CS4152-Deep Lerning and Neural Networks
Instructor: Dr. Jameel Ahmad
Class Activity-2: Neural Networks
Exercises and Activities Objectives

CA-2 Fall 2025

Important Instructions

- **Group Size:** Students should work in groups of **2 students**
- **Submission Deadline: November 7, 2025**
- **Submission Method:** Upload your reports to the following Google Drive folder:

Google Drive Submission Folder

- Please name your report file as:
- Please name your report file as: `Group--Name1+ID1+Name2+ID2--CA2.pdf`

Learning Objectives for Exercises and Activities

Exercise 2.01: Perceptron Implementation

- To understand and implement the basic building block of neural networks by:
  - Creating a single perceptron model in TensorFlow
  - Implementing the net input function using matrix multiplication
  - Applying the sigmoid activation function for binary classification
  - Setting up weights, biases, and input data structures

Exercise 2.02: Perceptron as a Binary Classifier

- To train a perceptron for binary classification by:
  - Implementing the complete training loop with forward and backward propagation
  - Using stochastic gradient descent (SGD) optimizer
  - Calculating loss using cross-entropy
  - Evaluating model performance using accuracy score and confusion matrix

Exercise 2.03: Multiclass Classification Using a Perceptron

- To extend perceptron capabilities for multiclass problems by:
  - Implementing one-hot encoding for multiple classes
  - Using Softmax activation function instead of sigmoid
  - Building a single layer with multiple neurons
  - Applying softmax cross-entropy loss function

Exercise 2.04: Classifying Handwritten Digits

- To apply neural networks to real-world image data by:
  - Working with the MNIST dataset of handwritten digits
  - Flattening 2D image data into 1D feature vectors
  - Building a 10-class classifier for digit recognition
  - Handling image preprocessing and normalization

Exercise 2.05: Binary Classification Using Keras

- To transition to high-level neural network APIs by:
  - Using Keras Sequential API for model building
  - Understanding the difference between TensorFlow and Keras implementations
  - Compiling models with appropriate optimizers and loss functions
  - Utilizing model summary and training visualization

Exercise 2.06: Multilayer Binary Classifier

- To build deep neural networks by:
  - Creating multiple hidden layers with ReLU activation
  - Understanding the architecture of deep neural networks
  - Comparing performance between single-layer and multilayer networks
  - Configuring the output layer for binary classification

Exercise 2.07: Deep Neural Network on MNIST Using Keras

- To implement comprehensive deep learning solutions by:
  - Building a deep neural network for multiclass image classification
  - Using Flatten layers for image input preprocessing
  - Configuring appropriate loss functions for multiclass problems
  - Making predictions and visualizing results on test data

Activity 2.01: Build a Multilayer Neural Network to Classify Sonar Signals

- To apply comprehensive neural network knowledge by:
  - Analyzing and preprocessing real-world sonar data
  - Designing an appropriate network architecture from scratch
  - Implementing data preprocessing and one-hot encoding
  - Achieving high accuracy (>95%) on a practical classification problem
  - Demonstrating end-to-end neural network development skills

Overall Learning Goals

- Understand the biological inspiration behind artificial neural networks
- Master the implementation of single-layer and multilayer perceptrons
- Learn to use both TensorFlow low-level APIs and Keras high-level APIs
- Develop skills in binary and multiclass classification problems
- Gain experience with real-world datasets (MNIST, Iris, Sonar)
- Understand key concepts: forward/backward propagation, activation functions, optimizers
- Learn practical considerations: overfitting, dropout, hyperparameter tuning

Key Concepts Covered

- Biological vs. Artificial Neurons
- Perceptron Architecture and Components
- Activation Functions (Sigmoid, ReLU, Softmax)
- Loss Functions (Binary Cross-Entropy, Categorical Cross-Entropy)
- Optimizers (SGD, Adam)
- Forward and Backward Propagation
- One-Hot Encoding
- Model Evaluation Metrics
- Hyperparameter Tuning
- Overfitting and Regularization (Dropout)

Submission Requirements

- Complete all **7 exercises** and **1 activity**
- Submit a single PDF report containing:
  - Code implementation for each exercise/activity-Add Google Colab link
  - Outputs and results with explanations
  - Model architectures and configurations
  - Performance metrics and analysis
  - Group member names and student IDs
- Ensure proper documentation and clear presentation of results

Submission Deadline: November 7, 2025

Dataset References

- **OR Table Data** - Basic logical operation dataset
- **data.csv** - Binary classification dataset
- **Iris Dataset** - Multiclass flower classification
- **MNIST Dataset** - Handwritten digit recognition
- **Sonar Dataset** - Real-world signal classification

Class Activity-2

NEURAL NETWORKS

OVERVIEW

This chapter starts with an introduction to biological neurons; we see how an artificial neural network is inspired by biological neural networks. We will examine the structure and inner workings of a simple single-layer neuron called a perceptron and learn how to implement it in TensorFlow. We will move on to building multilayer neural networks to solve more complex multiclass classification tasks and discuss the practical considerations of designing a neural network. As we build deep neural networks, we will move on to Keras to build modular and easy-to-customize neural network models in Python. By the end of this chapter, you'll be adept at building neural networks to solve complex problems.

INTRODUCTION

In the previous chapter, we learned how to implement basic mathematical concepts such as quadratic equations, linear algebra, and matrix multiplication in TensorFlow. Now that we have learned the basics, let's dive into **Artificial Neural Networks** (**ANNs**), which are central to artificial intelligence and deep learning.

Deep learning is a subset of machine learning. In supervised learning, we often use traditional machine learning techniques, such as support vector machines or tree-based models, where features are explicitly engineered by humans. However, in deep learning, the model explores and identifies the important features of a labeled dataset without human intervention. ANNs, inspired by biological neurons, have a layered representation, which helps them learn labels incrementally—from the minute details to the complex ones. Consider the example of image recognition: in a given image, an ANN would just as easily identify basic details such as light and dark areas as it would identify more complex structures such as shapes. Though neural network techniques are tremendously successful at tasks such as identifying objects in images, how they do so is a black box, as the features are learned implicitly. Deep learning techniques have turned out to be powerful at tackling very complex problems, such as speech/image recognition, and hence are used across industry in building self-driving cars, Google Now, and many more applications.

Now that we know the importance of deep learning techniques, we will take a pragmatic step-by-step approach to understanding a mix of theory and practical considerations in building deep-learning-based solutions. We will start with the smallest component of a neural network, which is an artificial neuron, also referred to as a perceptron, and incrementally increase the complexity to explore **Multi-Layer Perceptrons** (**MLPs**) and advanced models such as **Recurrent Neural Networks** (**RNNs**) and **Convolutional Neural Networks** (**CNNs**).

NEURAL NETWORKS AND THE STRUCTURE OF PERCEPTRONS

A neuron is a basic building block of the human nervous system, which relays electric signals across the body. The human brain consists of billions of interconnected biological neurons, and they are constantly communicating with each other by sending minute electrical binary signals by turning themselves on or off. The general meaning of a neural network is a network of interconnected neurons. In the current context, we are referring to ANNs, which are actually modeled on a biological neural network. The term artificial intelligence is derived from the fact that natural intelligence exists in the human brain (or any brain for that matter), and we humans are trying to simulate this natural intelligence artificially. Though ANNs are inspired by biological neurons, some of the advanced neural network architectures, such as CNNs and RNNs, do not actually mimic the behavior of a biological neuron. However, for ease of understanding, we will begin by drawing an analogy between the biological neuron and an artificial neuron (perceptron).

A simplified version of a biological neuron is represented in *Figure 2.1*:

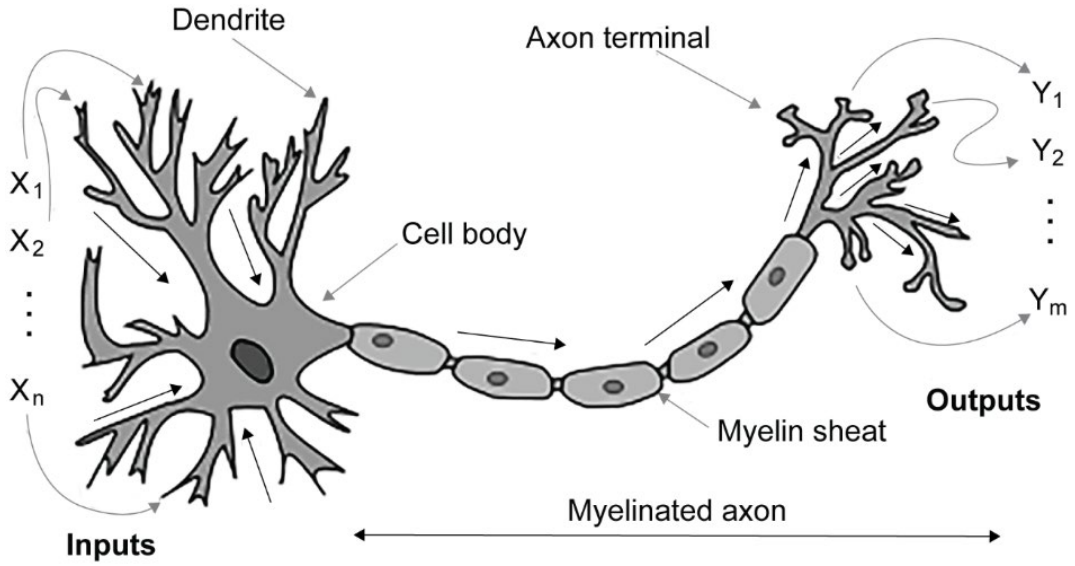

Figure 2.1: Biological neuron

This is a highly simplified representation. There are three main components:

- The dendrites, which receive the input signals
- The cell body, where the signal is processed in some form
- The tail-like axon, through which the neuron transfers the signal out to the next neuron

A perceptron can also be represented in a similar way, although it is not a physical entity but a mathematical model. *Figure 2.2* shows a high-level representation of an artificial neuron:

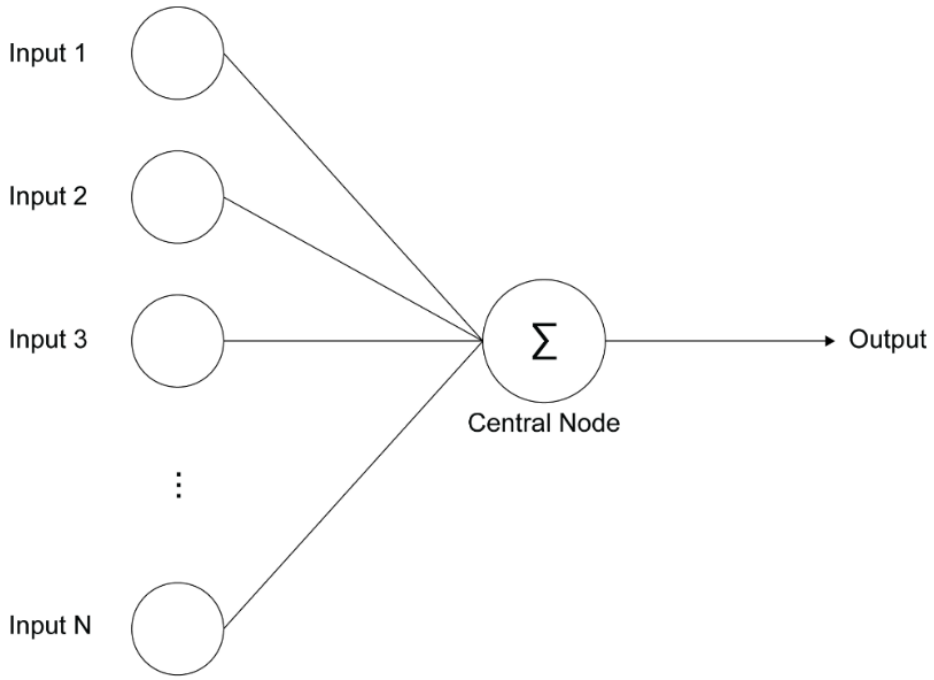

Figure 2.2: Representation of an artificial neuron

In an artificial neuron, as in a biological one, there is an input signal. The central node conflates all the signals and fires the output signal if it is above a certain threshold. A more detailed representation of a perceptron is shown in *Figure 2.3*. Each component of this perceptron is explained in the sections that follow:

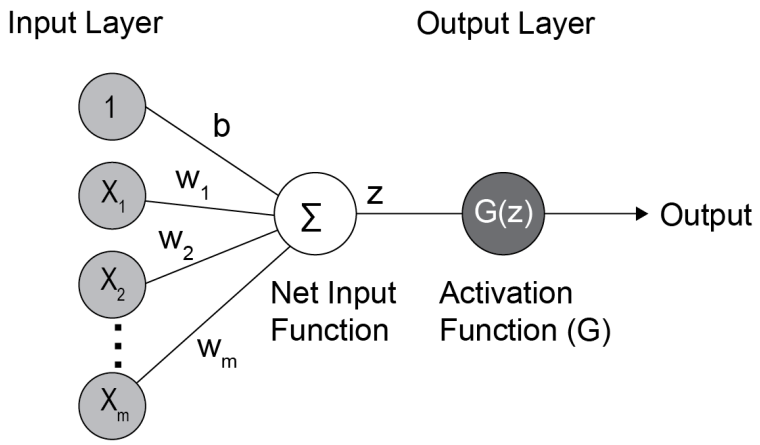

Figure 2.3: Representation of a perceptron

A perceptron has the following components:

- Input layer
- Weights
- Bias
- Net input function
- Activation function

Let's look at these components and their TensorFlow implementations in detail by considering an **OR** table dataset.

INPUT LAYER

Each example of input data is fed through the input layer. Referring to the representation shown in *Figure 2.3*, depending on the size of the input example, the number of nodes will vary from x_1 to x_m. The input data can be structured data (such as a CSV file) or unstructured data, such as an image. These inputs, x_1 to x_m, are called features (**m** refers to the number of features). Let's illustrate this with an example.

Let's say the data is in the form of a table as follows:

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

Figure 2.4: Sample input and output data – OR table

Here, the inputs to the neuron are the columns x_1 and x_2, which correspond to one row. At this point, it may be difficult to comprehend, but for now, accept it that the data is fed one row at a time in an iterative manner during training. We will represent the input data and the true labels (output **y**) with the TensorFlow **Variable** class as follows:

```
X = tf.Variable([[0.,0.],[0.,1.],\
                 [1.,0.],[1.,1.]], \
                 tf.float32)
y = tf.Variable([0, 1, 1, 1], tf.float32)
```

WEIGHTS

Weights are associated with each neuron, and the input features dictate how much influence each of the input features should have in computing the next node. Each neuron will be connected to all the input features. In the example, since there were two inputs (x_1 and x_2) and the input layer is connected to one neuron, there will be two weights associated with it: w_1 and w_2. A weight is a real number; it can be positive or negative and is mathematically represented as **R**. When we say that a neural network is learning, what is happening is that the network is adjusting its weights and biases to get the correct predictions by adjusting to the error feedback. We will see this in more detail in the sections that follow. For now, we will initialize the weights as zeros and use the same TensorFlow **Variable** class as follows:

```
number_of_features = x.shape[1]
number_of_units = 1
Weight = tf.Variable(tf.zeros([number_of_features, \
                               number_of_units]), \
                               tf.float32)
```

Weights would be of the following dimension: *number of input features* × *output size*.

BIAS

In *Figure 2.3*, bias is represented by *b*, which is called additive bias. Every neuron has one bias. When *x* is zero, that is, no information is coming from the independent variables, then the output should be biased to just *b*. Like the weights, the bias also a real number, and the network has to learn the bias value to get the correct predictions.

In TensorFlow, bias is the same size as the output size and can be represented as follows:

```
B = tf.Variable(tf.zeros([1, 1]), tf.float32)
```

NET INPUT FUNCTION

The net input function, also commonly referred to as the input function, can be described as the sum of the products of the inputs and their corresponding weights plus the bias. Mathematically, it is represented as follows:

$$z = \sum_{i=1}^{m} w_i x_i + b$$

Figure 2.5: Net input function in mathematical form

Here:

- x_i: input data—x_1 to x_m
- w_i: weights—w_1 to w_m
- *b*: additive bias

As you can see, this formula involves inputs and their associated weights and biases. This can be written in vectorized form, and we can use matrix multiplication, which we learned about in *Chapter 1*, *Building Blocks of Deep Learning*. We will see this when we start the code demo. Since all the variables are numbers, the result of the net input function is just a number, a real number. The net input function can be easily implemented using the TensorFlow **`matmul`** functionality as follows:

```
z = tf.add(tf.matmul(X, W), B)
```

W stands for weight, **X** stands for input, and **B** stands for bias.

ACTIVATION FUNCTION (G)

The output of the net input function (**z**) is fed as input to the activation function. The activation function squashes the output of the net input function (**z**) into a new output range depending on the choice of activation function. There are a variety of activation functions, such as sigmoid (logistic), ReLU, and tanh. Each activation function has its own pros and cons. We will take a deep dive into activation functions later in the chapter. For now, we will start with a sigmoid activation function, also known as a logistic function. With the sigmoid activation function, the linear output **z** is squashed into a new output range of (0,1). The activation function provides non-linearity between layers, which gives neural networks the ability to approximate any continuous function.

The mathematical equation of the sigmoid function is as follows, where *G(z)* is the sigmoid function and the right-hand equation details the derivative with respect to *z*:

$$G(z) = \frac{1}{1 + e^{-z}}$$

Figure 2.6: Mathematical form of the sigmoid function

As you can see in *Figure 2.7*, the sigmoid function is a more or less S-shaped curve with values between 0 and 1, no matter what the input is:

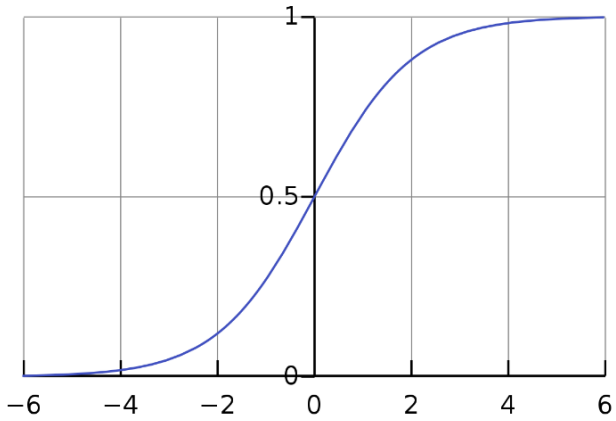

Figure 2.7: Sigmoid curve

And if we set a threshold (say **0.5**), we can convert this into a binary output. Any output greater than or equal to **.5** is considered **1**, and any value less than **.5** is considered **0**.

Activation functions such as sigmoid are provided out of the box in TensorFlow. A sigmoid function can be implemented in TensorFlow as follows:

```
output = tf.sigmoid(z)
```

Now that we have seen the structure of a perceptron and its code representation in TensorFlow, let's put all the components together to make a perceptron.

PERCEPTRONS IN TENSORFLOW

In TensorFlow, a perceptron can be implemented just by defining a simple function, as follows:

```
def perceptron(X):
    z = tf.add(tf.matmul(X, W), B)
    output = tf.sigmoid(z)
    return output
```

At a very high level, we can see that the input data passes through the net input function. The output of the net input function is passed to the activation function, which, in turn, gives us the predicted output. Now, let's look at each line of the code:

```
z = tf.add(tf.matmul(X, W), B)
```

The output of the net input function is stored in **z**. Let's see how we got that result by breaking it down further into two parts, that is, the matrix multiplication part contained in **tf.matmul** and the addition contained in **tf.add**.

Let's say we're storing the result of the matrix multiplication of **X** and **W** in a variable called **m**:

```
m = tf.matmul(X, W)
```

Now, let's consider how we got that result. For example, let's say **X** is a row matrix, like [X_1 X_2], and **W** is a column matrix, as follows:

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

Figure 2.8: Column matrix

Recall from the previous chapter that **tf.matmul** will perform matrix multiplication. So, the result is this:

```
m = x1*w1 + x2*w2
```

And then, we add the output, **m**, to the bias, **B**, as follows:

```
z = tf.add(m, B)
```

Note that what we do in the preceding step is the same as the mere addition of the two variables **m** and **b**:

```
m + b
```

Hence, the final output is:

```
z = x1*w1 + x2*w2 + b
```

z would be the output of the net input function.

Now, let's consider the next line:

```
output= tf.sigmoid(z)
```

As we learned earlier, **tf.sigmoid** is a readily available implementation of the sigmoid function. The net input function's output (**z**) computed in the previous line is fed as input to the sigmoid function. The result of the sigmoid function is the output of the perceptron, which is in the range of 0 to 1. During training, which will be explained later in the chapter, we will feed the data in batches to this function, which will calculate the predicted values.

EXERCISE 2.01: PERCEPTRON IMPLEMENTATION

In this exercise, we will implement the perceptron in TensorFlow for an **OR** table. Let's set the input data in TensorFlow and freeze the design parameters of perceptron:

1. Let's import the necessary package, which, in our case, is **tensorflow**:

```
import tensorflow as tf
```

2. Set the input data and labels of the **OR** table data in TensorFlow:

```
X = tf.Variable([[0.,0.],[0.,1.],\
                 [1.,0.],[1.,1.]], \
                 dtype=tf.float32)
print(X)
```

   As you can see in the output, we will have a 4 × 2 matrix of input data:

```
<tf.Variable 'Variable:0' shape=(4, 2) dtype=float32,
numpy=array([[0., 0.],
             [0., 1.],
             [1., 0.],
             [1., 1.]], dtype=float32)>
```

3. We will set the actual labels in TensorFlow and use the **reshape()** function to reshape the **y** vector into a 4 × 1 matrix:

```
y = tf.Variable([0, 1, 1, 1], dtype=tf.float32)
y = tf.reshape(y, [4,1])
print(y)
```

   The output is a 4 × 1 matrix, as follows:

```
tf.Tensor(
[[0.]
 [1.]
 [1.]
 [1.]], shape=(4, 1), dtype=float32)
```

4. Now let's design parameters of a perceptron.

   Number of neurons (units) = 1

   Number of features (inputs) = 2 (number of examples × number of features)

   The activation function will be the sigmoid function, since we are doing binary classification:

```
NUM_FEATURES = X.shape[1]
OUTPUT_SIZE = 1
```

   In the preceding code, **`X.shape[1]`** will equal **2** (since the indices start with zero, **1** refers to the second index, which is **2**).

5. Define the connections weight matrix in TensorFlow:

```
W = tf.Variable(tf.zeros([NUM_FEATURES, \
                          OUTPUT_SIZE]), \
                          dtype=tf.float32)
print(W)
```

   The weight matrix would essentially be a columnar matrix as shown in the following figure. It will have the following dimension: *number of features (columns)* × *output size*:

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

Figure 2.9: A columnar matrix

   The output size will be dependent on the number of neurons—in this case, it is **1**. So, if you are developing a layer of 10 neurons with two features, the shape of this matrix will be [2,10]. The **`tf.zeros`** function creates a tensor with the given shape and initializes all the elements to zeros.

   So, this will result in a zero columnar matrix like this:

```
<tf.Variable 'Variable:0' shape=(2, 1) dtype=float32, \
numpy=array([[0.], [0.]], dtype=float32)>
```

6. Now create the variable for the bias:

```
B = tf.Variable(tf.zeros([OUTPUT_SIZE, 1]), dtype=tf.float32)
print(B)
```

There is only one bias per neuron, so in this case, the bias is just one number in the form of a single-element array. However, if we had a layer of 10 neurons, then it would be an array of 10 numbers—1 for each neuron.

This will result in a 0-row matrix with a single element like this:

```
<tf.Variable 'Variable:0' shape=(1, 1) dtype=float32,
numpy=array([[0.]], dtype=float32)>
```

7. Now that we have the weights and bias, the next step is to perform the computation to get the net input function, feed it to the activation function, and then get the final output. Let's define a function called **`perceptron`** to get the output:

```
def perceptron(X):
    z = tf.add(tf.matmul(X, W), B)
    output = tf.sigmoid(z)
    return output

print(perceptron(X))
```

The output will be a 4 × 1 array that contains the predictions by our perceptron:

```
tf.Tensor(
[[0.5]
 [0.5]
 [0.5]
 [0.5]], shape=(4, 1), dtype=float32)
```

As we can see, the predictions are not quite accurate. We will learn how to improve the results in the sections that follow.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/3feF7MO.
>
> You can also run this example online at https://packt.live/2CkMiEE.
> You must execute the entire Notebook in order to get the desired result.

In this exercise, we implemented a perceptron, which is a mathematical implementation of a single artificial neuron. Keep in mind that it is just the implementation of the model; we have not done any training. In the next section, we will see how to train the perceptron.

TRAINING A PERCEPTRON

To train a perceptron, we need the following components:

- Data representation
- Layers
- Neural network representation
- Loss function
- Optimizer
- Training loop

In the previous section, we covered most of the preceding components: the **data representation** of the input data and the true labels in TensorFlow. For **layers**, we have the linear layer and the activation functions, which we saw in the form of the net input function and the sigmoid function respectively. For the **neural network representation**, we made a function called `perceptron()`, which uses a linear layer and a sigmoid layer to perform predictions. What we did in the previous section using input data and initial weights and biases is called **forward propagation**.
The actual neural network training involves two stages: forward propagation and backward propagation. We will explore them in detail in the next few steps. Let's look at the training process at a higher level:

- A training iteration where the neural network goes through all the training examples is called an Epoch. This is one of the hyperparameters to be tweaked in order to train a neural network.
- In each pass, a neural network does forward propagation, where data travels from the input to the output. As seen in *Exercise 2.01*, *Perceptron Implementation*, inputs are fed to the perceptron. Input data passes through the net input function and the activation function to produce the predicted output. The predicted output is compared with the labels or the ground truth, and the error or loss is calculated.

- In order to make a neural network learn, learning being the adjustment of weights and biases in order to make correct predictions, there needs to be a **loss function**, which will calculate the error between an actual label and the predicted label.
- To minimize the error in the neural network, the training loop needs an **optimizer**, which will minimize the loss on the basis of a loss function.
- Once the error is calculated, the neural network then sees which nodes of the network contributed to the error and by how much. This is essential in order to make the predictions better in the next epoch. This way of propagating the error backward is called **backward propagation** (backpropagation). Backpropagation uses the chain rule from calculus to propagate the error (the error gradient) in reverse order until it reaches the input layer. As it propagates the error back through the network, it uses gradient descent to make fine adjustments to the weights and biases in the network by utilizing the error gradient calculated before.

This cycle continues until the loss is minimized.

Let's implement the theory we have discussed in TensorFlow. Revisit the code in *Exercise 2.01*, *Perceptron Implementation,* where the perceptron we created just did one forward pass. We got the following predictions, and we saw that our perceptron had not learned anything:

```
tf.Tensor(
[[0.5]
 [0.5]
 [0.5]
 [0.5]], shape=(4, 1), dtype=float32)
```

In order to make our perceptron learn, we need additional components, such as a training loop, a loss function, and an optimizer. Let's see how to implement these components in TensorFlow.

PERCEPTRON TRAINING PROCESS IN TENSORFLOW

In the next exercise, when we train our model, we will use a **Stochastic Gradient Descent** (**SGD**) optimizer to minimize the loss. There are a few more advanced optimizers available and provided by TensorFlow out of the box. We will look at the pros and cons of each of them in later sections. The following code will instantiate a stochastic gradient descent optimizer using TensorFlow:

```
learning_rate = 0.01
optimizer = tf.optimizers.SGD(learning_rate)
```

The **`perceptron`** function takes care of the forward propagation. For the backpropagation of the error, we have used an optimizer. **`Tf.optimizers.SGD`** creates an instance of an optimizer. SGD will update the parameters of the networks—weights and biases—on each example from the input data. We will discuss the functioning of the gradient descent optimizer in greater detail later in this chapter. We will also discuss the significance of the **`0.01`** parameter, which is known as the learning rate. The learning rate is the magnitude by which SGD takes a step in order to reach the global optimum of the loss function. The learning rate is another hyperparameter that needs to be tweaked in order to train a neural network.

The following code can be used to define the epochs, training loop, and loss function:

```
no_of_epochs = 1000

for n in range(no_of_epochs):
    loss = lambda:abs(tf.reduce_mean(tf.nn.\
           sigmoid_cross_entropy_with_logits\
           (labels=y,logits=perceptron(X))))
    optimizer.minimize(loss, [W, B])
```

Inside the training loop, the loss is calculated using the loss function, which is defined as a lambda function.

The **`tf.nn.sigmoid_cross_entropy_with_logits`** function calculates the loss value of each observation. It takes two parameters: **`Labels = y`** and **`logit = perceptron(x)`**.

`perceptron(X)` returns the predicted value, which is the result of the forward propagation of the input, **`x`**. This is compared with the corresponding label value stored in **`y`**. The mean value is calculated using **`Tf.reduce_mean`**, and the magnitude is taken. The sign is ignored using the **`abs`** function. **`Optimizer.minimize`** takes the loss value and adjusts the weights and bias as a part of the backward propagation of the error.

The forward propagation is executed again with the new values of weights and bias. And this forward and backward process continues for the number of iterations we define.

During the backpropagation, the weights and biases are updated only if the loss is less than the previous cycle. Otherwise, the weights and biases remain unchanged. In this way, the optimizer ensures that even though it loops through the required number of iterations, it only stores the values of **`w`** and **`b`** for which the loss is minimal.

We have set the number of epochs for the training to 1,000 iterations. There is no rule of thumb for setting the number of epochs since the number of epochs is a hyperparameter. But how do we know when training has taken place successfully?

When we can see that the values of weights and biases have changed, we can conclude the training has taken place. Let's say we used a training loop for the **`OR`** data we saw in *Exercise 2.01*, *Perceptron Implementation*, we would see weights somewhat equal to the following:

```
[[0.412449151]
[0.412449151]]
```

And the bias would be something like this:

```
0.236065879
```

When the network has learned, that is, the weights and biases have been updated, we can see whether it is making accurate predictions using **`accuracy_score`** from the **`scikit-learn`** package. We can use it to measure the accuracy of the predictions as follows:

```
from sklearn.metrics import accuracy_score
print(accuracy_score(y, ypred))
```

Here, **accuracy_score** takes two parameters—the label values (**y**) and the predicted values (**ypred**)—and measures the accuracy. Let's say the result is **1.0**. This means the perceptron is 100% accurate.

In the next exercise, we will train our perceptron to perform a binary classification.

EXERCISE 2.02: PERCEPTRON AS A BINARY CLASSIFIER

In the previous section, we learned how to train a perceptron. In this exercise, we will train our perceptron to approximate a slightly more complicated function. We will be using randomly generated external data with two classes: class **0** and class **1**. Our trained perceptron should be able to classify the random numbers based on their class:

> **NOTE**
>
> The data is in a CSV file called **data.csv**. You can download the file from GitHub by visiting https://packt.live/2BVtxIf.

1. Import the required libraries:

```
import tensorflow as tf
import pandas as pd
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
%matplotlib inline
```

   Apart from **tensorflow**, we will need **pandas** to read the data from the CSV file, **confusion_matrix** and **accuracy_score** to measure the accuracy of our perceptron after the training, and **matplotlib** to visualize the data.

2. Read the data from the **data.csv** file. It should be in the same path as the Jupyter Notebook file in which you are running this exercise's code. Otherwise, you will have to change the path in the code before executing it:

```
df = pd.read_csv('data.csv')
```

3. Examine the data:

```
df.head()
```

The output will be as follows:

	label	x1	x2
0	1	2.6487	4.5192
1	1	1.5438	2.4443
2	1	1.8990	4.2409
3	1	2.4711	5.8097
4	1	3.3590	6.4423

Figure 2.10: Contents of the DataFrame

As you can see, the data has three columns. **`x1`** and **`x2`** are the features, and the **`label`** column contains the labels **0** or **1** for each observation. The best way to see this kind of data is through a scatter plot.

4. Visualize the data by plotting it using **`matplotlib`**:

```
plt.scatter(df[df['label'] == 0]['x1'], \
            df[df['label'] == 0]['x2'], \
            marker='*')
plt.scatter(df[df['label'] == 1]['x1'], \
            df[df['label'] == 1]['x2'], marker='<')
```

The output will be as follows:

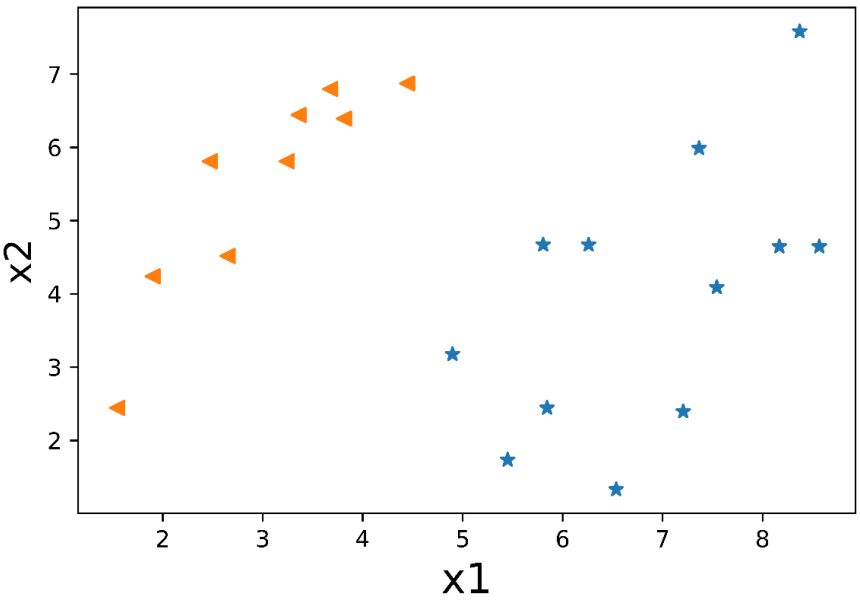

Figure 2.11: Scatter plot of external data

This shows the two distinct classes of the data shown by the two different shapes. Data with the label **0** is represented by a star, while data with the label **1** is represented by a triangle.

5. Prepare the data. This step is not unique to neural networks; you must have seen it in regular machine learning as well. Before submitting the data to a model for training, you split it into features and labels:

```
X_input = df[['x1','x2']].values
y_label = df[['label']].values
```

`x_input` contains the features, **`x1`** and **`x2`**. The values at the end convert it into matrix format, which is what is expected as input when the tensors are created. **`y_label`** contains the labels in matrix format.

6. Create TensorFlow variables for features and labels and typecast them to **`float`**:

```
x = tf.Variable(X_input, dtype=tf.float32)
y = tf.Variable(y_label, dtype=tf.float32)
```

7. The rest of the code is for the training of the perceptron, which we saw in *Exercise 2.01*, *Perceptron Implementation*:

Exercise2.02.ipynb

```
Number_of_features = 2
Number_of_units = 1
learning_rate = 0.01

# weights and bias
weight = tf.Variable(tf.zeros([Number_of_features, \
                               Number_of_units]))
bias = tf.Variable(tf.zeros([Number_of_units]))

#optimizer
optimizer = tf.optimizers.SGD(learning_rate)

def perceptron(x):
    z = tf.add(tf.matmul(x,weight),bias)
    output = tf.sigmoid(z)
    return output
```

The complete code for this step can be found at https://packt.live/3gJ73bY.

> **NOTE**
>
> The **#** symbol in the code snippet above denotes a code comment. Comments are added into code to help explain specific bits of logic.

8. Display the values of **`weight`** and **`bias`** to show that the perceptron has been trained:

```
tf.print(weight, bias)
```

The output is as follows:

```
[[-0.844034135]
 [0.673354745]] [0.0593947917]
```

9. Pass the input data to check whether the perceptron classifies it correctly:

```
ypred = perceptron(x)
```

10. Round off the output to convert it into binary format:

```
ypred = tf.round(ypred)
```

11. Measure the accuracy using the **`accuracy_score`** method, as we did in the previous exercise:

```
acc = accuracy_score(y.numpy(), ypred.numpy())
print(acc)
```

The output is as follows:

```
1.0
```

The perceptron gives 100% accuracy.

12. The confusion matrix helps to get the performance measurement of a model. We will plot the confusion matrix using the **`scikit-learn`** package.

```
cnf_matrix = confusion_matrix(y.numpy(), \
                              ypred.numpy())
print(cnf_matrix)
```

The output will be as follows:

```
[[12  0]
[ 0  9]]
```

All the numbers are along the diagonal, that is, 12 values corresponding to class 0 and 9 values corresponding to class 1 are properly classified by our trained perceptron (which has achieved 100% accuracy).

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/3gJ73bY.
>
> You can also run this example online at https://packt.live/2DhelFw.
> You must execute the entire Notebook in order to get the desired result.

In this exercise, we trained our perceptron into a binary classifier, and it has done pretty well. In the next exercise, we will see how to create a multiclass classifier.

MULTICLASS CLASSIFIER

A classifier that can handle two classes is known as a **binary classifier**, like the one we saw in the preceding exercise. A classifier that can handle more than two classes is known as a **multiclass classifier**. We cannot build a multiclass classifier with a single neuron. Now we move from one neuron to one layer of multiple neurons, which is required for multiclass classifiers.

A single layer of multiple neurons can be trained to be a multiclass classifier. Some of the key points are detailed here. You need as many neurons as the number of classes; that is, for a 3-class classifier, you need 3 neurons; for a 10-class classifier you need 10 neurons, and so on.

As we saw in binary classification, we used sigmoid (logistic layer) to get predictions in the range of 0 to 1. In multiclass classification, we use a special type of activation function called the **Softmax** activation function to get probabilities across each class that sums to 1. With the sigmoid function in a multiclass setting, the probabilities do not necessarily add up to 1, so Softmax is preferred.

Before we implement the multiclass classifier, let's explore the Softmax activation function.

THE SOFTMAX ACTIVATION FUNCTION

The Softmax function is also known as the **normalized exponential function**. As the word **normalized** suggests, the Softmax function normalizes the input into a probability distribution that sums to 1. Mathematically, it is represented as follows:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Figure 2.12: Mathematical form of the Softmax function

To understand what Softmax does, let's use TensorFlow's built-in **`softmax`** function and see the output.

So, for the following code:

```
values = tf.Variable([3,1,7,2,4,5], dtype=tf.float32)
output = tf.nn.softmax(values)
tf.print(output)
```

The output will be:

```
[0.0151037546 0.00204407098 0.824637055
 0.00555636082 0.0410562605 0.111602485]
```

As you can see in the output, the **`values`** input is mapped to a probability distribution that sums to 1. Note that **`7`** (the highest value in the original input values) received the highest weight, **`0.824637055`**. This is what the Softmax function is mainly used for: to focus on the largest values and suppress values that are below the maximum value. Also, if we sum the output, it adds up to ~ 1.

Illustrating the example in more detail, let's say we want to build a multiclass classifier with 3 classes. We will need 3 neurons connected to a Softmax activation function:

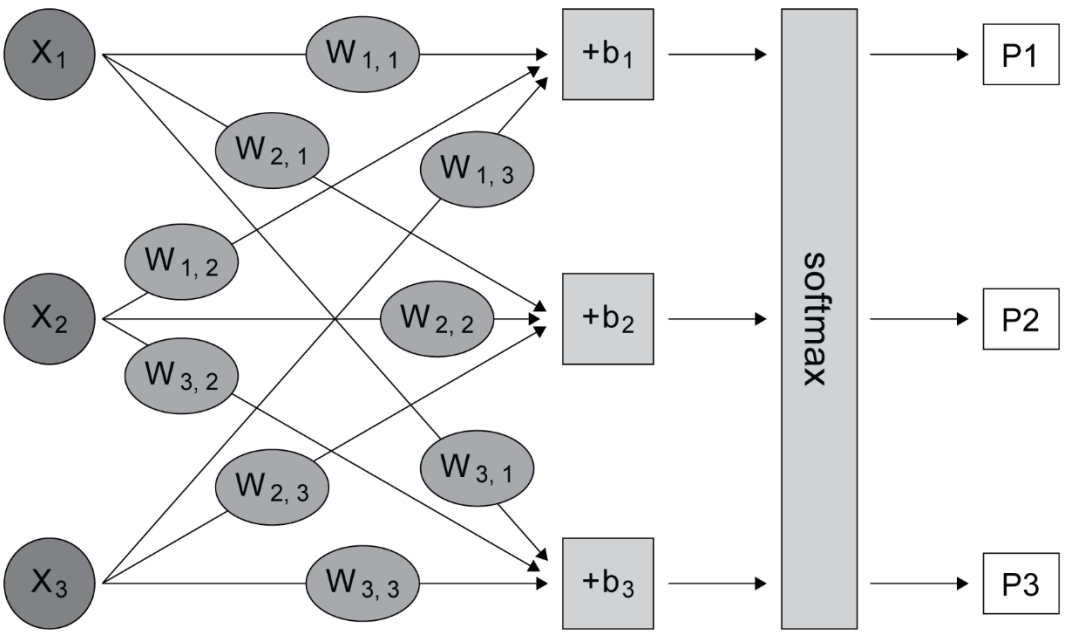

Figure 2.13: Softmax activation function used in a multiclass classification setting

As seen in *Figure 2.13*, $\mathbf{x}_1$, $\mathbf{x}_2$, and $\mathbf{x}_3$ are the input features, which go through the net input function of each of the three neurons, which have the weights and biases ($\mathbf{W}_{i,j}$ and $\mathbf{b}_i$) associated with it. Lastly, the output of the neuron is fed to the common Softmax activation function instead of the individual sigmoid functions. The Softmax activation function spits out the probabilities of the 3 classes: **P1**, **P2**, and **P3**. The sum of these three probabilities will add to 1 because of the Softmax layer.

As we saw in the previous section, Softmax highlights the maximum value and suppresses the rest of the values. Suppose a neural network is trained to classify the input into three classes, and for a given set of inputs, the output is class 2; then it would say that **P2** has the highest value since it is passed through a Softmax layer. As you can see in the following figure, **P2** has the highest value, which means the prediction is correct:

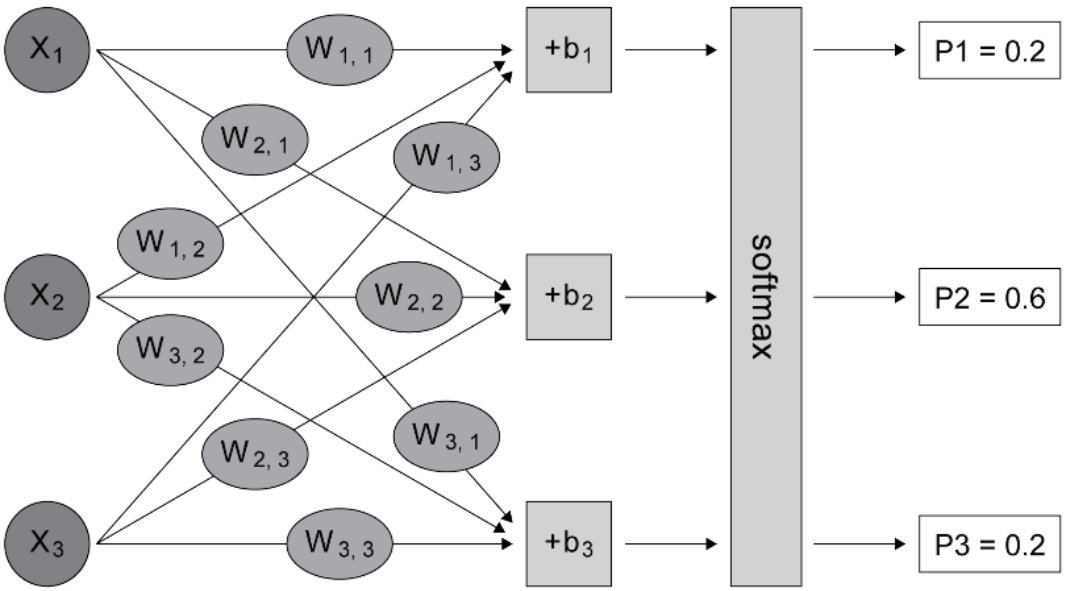

Figure 2.14: Probability P2 is the highest

An associated concept is one-hot encoding. As we have three different classes, **`class1`**, **`class2`**, and **`class3`**, we need to encode the class labels into a format that we can work with more easily; so, after applying one-hot encoding, we would see the following output:

	class 1	class 2	class 3
0	1	0	0
1	0	1	0
2	0	0	1

Figure 2.15: One-hot encoded data for three classes

This makes the results quick and easy to interpret. In this case, the output that has the highest value is set to 1, and all others are set to 0. The one-hot encoded output of the preceding example would be like this:

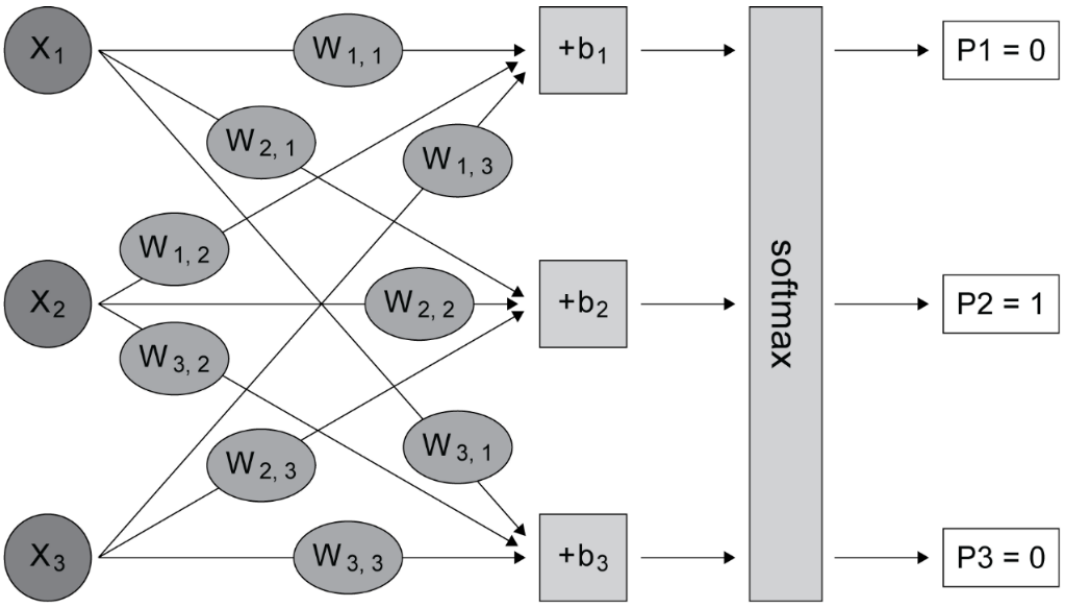

Figure 2.16: One-hot encoded output probabilities

The labels of the training data also need to be one-hot encoded. And if they have a different format, they need to be converted into one-hot-encoded format before training the model. Let's do an exercise on multiclass classification with one-hot encoding.

EXERCISE 2.03: MULTICLASS CLASSIFICATION USING A PERCEPTRON

To perform multiclass classification, we will be using the Iris dataset (https://archive.ics.uci.edu/ml/datasets/Iris), which has 3 classes of 50 instances each, where each class refers to a type of Iris. We will have a single layer of three neurons using the Softmax activation function:

> **NOTE**
>
> You can download the dataset from GitHub using this link: https://packt.live/3ekiBBf.

1. Import the required libraries:

```
import tensorflow as tf
import pandas as pd

from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score

import matplotlib.pyplot as plt
%matplotlib inline

from pandas import get_dummies
```

   You must be familiar with all of these imports as they were used in the previous exercise, except for **`get_dummies`**. This function converts a given label data into the corresponding one-hot-encoded format.

2. Load the **`iris.csv`** data:

```
df = pd.read_csv('iris.csv')
```

3. Let's examine the first five rows of the data:

```
df.head()
```

The output will be as follows:

	petallength	petalwidth	sepallength	sepalwidth	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

Figure 2.17: Contents of the DataFrame

4. Visualize the data by using a scatter plot:

```
plt.scatter(df[df['species'] == 0]['sepallength'],\
            df[df['species'] == 0]['sepalwidth'], marker='*')
plt.scatter(df[df['species'] == 1]['sepallength'],\
            df[df['species'] == 1]['sepalwidth'], marker='<')
plt.scatter(df[df['species'] == 2]['sepallength'], \
            df[df['species'] == 2]['sepalwidth'], marker='o')
```

The resulting plot will be as follows. The *x* axis denotes the sepal length and the *y* axis denotes the sepal width. The shapes in the plot represent the three species of Iris, setosa (star), versicolor (triangle), and virginica (circle):

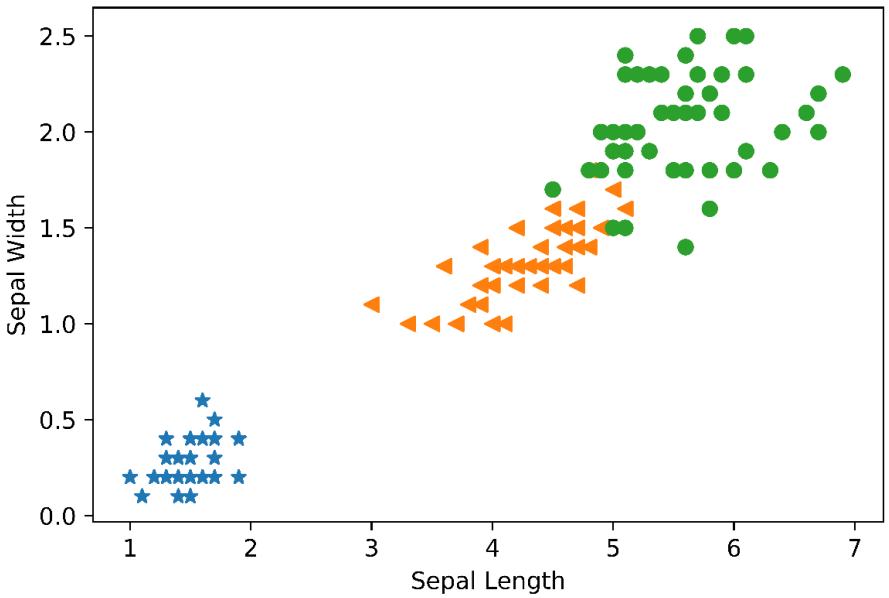

Figure 2.18: Iris data scatter plot

There are three classes, as can be seen in the visualization, denoted by different shapes.

5. Separate the features and the labels:

```
x = df[['petallength', 'petalwidth', \
        'sepallength', 'sepalwidth']].values
y = df['species'].values
```

`values` will transform the features into matrix format.

6. Prepare the data by doing one-hot encoding on the classes:

```
y = get_dummies(y)
y = y.values
```

`get_dummies(y)` will convert the labels into one-hot-encoded format.

7. Create a variable to load the features and typecast it to **`float32`**:

```
x = tf.Variable(x, dtype=tf.float32)
```

8. Implement the **`perceptron`** layer with three neurons:

```
Number_of_features = 4
Number_of_units = 3

# weights and bias
weight = tf.Variable(tf.zeros([Number_of_features, \
                               Number_of_units]))
bias = tf.Variable(tf.zeros([Number_of_units]))
def perceptron(x):
    z = tf.add(tf.matmul(x, weight), bias)
    output = tf.nn.softmax(z)
    return output
```

The code looks very similar to the single perceptron implementation. Only the **`Number_of_units`** parameter is set to **`3`**. Therefore, the weight matrix will be 4 x 3 and the bias matrix will be 1 x 3.

The other change is in the activation function:

`Output=tf.nn.softmax(x)`

We are using **`softmax`** instead of **`sigmoid`**.

9. Create an instance of the **`optimizer`**. We will be using the **`Adam`** optimizer. At this point, you can think of **`Adam`** as an improved version of gradient descent that converges faster. We will cover it in detail later in the chapter:

```
optimizer = tf.optimizers.Adam(.01)
```

10. Define the training function:

```
def train(i):
    for n in range(i):
        loss=lambda: abs(tf.reduce_mean\
                         (tf.nn.softmax_cross_entropy_with_logits(\
                          labels=y, logits=perceptron(x))))
        optimizer.minimize(loss, [weight, bias])
```

Again, the code looks very similar to the single-neuron implementation except for the loss function. Instead of **`sigmoid_cross_entropy_with_logits`**, we use **`softmax_cross_entropy_with_logits`**.

11. Run the training for **`1000`** iterations:

```
train(1000)
```

12. Print the values of the weights to see if they have changed. This is also an indication that our perceptron is learning:

```
tf.print(weight)
```

The output shows the learned weights of our perceptron:

```
[[0.684310317 0.895633 -1.0132345]
 [2.6424644 -1.13437736 -3.20665336]
 [-2.96634197 -0.129377216 3.2572844]
 [-2.97383809 -3.13501668 3.2313652]]
```

13. To test the accuracy, we feed the features to predict the output and then calculate the accuracy using **`accuracy_score`**, like in the previous exercise:

```
ypred=perceptron(x)
ypred=tf.round(ypred)
accuracy_score(y, ypred)
```

The output is:

```
0.98
```

It has given 98% accuracy, which is pretty good.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/2Dhes3U.
>
> You can also run this example online at https://packt.live/3iJJKkm.
> You must execute the entire Notebook in order to get the desired result.

In this exercise, we performed multiclass classification using our perceptron. Let's do a more complex and interesting case study of the handwritten digit recognition dataset in the next section.

MNIST CASE STUDY

Now that we have seen how to train a single neuron and a single layer of neurons, let's take a look at more realistic data. MNIST is a famous case study. In the next exercise, we will create a 10-class classifier to classify the MNIST dataset. However, before that, you should get a good understanding of the MNIST dataset.

Modified National Institute of Standards and Technology (**MNIST**) refers to the modified dataset that the team led by Yann LeCun worked with at NIST. This project was aimed at handwritten digit recognition using neural networks.

We need to understand the dataset before we get into writing the code. The MNIST dataset is integrated into the TensorFlow library. It consists of 70,000 handwritten images of the digits 0 to 9:

Figure 2.19: Handwritten digits

When we say images, you might think these are JPEG files, but they are not. They are actually stored in the form of pixel values. As far as the computer is concerned, an image is a bunch of numbers. These numbers are pixel values ranging from 0 to 255. The dimension of each of these images is 28 x 28. The images are stored in the form of a 28 x 28 matrix, each cell containing real numbers ranging from 0 to 255. These are grayscale images (commonly known as black and white). 0 indicates white and 1 indicates complete black, and values in between indicate a certain shade of gray. The MNIST dataset is split into 60,000 training images and 10,000 test images.

Each image has a label associated with it ranging from 0 to 9. In the next exercise, let's build a 10-class classifier to classify the handwritten MNIST images.

EXERCISE 2.04: CLASSIFYING HANDWRITTEN DIGITS

In this exercise, we will build a single-layer 10-class classifier consisting of 10 neurons with the Softmax activation function. It will have an input layer of 784 pixels:

1. Import the required libraries and packages just like we did in the earlier exercise:

```
import tensorflow as tf
import pandas as pd
from sklearn.metrics import accuracy_score

import matplotlib.pyplot as plt
%matplotlib inline

from pandas import get_dummies
```

2. Create an instance of the MNIST dataset:

```
mnist = tf.keras.datasets.mnist
```

3. Load the MNIST dataset's **`train`** and **`test`** data:

```
(train_features, train_labels), (test_features, test_labels) = \
mnist.load_data()
```

4. Normalize the data:

```
train_features, test_features = train_features / 255.0, \
                                test_features / 255.0
```

5. Flatten the 2-dimensional images into row matrices. So, a 28 × 28 pixel gets flattened to **`784`** using the **`reshape`** function:

```
x = tf.reshape(train_features,[60000, 784])
```

6. Create a **`Variable`** with the features and typecast it to **`float32`**:

```
x = tf.Variable(x)
x = tf.cast(x, tf.float32)
```

7. Create a one-hot encoding of the labels and transform it into a matrix:

```
y_hot = get_dummies(train_labels)
y = y_hot.values
```

8. Create the single-layer neural network with **10** neurons and train it for **1000** iterations:

Exercise2.04.ipynb

```
#defining the parameters
Number_of_features = 784
Number_of_units = 10

# weights and bias
weight = tf.Variable(tf.zeros([Number_of_features, \
                               Number_of_units]))
bias = tf.Variable(tf.zeros([Number_of_units]))
```

The complete code for this step can be accessed from https://packt.live/3efd7Yh.

9. Prepare the test data to measure the accuracy:

```
# Prepare the test data to measure the accuracy.
test = tf.reshape(test_features, [10000, 784])
test = tf.Variable(test)
test = tf.cast(test, tf.float32)

test_hot = get_dummies(test_labels)
test_matrix = test_hot.values
```

10. Run the predictions by passing the test data through the network:

```
ypred = perceptron(test)
ypred = tf.round(ypred)
```

11. Calculate the accuracy:

```
accuracy_score(test_hot, ypred)
```

The predicted accuracy is:

```
0.9304
```

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/3efd7Yh.
>
> You can also run this example online at https://packt.live/2Oc83ZW.
> You must execute the entire Notebook in order to get the desired result.

In this exercise, we saw how to create a single-layer multi-neuron neural network and train it as a multiclass classifier.

The next step is to build a multilayer neural network. However, before we do that, we must learn about the Keras API, since we use Keras to build dense neural networks.

KERAS AS A HIGH-LEVEL API

In TensorFlow 1.0, there were several APIs, such as Estimator, Contrib, and layers. In TensorFlow 2.0, Keras is very tightly integrated with TensorFlow, and it provides a high-level API that is user-friendly, modular, composable, and easy to extend in order to build and train deep learning models. This also makes developing code for neural networks much easier. Let's see how it works.

EXERCISE 2.05: BINARY CLASSIFICATION USING KERAS

In this exercise, we will implement a very simple binary classifier with a single neuron using the Keras API. We will use the same `data.csv` file that we used in *Exercise 2.02*, *Perceptron as a Binary Classifier*:

> **NOTE**
>
> The dataset can be downloaded from GitHub by accessing the following GitHub link: https://packt.live/2BVtxIf.

1. Import the required libraries:

```
import tensorflow as tf

import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline

# Import Keras libraries
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

In the code, **`Sequential`** is the type of Keras model that we will be using because it is very easy to add layers to it. **`Dense`** is the type of layer that will be added. These are the regular neural network layers as opposed to the convolutional layers or pooling layers that will be used later on.

2. Import the data:

```
df = pd.read_csv('data.csv')
```

3. Inspect the data:

```
df.head()
```

The following will be the output:

	label	x1	x2
0	1	2.6487	4.5192
1	1	1.5438	2.4443
2	1	1.8990	4.2409
3	1	2.4711	5.8097
4	1	3.3590	6.4423

Figure 2.20: Contents of the DataFrame

4. Visualize the data using a scatter plot:

```
plt.scatter(df[df['label'] == 0]['x1'], \
            df[df['label'] == 0]['x2'], marker='*')
plt.scatter(df[df['label'] == 1]['x1'], \
            df[df['label'] == 1]['x2'], marker='<')
```

The resulting plot is as follows, with the *x* axis denoting **`x1`** values and the y-axis denoting **`x2`** values:

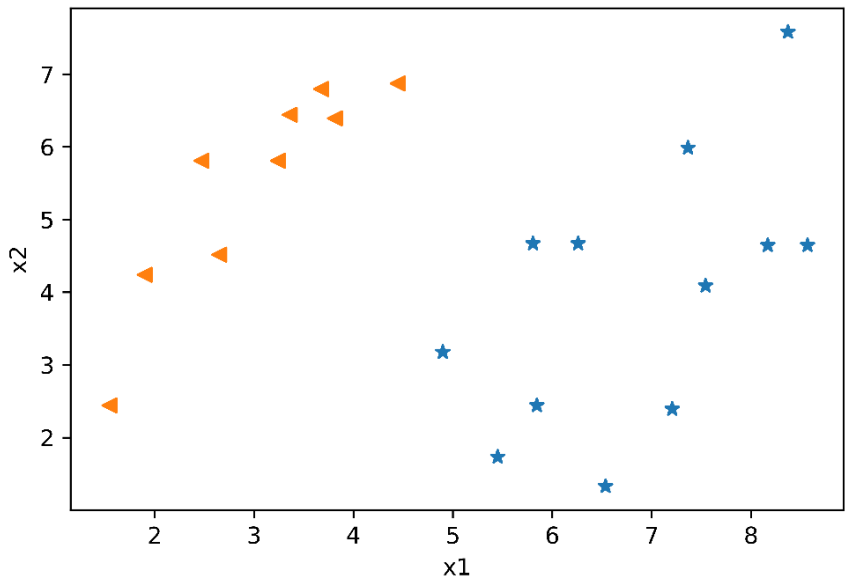

Figure 2.21: Scatter plot of the data

5. Prepare the data by separating the features and labels and setting the **`tf`** variables:

```
x_input = df[['x1','x2']].values
y_label = df[['label']].values
```

6. Create a neural network model consisting of a single layer with a neuron and a sigmoid activation function:

```
model = Sequential()
model.add(Dense(units=1, input_dim=2, activation='sigmoid'))
```

The parameters in **`mymodel.add(Dense())`** are as follows: **`units`** is the number of neurons in the layer; **`input_dim`** is the number of features, which in this case is **2**; and **`activation`** is **`sigmoid`**.

7. Once the model is created, we use the **`compile`** method to pass the additional parameters that are needed for training, such as the type of the optimizer, the loss function, and so on:

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

   In this case, we are using the **`adam`** optimizer, which is an enhanced version of the gradient descent optimizer, and the loss function is **`binary_crossentropy`**, since this is a binary classifier.

   The **`metrics`** parameter is almost always set to **`['accuracy']`**, which is used to display information such as the number of epochs, the training loss, the training accuracy, the test loss, and the test accuracy during the training process.

8. The model is now ready to be trained. However, it is a good idea to check the configuration of the model by using the **`summary`** function:

```
model.summary()
```

   The output will be as follows:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 1)                 3
=================================================================
Total params: 3
Trainable params: 3
Non-trainable params: 0
_________________________________________________________________
```

Figure 2.22: Summary of the sequential model

9. Train the model by calling the **`fit()`** method:

```
model.fit(x_input, y_label, epochs=1000)
```

It takes the features and labels as the data parameters along with the number of epochs, which in this case is **`1000`**. The model will start training and will continuously provide the status as shown here:

```
Epoch 994/1000
21/21 [==============================] - 0s 144us/sample - loss: 0.2458 - accuracy: 1.0000
Epoch 995/1000
21/21 [==============================] - 0s 144us/sample - loss: 0.2456 - accuracy: 1.0000
Epoch 996/1000
21/21 [==============================] - 0s 123us/sample - loss: 0.2454 - accuracy: 1.0000
Epoch 997/1000
21/21 [==============================] - 0s 146us/sample - loss: 0.2451 - accuracy: 1.0000
Epoch 998/1000
21/21 [==============================] - 0s 143us/sample - loss: 0.2449 - accuracy: 1.0000
Epoch 999/1000
21/21 [==============================] - 0s 134us/sample - loss: 0.2447 - accuracy: 1.0000
Epoch 1000/1000
21/21 [==============================] - 0s 174us/sample - loss: 0.2444 - accuracy: 1.0000
```

Figure 2.23: Model training logs using Keras

10. We will evaluate our model using Keras's **`evaluate`** functionality:

```
model.evaluate(x_input, y_label)
```

The output is as follows:

```
21/21 [==============================] - 0s 611us/sample - loss:
  0.2442 - accuracy: 1.0000
[0.24421504139900208, 1.0]
```

As you can see, our Keras model is able to train well, as our accuracy is 100%.

In this exercise, we have learned how to build a perceptron using Keras. As you have seen, Keras makes the code more modular and more readable, and the parameters easier to tweak. In the next section, we will see how to build a multilayer or deep neural network using Keras.

MULTILAYER NEURAL NETWORK OR DEEP NEURAL NETWORK

In the previous example, we developed a single-layer neural network, often referred to as a shallow neural network. A diagram of this follows:

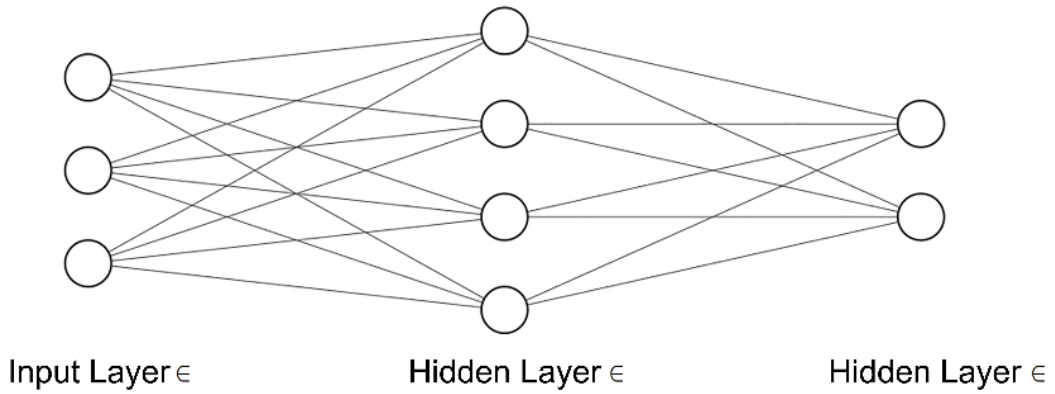

Figure 2.24: Shallow neural network

One layer of neurons is not sufficient to solve more complex problems, such as face recognition or object detection. You need to stack up multiple layers. This is often referred to as creating a deep neural network. A diagram of this follows:

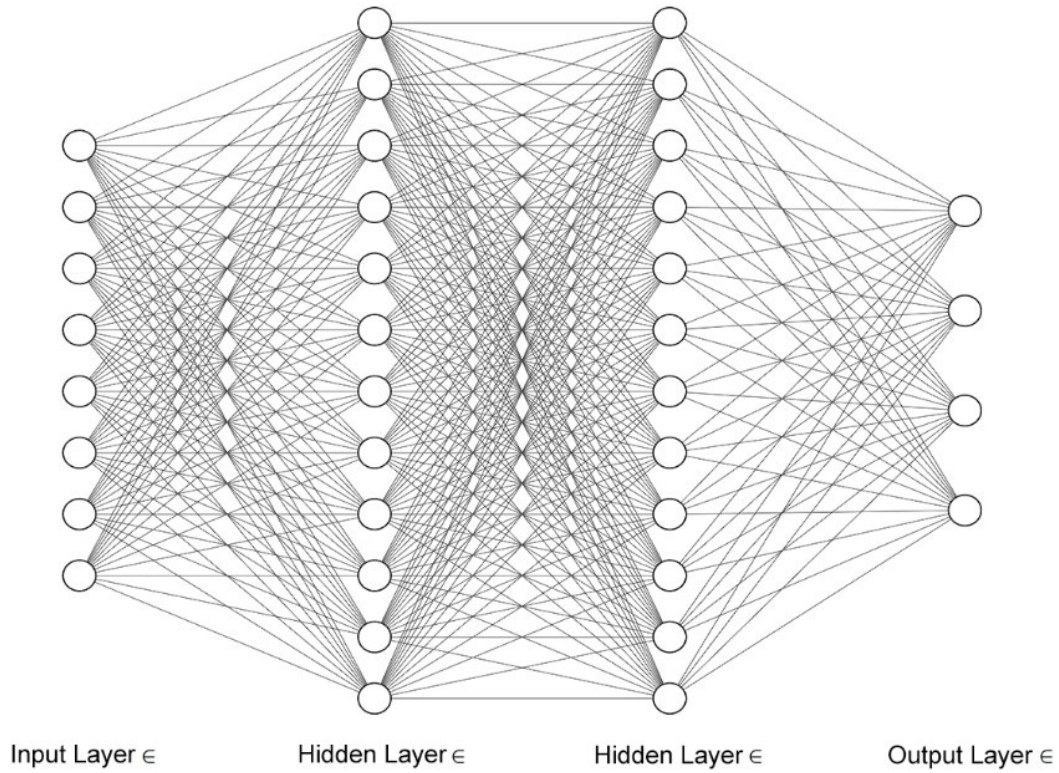

Figure 2.25: Deep neural network

Before we jump into the code, let's try to understand how this works. Input data is fed to the neurons in the first layer. It must be noted that every input is fed to every neuron in the first layer, and every neuron has one output. The output from each neuron in the first layer is fed to every neuron in the second layer. The output of each neuron in the second layer is fed to every neuron in the third layer, and so on.

That is why this kind of network is also referred to as a dense neural network or a fully connected neural network. There are other types of neural networks with different workings, such as CNNs, but that is something we will discuss in the next chapter. There is no set rule about the number of neurons in each layer. This is usually determined by trial and error in a process known as hyperparameter tuning (which we'll learn about later in the chapter). However, when it comes to the number of neurons in the last layers, there are some restrictions. The configuration of the last layer is determined as follows:

Binary classification	**Multiclass classification**
No. of Neurons = 1 Activation function – Sigmoid	No. of Neurons = number of classes Activation function - Softmax

Figure 2.26: Last layer configuration

RELU ACTIVATION FUNCTION

One last thing to do before we implement the code for deep neural networks is learn about the ReLU activation function. This is one of the most popular activation functions used in multilayer neural networks.

ReLU is a shortened form of **Rectified Linear Unit**. The output of the ReLU function is always a non-negative value that is greater than or equal to 0:

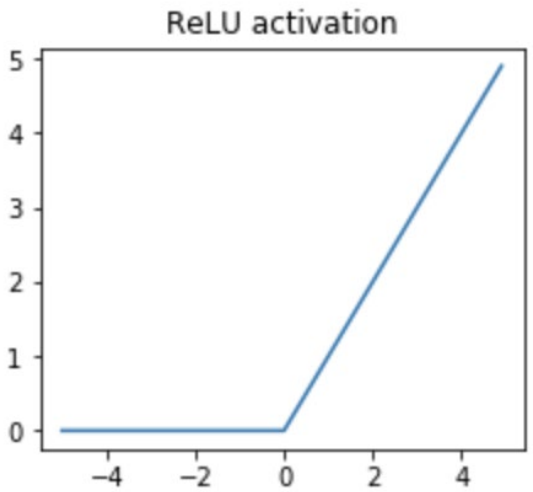

Figure 2.27: ReLU activation function

The mathematical expression for ReLU is:

$$f(x) = max(0,x)$$

Figure 2.28: ReLU activation function

ReLU converges much more quickly than the sigmoid activation function, and therefore it is by far the most widely used activation function. ReLU is used in almost every deep neural network. It is used in all the layers except the last layer, where either sigmoid or Softmax is used.

The ReLU activation function is provided by TensorFlow out of the box. To see how it is implemented, let's give some sample input values to a ReLU function and see the output:

```
values = tf.Variable([1.0, -2., 0., 0.3, -1.5], dtype=tf.float32)
output = tf.nn.relu(values)
tf.print(output)
```

The output is as follows:

```
[1 0 0 0.3 0]
```

As you can see, all the positive values are retained, and the negative values are suppressed to zero. Let's use this ReLU activation function in the next exercise to do a multilayer binary classification task.

EXERCISE 2.06: MULTILAYER BINARY CLASSIFIER

In this exercise, we will implement a multilayer binary classifier using the **`data.csv`** file that we used in *Exercise 2.02*, *Perceptron as a Binary Classifier*.

We will build a binary classifier with a deep neural network of the following configuration. There will be an input layer with 2 nodes and 2 hidden layers, the first with 50 neurons and the second with 20 neurons, and lastly a single neuron to do the final prediction belonging to any binary class:

> **NOTE**
>
> The dataset can be downloaded from GitHub using the following link: https://packt.live/2BVtxIf .

1. Import the required libraries and packages:

```
import tensorflow as tf
import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline

##Import Keras libraries
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

2. Import and inspect the data:

```
df = pd.read_csv('data.csv')
df.head()
```

The output is as follows:

	label	x1	x2
0	1	2.6487	4.5192
1	1	1.5438	2.4443
2	1	1.8990	4.2409
3	1	2.4711	5.8097
4	1	3.3590	6.4423

Figure 2.29: The first five rows of the data

3. Visualize the data using a scatter plot:

```
plt.scatter(df[df['label'] == 0]['x1'], \
            df[df['label'] == 0]['x2'], marker='*')
plt.scatter(df[df['label'] == 1]['x1'], \
            df[df['label'] == 1]['x2'], marker='<')
```

The resulting output is as follows, with the *x* axis showing **`x1`** values and the *y* axis showing **`x2`** values:

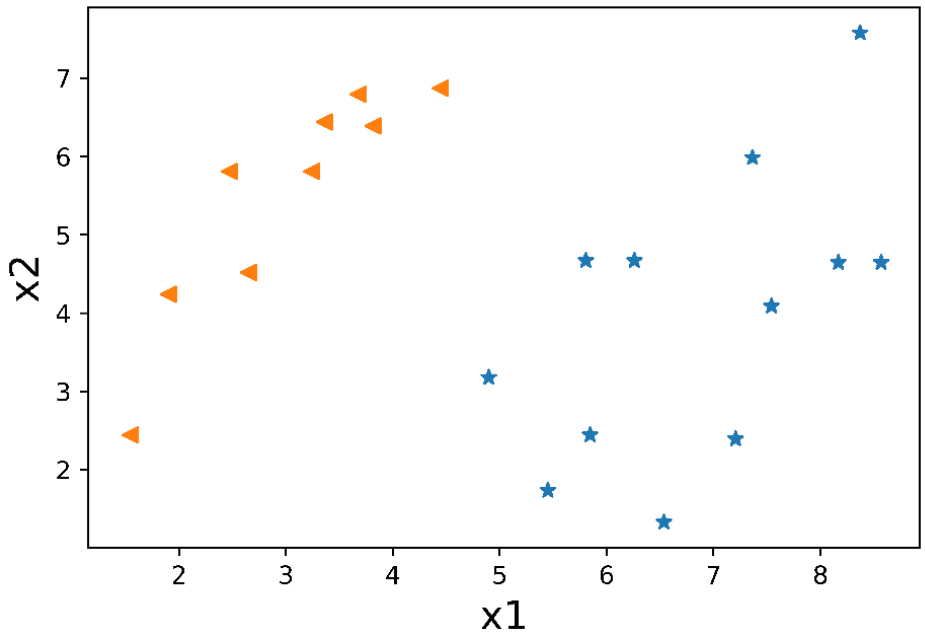

Figure 2.30: Scatter plot for given data

4. Prepare the data by separating the features and labels and setting the **`tf`** variables:

```
x_input = df[['x1','x2']].values
y_label = df[['label']].values
```

5. Build the **`Sequential`** model:

```
model = Sequential()
model.add(Dense(units = 50,input_dim=2, activation = 'relu'))
model.add(Dense(units = 20 , activation = 'relu'))
model.add(Dense(units = 1,input_dim=2, activation = 'sigmoid'))
```

Here are a couple of points to consider. We provide the input details for the first layer, then use the ReLU activation function for all the intermediate layers, as discussed earlier. Furthermore, the last layer has only one neuron with a sigmoid activation function for binary classifiers.

6. Provide the training parameters using the **`compile`** method:

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

7. Inspect the **`model`** configuration using the **`summary`** function:

```
model.summary()
```

The output will be as follows:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 50)                150
_________________________________________________________________
dense_1 (Dense)              (None, 20)                1020
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 21
=================================================================
Total params: 1,191
Trainable params: 1,191
Non-trainable params: 0
_________________________________________________________________
```

Figure 2.31: Deep neural network model summary using Keras

In the model summary, we can see that there are a total of **`1191`** parameters—weights and biases—to learn across the hidden layers to the output layer.

8. Train the model by calling the **`fit()`** method:

```
model.fit(x_input, y_label, epochs=50)
```

Notice that, in this case, the model reaches 100% accuracy within **`50`** epochs, unlike the single-layer model, which needed about 1,000 epochs:

```
Epoch 8/50
21/21 [==============================] - 0s 285us/sample - loss: 0.4644 - accuracy: 0.5714
Epoch 9/50
21/21 [==============================] - 0s 285us/sample - loss: 0.4502 - accuracy: 0.5714
Epoch 10/50
21/21 [==============================] - 0s 380us/sample - loss: 0.4378 - accuracy: 0.5714
Epoch 11/50
21/21 [==============================] - 0s 380us/sample - loss: 0.4267 - accuracy: 0.7143
Epoch 12/50
21/21 [==============================] - 0s 380us/sample - loss: 0.4163 - accuracy: 0.8571
Epoch 13/50
21/21 [==============================] - 0s 285us/sample - loss: 0.4059 - accuracy: 0.9524
Epoch 14/50
21/21 [==============================] - 0s 190us/sample - loss: 0.3960 - accuracy: 0.9524
Epoch 15/50
21/21 [==============================] - 0s 238us/sample - loss: 0.3863 - accuracy: 1.0000
Epoch 16/50
21/21 [==============================] - 0s 332us/sample - loss: 0.3769 - accuracy: 1.0000
Epoch 17/50
21/21 [==============================] - 0s 380us/sample - loss: 0.3677 - accuracy: 1.0000
Epoch 18/50
21/21 [==============================] - 0s 237us/sample - loss: 0.3585 - accuracy: 1.0000
Epoch 19/50
21/21 [==============================] - 0s 236us/sample - loss: 0.3495 - accuracy: 1.0000
```

Figure 2.32: Multilayer model train logs

9. Let's evaluate the model's performance:

```
model.evaluate(x_input, y_label)
```

The output is as follows:

```
21/21 [==============================] - 0s 6ms/sample - loss:
  0.1038 - accuracy: 1.0000
[0.1037961095571518, 1.0]
```

Our model has now been trained and demonstrates 100% accuracy.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/2ZUkM94.
>
> You can also run this example online at https://packt.live/3iKsD1W. You must execute the entire Notebook in order to get the desired result.

In this exercise, we learned how to build a multilayer neural network using Keras. This is a binary classifier. In the next exercise, we will build a deep neural network for a multiclass classifier with the MNIST dataset.

EXERCISE 2.07: DEEP NEURAL NETWORK ON MNIST USING KERAS

In this exercise, we will perform a multiclass classification by implementing a deep neural network (multi-layer) for the MNIST dataset where our input layer comprises 28 × 28 pixel images flattened to 784 input nodes followed by 2 hidden layers, the first with 50 neurons and the second with 20 neurons. Lastly, there will be a Softmax layer consisting of 10 neurons since we are classifying the handwritten digits into 10 classes:

1. Import the required libraries and packages:

```
import tensorflow as tf
import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline

# Import Keras libraries
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten
```

2. Load the MNIST data:

```
mnist = tf.keras.datasets.mnist
(train_features,train_labels), (test_features,test_labels) = \
mnist.load_data()
```

`train_features` has the training images in the form of 28 x 28 pixel values.

`train_labels` has the training labels. Similarly, **`test_features`** has the test images in the form of 28 x 28 pixel values. **`test_labels`** has the test labels.

3. Normalize the data:

```
train_features, test_features = train_features / 255.0, \
                                test_features / 255.0
```

The pixel values of the images range from 0-255. We need to normalize the values by dividing them by 255 so that the range goes from 0 to 1.

4. Build the **sequential** model:

```
model = Sequential()
model.add(Flatten(input_shape=(28,28)))
model.add(Dense(units = 50, activation = 'relu'))
model.add(Dense(units = 20 , activation = 'relu'))
model.add(Dense(units = 10, activation = 'softmax'))
```

There are couple of points to note. The first layer in this case is not actually a layer of neurons but a **Flatten** function. This flattens the 28 x 28 image into a single array of **784**, which is fed to the first hidden layer of **50** neurons. The last layer has **10** neurons corresponding to the 10 classes with a **softmax** activation function.

5. Provide training parameters using the **compile** method:

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

> **NOTE**
>
> The loss function used here is different from the binary classifier. For a multiclass classifier, the following loss functions are used: **sparse_categorical_crossentropy**, which is used when the labels are not one-hot encoded, as in this case; and, **categorical_crossentropy**, which is used when the labels are one-hot encoded.

6. Inspect the model configuration using the **summary** function:

```
model.summary()
```

The output is as follows:

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_2 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_9 (Dense)              (None, 50)                39250
_________________________________________________________________
dense_10 (Dense)             (None, 20)                1020
_________________________________________________________________
dense_11 (Dense)             (None, 10)                210
=================================================================
Total params: 40,480
Trainable params: 40,480
Non-trainable params: 0
```

Figure 2.33: Deep neural network summary

In the model summary, we can see that there are a total of 40,480 parameters—weights and biases—to learn across the hidden layers to the output layer.

7. Train the model by calling the **`fit`** method:

```
model.fit(train_features, train_labels, epochs=50)
```

The output will be as follows:

```
Train on 60000 samples
Epoch 1/50
60000/60000 [==============================] - 3s 58us/sample - loss: 0.3271 - accuracy: 0.9064
Epoch 2/50
60000/60000 [==============================] - 3s 49us/sample - loss: 0.1521 - accuracy: 0.9553s - loss: 0.153
Epoch 3/50
60000/60000 [==============================] - 3s 52us/sample - loss: 0.1130 - accuracy: 0.9660
Epoch 4/50
60000/60000 [==============================] - 4s 72us/sample - loss: 0.0925 - accuracy: 0.9717
Epoch 5/50
60000/60000 [==============================] - 3s 52us/sample - loss: 0.0779 - accuracy: 0.9761
Epoch 6/50
60000/60000 [==============================] - 3s 49us/sample - loss: 0.0672 - accuracy: 0.9789
Epoch 7/50
60000/60000 [==============================] - 4s 64us/sample - loss: 0.0589 - accuracy: 0.9814
Epoch 8/50
60000/60000 [==============================] - 3s 50us/sample - loss: 0.0520 - accuracy: 0.9844
Epoch 9/50
60000/60000 [==============================] - 3s 53us/sample - loss: 0.0466 - accuracy: 0.9851
Epoch 10/50
60000/60000 [==============================] - 4s 62us/sample - loss: 0.0408 - accuracy: 0.9868
Epoch 11/50
60000/60000 [==============================] - 3s 47us/sample - loss: 0.0378 - accuracy: 0.9882
```

Figure 2.34: Deep neural network training logs

8. Test the model by calling the **`evaluate()`** function:

```
model.evaluate(test_features, test_labels)
```

The output will be:

```
10000/10000 [==============================] - 1s 76us/sample - loss:
  0.2072 - accuracy: 0.9718
[0.20719025060918111, 0.9718]
```

Now that the model is trained and tested, in the next few steps, we will run the prediction with some images selected randomly.

9. Load a random image from a test dataset. Let's locate the 200th image:

```
loc = 200
test_image = test_features[loc]
```

10. Let's see the shape of the image using the following command:

```
test_image.shape
```

The output is:

```
(28,28)
```

We can see that the shape of the image is 28 x 28. However, the model expects 3-dimensional input. We need to reshape the image accordingly.

11. Use the following code to reshape the image:

```
test_image = test_image.reshape(1,28,28)
```

12. Let's call the **`predict()`** method of the model and store the output in a variable called **`result`**:

```
result = model.predict(test_image)
print(result)
```

`result` has the output in the form of 10 probability values, as shown here:

```
[[2.9072076e-28 2.1215850e-29 1.7854708e-21
  1.0000000e+00 0.0000000e+00 1.2384960e-15
  1.2660366e-34 1.7712217e-32 1.7461657e-08
  9.6417470e-29]]
```

13. The position of the highest value will be the prediction. Let's use the **`argmax`** function we learned about in the previous chapter to find out the prediction:

```
result.argmax()
```

In this case, it is **3**:

```
3
```

14. In order to check whether the prediction is correct, we check the label of the corresponding image:

```
test_labels[loc]
```

Again, the value is **3**:

```
3
```

15. We can also visualize the image using **`pyplot`**:

```
plt.imshow(test_features[loc])
```

The output will be as follows:

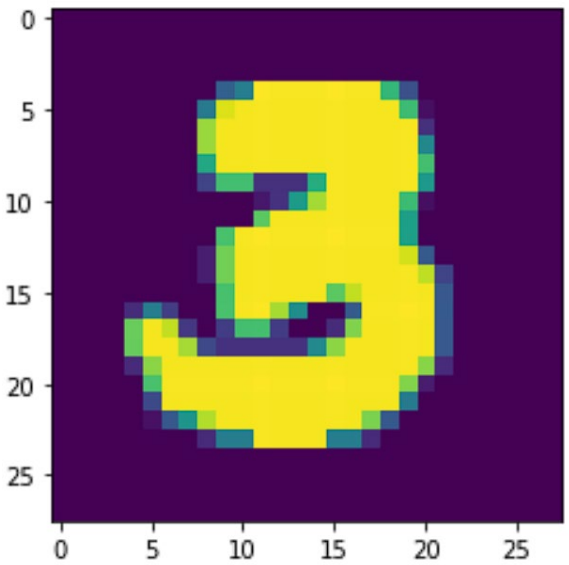

Figure 2.35: Test image visualized

And this shows that the prediction is correct.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/2O5KRgd.
>
> You can also run this example online at https://packt.live/2O8JHR0.
> You must execute the entire Notebook in order to get the desired result.

In this exercise, we created a multilayer multiclass neural network model using Keras to classify the MNIST data. With the model we built, we were able to correctly predict a random handwritten digit.

EXPLORING THE OPTIMIZERS AND HYPERPARAMETERS OF NEURAL NETWORKS

Training a neural network to get good predictions requires tweaking a lot of hyperparameters such as optimizers, activation functions, the number of hidden layers, the number of neurons in each layer, the number of epochs, and the learning rate. Let's go through each of them one by one and discuss them in detail.

GRADIENT DESCENT OPTIMIZERS

In an earlier section titled *Perceptron Training Process in TensorFlow*, we briefly touched upon the gradient descent optimizer without going into the details of how it works. This is a good time to explore the gradient descent optimizer in a little more detail. We will provide an intuitive explanation without going into the mathematical details.

The gradient descent optimizer's function is to minimize the loss or error. To understand how gradient descent works, you can think of this analogy: imagine a person at the top of a hill who wants to reach the bottom. At the beginning of the training, the loss is large, like the height of the hill's peak. The functioning of the optimizer is akin to the person descending the hill to the valley at the bottom, or rather, the lowest point of the hill, and not climbing up the hill that is on the other side of the valley.

Remember the learning rate parameter that we used while creating the optimizer? That can be compared to the size of the steps the person takes to climb down the hill. If these steps are large, it is fine at the beginning since the person can climb down faster, but once they near the bottom, if the steps are too large, the person crosses over to the other side of the valley. Then, in order to climb back down to the bottom of the valley, the person will try to move back but will move over to the other side again. This results in going back and forth without reaching the bottom of the valley.

On the other hand, if the person takes very small steps (a very small learning rate), they will take forever to reach the bottom of the valley; in other words, the model will take forever to converge. So, finding a learning rate that is neither too small nor too big is very important. However, unfortunately, there is no rule of thumb to find out in advance what the right value should be—we have to find it by trial and error.

There are two main types of gradient-based optimizers: batch and stochastic gradient descent. Before we jump into them, let's recall that one epoch means a training iteration where the neural network goes through all the training examples:

- In an epoch, when we reduce the loss across all the training examples, it is called **batch gradient descent**. This is also known as **full batch gradient descent**. To put it simply, after going through a full batch, we take a step to adjust the weights and biases of the network to reduce the loss and improve the predictions. There is a similar form of it called mini-batch gradient descent, where we take steps, that is, we adjust weights and biases, after going through a subset of the full dataset.
- In contrast to batch gradient descent, when we take a step at one example per iteration, we have **stochastic gradient descent** (**SGD**). The word *stochastic* tells us there is randomness involved here, which, in this case, is the batch that is randomly selected.

Though SGD works relatively well, there are advanced optimizers that can speed up the training process. They include SGD with momentum, Adagrad, and Adam.

THE VANISHING GRADIENT PROBLEM

In the *Training a Perceptron* section, we learned about the forward and backward propagation of neural networks. When a neural network performs forward propagation, the error gradient is calculated with respect to the true label, and backpropagation is performed to see which parameters (the weights and biases) of the neural network have contributed to the error and the extent to which they have done so. The error gradient is propagated from the output layer to the input layer to calculate gradients with respect to each parameter, and in the last step, the gradient descent step is performed to adjust the weights and biases according to the calculated gradient. As the error gradient is propagated backward, the gradients calculated at each parameter become smaller and smaller as it advances to the lower (initial) layers. This decrease in the gradients means that the changes to the weights and biases become smaller and smaller. Hence, our neural network struggles to find the global minimum and does not give good results. This is called the vanishing gradient problem. The problem happens with the use of the sigmoid (logistic) function as an activation function, and hence we use the ReLU activation function to train deep neural network models to avoid gradient complications and improve the results.

HYPERPARAMETER TUNING

Like any other model training process in machine learning, it is possible to perform hyperparameter tuning to improve the performance of the neural network model. One of the parameters is the learning rate. The other parameters are as follows:

- **Number of epochs**: Increasing the number of epochs generally increases the accuracy and lowers the loss
- **Number of layers**: Increasing the number of layers increases the accuracy, as we saw in the exercises with MNIST
- **Number of neurons per layer**: This also increases the accuracy

And once again, there is no way to know in advance what the right number of layers or the right number of neurons per layer is. This has to be figured out by trial and error. It has to be noted that the larger the number of layers and the larger the number of neurons per layer, the greater the computational power required. Therefore, we start with the smallest possible numbers and slowly increase the number of layers and neurons.

OVERFITTING AND DROPOUT

Neural networks with complex architectures and too many parameters tend to fit on all the data points, including noisy labels, leading to the problem of overfitting and neural networks that are not able to generalize well on unseen datasets. To tackle this issue, there is a technique called **dropout**:

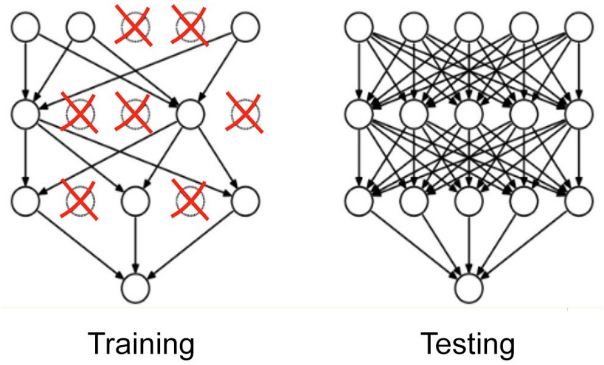

Figure 2.36: Dropout illustrated

In this technique, a certain number of neurons are deactivated randomly during the training process. The number of neurons to be deactivated is provided as a parameter in the form of a percentage. For example, **`Dropout = .2`** means 20% of the neurons in that layer will be randomly deactivated during the training process. The same neurons are not deactivated more than once, but a different set of neurons is deactivated in each epoch. During testing, however, all the neurons are activated.

Here is an example of how we can add **`Dropout`** to a neural network model using Keras:

```
model.add(Dense(units = 300, activation = 'relu')) #Hidden layer1
model.add(Dense(units = 200, activation = 'relu')) #Hidden Layer2
model.add(Dropout(.20))
model.add(Dense(units = 100, activation = 'relu')) #Hidden Layer3
```

In this case, a dropout of 20% is added to **`Hidden Layer2`**. It is not necessary for the dropout to be added to all layers. As a data scientist, you can experiment and decide what the **`dropout`** value should be and how many layers need it.

> **NOTE**
>
> A more detailed explanation of dropout can be found in the paper by Nitish Srivastava et al. available here: http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf.

As we have come to the end of this chapter, let's test what we have learned so far with the following activity.

ACTIVITY 2.01: BUILD A MULTILAYER NEURAL NETWORK TO CLASSIFY SONAR SIGNALS

In this activity, we will use the Sonar dataset (https://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+(Sonar,+Mines+vs.+Rocks)), which has patterns obtained by bouncing sonar signals off a metal cylinder at various angles and under various conditions. You will build a neural network-based classifier to classify between sonar signals bounced off a metal cylinder (the Mine class), and those bounced off a roughly cylindrical rock (the Rock class). We recommend using the Keras API to make your code more readable and modular, which will allow you to experiment with different parameters easily:

> **NOTE**
>
> You can download the sonar dataset from this link https://packt.live/31Xtm9M.

1. The first step is to understand the data so that you can figure out whether this is a binary classification problem or a multiclass classification problem.
2. Once you understand the data and the type of classification that needs to be done, the next step is network configuration: the number of neurons, the number of hidden layers, which activation function to use, and so on.

   Recall the network configuration steps that we've covered so far. Let's just reiterate a crucial point, the activation function part: for the output (the last) layer, we use sigmoid to do binary classification and Softmax to do multiclass classification.

3. Open the **`sonar.csv`** file to explore the dataset and see what the target variables are.
4. Separate the input features and the target variables.
5. Preprocess the data to make it neural network-compatible. Hint: one-hot encoding.
6. Define a neural network using Keras and compile it with the right loss function.
7. Print out a model summary to verify the network parameters and considerations.

You are expected to get an accuracy value above 95% by designing a proper multilayer neural network using these steps.

> **NOTE**
>
> The detailed steps for this activity, along with the solutions and additional commentary, are presented on page 390.

SUMMARY

In this chapter, we started off by looking at biological neurons and then moved on to artificial neurons. We saw how neural networks work and took a practical approach to building single-layer and multilayer neural networks to solve supervised learning tasks. We looked at how a perceptron works, which is a single unit of a neural network, all the way to a deep neural network capable of performing multiclass classification. We saw how Keras makes it very easy to create deep neural networks with a minimal amount of code. Lastly, we looked at practical considerations to take into account when building a successful neural network, which involved important concepts such as gradient descent optimizers, overfitting, and dropout.

In the next chapter, we will go to the next level and build a more complicated neural network called a CNN, which is widely used in image recognition.