = train_datagen.flow_from_directory(

        train_dir,

        target_size=(150, 150)

        batch_size=20,
```

```
        class_mode='binary')
validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')
```

The image data generator used in this code will keep generating these data batches infinitely. Now we will break the generating loop once we have gotten the converted data:

```
>>> for data_batch, labels_batch in train_generator:
>>> print('data batch shape:', data_batch.shape)
>>> print('labels batch shape:', labels_batch.shape)
>>> break
data batch shape: (20, 150, 150, 3)
labels batch shape: (20,)
```

Now, to fit the deep learning model using a batch generator:

```
history = model.fit_generator(
train_generator,
steps_per_epoch=100,
epochs=30,
validation_data=validation_generator,
validation_steps=50)
```

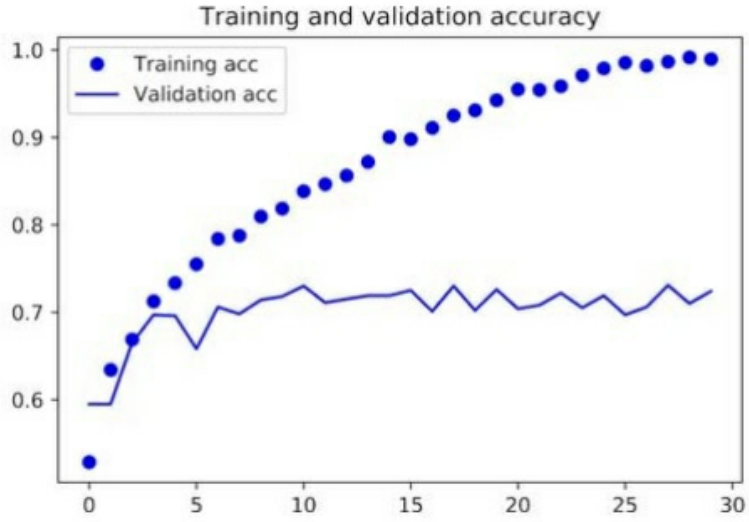
Never a bad idea to save the model:

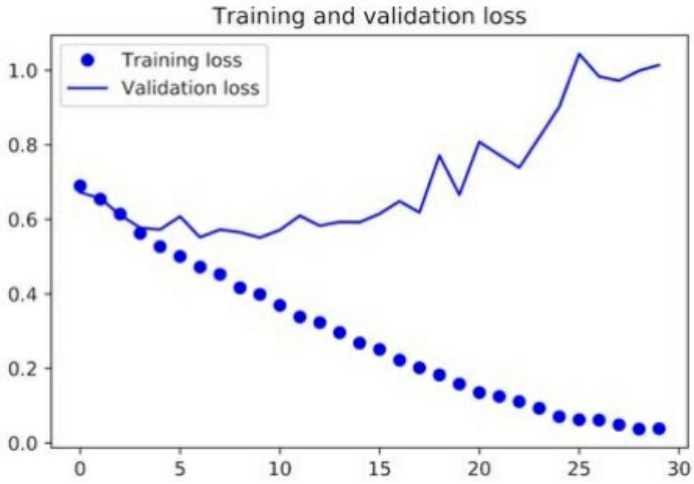
```
model.save('cats_and_dogs_small_1.h5')
```

We will now plot the loss and accuracy data of this deep learning model to analyze it and make changes to the model accordingly:

```
import matplotlib.pyplot as plt


acc = history.history['acc']

val_acc = history.history['val_acc']

loss = history.history['loss']

val_loss = history.history['val_loss']


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


plt.plot(epochs, acc, 'bo', label='Training acc')

plt.plot(epochs, val_acc, 'b', label='Validation acc')

plt.title('Training and validation accuracy')

plt.legend()


plt.figure()


plt.plot(epochs, loss, 'bo', label='Training loss')

plt.plot(epochs, val_loss, 'b', label='Validation loss')

plt.title('Training and validation loss')

plt.legend()


plt.show()
```

By running this code, we get the following graphical representations of the loss and accuracy data of the model:





In our current model, the problem of overfitting is the main concern at this point. We have learned several techniques to deal with overfittings, such as the weight decay method and the dropout method. However, in this example, we will be using a new technique known as data augmentation, and this technique is used almost universally for deep learning models that are dealing with image-classification tasks.

Mitigating Overfitting by Data Augmentation

Overfitting is very prominent in models that are using a small number of

samples for training their Neural network. Data augmentation diminishes overfitting by opting for the approach that, if we have a larger number of training samples, then there will be less overfitting. Hence data augmentation generates new training samples from the existing set of training samples through randomly transforming data into similarly structured data (known as augmentation).

We will now establish a data augmentation configuration for our model using the ImageDataGenerator function:

```
datagen = ImageDataGenerator(
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')
```

Now to display some of the augmented images which we have created for the network's training:

```
from keras.preprocessing import image

fnames = [os.path.join(train_cats_dir, fname) for
         fname in os.listdir(train_cats_dir)]

img_path = fnames[3]

img = image.load_img(img_path, target_size=(150, 150))
```

```
x = image.img_to_array(img)

x = x.reshape((1,) + x.shape)

i=0
for batch in datagen.flow(x, batch_size=1):
        plt.figure(i)
        imgplot = plt.imshow(image.array_to_img(batch[0]))
        i += 1
        if i % 4 == 0:
                break

plt.show()
```

These are the cat pictures that have been generated through data augmentation. However, this will not completely diminish overfitting because we are just remixing the existing information and training the network on it, so no new data is being produced. To further reduce overfitting in the model, we will add in a **dropout** layer and place it just before the densely connected classifier.

We will now define a convnet and this time, it will include the dropout layer:

```
model = models.Sequential()

model.add(layers.Conv2D(32, (3, 3), activation='relu',

                        input_shape=(150, 150, 3)))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2, 2)))
```

```
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])
```

Now to train the network by using data augmentation and dropout:

```
train_datagen = ImageDataGenerator(
      rescale=1./255,
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,)
```

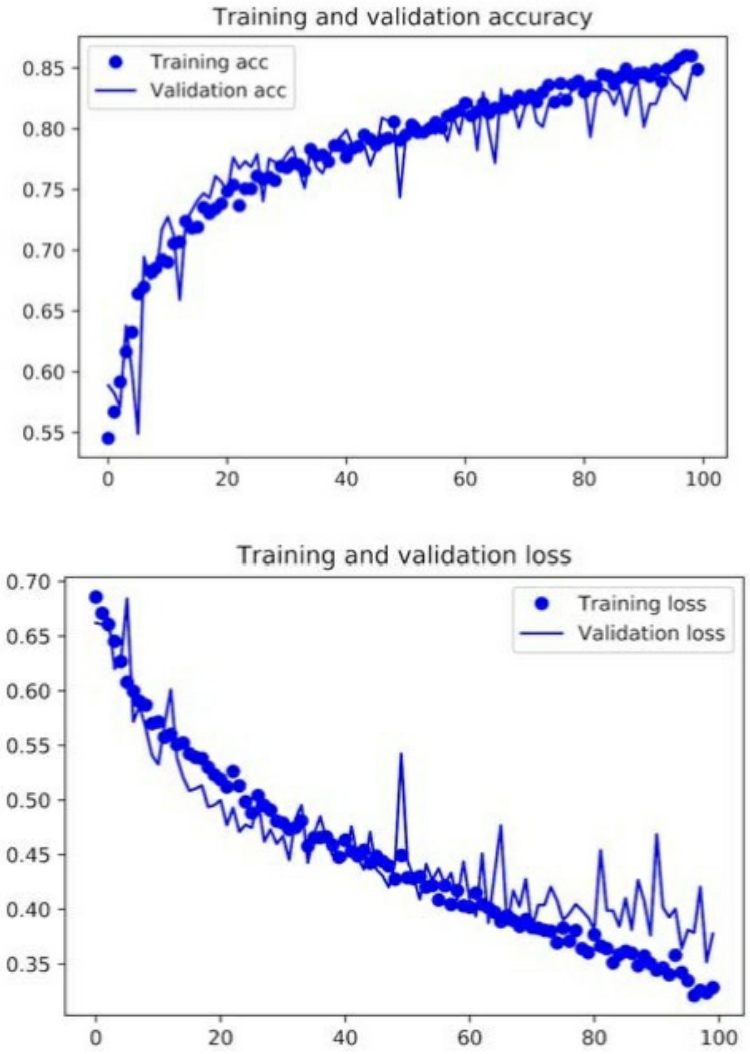
```
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        train_dir,
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

history = model.fit_generator(
        train_generator,
        steps_per_epoch=100,
        epochs=100,
        validation_data=validation_generator,
        validation_steps=50)
```

A very important note, never augment the validation data.

Let’s save the model in case we need it for future purposes:

```
model.save('cats_and_dogs_small_2.h5')
```

The following are the results plotted onto a graph:



We can see that the training curves are closely following the validation curves, and we have eradicated overfitting from our deep learning model by using data augmentation and dropout.

Working with a Pretrained Convnet

This another effective, efficient, and very commonly practiced approach for people using deep learning models in computer vision. This approach is self-definitive. We are using a convnet that has already been trained on a large-scale dataset used primarily for tackling a complex and huge image-classification task, and later, this network has been saved. We can use this network for small datasets with a little change here and there. Moreover, a pre-trained convnet most probably has a sizable spatial hierarchy learned,

meaning that it can perform general classification.

A pre-trained network can be used by following either of the two ways;

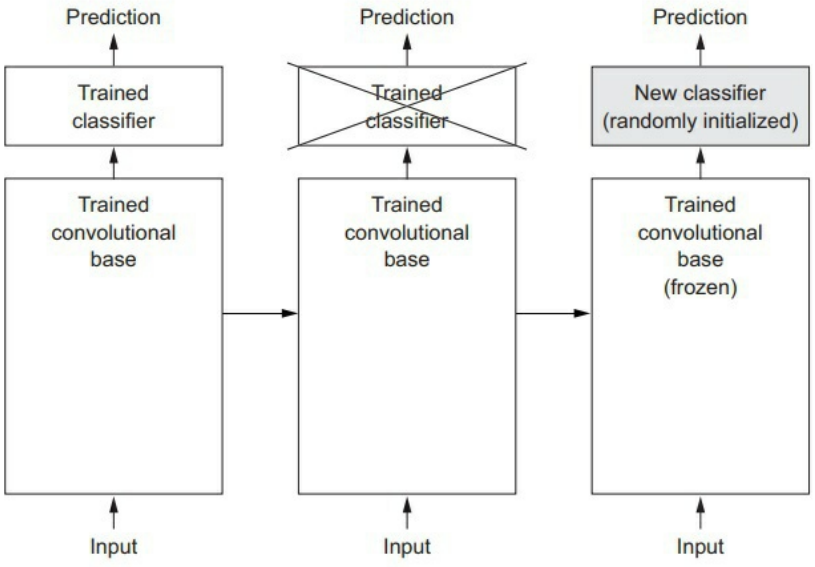
1. Feature Extraction
2. Fine Tuning

Feature Extraction

In feature extraction, we use representations of the pre-trained convnet to extract some new features in the current dataset we are working on. Once the features have been extracted, we then train a new classifier and run these features through it.

To refresh our memory, a typical convnet begins with a bunch of convolution and max-pooling layers and end with a densely connected classifier. In feature extraction, we will use the convolutional base of the pre-trained convnet and run our data through it. Afterward, we will train a new classifier and put it on top of the output yielded by the pre-trained convnets convolutional base.

We are essentially swapping out the pre-trained convnet's classifier with a newly trained classifier while keeping the same convolutional base as shown below;



Let’s being using a pre-trained convnet. There are several pre-trained convnets models available in Keras which are;

1. Xception
2. Inception V3
3. ResNet50
4. VGG16
5. VGG19
6. MobileNet

We will be using a VGG16 convnet, which has been pre-trained on the ImageNet dataset. As per feature extraction, we will be extracting the features of cats and dogs from this convnet's convolutional base. After extracting the features, we will train a dog vs. cat classifier and place it on these extracted features.

We do not need to download the VGG16 model as it is already available for use in Keras. To import it, we will use the module **keras.applications**.

We will now proceed to begin instantiating the convolutional base of VGG16;

```
from keras.applications import VGG16


conv_base = VGG16(weights='imagenet',

                  include_top=False,

                  input_shape=(150, 150, 3))
```

To understand the VGG16's convolutional base better, take a look at a detailed architecture of it;

```
>>> conv_base.summary()

Layer (type) Output Shape Param #

=================================================================

input_1 (InputLayer) (None, 150, 150, 3) 0

_________________________________________________________________

block1_conv1 (Convolution2D) (None, 150, 150, 64) 1792
```

block1_conv2 (Convolution2D) (None, 150, 150, 64) 36928
block1_pool (MaxPooling2D) (None, 75, 75, 64) 0
block2_conv1 (Convolution2D) (None, 75, 75, 128) 73856
block2_conv2 (Convolution2D) (None, 75, 75, 128) 147584
block2_pool (MaxPooling2D) (None, 37, 37, 128) 0
block3_conv1 (Convolution2D) (None, 37, 37, 256) 295168
block3_conv2 (Convolution2D) (None, 37, 37, 256) 590080
block3_conv3 (Convolution2D) (None, 37, 37, 256) 590080
block3_pool (MaxPooling2D) (None, 18, 18, 256) 0
block4_conv1 (Convolution2D) (None, 18, 18, 512) 1180160
block4_conv2 (Convolution2D) (None, 18, 18, 512) 2359808

```
_________________________________________________________________
block4_conv3 (Convolution2D) (None, 18, 18, 512) 2359808
_________________________________________________________________
block4_pool (MaxPooling2D) (None, 9, 9, 512) 0
_________________________________________________________________
block5_conv1 (Convolution2D) (None, 9, 9, 512) 2359808
_________________________________________________________________
block5_conv2 (Convolution2D) (None, 9, 9, 512) 2359808
_________________________________________________________________
block5_conv3 (Convolution2D) (None, 9, 9, 512) 2359808
_________________________________________________________________
block5_pool (MaxPooling2D) (None, 4, 4, 512) 0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

As we can see the final block has a feature map of (4, 4, 512). We will place a densely connected classifier on top of this feature map.

Now we can proceed with feature extraction in two ways;

- Feature extraction without data augmentation
- Feature extraction with data augmentation

Feature Extraction Without Data Augmentation

This method is particularly suitable in cases where you do not have access to

a GPU, or you can only run your code on the CPU. First of all, we will begin by extracting the images along with their labels as NumPy arrays by using the ImageDataGenerator. The features will be extracted by using the **predict** method of the **conv_base** model.

```
import os
import numpy as np
from keras.preprocessing.image import ImageDataGenerator

base_dir = '/Users/fchollet/Downloads/cats_and_dogs_small'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')

datagen = ImageDataGenerator(rescale=1./255)
batch_size = 20

def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 4, 4, 512))
    labels = np.zeros(shape=(sample_count))
    generator = datagen.flow_from_directory(
        directory,
        target_size=(150, 150),
        batch_size=batch_size,
        class_mode='binary')
```

```
    i=0
    for inputs_batch, labels_batch in generator:
        features_batch = conv_base.predict(inputs_batch)
        features[i * batch_size : (i + 1) * batch_size] = features_batch
        labels[i * batch_size : (i + 1) * batch_size] = labels_batch
        i += 1
        if i * batch_size >= sample_count:
            break
        return features, labels
train_features, train_labels = extract_features(train_dir, 2000)
validation_features, validation_labels = extract_features(validation_dir, 1000)
test_features, test_labels = extract_features(test_dir, 1000)
```

Since we are now going to use a densely connected classifier on the extracted features, we will first need to flatten their shape. The shape of the extracted features is (4, 4, 512) and we will flatten this shape into (samples, 8192) as shown below;

```
train_features = np.reshape(train_features, (2000, 4*4* 512))
validation_features = np.reshape(validation_features, (1000, 4*4* 512))
test_features = np.reshape(test_features, (1000, 4*4* 512))
```

Now that we have the required form of input for the densely connected classifier, we can now define it and use the training data and labels we recently recorded for training this classifier;

```
from keras import models
from keras import layers
```

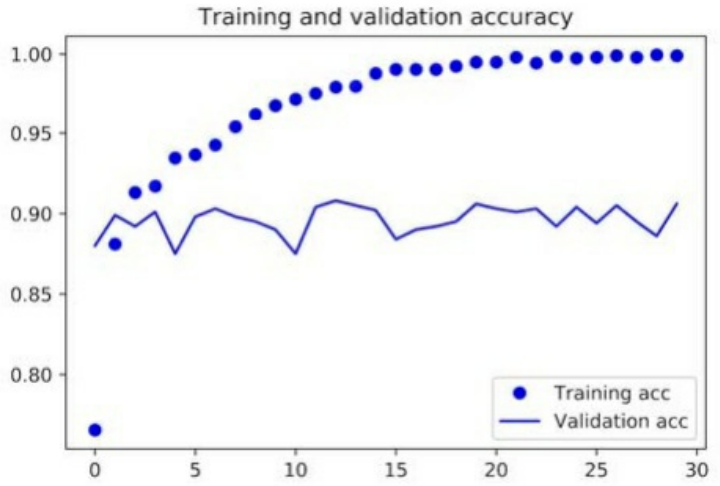
```
from keras import optimizers

model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=4 * 4 * 512))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer=optimizers.RMSprop(lr=2e-5),
              loss='binary_crossentropy',
              metrics=['acc'])

history = model.fit(train_features, train_labels,
                    epochs=30,
                    batch_size=20,
                                        validation_data=(validation_features,
validation_labels))
```

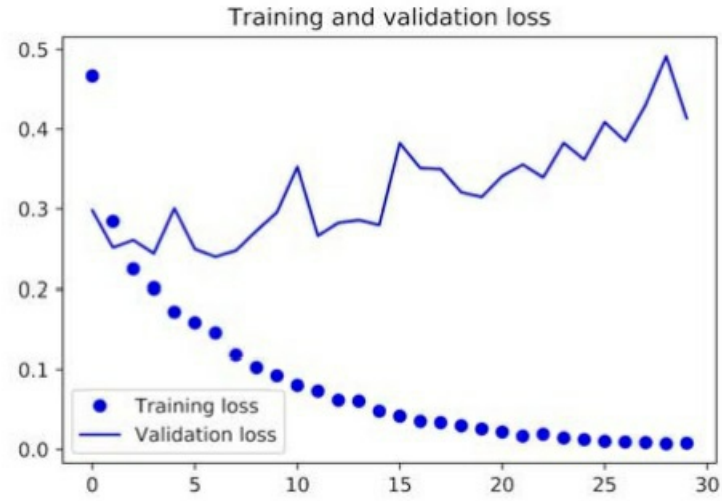
Since we are only using two dense layers, hence, the speed of the training is very speedy such that the time taken by each epoch to complete is lesser than a second even though we are running our code only on the CPU.

Now, we will analyze the loss and accuracy of the model by plotting it graphically;

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
```

```
loss = history.history['loss']
val_loss = history.history['val_loss']

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

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```





By using a pre-trained convnet, we reached a validation accuracy of 90% on our first go without using data augmentation for eradicating the overfitting.

Feature Extraction with Data Augmentation

Apart from the fact that in this method, we will implement data augmentation into the convnet, this method is recommended to be used only if the machine has access to a GPU as it is very resource-intensive. Generally, feature extraction is a slower and expensive process, but the upside to this method is that we can use data augmentation. Feature extraction without data augmentation is faster, and a little less resource-intensive, so bear this in mind when choosing which method to go for.

We will first add the conv_base model to the sequential models just as how we would add layers on top of each other;

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

The architecture summary of the model so far is;

```
>>> model.summary()
Layer (type) Output Shape Param #
=================================================================
vgg16 (Model) (None, 4, 4, 512) 14714688
_________________________________________________________________
flatten_1 (Flatten) (None, 8192) 0
_________________________________________________________________
dense_1 (Dense) (None, 256) 2097408
_________________________________________________________________
dense_2 (Dense) (None, 1) 257
=================================================================
Total params: 16,812,353
Trainable params: 16,812,353
```

```
Non-trainable params: 0
```

From the model summary, we can see that the VGG16 model's convolutional base has over a million parameters and the densely connected classifier we are adding has 2 million parameters. In such a scenario, we can just freeze the convolutional base before proceeding with the training and compilation of the model.

Freezing a network can be done in Keras by simply setting the **trainable** attribute of the network to **false** as shown below;

```
>>> print('This is the number of trainable weights '

'before freezing the conv base:', len(model.trainable_weights))

This is the number of trainable weights before freezing the conv base: 30

>>> conv_base.trainable = False

>>> print('This is the number of trainable weights '

'after freezing the conv base:', len(model.trainable_weights))

This is the number of trainable weights after freezing the conv base: 4
```

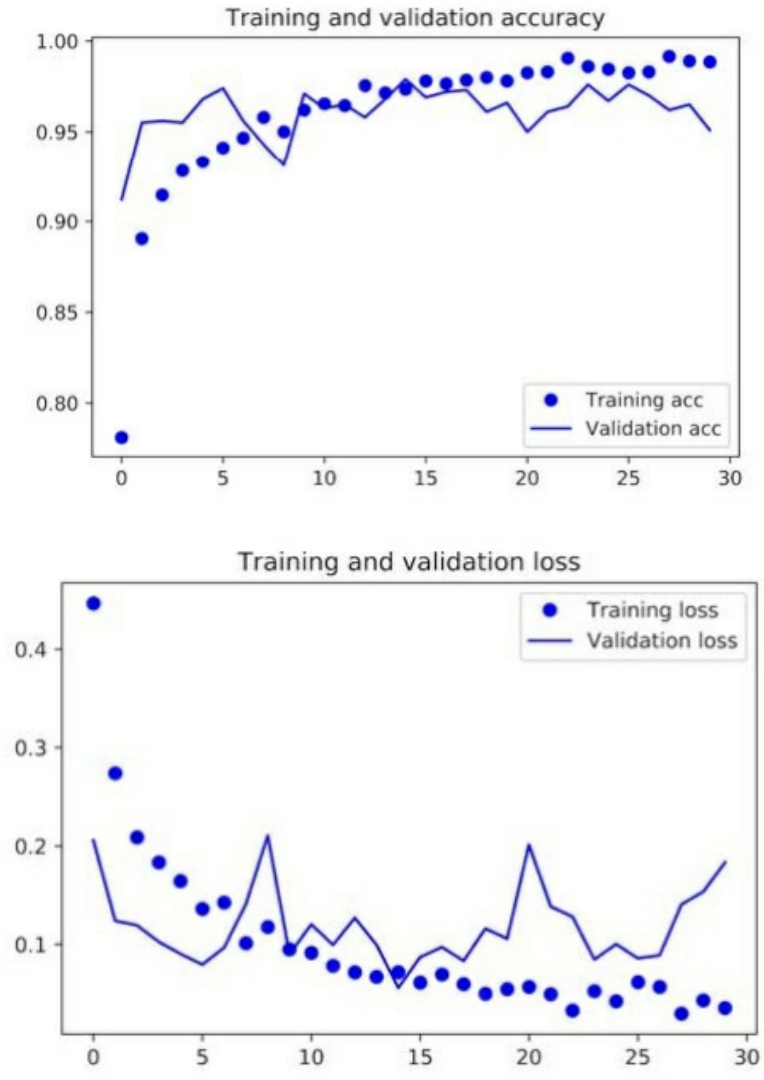
To bring the changes we just made into effect, we first need to compile the model; otherwise, the changes will be ignored. After compiling and applying this change to the model, we will now start training the model with a frozen convolutional base;

```
from keras.preprocessing.image import ImageDataGenerator

from keras import optimizers


train_datagen = ImageDataGenerator(

        rescale=1./255,

        rotation_range=40,

        width_shift_range=0.2,
```

```
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        train_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')

model.compile(loss='binary_crossentropy',
        optimizer=optimizers.RMSprop(lr=2e-5),
        metrics=['acc'])
```

```
history = model.fit_generator(
        train_generator,
        steps_per_epoch=100,
        epochs=30,
        validation_data=validation_generator,
        validation_steps=50)
```

Now let’s see the results after plotting the loss and accuracy of the model;



From the above curves, we can see that the validation accuracy of the model

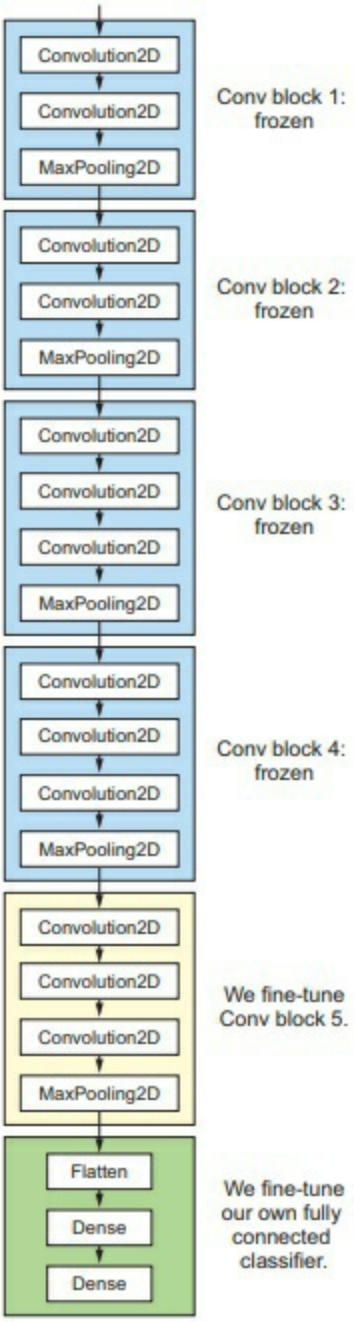
is a whopping 96%. Really big improvement as compared to the convnet we trained on a small dataset.

Fine Tuning

We will try to keep this as brief and to the point as possible because it correlates to the concepts we discussed previously in feature extraction.

A pre-trained convnet can be used by the method of fine-tuning, which is essentially the unfreezing of the top layers which have been frozen when using the feature extraction method. After unfreezing, we train these top layers and the new fully-connected classifier. In essence, we are taking the pre-trained convnet and adjusting its abstract representations ever so slightly to increase the relevance of the representations for the current task.

Here’s a visual representation of how a last convolutional block of the VGG16 convnet is fine-tuned;



An important note to remember in fine-tuning is that it is necessary to train the classifier before adding it to the top layers. Otherwise, during training, there will be a huge error signal propagating through the entire network, eradicating the representations which have been previously learned by the layers that we are fine-tuning. Hence, it is important to follow these steps to fine-tune a pre-trained convnet properly;

1. When taking a pre-trained network, put in your custom network on

top of it.

2. Then proceed to freeze the first network (the base).
3. Proceed with training the custom network which you recently added.
4. Go to the frozen base network and unfreeze some of the layers in there.
5. Finally, train these unfrozen layers and the recently added custom network together.

In feature extraction, we performed the initial 3 steps, in fine-tuning we will have to perform all of these steps. As we already know how to perform the first 3 steps, let's start directly on step 4 i.e, unfreezing the conv_base and freezing some of the individual layers of this model. The summary of our model so far should look like this;

```
>>> conv_base.summary()
Layer (type) Output Shape Param #
=================================================================
input_1 (InputLayer) (None, 150, 150, 3) 0
_________________________________________________________________
block1_conv1 (Convolution2D) (None, 150, 150, 64) 1792
_________________________________________________________________
block1_conv2 (Convolution2D) (None, 150, 150, 64) 36928
_________________________________________________________________
block1_pool (MaxPooling2D) (None, 75, 75, 64) 0
_________________________________________________________________
block2_conv1 (Convolution2D) (None, 75, 75, 128) 73856
_________________________________________________________________
block2_conv2 (Convolution2D) (None, 75, 75, 128) 147584
```

block2_pool (MaxPooling2D) (None, 37, 37, 128) 0

block3_conv1 (Convolution2D) (None, 37, 37, 256) 295168

block3_conv2 (Convolution2D) (None, 37, 37, 256) 590080

block3_conv3 (Convolution2D) (None, 37, 37, 256) 590080

block3_pool (MaxPooling2D) (None, 18, 18, 256) 0

block4_conv1 (Convolution2D) (None, 18, 18, 512) 1180160

block4_conv2 (Convolution2D) (None, 18, 18, 512) 2359808

block4_conv3 (Convolution2D) (None, 18, 18, 512) 2359808

block4_pool (MaxPooling2D) (None, 9, 9, 512) 0

block5_conv1 (Convolution2D) (None, 9, 9, 512) 2359808

```
block5_conv2 (Convolution2D) (None, 9, 9, 512) 2359808
_________________________________________________________________
block5_conv3 (Convolution2D) (None, 9, 9, 512) 2359808
_________________________________________________________________
block5_pool (MaxPooling2D) (None, 4, 4, 512) 0
=================================================================
Total params: 14714688
```

From this model, we will be fine-tuning only the ending three layers and freeze the rest of the layers up till block4_pool will be frozen.

We will now freeze all of the layers until the block5_conv1 layer;

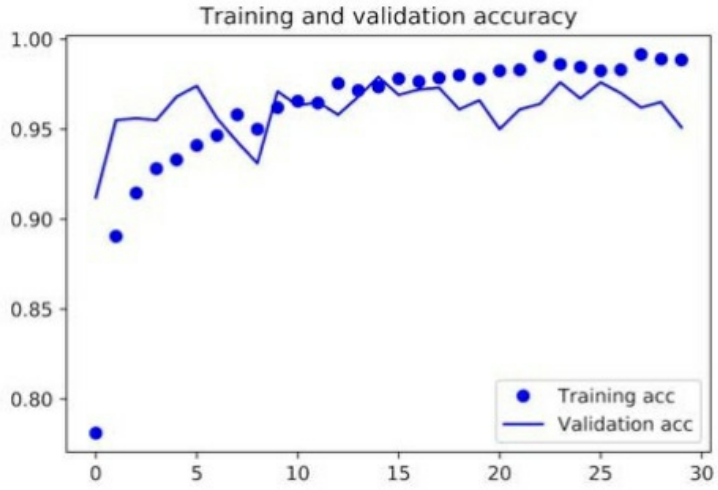
```
conv_base.trainable = True
set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
else:
    layer.trainable = False
```

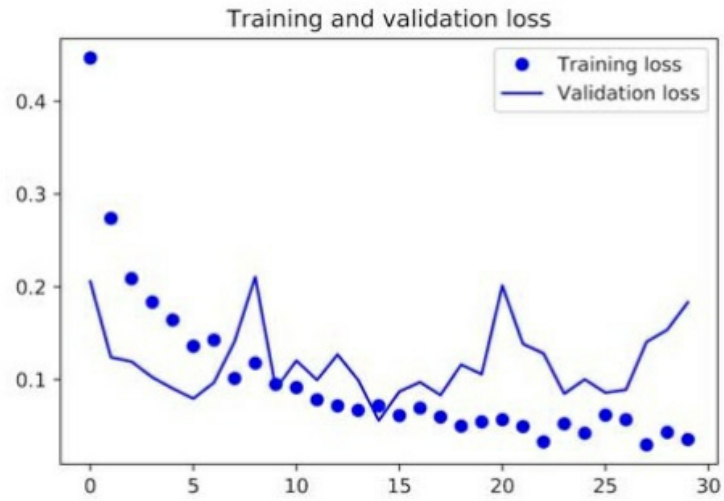
We will now proceed with fine-tuning the deep learning model;

```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-5),
              metrics=['acc'])
```

```
history = model.fit_generator(
        train_generator,
        steps_per_epoch=100,
        epochs=100,
        validation_data=validation_generator,
        validation_steps=50)
```

Let’s check the results by plotting the model’s loss and accuracy



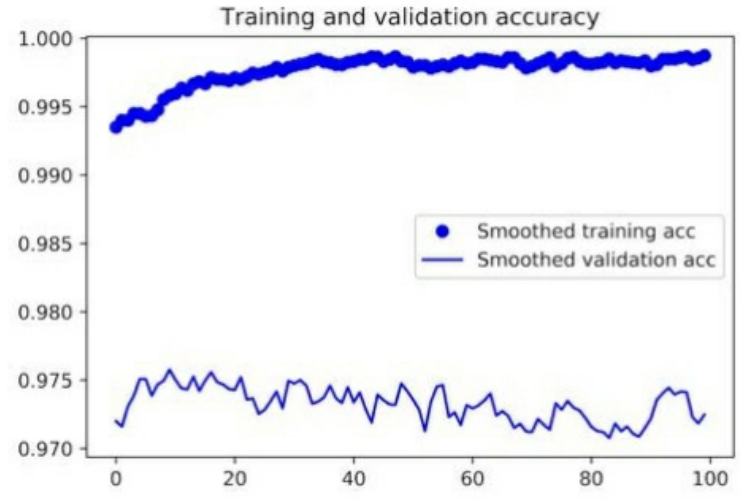


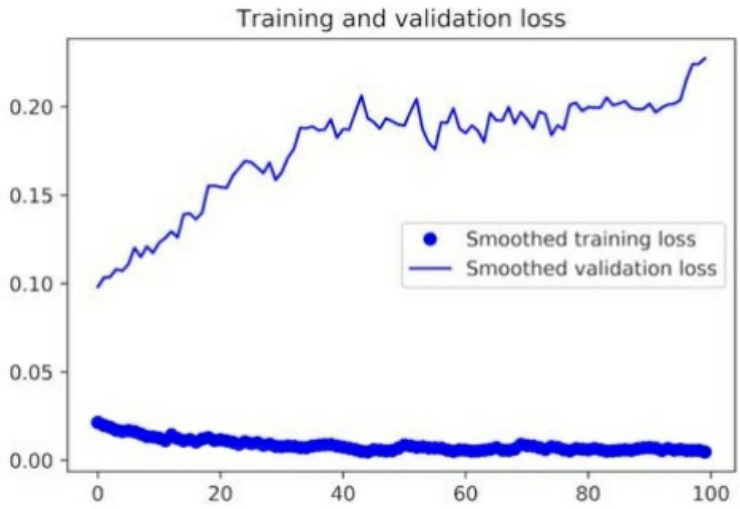
Let’s smoothen out these curves by replacing each instance of loss and

accuracy with the exponential moving averages to make the data more understandable;

```
def smooth_curve(points, factor=0.8):
    smoothed_points = []
    for point in points:
        if smoothed_points:
            previous = smoothed_points[-1]
            smoothed_points.append(previous * factor + point * (1 - factor))
        else:
            smoothed_points.append(point)
    return smoothed_points

plt.plot(epochs,
         smooth_curve(acc), 'bo', label='Smoothed training acc')
plt.plot(epochs,
         smooth_curve(val_acc), 'b', label='Smoothed validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs,
         smooth_curve(loss), 'bo', label='Smoothed training loss')
```

```
plt.plot(epochs,
         smooth_curve(val_loss), 'b', label='Smoothed validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```





The curves look more readable and clean, moreover, the accuracy has increased from 96% to more than 97%

Finally, we can now check the performance of this model on a testing dataset;

```
test_generator = test_datagen.flow_from_directory(
        test_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')

test_loss, test_acc = model.evaluate_generator(test_generator, steps=50)
print('test acc:', test_acc)
```

The results of the evaluation of the model on entirely new data will come in as an accuracy of 97%.

Chapter 5: Mastering Advanced Practices in Deep Learning

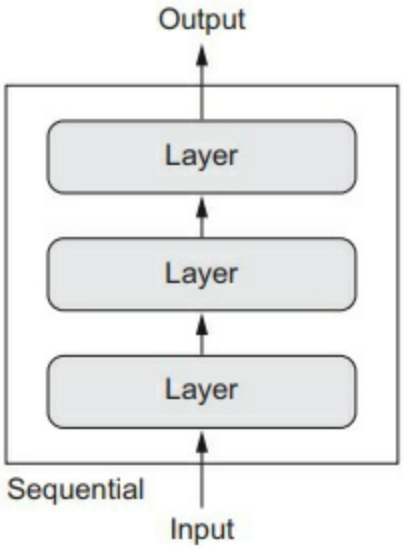
So far, we have discussed the various uses of deep learning in solving real-world problems and demonstrated we could set up a model for solving some of the most common real-world problems. In this chapter, we will take our discussion to the next level and begin exploring some amazing tools with advanced usability. These tools will help us lay the foundation for building some amazing and high caliber deep learning models with which we can deal with some very difficult problems. We will focus our discussion on a variety of these advanced practices such as;

- Batch normalization
- Residual connections
- Hyperparameter optimization
- Model ensembling

Keras Functional API

A functional API is an alternative to using sequential deep learning models. A sequential model assumes that the Neural network of the deep learning model has been architected in such a way that it receives only one input and gives a corresponding single output.

Moreover, it also assumes that the network is made up of layers in the form of a single stack, as shown in the figure below;



Notice that up until this point in the book, all the deep learning models we have seen are sequential models. Although this is assumption is true in most of the cases for deep learning models. However, it is still a fact that this assumption is inflexible when considering difficult problems to solve. For instance, a network can need multiple inputs to perform a task effectively, or it can require multiple outputs too. This case is observed in problems that can only be solved by using multimodal inputs (the data coming for various input sources are merged, and each type of data is then processed accordingly by different types of layers).

Using the Keras Functional API

A functional API allows us to a direct influence on how the tensors are being manipulated. In other words, we are controlling the tensors directly, and the layers serve the purpose of functions. In functional API, a layer is given a tensor input, and a corresponding tensor is given as an output as shown below;

```
from keras import Input, layers

input_tensor = Input(shape=(32,))

dense = layers.Dense(32, activation='relu')

output_tensor = dense(input_tensor)
```

To understand how functional API is different from a Sequential model, let’s make a comparison between how the two networks;

A sequential model;

```
from keras.models import Sequential, Model

from keras import layers

from keras import Input


seq_model = Sequential()

seq_model.add(layers.Dense(32, activation='relu', input_shape=(64,)))

seq_model.add(layers.Dense(32, activation='relu'))

seq_model.add(layers.Dense(10, activation='softmax'))
```

A functional API which is the equivalent of the model detailed above;

```
input_tensor = Input(shape=(64,))

x = layers.Dense(32, activation='relu')(input_tensor)

x = layers.Dense(32, activation='relu')(x)

output_tensor = layers.Dense(10, activation='softmax')(x)

model = Model(input_tensor, output_tensor)

model.summary()
```

By opening the model summary, here’s what we get;

```
_________________________________________________________________

Layer (type) Output Shape Param #

=================================================================

input_1 (InputLayer) (None, 64) 0
```

```
_________________________________________________________________
dense_1 (Dense) (None, 32) 2080
_________________________________________________________________
dense_2 (Dense) (None, 32) 1056
_________________________________________________________________
dense_3 (Dense) (None, 10) 330
=================================================================
Total params: 3,466
Trainable params: 3,466
Non-trainable params: 0
_________________________________________________________________
```

Moreover, the functional API is essentially the same as the sequential model when we talk about the process of compilation, training, and evaluation:

```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

import numpy as np

x_train = np.random.random((1000, 64))

y_train = np.random.random((1000, 10))

model.fit(x_train, y_train, epochs=10, batch_size=128)

score = model.evaluate(x_train, y_train)
```

Multi-Input Models

The main feature of the functional API is its ability to lay a foundation for the network to build into a deep learning model that has multiple input sources. In such models, the input branches are later merged into a single entity, which is essentially combining multiple tensors deeper into the Neural

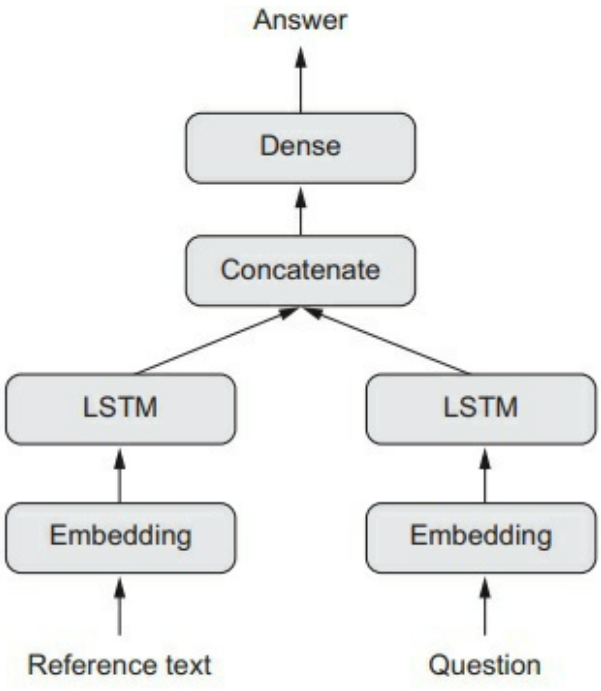
network. This merging is usually done by methods such as addition, concatenation, etc. by calling upon the merge operation in Keras such as;

- keras.layers.add
- keras.layers.concatenate

To understand this concept better, let's talk about an actual multi-input model like the question-answering model. A question answering model has two input sources. These two inputs are;

1. Question
2. Text snippet

These two input sources collectively provide the model with the necessary information so that it can answer the question. The next plan of action for the model is to provide an answer to the question being posed. In a very plain and simple multi-input, the output is given in the form of an answer consisting of only a single word by utilizing a softmax function with a preloaded dictionary. The figure below depicts a question-answering model;



To build a multi-input question answering model by using a Functional API, we will need to define two input sources as independent branches. One

branch will be responsible for encoding the text input, and the other will encode the question input. The data will be encoded into representation vectors, and these vectors will then be concatenated. Afterward, a softmax classifier will be on these representations. The code for doing this is as follows;

```
from keras.models import Model

from keras import layers

from keras import Input

text_vocabulary_size = 10000

question_vocabulary_size = 10000

answer_vocabulary_size = 500

text_input = Input(shape=(None,), dtype='int32', name='text')

embedded_text = layers.Embedding(

64, text_vocabulary_size)(text_input)

encoded_text = layers.LSTM(32)(embedded_text)

question_input = Input(shape=(None,),

dtype='int32',

name='question')

embedded_question = layers.Embedding(

32, question_vocabulary_size)(question_input)

encoded_question = layers.LSTM(16)(embedded_question)

concatenated = layers.concatenate([encoded_text, encoded_question],

axis=-1)
```

```
answer = layers.Dense(answer_vocabulary_size,

activation='softmax')(concatenated)

model = Model([text_input, question_input], answer)

model.compile(optimizer='rmsprop',

loss='categorical_crossentropy',

metrics=['acc'])
```

In regards to training this model, we can do so by either of the two approaches;

1. Feeding the model inputs in the form of NumPy arrays lists.
2. Feeding the model a dictionary that automatically maps the input names to the corresponding NumPy arrays (can only be done if you already named the inputs).

We will now demonstrate both of these approaches;

```
import numpy as np


num_samples = 1000

max_length = 100


text = np.random.randint(1, text_vocabulary_size,

                         size=(num_samples, max_length))

question = np.random.randint(1, question_vocabulary_size,

                             size=(num_samples, max_length))

answers = np.random.randint(0, 1,

                                                 size=(num_samples,
answer_vocabulary_size))
```

```
model.fit([text, question], answers, epochs=10, batch_size=128)

model.fit({'text': text, 'question': question}, answers,

                    epochs=10, batch_size=128)
```

The first model.fit argument shows how you can feed a list of NumPy arrays as input and the last model.fit argument shows how you can feed a dictionary of inputs respectively.

Multi-Output Models

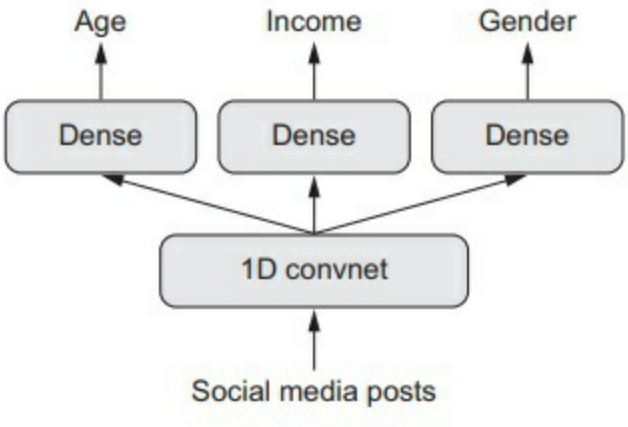
So far, we have talked about multi-input models, such as the question-answer model. Similarly, a functional API can also be used to construct a deep learning model that has multiple outputs, also referred to as heads. For instance, a multi-output model can be a deep learning model that analyzes the social media posts of an unknown person and give multiple predictions regarding the person's age, profession, and gender, etc. These multiple predictions are, in essence, multiple outputs of the model.

Let's see how a multi-output model of a maximum of three outputs can be built by using the functional API;

```
from keras import layers

from keras import Input

from keras.models import Model


vocabulary_size = 50000

num_income_groups = 10


posts_input = Input(shape=(None,), dtype='int32', name='posts')

embedded_posts = layers.Embedding(256, vocabulary_size)(posts_input)

x = layers.Conv1D(128, 5, activation='relu')(embedded_posts)

x = layers.MaxPooling1D(5)(x)
```

```
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dense(128, activation='relu')(x)

age_prediction = layers.Dense(1, name='age')(x)
income_prediction = layers.Dense(num_income_groups,
                                 activation='softmax',
                                 name='income')(x)
gender_prediction = layers.Dense(1, activation='sigmoid', name='gender')(x)

model = Model(posts_input,
              [age_prediction, income_prediction, gender_prediction])
```

A figurative representation of this model is shown below;



In such deep learning models, we have to specify a different loss function for each corresponding output or the head of the network. Take the gender prediction output given by the model, for the Neural network; this is a binary classification task. At the same time, giving an age prediction output is a scalar regression task, and the training process is also entirely different. Hence the nature of the task being performed by the model is different for each head, and that is why each head requires a specific loss function. Moreover, a primary requirement of gradient descent is the minimizing of the scalar. To do this, we will have to merge these different losses into a singular value, and only then can we proceed to train the model.

The most straight-forward and simple approach towards combining the losses is just to sum them up and to do so, Keras gives us the option of using the lists or dictionary of the ‘compile’ function so that we can specify the multiple outputs to the corresponding object. Afterward, we take the loss values from the outputs and sum them up into one global loss value. This can be then minimized, and the model can be trained.

The two options through which we can compile the losses of the multi-output model are as follows;

```
model.compile(optimizer='rmsprop',
                                    loss=['mse',       'categorical_crossentropy',
'binary_crossentropy'])
model.compile(optimizer='rmsprop',
                    loss={'age': 'mse',
```

```
                    'income': 'categorical_crossentropy',
                    'gender': 'binary_crossentropy'})
```

The latter is only possible if you have tagged the output layers with specific names.

A very important thing to note regarding losses in a multi-output model is that if there is an imbalance loss contribution, the representations of the deep learning model will inherently be optimized for the task that has the current biggest individual loss value. This optimization comes at the expense of other tasks. We can avoid this by taking the multiple loss values and assigning each value with a level of importance, which defines its contribution to the global loss value.

The following lines of code show the loss weighting option in the compilation of a multi-output model;

```
model.compile(optimizer='rmsprop',
                    loss=['mse', 'categorical_crossentropy', 'binary_crossentropy'],
                    loss_weights=[0.25, 1., 10.])

model.compile(optimizer='rmsprop',
                    loss={'age': 'mse',
                          'income': 'categorical_crossentropy',
                          'gender': 'binary_crossentropy'},
                    loss_weights={'age': 0.25,
                                  'income': 1.,
                                  'gender': 10.})
```

To train the model, we can use the same two approaches defined for multi-input models, i.e; using a list of NumPy arrays or using a dictionary of

NumPy arrays;

```
model.fit(posts, [age_targets, income_targets, gender_targets],
          epochs=10, batch_size=64)

model.fit(posts, {'age': age_targets,
                  'income': income_targets,
                  'gender': gender_targets},
          epochs=10, batch_size=64)
```

The latter is only possible if you have tagged the output layers with specific names.

Directed Acyclic Graphs of Layers

Apart from being used as the gateway for building models that have multiple inputs and multiple outputs, the functional API is also capable of allowing the user to implement neural networks that feature a complex internal topology. This makes the full use of Keras's ability to support arbitrary neural networks. Moreover, the conceptual foundation of such a network is "acyclic," meaning that no tensor will become the input of the layer that generated it.

The two notable neural network components which are implemented as graphs are;

1. Inception Modules
2. Residual connections

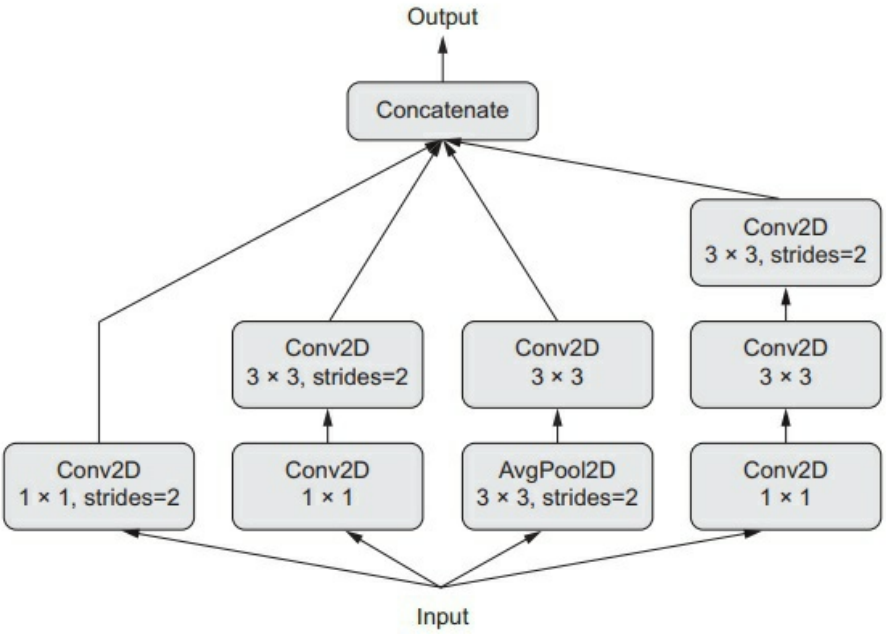
Understanding these two components is key to learning how we can implement a functional API to build a graph of layers.

Inception Modules

Inception is actually a network architecture which is chiefly used in convolutional neural networks. The architecture is made up of a stack of modules that break into several parallel branches. A basic setup of an inception module is as follows;

- Three or four branches, beginning with 1x1 convolution.
- Following up with a 3x3 convolution.
- Finishing with the result, which is an overall concatenation of the results accumulated from the convolutions.

The setup described enables the neural network to learn the two features; spatial and channel-wise features in a separate manner; this is way more efficient for the network rather than learning these features jointly. An inception module can be set up to be even more complex by adding in some pooling operations, making the sizes of the convolutions different, and using branches that do not have a spatial convolution. The figure below shows a module that has been taken from the Inception V3 model;



To implement this module, we will use a functional API and assume that there is an input tensor ‘x,’ which is a 4D tensor. The following lines of code demonstrate this assumption;

```
from keras import layers


branch_a = layers.Conv2D(128, 1,

                         activation='relu', strides=2)(x)

branch_b = layers.Conv2D(128, 1, activation='relu')(x)
```

```
branch_b = layers.Conv2D(128, 3, activation='relu', strides=2)(branch_b)

branch_c = layers.AveragePooling2D(3, strides=2)(x)
branch_c = layers.Conv2D(128, 3, activation='relu')(branch_c)

branch_d = layers.Conv2D(128, 1, activation='relu')(x)
branch_d = layers.Conv2D(128, 3, activation='relu')(branch_d)
branch_d = layers.Conv2D(128, 3, activation='relu', strides=2)(branch_d)

output = layers.concatenate(
        [branch_a, branch_b, branch_c, branch_d], axis=-1)
```

If you want to analyze this module even further, you can access the full architecture of the Inception V3 model in Keras by using the argument;

keras.applications.inception_v3.InceptionV3

This architecture includes pre-trained weights.

Residual Connections

Residual connections are basically one of the common components found in a graph-like network; for instance, in deep learning models such as Xception, residual connections can be found in the architecture of the network. This component is an effective solution to the most observed problems that are found in most large-scale deep learning models;

1. Vanishing gradients
2. Representational bottlenecks

In any model that features more than ten layers in its network architecture, residual connections can benefit the model in one way or the other.

Residual connections essentially function to give the layers easy access to the output of a layer preceding it. This output is taken as input by this layer easily because of residual connections. In other words, a residual connection creates

shortcuts between the layers in a sequential model.

Let’s consider a network which has same sized feature maps and implement a residual connection. For its implementation, we will be using identity residual connections. Note that this demonstration has an assumption that there is a 4D input tensor ‘x’;

```
from keras import layers

x = ...

y = layers.Conv2D(128, 3, activation='relu', padding='same')(x)

y = layers.Conv2D(128, 3, activation='relu', padding='same')(y)

y = layers.Conv2D(128, 3, activation='relu', padding='same')(y)

y = layers.add([y, x])
```

The above lines of code is for using residual connections with feature maps that are of the same size. A residual connection can also be implemented in a network when the feature maps are of different sizes, for such a case we simply use linear residual connection instead of identity;

```
from keras import layers

x = ...

y = layers.Conv2D(128, 3, activation='relu', padding='same')(x)

y = layers.Conv2D(128, 3, activation='relu', padding='same')(y)

y = layers.MaxPooling2D(2, strides=2)(y)

residual = layers.Conv2D(128, 1, strides=2, padding='same')(x)

y = layers.add([y, residual])
```

Layer Weight Sharing

Another one of functional API's notable features is layer weight sharing. This basically refers to the API's ability to reusing a certain layer many times. This is evident when we proceed to call on a layer two times. Normally, we would instantiate the layer on each call; however, in this case, we call on them without instantiating, and this results in the usage of the same weights on each call. By doing this, we can construct deep learning models that shared branches.

Let's make this concept even more clear by discussing an example. Let's say we have a model that is given a task to identify the similarity between two sentences in terms of semantics. In such a scenario, the model is dealing with two inputs, and the output is given either 0 or 1, meaning unrelated sentences and identical or reformulated sentences, respectively. In such a setup, the two inputs we are dealing with are interchangeable because the semantic relationship of sentences is commutative. Hence we do not need to build two separate models for dealing with the processing of each of these two inputs. In such a case, we would use a single layer to process both of these inputs, and this layer would be the LTSM layer. The representations of the LTSM layers are learned while taking both of the inputs into consideration. This is also known as the "Siamese LTSM model" or a "Shared LTSL model."

The following lines of code detail how you can implement such a model using functional API;

```
from keras import layers

from keras import Input

from keras.models import Model


lstm = layers.LSTM(32)


left_input = Input(shape=(None, 128))

left_output = lstm(left_input)


right_input = Input(shape=(None, 128))
```

```
right_output = lstm(right_input)


merged = layers.concatenate([left_output, right_output], axis=-1)

predictions = layers.Dense(1, activation='sigmoid')(merged)


model = Model([left_input, right_input], predictions)

model.fit([left_data, right_data], targets)
```

Using Models as Layers

Another ability of the functional API is allowing models to be effectively used as layers, as is in the case for sequential and model classes. Consequently, we can use an input tensor and call on a model, in turn, receiving an output tensor. This can be done by using the following argument;

```
y = model(x)
```

If the model we are using as a layer itself has several inputs and outputs tensors, then the method to call such a model should be through a list of tensors as shown below;

```
y1, y2 = model([x1, x2])
```

Similar to when we call an instance of a layer, calling upon an instance of a model uses the same weights the model has been trained upon. In other words, no matter if we call upon a layer or a model, the representations will remain the same and can be reused.

An example of what we can do by reusing a model is building a ‘vision’ model. This model has two inputs by using a dual camera as its source. To process the data coming from the two cameras, we don’t need to build two separate models and then merge them later on. A simple stream of data like this can be easily handled by using a low-tier processing technique, such as using layers that have them weights and representations. To implement a Siamese vision model, follow the lines of code shown below;

```
from keras import layers
```

```
from keras import applications

from keras import Input


xception_base = applications.Xception(weights=None,

                                      include_top=False)


left_input = Input(shape=(250, 250, 3))

right_input = Input(shape=(250, 250, 3))


left_features = xception_base(left_input)

right_input = xception_base(right_input)


merged_features = layers.concatenate(

        [left_features, right_input], axis=-1)
```

Inspection of Deep Learning Models Using Keras Callbacks and Tensorboards

This section will primarily focus on the ways through which we can better control the processes within our deep learning model and access its components more easily. In other words, we will explore the ways that will help us manipulate deep learning models more effectively. The reason why this is important is because when training deep learning models on large datasets with many epochs, we mainly use the model.fit() and model.fit_generator() arguments to control it. However, beyond the initial impulse, we do not have any control over the model, and hence, it is impossible always to avoid bad outcomes. The techniques detailed in this section will turn this model.fit() argument from a passively uncontrollable mechanism to a useful and autonomously smart enough argument that can deter bad outcomes

Using Callbacks to Act on an In-Training Model

Training a model is never a predictable process. We do not know how many epochs are needed for an optimal validation loss beforehand; we can only come to know after some trial and error. So far, we have practiced the approach of training the model just before it beings overfitting by plotting the validation and loss data to determine the number of epochs required to do so. This approach is inefficient and takes up a lot of resources.

So, an alternative to this approach is that instead of completing the entire training process to find out where the overfitting begins, we can just stop the training at the point where the validation loss values no longer show any improvement, saving us a lot of time and effort. To do this, we will need to use the Keras callback. Callbacks are basically objects which are given to the fit argument. The deep learning model then calls upon this object at different points during the training process. The features which make a callback so useful are;

- They have access to data which details the information regarding the model's current state and performance
- It has the authorization of acting according to the situation, such as stop the training, save the model's current state, bring in some other set of weights for the model to use, or even change the current state of the deep learning model.

Keeping these features of a callback in mind, we can use it for the following purposes;

1. **As Checkpoints**, as a callback, has the capability of saving a model's current state by simply saving its current weight set, it can make frequent saves giving us a checkpoint to revert the model if something goes wrong.
2. **Premature Interruption**; a callback can step in and stop the training process of the model. This can be used for stopping the training of the model as soon as the improvement of the validation loss becomes stagnant.
3. **Dynamic Adjustment**; a callback can change the values of certain parameters during the training process to make necessary adjustments, for instance, adjusting the learning rate of the optimizer during the network's training.

4. Remember the Keras bar? This is a practical implementation of a callback as it can log data and visualize representations.

Here's a list of callbacks that are included in the **keras.callback** module;

```
keras.callbacks.ModelCheckpoint

keras.callbacks.EarlyStopping

keras.callbacks.LearningRateScheduler

keras.callbacks.ReduceLROnPlateau

keras.callbacks.CSVLogger
```

We will just explain only a select few of these different callbacks.

The ModelCheckpoint and EarlyStopping Callbacks

The primary purpose of the EarlyStopping callback is to make the approach of achieving optimal validation loss, viable. In essence, we define a metric for the callback to monitor during the training. As soon as it detects that the metric value has reached its optimal point and can no longer improve, then it immediately interrupts the training. The EarlyStopping callback is commonly used in pairs with the ModelCheckpoint callback, the latter basically creating saved model checkpoints.

To understand these two callbacks better, let's see how they are implemented in code;

```
import keras

callbacks_list = [

        keras.callbacks.EarlyStopping(

                monitor='acc',

                patience=1,

        ),
```

```
        keras.callbacks.ModelCheckpoint(
                filepath='my_model.h5',
                monitor='val_loss',
                save_best_only=True,
        )
]

model.compile(optimizer='rmsprop',
                loss='binary_crossentropy',
                metrics=['acc'])
model.fit(x, y,
                epochs=10,
                batch_size=32,
                callbacks=callbacks_list,
                validation_data=(x_val, y_val))
```

The ReduceLROnPlateau Callback

A common use of this callback is to reduce the model's rate of learning when it detects that the metric being monitored, in this case, the validation loss, has stagnated in terms of improvement. This sets up for a very effective strategy of escaping the in-training local minima as we can control the learning rate in a loss plateau. Here's an example of how we can use this callback;

```
callbacks_list = [
        keras.callbacks.ReduceLROnPlateau(
                monitor='val_loss'
```

```
                factor=0.1,
                patience=10,
        )
]
model.fit(x, y,
                epochs=10,
                batch_size=32,
                callbacks=callbacks_list,
                    validation_data=(x_val, y_val))
```

Writing a Custom Callback

If you're looking for a callback that can do a specific function but can't find one in the Keras' list of callbacks, then you can just create a custom callback specifically for your purpose. A callback is basically implemented in a network by simply sub-classing the keras.callbacks.Callback class. A callback can be implemented at any point in the training by using these prenamed methods;

- on_epoch_begin; gets called on at the start of every iterating epoch
- on_epoch_end; gets called on at the end of every iterating epoch
- on_batch_begin; gets called on just before each batch's processing
- on_batch_end; gets called on as soon as the processing of a batch is finished
- on_train_beign; gets called on at the start of the training
- on_training_end; gets called on at the end of the training

All of these methods, which are listed above, are called along with a **logs** argument (a dictionary that has data about the preceding batch, epoch, or training iteration).

Moreover, the two attributes listed below are easily accessible by a callback;

1. self.model; the instance of the deep learning model from where the

callback is being called upon.
2. self.validation_data; the value of the validation data, which was passed onto the fit argument.

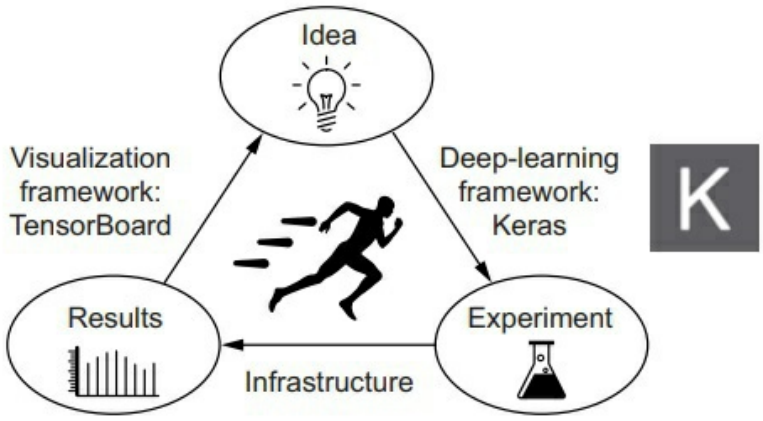
Let’s look at the following example of a custom-made callback. This callback takes the activation values of the layers at the end of every epoch and saves them on a disk as NumPy arrays;

```
import keras

import numpy as np


class ActivationLogger(keras.callbacks.Callback):


        def set_model(self, model):

                self.model = model

                layer_outputs = [layer.output for layer in model.layers]

                self.activations_model = keras.models.Model(model.input,

                                                                layer_outputs)


def on_epoch_end(self, epoch, logs=None):

        if self.validation_data is None:

                raise RuntimeError('Requires validation_data.')

validation_sample = self.validation_data[0][0:1]

activations = self.activations_model.predict(validation_sample)

f = open('activations_at_epoch_' + str(epoch) + '.npz', 'w')

np.savez(f, activations)

f.close()
```

With regards to conceptual knowledge to use callbacks, up until now we have covered it, the remaining details are basically technical details and you can look them up easily.

Tensorboard: The TensorFlow Visualization Network

The key element in building an effective deep learning model for any experiment is to be aware of the internal situation of the model's current state. Moreover, this is the exact purpose of experiments, which is to find out how effective is the deep learning model at handling the information and performing the required tasks. Similarly, a model's improvement is an iterative process, not a defined process that you architect it one time, and it turns out to be either good or bad. Instead, you come up with an idea; you build a suitable experiment to test that idea. You perform this experiment and check the resulting information you obtain, and from this, you get another idea and repeat the process. In this way, you iterate this entire process and refine your deep learning model with even more powerful and effective ideas being the foundation of the model. The reason we discussed this topic is because of the role the Tensorboard plays here; it basically fulfills the job of processing the experimental results in this iterative process.



A TensorBoard is a visualization tool that comes prepackaged in the Keras framework, and this tool is browser-based. However, this tool is only accessible if the deep learning model is using the Keras framework along with a TensorFlow backend engine. Tensorboards are mainly used for displaying what is actually happening in the model during the training session.

Furthermore, using the Tensorboard to monitor several other metrics apart from the validation loss will provide a better insight into how your model is working and a better understanding of how it can be improved. Tensorboard gives convenient access right on the browser to several features such as;

1. Displaying visualizations of the in-training metrics to monitor them.
2. Displaying the makeup of the deep learning model.
3. Displaying both the activation and gradient histograms.
4. 3D surveying the embeddings.

Lets put these features to use in a 1D convnet being trained on the IMDB sentiment analysis task.

The makeup of the deep learning model will be primarily to make the word embeddings visualizations more tractable. ;

```
import keras

from keras import layers

from keras.datasets import imdb

from keras.preprocessing import sequence


max_features = 2000

max_len = 500


(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)

x_train = sequence.pad_sequences(x_train, maxlen=max_len)

x_test = sequence.pad_sequences(x_test, maxlen=max_len)


model = keras.models.Sequential()

model.add(layers.Embedding(max_features, 128,
```

```
                                    input_length=max_len,
                                    name='embed'))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))
model.summary()
model.compile(optimizer='rmsprop',
                    loss='binary_crossentropy',
                     metrics=['acc'])
```

Before we can start using the Tensorboard, we first need to define a directory for it to store the log files generated by the Tensorboard;

```
$ mkdir my_log_dir
```

We will now begin the training for the model by using the **Tensorboard** callback. The purpose of this callback is to take the log event files generated and save them at the specified directory on the disk.

```
callbacks = [
        keras.callbacks.TensorBoard(
                log_dir='my_log_dir',
                histogram_freq=1,
                embeddings_freq=1,
        )
```

```
]

history = model.fit(x_train, y_train,
                    epochs=20,
                    batch_size=128,
                    validation_split=0.2,
                    callbacks=callbacks)
```

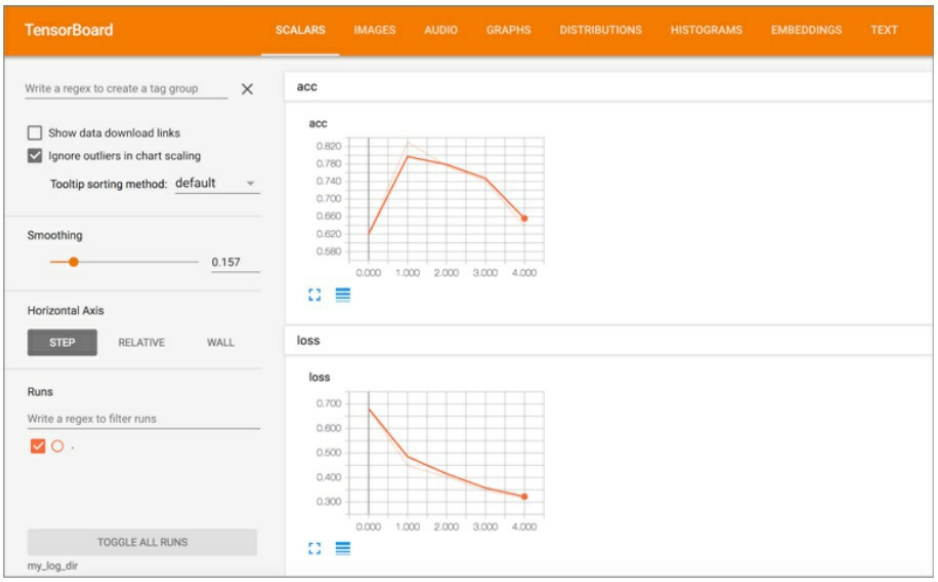
If you have installed the TensorFlow backend engine, then the **tensorboard** utility has also been automatically installed on your system. We can now proceed to open the Tensorboard utility by using the command line.

```
$ tensorboard --logdir=my_log_dir
```

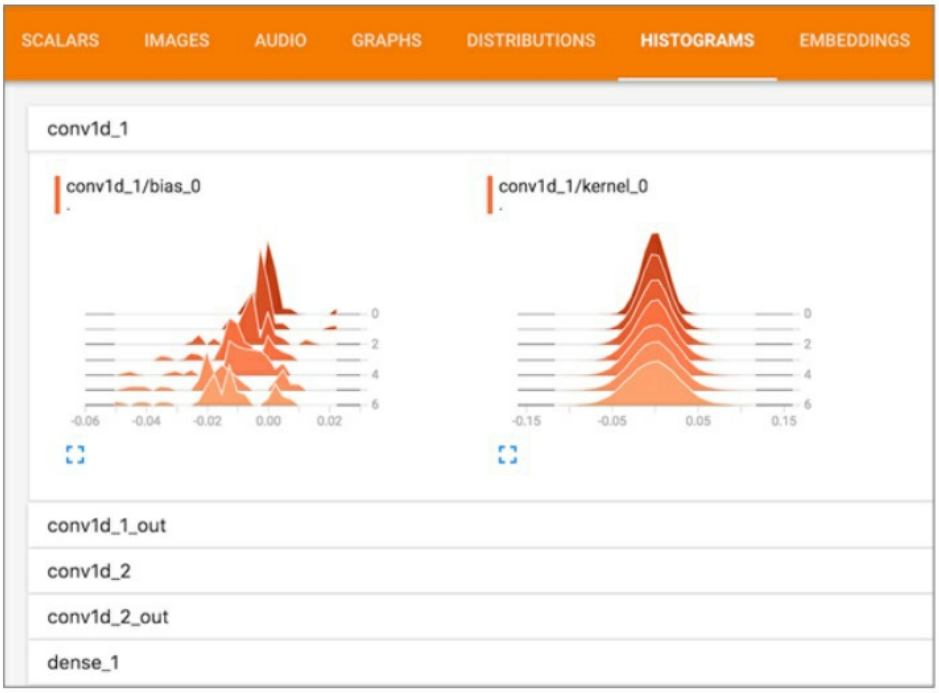
To see the visualized training session of the deep learning model, open the browser and enter the address;

http://localhost:6006

You can see all kinds of useful metric visualizations and other stuff as shown in the figures below;



(Tensorboard Metric Monitoring)



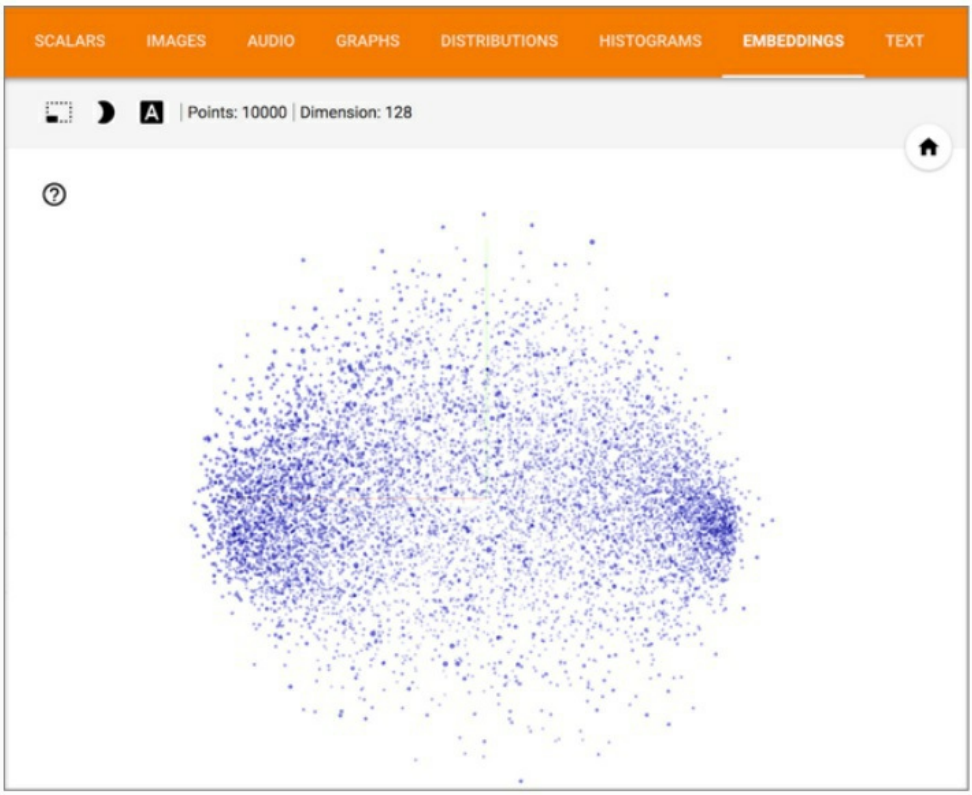
(Tensorboard Visualizing the Activation of Histograms)

If we go the embeddings tab in the Tensorboard, we can easily analyze the attributes of the ten thousand word vocabulary input of our model such as;

- Embedding locations
- Spatial relationships

One more point to note is that originally, the embedding space is of a higher dimension, i.e., 128-dimensional. To visualize and display it, the Tensorboard reduces the dimensional size down to either 2D or 3D. This is done by using algorithms such as the ‘dimensionality-reduction.’

Moreover, you can choose which dimensionality reduction algorithm you want to use as the Tensorboard offers two choices in this regard, namely the PCA (Principal Component Analysis) or the t-SNE (t-distributed Stochastic Neighbor Encoding). In the figure below, we can see a visualization of the embedding space of the words with positive and negative connotations.



Just as how we accessed to useful information by going to the embedding tab in the Tesnorboard, you can explore the other tabs and see what kind of information they have to offer.

Working with Advanced Methods and Getting Optimized Results

Most of the time, programmers tend to try out various deep learning models without using any proper techniques or methods just to find something that simply works. In this section, we will explore the advanced methods, which are essentially the building blocks or the foundation of making amazing and effective deep learning models.

The Advanced Architecture Patterns

In the preceding sections, we explored a very important and effective network architecture for building competitive deep learning models, and this design pattern is the ‘residual connections.’ Apart from this pattern, we will discuss two more architecture patterns, namely;

1. Normalization

2. Depthwise separable convolution

Although these architecture patterns are also commonly found in good deep learning models, they basically set up the foundation for a flagship tier deep learning convnets.

Batch Normalization

Normalization does not refer to one specific pattern or method. Instead, it covers a broad range of methods. However, the goal of these methods is essentially the same, i.e., to normalize the various samples being inputted into the deep learning model. In other words, it takes different samples and converts them into similar samples for the model to train on. In this way, the model copes well when dealing with new data and generalizing predictions effectively and accordingly.

Out of this broad category of methods, the most common normalization method we have seen being used not only in the examples demonstrated in this book but also in some exemplary deep learning models as well and this normalization method is the one where we consider a data sample and take out the mean value from the data, hence centering it on 0. Afterward, we equip this data with a 'unit standard deviation,' and this is obtained by simply taking the standard deviation and dividing the data on it. The result is an assumption that considers the data to now be following a gaussian distribution (a normal distribution) while being centered and scaled to unit variance.

```
normalized_data = (data - np.mean(data, axis=...)) / np.std(data, axis=...)
```

Previously, we saw that the examples using normalization would only feed the data to the network only after it had been normalized. However, for it to be more effective, data normalization should be done after every transformation that is functioned by the neural network.

Batch normalization is essentially a type of layer in Keras (**BatchNormalization**) that can adapt to the shifting mean and variance attributes during the training session and manage to normalize the data even then. Its working is basically dependant on being the maintainer of a steady exponential moving average of the two internal metrics; the mean (according to each batch) and the data being learned during training's variance. The primary goal of a batch normalizer is similar to the residual connections in

the sense that batch normalization tends to facilitate the gradient propagation, making it possible to build even deeper neural networks for a model. Similarly, certain phenomenally deep neural networks can only go through training if there are several **BatchNormalization** layers present. Otherwise, it cannot be trained. We also see the use of this batch normalization layers in the architectures of the popular advanced convnets such as ResNet50, Inception V3, and Xception.

Usually, a **BatchNormalization** layer is implemented in such a way that it either succeeds a convolutional layer or a densely connected layer as shown below;

```
conv_model.add(layers.Conv2D(32, 3, activation='relu'))

conv_model.add(layers.BatchNormalization())


dense_model.add(layers.Dense(32, activation='relu'))

dense_model.add(layers.BatchNormalization())
```

From the lines of code, you can see that the first one shows a batch normalization layer coming after a convolutional layer and the second shows it coming after a desenly connected layer.

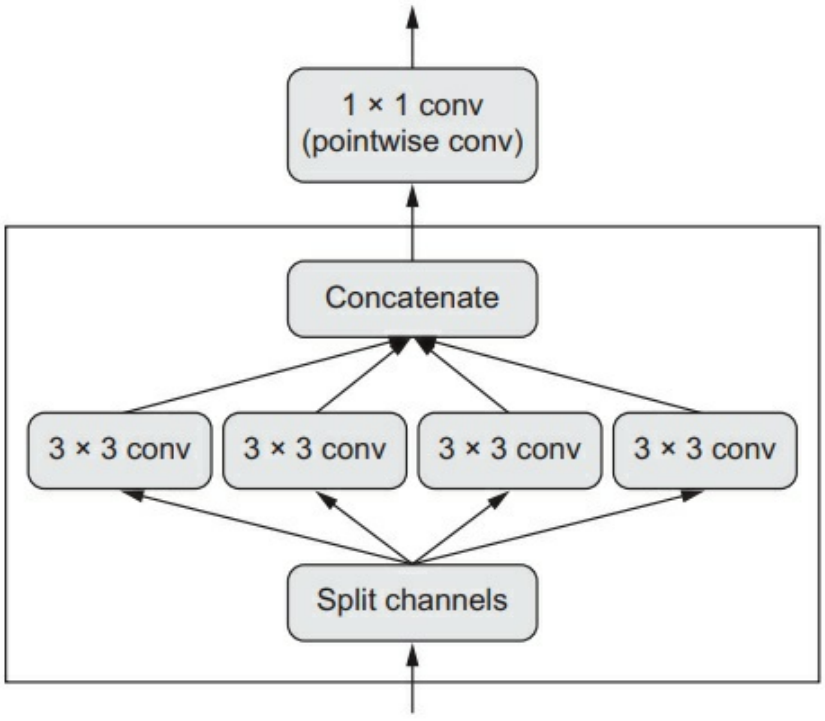
Furthermore, this normalization layer identifies and specifies the feature axis to be normalized by using the **axis** argument with a default value set to -1 (this value refers to the very last layer in the input tensor). This specific value is accurate for using after layers after the following layers; ‘Dense’ layers, ‘Conv1D’ layers, ‘RNN’ layers, and ‘Conv2D’ layers (with a pre-requisite that the argument data_format is specified to “channels_last”). However, when using this normalization layer in niche cases, the axis argument’s value is set to 1 instead of -1. In this case, the data_format argument of the Conv2D layers should be inverted, i.e., set to “channels_first.”

Depthwise Separable Convolution

In topic, we will explore a unique layer that, when added to a network by replacing the convolutional layer Conv2D, can not only improve the speed of the network in which it is being used but also make it several degrees lighter. The network becomes faster because are now lesser floating-point operations,

and it becomes lighter as there now is a smaller set of trainable weight parameters, making it perform better by several percentages on specific tasks. A layer with such properties is none other than the depthwise separable convolution layer, also known as **SeperableConv2D.**

The way this layer operates is that it takes each independent channel of the input and executes spatial convolution on each of the corresponding channels preceding the use of pointwise convolution to mix the output channels (essentially a 1x1 convolution). This process is the alternative equivalent of segregating two features - the spatial and channel-wise features. This step is sensible under the assumption that although the spatial locations are intricately correlated in the input, the varying channels remain independent. This ultimately results in a lighter requirement for the network to use fewer representations and learn better, perform convolutions and ultimately, resulting in a high-performance deep learning model by hardly requiring any parameters and even involving a lesser number of computations, making up a model that is small yet speedy.



(A Depthwise Convolution Followed by a Pointwise Convolution)

The usability, effectiveness, and importance of this convolution layer become

evident when we are working with small models and training on them from the ground up on a limited amount of data. To understand depthwise separable convolutional layers even better, let's see a demonstration of how a lightweight deep learning model can be built by using a depthwise separable convnet and train it for the task of image-classification (in essence, softmax categorical classification) on a limited dataset;

```
from keras.models import Sequential, Model

from keras import layers


height = 64

width = 64

channels = 3

num_classes = 10


model = Sequential()

model.add(layers.SeparableConv2D(32, 3,

                                 activation='relu',

                                 input_shape=(height, width, channels,)))


model.add(layers.SeparableConv2D(64, 3, activation='relu'))

model.add(layers.MaxPooling2D(2))


model.add(layers.SeparableConv2D(64, 3, activation='relu'))

model.add(layers.SeparableConv2D(128, 3, activation='relu'))

model.add(layers.MaxPooling2D(2))
```

```
model.add(layers.SeparableConv2D(64, 3, activation='relu'))

model.add(layers.SeparableConv2D(128, 3, activation='relu'))

model.add(layers.GlobalAveragePooling2D())


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

model.add(layers.Dense(num_classes, activation='softmax'))


model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
```

Depthwise separable convolutional layers are not exclusive to being only used in small deep learning models. On the contrary, the depthwise convolutions are also the building blocks network architectures for large scale deep learning models such as the Xception model, a high-speed convnet that can be accessed through the Keras framework as it comes pre-packaged in it.

Hyperparameter Optimization

The process of building a deep learning model and architecting a neural network usually involves arbitrary decisions and guess-work. For example, you might ask yourself how many stack layers does the model need, what's the optimum number of units and filters for each layer I am using? You might find yourself choosing between a relu activation or some other function for the best result, the decision of either using a **BatchNormalization** layer or not or even the number of dropouts you should use and this list continues. All of these parameters, you are pondering on come under the category of "Hyperparameters." The reason why they are termed as such is for avoiding confusion between these architecture-level parameters and a model's parameters that are trained using backpropagation.

Architecting a good neural network can be done by intuition, and such intuition can only be developed by repeated practice and experience. This leads to the development of skills for hyperparameter tuning.

However, there are no set rules for doing this. If we want to push our model to its very limits and get the most out of it, we cannot settle for arbitrary

choices defining our deep learning model as human choices are always subject to fallacy and error in one way or the other. In other words, even if a person ends up developing a commendable intuition, in the end, the initial decision will always end up being sub-optimal. In such a scenario, the engineers and researchers of machine learning will have to grind their time in repeatedly improving their deep learning model. At the end of the day, the task of tweaking the hyperparameters to optimize them is not suitable for humans and is best left to machines themselves.

In short, the jobs we should spend our time perfecting is the exploration of the realm of possible decisions in a systematic and principled way. To do so, we are required to scavenge through the network's architecture and look for the most optimally functioning space empirically. This is the essence and core of hyperparameter optimization, a critical and effective field of research. A typical hyperparameter optimization procedure is as follows;

1. Automating the nomination of a hyperparameter.
2. Constructing the deep learning model accordingly.
3. Proceeding to fit these parameters to the input training data and calculating the model's performance on the corresponding validation dataset.
4. Automating the nomination of another set of hyperparameters for the model to try out.
5. Repeating steps 2 and 3.
6. Gradually moving on to analyzing the model's performance on the testing dataset.

The key to performing the hyperparameter optimization process successfully is twofold:

- The algorithm is not responsible for nominating the sets of hyperparameters for the model to try
- Careful consideration is given to the historical validation performance for the different hyperparameters sets used so far.

As such, there are several techniques available to use, such as Bayesian optimization, genetic algorithms, simple random search, etc.

Training the model's weight is comparatively easy and simple. All you have

to do is just calculate the loss functions on the current mini-batch data and use the backpropagation algorithm so that you can push these weights in the right direction. On the other hand, updating a hyperparameter is anything but easy. To understand this, try to analyze these two situations;

1. It can be expensive in resource terms to calculate a feedback signal to determine whether the current set of hyperparameters creates an optimal model for the task at hand. This means that it will require the machine learning engineer to repeatedly build and train new models from the ground up on the given dataset.
2. A hyperparameter space is essentially a network of discrete decisions. This means that it cannot be differentiable or continuous. Due to this, the gradient descent optimization is not an option for use with a hyperparameter space. Hence, we are left with using optimization methods and techniques that are gradient-free, and these techniques are inefficient as compared to gradient descent.

Due to the great difficulty of these challenges being faced by machine learning engineers while the field is relatively is young and not well-explored, we are stuck with using only a limited set of tools to optimize deep learning models with. Most of the time, random search becomes the only viable option. However, it is a very naïve technique as we are essentially choosing random hyperparameters to try out and keep on repeating this process. However, a tool more reliable than random search and can perform arguably better than it is a Python library for hyperparameter optimization that predicts a set of hyperparameters likely to be optimal for the model by using Parzen estimators is the **Hyperopt** tool and this tool can be accessed from;

https://github.com/hyperopt/hyperopt

In Keras deep learning models, there is a similar tool that essentially integrates Hyperopt so that they can be used with deep learning models using Keras and this library is known as **Hyperas** and can be accessed from;

https://github.com/maxpumperla/hyperas

In short, hyperparameter optimization is an essential and very important tool for building top tier and high-performance deep learning models.

Model Ensembling

The last technique we will be discussing for this book is the model ensembling, a robust tool that can bring out the maximum potential of your deep learning model. Model ensembling involves taking the predictions of different models and pooling them to produce an overall better prediction.

The core idea of model ensembling that all the good models are designed to be optimal in their own ways. For example, each of them looks at a certain aspect of the data to give good predictions, by combining these predictions looking at different aspects together, we can produce an even better prediction that includes all these different aspects of the data. For example, let's look at a typical classification task. To ensemble the different sets of classifiers, we can just average their predictions by setting up a meantime;

```
preds_a = model_a.predict(x_val)

preds_b = model_b.predict(x_val)

preds_c = model_c.predict(x_val)

preds_d = model_d.predict(x_val)


final_preds = 0.25 * (preds_a + preds_b + preds_c + preds_d)
```

This is only viable if all the classifiers have more or less the same level of performance. If one is worse than the rest, the resulting prediction will be heavily affected.

A more efficient and optimal method of ensembling different classifiers is by performing a weighted average in such a way that good classifiers are given a heavier weight set, and bad classifiers are given low weight sets. To find an optimal set of ensembling weights, we can either use the random search or a simple optimization algorithm like the Nelder-Mead;

```
preds_a = model_a.predict(x_val)

preds_b = model_b.predict(x_val)

preds_c = model_c.predict(x_val)
```

```
preds_d = model_d.predict(x_val)


final_preds = 0.5 * preds_a + 0.25 * preds_b + 0.1 * preds_c + 0.15 * preds_d
```

The methods through which we can approach model ensembling is very diverse. However, the method shown above is seen to be a very strong foundation for performing a good model ensembling.

Conclusion

Until now, we have discussed the majority of the techniques, concepts, and methods that fall into the realm of advanced practices when building deep learning models for Python. We have explored the intricacies of the neural network in a mode. We also explored its internal realms by experimenting on how we can change its structure, add in different layers, functions, and arguments and other such elements to build a model that effectively works to perform the task at hand efficiently and optimally. To wrap things up, we introduced a final bundle of techniques that require practice to master and ingenuity to implement correctly, but when used correctly, they will prove to be the building blocks of a deep learning model that will foster the aspirations of the engineer in its truest form.

As a final note, we could only cover so many topics in the span of one book, in the upcoming series we will delve even deeper into the complex and intricate understandings of deep learning systems and rebuild our current understanding into something even grander.

References

1). Deep Learning with python by Author Francois Chollet; ISBN: 9781617294433
https://people.sc.fsu.edu/~jburkardt/keras_src/newswire/newswire.py

2). Python Script using Data from Hate Crimes in India by Regression Models (Mohamed Khalid); https://www.kaggle.com/khalidative/regressionmodels

3). Dog vs Cat: How is the data labelled? By Petezurich and Peyman. H (Stackoverflow) https://stackoverflow.com/a/52201865

4). Docker example by Oliver LeDiouris; https://github.com/OlivierLD/raspberry-coffee/blob/master/docker/